Acting and Learning with Goal and Task Decomposition

by

Maciej Wojnar

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2011

Abstract

Two central problems of creating artificial intelligent agents that can operate in the human world are learning the necessary knowledge to achieve routine tasks, and using that knowledge effectively in a complex and unpredictable domain.

The thesis argues that an important part of this domain knowledge should be represented in the form of decomposition rules that decompose tasks into subgoals.

The thesis presents HOPPER, an implemented planning system that uses decomposition rules and a least-commitment decomposition strategy that strikes a balance between reactive and deliberative planning. Like reactive planners, HOPPER is able to robustly handle and recover from unexpected events with minimal disruption to its plan. Like deliberative planners, it is also able to plan ahead to take advantage of opportunities to interleave and shorten its sub-plans.

The thesis also presents TADPOLE, an implemented learning system that learns both the structure and preconditions of new decomposition rules from a small number of lessons demonstrated by a teacher. It learns by parsing and interpreting the teacher's behaviour in terms of decomposition rules it already knows. It extends its rule set by filling in the holes in its parses of the teacher's lessons.

Both HOPPER and TADPOLE have been evaluated together in two different domains: a kitchen domain that emphasizes complexity, and a logistics domain that emphasizes plan efficiency. Every rule used by HOPPER was learned by TADPOLE and every rule learned by TADPOLE was successfully used by HOPPER to achieve various tasks, showing that TADPOLE is able to learn effective decomposition rules from minimal lessons from a teacher, and that HOPPER is able to robustly make use of them even in the face of unexpected events.

To our future robotic overlords,
may they have superior mercy as well as
superior intelligence.

Acknowledgements

This thesis has taken some unexpected twists and turns, seeming to grow in size and scope with each metamorphosis. Keeping it under control has been an exhausting but ultimately rewarding endeavour.

I am especially grateful to my supervisor, Peter Andreae (Pondy). His clarity of thought has been particularly valuable; and after every meeting I always left energized, motivated, and excited about the thesis. I very much enjoyed our many wide-ranging philosophical discussions which every meeting inevitably ended in. I found Pondy's outlook to be very different from my own, challenging my preconceptions and broadening my horizons almost as much as the thesis itself.

In turn, my fellow graduate students were great at exchanging and debating ideas. The whiteboards were always covered with incomprehensible doodles, and there were often two people loudly arguing about some minor point. The Festival of Doubt meetings (when we actually managed to organize them) were extremely enjoyable and provided a range of perspectives from people working in different fields of Artificial Intelligence.

I am very grateful to the Tertiary Education Commission for the Bright Future Top Achiever Scholarship that has funded my research.

Finally, thank you Mum for bolstering my spirits with your delicious Polish cooking. Thank you Dad for politely listening to me ramble on about the minutiae of my thesis and my various philosophical musings about the nature of the mind. And I want to thank both of my sisters and my grandmother for their wholehearted encouragement and support.

Contents

Chapter 1

Introduction

This thesis describes the problem of learning the knowledge needed to effectively achieve routine tasks in a complex, unpredictable domain. The thesis presents two implemented algorithms that address this problem: HOPPER, a system that uses decomposition rules to robustly achieve tasks, recovering from unexpected events and exploiting unexpected opportunities; and TADPOLE, a system that interprets and learns decomposition rules from the demonstrated behaviour of a teacher.

Organization of the chapter

This chapter first discusses what makes designing intelligent agents that are effective in a general, human environment an interesting and important field of research. It then highlights the characteristics of the human planning domain that make this such a challenging task. The chapter goes on to broadly cover the previous approaches that have been taken and then presents the contributions of this thesis.

- Section 1.1 explains why the problem of making an intelligent agent that learns and acts effectively in the human domain is interesting and important. It highlights both the practical and theoretical benefits such an intelligent agent would bring.

- Section 1.2 describes the characteristics of the human domain that make creating an intelligent agent that can act effectively in this domain a particularly challenging problem.
- Section 1.3 presents the two classical approaches to the planning problem: classical planners and reactive systems.
- Classical planners and reactive systems ignore important aspects of the human domain. Section 1.4 describes the problem of scaling these systems to the human domain.
- Section 1.5 describes hierarchical decomposing systems which are more appropriate to achieving tasks in the human domain. The section then discusses the problem of learning the necessary domain knowledge for them to be effective.
- Section 1.6 outlines the key contributions of the thesis.
- Section 1.7 concludes the chapter by presenting an outline of the rest of the thesis.

1.1 Intelligent Agents

Creating an intelligent agent that can learn, plan and act effectively in domains that humans excel at (which I will refer to as the Human Planning Domain, or HPD) has been one of the central problems in Artificial Intelligence since the field's beginning.

1.1.1 Developing an intelligent agent would be practically useful

Intelligent agents that could duplicate the versatility and adaptability of human intelligence would allow robotics to move from the restricted and specialized domains it is applied in now, such as manufacturing assembly lines, and into the human world. Such intelligent agents could be used as robotic assistants to help people with their daily tasks including doing the shopping, preparing meals, and cleaning the house. Robotic assistants would be particularly helpful for people of limited capabilities, such as the elderly and disabled.

As well as helping with mundane tasks, intelligent agents could also be used for tasks that are too dangerous for humans but that nevertheless require creativity and autonomy to be successfully accomplished. For example, remote probes for exploring extra-terrestrial environments would have to be able to make autonomous decisions in potentially novel environments (what the atmosphere of Jupiter turns out to be like beneath the surface may turn out to be a complete surprise).

1.1.2 Developing an intelligent agent would help us understand intelligence

An important way of learning about a system is building something that duplicates its abilities. In the same way that the invention of the airplane helped us to understand how birds fly and the nature of aerodynamics in

general, intelligent agents could help us to understand how people think and the nature of intelligence in general.

1.2 The Human Planning Domain

The HPD spans the range of environments and domains that humans excel at, and this section describes the properties and characteristics of this domain that constrain the design of an intelligent agent.

1.2.1 The HPD includes a large number of different tasks

Being successful in the HPD means being able to accomplish an indefinitely wide variety of tasks (as many and as varied as people are able to achieve) in indefinitely varied situations. This means that an effective intelligent agent needs a mechanism for acquiring new knowledge for achieving novel tasks and it needs to be able to apply it in novel situations.

1.2.2 The HPD is complex

The HPD is very complicated with a vast number of objects each with many properties and many rich relationships with the objects around them. A single room in a household, such as a kitchen, can contain hundreds or thousands of objects. Although most of these objects and their properties and relationships are irrelevant to any given task, what aspects of the state are relevant or irrelevant depends on what task is being achieved. For example, the sink is important for washing the dishes, but completely irrelevant to vacuuming the house. Any algorithms used by the intelligent agent must be able to scale to domains with many objects, properties, and relationships. The intelligent agent also needs to have a way of distinguishing the relevant and irrelevant features of the world depending on the task it is trying to achieve or learning how to achieve.

1.2.3 The HPD is unpredictable

An intelligent agent can only observe a small part of the state in the HPD, and of the objects that its sensors *can* observe it may not be able to observe all of their relevant properties and relationships (*e.g.* whether or not a container is locked). Therefore the agent can never be certain what the true state of the world actually is. Furthermore, even if the agent knew what the current state is, it would need to fully understand the physical processes governing the domain to be able to accurately determine how the state will change over time. In domains as complex and rich as the HPD, this is not feasible. However, even this would not be enough if there are other agents in the domain affecting world, because other agents are never completely predictable.

Because the agent does not have complete information about the state of the world and it does not have complete information about how the state will change over time, it must be able to deal with and adapt to unexpected and unpredictable events that may be disruptive to the tasks it is achieving.

1.3 Planning and Reacting

Classical planners and reactive systems take diametrically opposed approaches to the planning problem. Classical planners attempt to predict all of the atomic actions that are needed to achieve a task. Reactive systems look only at the current state to find the appropriate action to execute next. Neither of these two extremes is appropriate for the HPD.

1.3.1 Classical planners search for sequences of atomic actions

Classical planners construe the planning problem as the problem of generating a sequence of atomic actions that, when executed in order, transform an initial state to a final state that satisfies some goal constraint(s). The archetypal example of such a classical planner is STRIPS [16].

Classical planners are characterized by having a limited amount of fixed domain knowledge in the form of action rules that specify the effects and preconditions of atomic actions. This knowledge is hand-coded and the agent does not need to do any learning.

Classical planners search for a sequence of actions that will transform an initial state into a goal state, using only the knowledge expressed in the action rules. The depth of the search tree depends on the number of atomic actions in the sequence, and the branching factor is determined by the number of ways that one state can be transformed into another with an atomic action.

1.3.2 Reactive systems use a policy mapping states to actions

Instead of trying to plan out every atomic action from the current state all the way to the goal state (and possibly plan out contingency plans for every way that the state could unexpectedly change while the plan is being executed), reactive systems instead approach the planning problem from the opposite extreme and do no look ahead at all. Reactive systems use a policy that maps states to atomic actions to determine what action to execute in any given state. Because reactive systems have no look ahead, they make no predictions about future states, and so have no problem dealing with unexpected states (indeed they do not even make the distinction between expected and unexpected states). A classical example of such a reactive system is Pengi [1].

Reactive systems are characterized by having an unlimited amount of domain knowledge (encoded in their policies). They do not do any search for the appropriate atomic action to execute. Instead they only have to match the current state to their policy to determine what the appropriate action is.

Because the policies of reactive systems are extensive and unlimited, they cannot be specified or hand-coded for any domain of even moderate complexity. Instead the agent has to learn its policy by interacting with its environment. The agent can learn by trial and error, experimenting with various actions in different states and seeing which bring it closer to its goal. The agent's learning can also be sped up with guidance from a teacher.

1.4 Scaling to the HPD

Classical planners and reactive systems do not take account of one or more of the characteristics of the HPD described in Section 1.2, and so have difficulty scaling to the real world.

1.4.1 Classical planners do not take the HPD's complexity into account

The problem with applying classical planners to the HPD is that they do not take the HPD's complexity and unpredictability into account.

An intelligent agent may need to achieve a very wide range of different tasks in the HPD. In order for a classical planners to be that versatile with a limited number of action rules, the action rules that the agent knows have to be very general and low-level. Because of this, complex tasks require a long sequence of atomic actions to be achieved.

The HPD has hundreds or thousands of objects that the agent can manipulate (apply atomic actions to) in any given state. Each atomic action

applied to each object in the state generally results in a different state transformation.

The length of the sequence of atomic actions and the large number of possible state transformations possible in each state means that the search tree to find the appropriate sequence is very deep and its branching factor is very wide, making classical planning completely intractable for all but the simplest problems [15].

Domain knowledge is necessary to guide the search, but no limited set of hand-crafted rules is versatile enough to apply to every task in the HPD.

1.4.2 Classical planners do not take the HPD's unpredictability into account

When producing a sequence of atomic actions, a classical planner implicitly makes predictions about what the state will be when each of the actions will be executed. If an unexpected event causes any of these predictions to fail, then the entire plan is invalidated and the agent must re-plan from the new state. The longer the plan, the higher the chance that one of its predictions will fail making it more brittle and unreliable.

Contingency planning helps to ameliorate this problem by trying to predict in advance where the plan may fail and generating backup plans that will still achieve the goal. This approach is fundamentally limited by the need to know all the ways in which a plan could fail; in the HPD this is an impossible requirement to satisfy.

During execution, the unexpected invalidity of a plan is normally detected at the moment of plan failure: when the precondition of one of the atomic actions fails. However the event invalidating the plan may have occurred much earlier in the execution of the plan. The later the failure is detected, the greater the disruptive effect on the plan, because the actions after the invalidating event are no longer guaranteed to lead the agent closer to the goal. These actions will, at best, be an inefficient waste of

time, and at worst, can lead to a state that the agent cannot recover from. For example, if during the execution of a plan to fly to another country the agent discovers that its flight will leave an hour earlier, but the agent does not identify the invalidity of its plan until it tries to board the non-existent plane at the airport, then there will be no easy way for it to re-plan to achieve the original goal because by that time the plane will have already left.

In a dynamic domain where unexpected, usually innocuous, events are constantly occurring, it is difficult to determine whether a given event will invalidate the plan, and so the agent is forced to constantly simulate executing the rest of its plan to make sure that it is still viable. Furthermore, because each atomic action in the sequence depends on the state in which it will be executed, which in turn depends on all of the previous actions, altering an action will usually invalidate the rest of the plan. In general, when a plan is invalidated, then the agent has to re-plan from scratch as there is no real way to re-use the remaining invalidated action sequence.

A more insidious problem that classical planners face in an unpredictable domain is that of unexpected opportunities. These events change the state in a way that makes it possible for the goal to be achieved more efficiently (*e.g.* with a fewer number of atomic actions). However, because these actions do not invalidate the original plan, they are very hard for a classical planner to detect, let alone exploit. For example, if during the execution of a plan to fly to another country the agent discovers that a friend is driving to the airport at the same time, then the agent should somehow detect and exploit this opportunity even though its original plan of calling a taxi has not been invalidated.

1.4.3 Reactive systems do not take the large number of tasks in the HPD into account

Because a reactive policy is a mapping from states to actions, it is only useful for achieving a single task. The appropriate atomic action to execute in a particular state may be completely different depending on what the agent wants to achieve. For example, if the agent is in a kitchen and wants a pizza, then an appropriate atomic action is for it to pick up a phone (to order the pizza); in the same state if the agent wants to clean the kitchen, then an appropriate action is for it to pick up a broom (to sweep the floor). A single policy is inappropriate for a domain with multiple tasks for the agent to achieve.

A possible way to address multiple tasks is to extend the agent's domain knowledge to multiple policies, one for each task (or equivalently extend the agent's policy to be a mapping from states and tasks to actions). However this approach does not scale to domains with a large number of complex tasks.

The first problem is that many plans for achieving different tasks have much in common with each other. For example, most tasks an agent may have to achieve in a kitchen (*e.g.* baking a cake, washing the dishes) will involve opening and closing containers, and picking up and putting down objects. If the knowledge of how to achieve different tasks is stored in separate policies, then the agent has to redundantly re-learn such shared knowledge every time it learns how to achieve a new task.

The second problem is that the agent has to have a way to determine which task it should be trying to achieve at any given moment. Traditionally, tasks are simply assigned to a reactive agent. If this approach is to be at all applicable to the HPD, the agent can only be assigned high-level tasks such as "prepare a meal" or "pack for a trip". The lower the level of the tasks assigned to the agent, the more work is done for the agent, and the less versatile it is. An agent that has to be told in explicit detail which

low-level tasks to achieve (*e.g.* "open that cupboard", "pick up that cup", and so on) is of little use.

However, high-level tasks in the HPD are complex; different atomic actions are appropriate for achieving different parts of the task, and the current state of the world often does not contain enough information to disambiguate between the different cases. For example, the state of having a half-packed suitcase will occur for both packing and unpacking a suitcase, but the appropriate actions for these two cases are diametrically opposed: adding another item to the suitcase or removing one, respectively. Both packing and unpacking a suitcase may be necessary when achieving the task of "packing for a trip" (if the agent accidentally packed some belongings into a suitcase that was too small, then it would need to unpack that suitcase and move the items to a bigger suitcase), and so even if the domain knowledge for the task were encoded in a separate policy, the agent would still be unable to determine what to do in a state with a half-packed suitcase.

The fundamental problem is that packing and unpacking a suitcase are tasks themselves and the domain knowledge of how to achieve them ought to be encoded in a separate policy. However, because these tasks are too low-level to be directly assigned, a reactive agent has no way of learning appropriate policies for them, and it has no way of knowing when it should apply such policies even if it could learn them.

1.5 Hierarchical Decomposing Systems

Hierarchical decomposing systems encode their domain knowledge in decomposition rules. They encode richer domain knowledge, particularly knowledge of how to achieve goals rather than just the effects of actions, making them much more efficient than classical planners. They also deal with goals and not just states, so they can be flexibly applied to a wide variety of tasks, unlike policy-based systems.

With an extensive enough rule set, a hierarchical decomposing system can handle the complexity and the large number of tasks in the HPD. If it applies its rules reactively or semi-reactively, then it can deal with the unpredictability of the HPD as well. Because an extensive rule set is necessary for an agent to be effective in the HPD, learning decomposition rules is of fundamental importance to hierarchical decomposing systems; however, most research has been on applying decomposition rules, and not on learning them.

1.5.1 Decomposition rules break a problem down into a hierarchy of sub-problems

Decomposition rules are a powerful way of encoding domain knowledge. Each rule specifies how to solve a problem by solving a number of sub-problems. Given the appropriate rules, an agent can solve problems of arbitrary complexity by matching a problem with an appropriate rule, determining the corresponding sub-problems that need to be solved, and then recursively matching the sub-problems with its rules to get their corresponding sub-sub-problems. The agent continually does this, building up a hierarchy of problems and sub-problems, until it reaches atomic problems that it can solve directly (with an atomic action or by some other means).

1.5.2 Decomposition rules can be applied reactively or deliberatively

Hierarchical decomposing systems have applied their rules either completely reactively or completely deliberatively, but neither extreme is appropriate to the HPD.

Hierarchical decomposing systems that apply their rules deliberatively produce a complete plan of atomic actions from an initial state to a final

state similar to the plans that classical planners generate. This results in the same difficulties that classical planners face in unpredictable domains.

Systems that apply their rules completely reactively tend to generate sub-optimal behaviour. This is because a system without foresight that does not consider the future aspects of a plan will tend to produce inefficient plans with needless, redundant steps.

1.5.3 Decomposition rules need to be learned

Each decomposition rule specifies how to achieve a task. In the HPD an agent may need to achieve an indefinite number of tasks, and it cannot know ahead of time what those tasks may be. It is not feasible to hand-code all of the necessary decomposition rules that the agent may need, so it has to have a way of learning new decomposition rules and extending its rule set.

1.5.4 Decomposition rules are appropriate for achieving routine tasks

An agent can use decomposition rules to achieve routine tasks — tasks that it has experience solving (whether through its own efforts or by observing another). Although good decomposition rules can be used to achieve routine tasks in novel situations, by themselves they are insufficient for achieving truly novel tasks.

In the HPD, an agent can expect to be faced with problems and tasks that it has never seen before and has no experience of. In such cases, a hierarchical decomposing system will not be enough by itself. However, routine tasks make up the majority of the tasks an agent has to achieve in the HPD, and the algorithms used to achieve such tasks in a complex and unpredictable domain can serve as a foundation for reasoning that is necessary for solving novel tasks.

This thesis focuses on the problem of learning how to achieve routine tasks, and future work will focus on extending it to allow an agent to search for ways of achieving completely novel tasks.

1.6 Contributions

This thesis presents three core contributions:

- a rule framework that decouples tasks and goals, resulting in rules that are less constrained and more re-usable than previous rule frameworks.
- a decomposition planner called HOPPER that overcomes the shortcomings that other decomposition planners face in unpredictable domains.
- a learner called TADPOLE that places a much smaller burden on the teacher than previous learners, requiring only unannotated, demonstrated lessons to learn complete decomposition rules.

1.6.1 De-coupled rules are more flexible and re-usable

Previous decomposing systems have used rules that decompose goals into sub-goals, coupling goals with the way to achieve them. The consequence of this approach is that each goal and sub-goal requires a separate rule, and each rule can only ever be used to achieve a single goal.

The rules learned by TADPOLE and applied by HOPPER decompose tasks into sub-goals. Each rule specifies how to achieve a state change rather than any particular goal. Because the rules do not specify what goal they are applicable to, the agent can determine this dynamically and re-use the same rule to achieve different goals. Although decoupling tasks from goals complicates both the learning and planning algorithms that make use of such rules, it results in more powerful and more flexible rules,

and ultimately requires fewer lessons for the agent to become capable in a given domain.

1.6.2 HOPPER uses a least-commitment decomposition strategy to react to unexpected events

Previous decomposition planners have insisted on generating the entire atomic plan and proving its correctness before beginning to execute it. However, this approach is not appropriate in an unpredictable domain. Previous decomposition planners have had difficulty detecting let alone responding to unexpected disruptions, and exploiting unexpected opportunities. Though decomposition planners have previously interleaved decompositions to make more efficient plans, they have relied on hand-crafted rules annotated with interleaving points.

HOPPER improves upon these previous algorithms by using a least-commitment decomposition strategy — generating the details of the plan only when necessary. It fully decomposes the current part of the plan being executed, with future parts of the plan being increasingly less decomposed, more abstract, and less specified. HOPPER constantly makes and verifies its predictions of future states. This allows it to detect unexpected events, recover from disruptions, and to exploit opportunities by making minimal changes to its plan. HOPPER takes advantage of opportunities to shorten its plan by employing a novel algorithm that can interleave the decompositions of completely unannotated decomposition rules.

HOPPERs least-commitment decomposition strategy does not allow it to prove plan correctness, but in an unpredictable domain it is not possible to guarantee that a plan will be executed successfully anyway.

1.6.3 TADPOLE learns complete rules and places a minimal burden on the teacher

Previous attempts to learn decomposition rules have focused on learning and refining the preconditions of rules the agent was already familiar with. The remaining algorithms for learning the structure of decomposition rules have placed an unreasonable burden on the teacher, requiring an inordinate amount of information such as plan traces and the complete, annotated decomposition hierarchies used to generate them.

TADPOLE improves upon these previous algorithms by using its previously learned decomposition rules to parse and interpret a teachers demonstrated lesson. TADPOLE reconstructs the decomposition hierarchy used by the teacher from their unannotated plan, and then uses that decomposition hierarchy to learn new decomposition rules. TADPOLE is able to learn complete decomposition rules (including their tasks, goals, and preconditions) from demonstrations consisting of nothing more than unannotated, atomic, sequences of states; greatly easing the burden placed on the teacher.

1.7 Outline of the thesis

The rest of the thesis is organized as follows:

- Chapter 2 covers the related work that has been done on hierarchical decomposing systems. Most previous systems have focused on applying decomposition rules and not on learning them. The chapter focuses on Icarus and HTNs which exemplify the two extremes of applying decomposition rules reactively and deliberatively.
- Chapter 3 describes the rule framework used by HOPPER and TADPOLE, and explains how decomposing tasks into sub-goals results in more re-usable decomposition rules.

- Chapter 4 describes HOPPER in detail. HOPPER applies decomposition rules to generate a partial goal-decomposition hierarchy. It modifies and adapts its plan to any unexpected events that arise, and it interleaves parallel sub-plans to make its overall plan more efficient.

- Chapter 5 describes TADPOLE in detail. TADPOLE parses the demonstrated lessons of a teacher, reconstructing a decomposition hierarchy that corresponds to the lesson. It uses the parsed lessons to refine its rules and to learn new ones.

- Chapter 6 demonstrates HOPPER and TADPOLE in a kitchen and a logistics domain. It gives examples of TADPOLE parsing complex lessons and learning novel rules. It also shows HOPPER applying learned rules to achieve tasks despite unexpected and disruptive events, and optimizing its plan by interleaving its sub-plans.

- Chapter 7 concludes the thesis and discusses avenues of future research.

Chapter 2

Related Work on Decomposition Systems

This chapter describes previous research on systems that use decomposition rules (rules that specify how to break a problem down into subproblems) to achieve tasks in the HPD, and it discusses their limitations and their relation to HOPPER and TADPOLE. The research on these systems is generally seen as (eventually) leading towards a Human-Level Intelligence (HLI) system, an artificial intelligent agent that can duplicate the competence and versatility that humans show in the same wide range of domains.

There have been two approaches to creating an HLI system (or at least one that is effective in the HPD): using humans as an explicit benchmark in an attempt to model and predict human behaviour directly, and using humans as an inspiration in an attempt to create a system that exhibits the same range of cognitive abilities (this is the approach taken in the development of HOPPER and TADPOLE).

The resulting systems fall into two further sub-types depending on how they interact with the environment: completely reactive systems that determine how to act without predicting future states at all, and completely deliberative systems that generate a complete plan of atomic ac-

tions before executing it. Both of these extremes are problematic in the HPD.

The performance of hierarchical decomposing systems critically depends on the quality and quantity of the domain skill knowledge encoded in their decomposition rules. The research done on such systems has focused on execution: how best to use already present (hand-crafted) rules to achieve a variety of tasks. Research on learning the decomposition rules has been limited.

Organization of the chapter

- Section 2.1 describes the ACT-R cognitive architecture. ACT-R uses production rules to model human cognition, predicting the time, accuracy, errors, and even the activity of different parts of the brain during task execution.
- Section 2.2 describes HTNs, a family of planning algorithms that use decomposition rules as domain knowledge to constrain the search for a classical plan (a sequence of atomic actions).
- Section 2.3 describes Icarus, a cognitive architecture that uses a reactive decomposition strategy to achieve tasks in dynamic domains using decomposition rules.
- Section 2.4 concludes the chapter by describing a number of miscellaneous systems related to HOPPER and TADPOLE.

2.1 ACT-R

ACT-R (Adaptive control of thought-rational) is a cognitive architecture used to model human cognition [2][5]. ACT-R makes neurally plausible assumptions about how it (or something similar) could be implemented in the human brain, and psychologically plausible assumptions about how the model could be acquired.

The core of the ACT-R architecture are hierarchical production rules similar to those used by HOPPER and TADPOLE. I developed the rules used by HOPPER and TADPOLE independently of the rules used by ACT-R, so it is noteworthy that the rules used by HOPPER and TADPOLE to learn how to solve tasks in the human domain bear such a striking resemblance to the rules used by ACT-R to model human cognition.

2.1.1 ACT-R divides cognition into separate modules

The ACT-R architecture is divided into a number of independent modules responsible for processing different kinds of information. These are the perceptual module, the motor module, declarative memory module, and goal module. Based on fMRI experiments, some of these modules have been mapped to particular regions of the human brain.

The perceptual module is responsible for parsing raw input data into a structured representation of the world, and the motor module is responsible for converting atomic actions to raw signals to the agent's mechanical effectors. The declarative memory module is responsible for storing and retrieving the agent's factual knowledge about the world, such as "2 + 3 = 5" and "France is a country". The goal module is responsible for keeping track of the agent's goals and sub-goals throughout the execution of the agent's tasks.

These modules do most of the processing of their respective information independently of each other, and deposit the result of their computation into their buffers. For example, the declarative memory module could

store the currently most pertinent fact in its buffer, and the goal module could store the most relevant and important goal that the agent is currently working on achieving, and so on.

The entire system is integrated by a central repository of production rules. Different production rules activate depending on the state of the different buffers, and the active rules change the state of the modules depending on what their effect is, which results in the buffers of the modified modules being updated and activating new rules.

2.1.2 Production rules form the core of the ACT-R architecture

An ACT-R production rule has a precondition that specifies what the contents of the different buffers has to be for the rule to activate, and a production that specifies how the different buffers will be modified when the rule is activated. Although these rules are more general than those used by HOPPER and TADPOLE, the rules that deal with how to achieve goals have almost the same form: they decompose goals into sub-goals and atomic actions (atomic actions modify the buffer of the motor module and result in the agent generating effects in the world). In comparison, the rules used by HOPPER and TADPOLE decompose tasks into sub-goals and atomic actions. Chapter 3 explains why rules that decompose tasks are more appropriate to the HPD than rules that decompose goals.

2.1.3 ACT-R predicts the results of human cognition

The ACT-R architecture has been used in a wide range of experiments to model human cognition in solving different tasks. Both people and ACT-R were given the same tasks to complete, and ACT-R was able to successfully predict the time it took humans to achieve the tasks, the speed-up that occurred after people had practice solving the tasks and became more proficient at them, the errors committed, and the accuracy with which the

tasks were solved. These tasks included: solving the Tower of Hanoi problem [4], managing aircraft during wartime as an Anti-Air Warfare Coordinator [5], solving algebra equations [3], reasoning about the spatial orientation of blocks [9], reasoning about diagrams and maps [13], playing the Rock-Paper-Scissors game [50], and reasoning about taxiing planes on a runway [10].

Recently, ACT-R has been used in conjunction with fMRI scans of brain activity to identify which parts of the brain correspond to ACT-R modules. ACT-R was then able to predict which parts of the brain would activate and for how long during the completion of the various tasks.

Although this research is promising it has only been applied to extremely limited and simplistic problem domains where the task to be achieved is clearly defined and only the relevant information is presented to the subject. It will be important to model human cognition with the ACT-R architecture in more complex and realistic domains to ensure that it can explain how humans can solve real-world tasks.

2.1.4 ACT-R achieves tasks reactively

ACT-R applies its production rules reactively — it chooses an applicable rule and applies the production immediately with no look-ahead and without predicting future states. This can be problematic because often the way to achieve a goal is to achieve a number of sub-goals in a particular order. If a production rule for achieving such a goal posts its sub-goals to the goal buffer and more than one of the sub-goals is achievable in the current state, then ACT-R could end up achieving the sub-goals out of order by simply selecting the first applicable production rule. For example, when mixing with a blender one should first open the blender, pour in the ingredients, close the blender, and only then turn it on. The sub-goal for turning on the blender is achievable throughout this process but it is important that it is not achieved until all the rest of the sub-goals have been.

ACT-R's production rules need to be crafted carefully with the appropriate preconditions to ensure that sub-goals are achieved in the correct order.

A more significant problem with reactively applying production rules is the inability to interleave the execution of tasks. Section 2.3.3 describes Icarus' reactive rule application strategy, and it discusses this limitation in more detail.

2.1.5 ACT-R estimates the utility of production rules

If multiple production rules are applicable at the same time, then ACT-R selects the one with the highest utility to execute. The utility of a production rule is its probability of successfully achieving its goal multiplied by the importance of the goal minus the cost of the rule. Although this is a useful way of selecting which rule to execute, it is not clear how to accurately estimate the probability that a rule will successfully achieve its goal nor the cost of the plan to achieve it.

ACT-R estimates the probability of success and the cost of a rule by simply tallying up the number of times the rule successfully achieved its goal and the number of time steps it took in the past. However, this mechanism does not take into account that the likelihood of the rule succeeding and the cost of its execution heavily depends on the nature of the current state. For example, making dinner may usually be very easy (high success rate and low cost), but if critical ingredients are missing, forcing the agent to go to the store to buy them, then both the likelihood of success and the cost of the rule will change drastically.

The preconditions of the rules used by HOPPER are learned by TADPOLE (this is described in detail in Chapter 5). The preconditions are generalizations of the states in which their corresponding rules were successfully executed. The descriptions of the states in the domains that TADPOLE learns rules are very rich (Chapter 6 describes the domains that HOPPER and TADPOLE have been evaluated in), and the preconditions

themselves have rich descriptions. TADPOLE learns the preconditions for different rules from different sets of demonstrated states, and the odds that two preconditions will have rich descriptions that will match equally well to the current state is remote. This means that HOPPER never has to choose between equally applicable rules; one will always match better than the others. However, estimating the utility of closely matching rules would be an interesting extension of HOPPER that would be of particular use with well-learned rules whose preconditions have been simplified and trimmed of irrelevant details.

2.1.6 ACT-R learns how to speed up task achievement

As well as learning the utilities of different production rules through experience and reinforcement learning [19], ACT-R also compiles multiple rules that it used to achieve a task into a single new production rule. This is analogous to Soar's chunking mechanism (section 2.4 covers this in more detail), and it allows ACT-R to directly achieve the task with less cognitive effort in the future [34]. However, this is a very limited method for learning new production rules because it depends on the agent already being able to solve the task in question. ACT-R has no way of learning completely novel skills to solve completely novel tasks.

2.2 Hierarchical Task Networks

Hierarchical Task Networks (HTNs) [38] are a family of practical planning systems that use decomposition rules to generate classical plans. HTNs were first developed more than twenty years ago [51] with UMCP [14] being the first provably sound and complete HTN algorithm. Since then, HTNs have been successfully applied to a wide range of practical domains, and a number of variants of the basic HTN algorithm have been developed to deal with the requirements of the various domains.

2.2.1 HTNs store domain knowledge as decomposition rules

The domain knowledge of an HTN planner consists of a number of decomposition rules, where each rule specifies how a procedure can be achieved by achieving a sequence of sub-procedures. Usually a decomposition also has a precondition specifying in what states it is applicable. For example, a rule for delivering a package in a logistics domain could look like:

```
deliver-package(P, L1)
    PRE: isPackage(P), atLocation(P, L0),
    isLocation(L1), isTruck(T) →
        move-truck(T,L0),
        load-package(P,T),
        move-truck(T,L1),
        unload-package(P,T)
```

As well as decomposing into a sequence of sub-procedures, a procedure can also be decomposed into a sequence of atomic actions (or into a sequence of both sub-procedures and atomic actions).

2.2.2 HTNs recursively decompose procedures into sub-procedures down to atomic actions

Rather than searching through atomic actions that achieve states like a classical planner, the HTN planning algorithm searches for decompositions that achieve the main task and are applicable in the current state. It then replaces the main task with the specified sequence of sub-procedures. The algorithm continues to recursively find applicable decompositions for achieving each sub-procedure, replacing it with the specified sequence of sub-sub-procedures and so on until only atomic procedures are left that are achievable with a single action. The algorithm stops decomposing

when there are no more non-atomic procedures to achieve and the sequence of atomic goals can be converted directly into a sequence of atomic actions. Depending on the specific kind of HTN, the algorithm may also process the final sequence of atomic actions to resolve any conflicts, bind variables, and so on.

A procedure can often be decomposed in several different ways by several different decomposition rules, and it may be able to bind its variables in different ways for each rule. Whether or not a decomposition rule is applicable depends on whether or not its precondition is satisfied (or satisfiable). If there are no applicable rules, then the algorithm backtracks to an earlier choice point. If there are multiple applicable rules, then the algorithm selects one and continues. If every choice ends up failing, then the algorithm backtracks and selects another applicable decomposition rule.

2.2.3 A deliberative decomposition strategy requires predictable domains

It is important to note that a decomposition rule used by HTNs does not specify the goal that will be satisfied nor the effects that will occur when the rule is executed. The only thing that distinguishes one procedure or sub-procedure from another is its label. An important consequence of such decomposition rules is that the agent has to generate its entire plan before executing it. This is because the agent does not even know what it is trying to accomplish until the plan is complete. Because of this, HTNs have to use a deliberative decomposition strategy and generate the entire plan of atomic actions before executing them in the same way that a classical planner does. This results in the same difficulties that classical planners face in unpredictable domains described in Section 1.4.2.

2.2.4 Ordered HTNs allow for more powerful preconditions

The SHOP algorithm [39] found that constraining the HTN search to decompose sub-procedures in the same order that they will later be executed allows the use of more expressive preconditions — if procedures are decomposed in the order they are executed, then all of their variables are already bound, and all of the variables of the sub-procedures are also bound. In particular, this means that the atomic actions at the bottom of the decomposition hierarchy also have all of their variables bound and so their effects can be directly determined (the effects of atomic actions in HTNs are specified in the same way as in classical planners), and the intermediate state corresponding to that stage of the planning process can be determined.

If the intermediate states are completely known, then sophisticated preconditions and reasoning can be applied to those states. For example, the procedure to get to some destination (*e.g.* to be at the airport as part of the plan of going on a trip) can be achieved with decompositions for walking there, calling a taxi, catching a bus and so on. Which of these decompositions is appropriate or even possible depends crucially on where the agent will be at that stage of the plan. For example, if the agent will be within walking distance of the airport, then it should walk; if it will have enough money for a taxi, then it should call a taxi; otherwise if it will have enough money for a bus, then it should take a bus to the airport:

```
getToDestination(X)
    PRE: at(Y), walkingDistance(Y,X) →
        Action: Walk(X)
    PRE: at(Y), enoughMoneyForTaxi(Y,X) →
        getToDestination(taxiStand), hailTaxi(T), getInVehicle(T),
        giveDirections, exitVehicle(T)
    PRE: at(Y), enoughMoneyForBus(Y,X) →
        getToDestination(busStop), getInVehicle(B), exitVehicle(B)
```

To determine which precondition and hence which decomposition rule is applicable depends crucially on where the agent will be (location Y) and how much money it will have at the time that it will execute this part of the plan. This depends on both the initial state and on all the actions the agent will have executed between then and the time the decomposition is executed. The only way to determine all of the previous actions is to fully decompose and bind all of the previous sub-procedures.

Though decomposing sub-procedures in order allows for powerful, expressive preconditions, the M-SHOP algorithm [40] found that the algorithm was limited by not being able to interleave sub-procedures properly. For example, when achieving the task of getting two packages to the same location in a logistics domain, the planner would return a sub-optimal plan of picking up each package and dropping it off separately rather than combining both activities into one shorter, more efficient plan (pick up the first package, pick up the second package, go to the destination, and unload both packages at the same time).

M-SHOP and the later SHOP2 [41] are extensions of the in-order HTN algorithm that resolve this problem. The sub-procedures of a decomposition can be arranged in a partial-order, and the decompositions of co-ordered sub-procedures could be interleaved to produce a more efficient plan. To ensure that the interleaved sub-plans do not interfere with each other, the preconditions of the sub-procedures of a decomposition can be labeled as "protected" and "immediate". Protected conditions cannot be undone by subsequent sub-procedures until an operator explicitly removes the protection (*e.g.* when picking up a package, the location of the truck could be protected until the package has been loaded into the truck). Immediate sub-procedures are executed immediately without doing any interleaving (*e.g.* when picking up a package, the load sub-procedure could be made immediate so that the package is loaded into the truck as soon as the truck arrives).

Although this approach does allow the planner to interleave the execution of sub-procedures, it depends on each decomposition being hand-coded to specify how its sub-procedures can be interleaved.

2.2.5 HTNs have been successfully applied to a wide range of practical domains

Because decomposition rules offer a powerful and intuitive framework for encoding domain knowledge, Hierarchical Task Networks have seen extensive real-world use. HTN systems have performed well in international planning competitions [38] as well as being applied to a wide range of practical domains. This includes planning the electronic and mechanical design of microwave modules [21], planning the declarer play of contract bridge [48], planning noncombatant evacuation operations (although the goal decomposition had to be human-supervised) [36], the automated composition of web services [52], evaluating terrorist threats, fighting forest fires, and controlling multiple UAVs (unmanned aerial vehicles) [37].

2.2.6 HTNs have been extended to non-deterministic domains

Though most HTN applications have been in deterministic, fully observable, and fully-controlled domains, there have been a number of extensions where some of these constraints have been relaxed. The YoYo algorithm [26][25] extended the classical planning model so that the result of atomic actions could not be deterministically predicted (whether due to random environmental influences or the actions of other agents). However, while the result of an atomic action was extended from a single possibility to a set, the set of possibilities was constrained to be a finite and *known* set. HTNs have also been extended to partially observable domains [27], but, again, the unknown variables in those states and their possible

values had to be known in advance. Although these extensions made to the HTN algorithm can handle this limited form of non-determinism, it does not scale to a truly unpredictable domain where the set of possible states following the execution of an action is unlimited and unknown.

The deliberative HTN decomposition strategy does not address the wider issue that states predicted in a plan become more and more uncertain the further they are in the future in any domain that is not completely deterministic. This means that atomic actions planned for states in the distant future will almost certainly be invalidated before they are executed.

2.2.7 HOTRiDE repairs failed plans

HOTRiDE is another HTN planning system that addresses some of the limitations of classical planners in non-deterministic domains. It uses SHOP to generate plans, but is then able to respond to unexpected disruptive events during the execution of a plan [7]. If an atomic action fails (because of an unexpected event), then HOTRiDE is able to repair the existing plan without the need of replanning completely from scratch.

When generating the initial plan, HOTRiDE keeps track of not only the sequence of atomic actions to be executed, but also the task decomposition hierarchy used to derive the plan. It also keeps track of the causal links between nodes in the graph. An atomic node in the decomposition hierarchy may be causally linked with a node in future parts of the hierarchy if the effects of the operator are not disjoint from the precondition of the task of the other node. These causal links are useful for determining which parts of the plan are affected by the failure of an atomic operator, allowing HOTRiDE to repair only the affected parts of the plan and preserving the rest.

When an atomic action fails, HOTRiDE searches for the actions minimal failed parent task (the task is marked as failed — its precondition is not satisfied — but its parent task is not marked as failed) and uses SHOP

to replan to achieve this failed task. HOTRiDE then considers the causal links of the plan. If any no longer hold, then HOTRiDE uses SHOP to replan the future nodes of the decomposition hierarchy until all the causal links of the plan are valid. If this is not possible, then HOTRiDE recursively tries to replan the next highest node above the failed task. After the plan is repaired, HOTRiDE the continues executing it.

HOTRiDE's most significant limitation is caused by HOTRiDE's insistence on generating the entire plan before executing it. This forces HOTRiDE to consider all of the causal connections between earlier parts of the plan with later parts. This becomes particularly problematic when for extensive plans such as planning a trip overseas which can involve many thousands of atomic actions. The atomic actions generated (and re-generated for plan failures) in the future parts of the plan constitute wasted effort because most of them are unlikely to hold when it actually comes time to execute them. For example, when planning an overseas trip, its a waste of time planning to get on a bus after getting off a plane at your overseas destination, because you may end up being forced to take a taxi instead.

Because HTNs do not keep track of goals at a lower level than the top goal and they do not keep track of tasks at a higher level than the atomic operators, HOTRiDE has no way of knowing what it is trying to achieve in any part of the plan nor what the likely effects will be of achieving it. This forces it to deal with unexpected events at the atomic level of the plan. However, it may be too late to effectively deal with an unexpected disruption when a particular atomic action fails. For example, if your plane has been canceled, then you should immediately alter your travel plans, rather than waiting for the non-existent plane and then having the atomic action of boarding the plane fail.

Finally, HOTRiDE only addresses the issue of unexpected atomic disruptions to a plan. It does not handle the more subtle problem of detecting and exploiting unexpected opportunities — opportunities to significantly reduce the length of a plan.

HOPPER addresses these shortcomings by not decomposing the entire plan before beginning to execute it (chapter 4). This strategy of decomposition by least-commitment is more appropriate for long-term plans whose future parts are difficult to determine precisely in advance. HOPPER detects unexpected events by failed predictions rather than failed operators. This combined with keeping track of the sub-goals it is trying to achieve and the effects of sub-tasks it is executing, allows HOPPER to redecompose the affected parts of the plan and handle unexpected disruptions before they become operator failures. This also allows it to exploit unexpected opportunities.

2.2.8 There has been relatively little research into learning HTN decomposition rules

In the majority of HTN systems, the decomposition rules that were used were handcrafted to suit the problem domain. Though the decomposition rules proved to be a powerful and intuitive method for encoding useful and versatile domain knowledge, there has been a lot less research focused on the problem of learning the decomposition rules autonomously. The research on learning HTN decomposition rules has focused only on learning the preconditions of HTN decomposition rules, while still relying on their structure to be hand-crafted.

CaMeL [24] and CaMeL++ [23] are HTN systems that have been developed to learn parts of decomposition rules. However, both of these systems are quite limited in scope. They require substantial input and prior knowledge and only learn and refine the preconditions of decomposition rules and not the rules themselves. Specifically the input includes knowledge of all tasks and decomposition rules and complete demonstration traces of various problems being solved (by an expert). Each decomposition trace specifies how and in what state each procedure and subprocedure was decomposed. This is an unreasonable amount of informa-

tion to expect from a teacher. The algorithms were also tested in domains with states that had a limited number of objects. It is unclear whether the candidate elimination algorithm they use would scale to rich and complex domains with a large number of objects.

2.3 Icarus

Icarus [31] is a cognitive architecture that uses decomposition rules to achieve tasks in dynamic domains. It has successfully been applied to a range of domains including blocks world, in-city driving, multi-column subtraction, and the towers of Hanoi [32].

2.3.1 Icarus uses a concept hierarchy

Icarus organizes concepts into a hierarchy with primitive concepts defined in terms of perceptual information and more complex concepts defined in terms of simpler concepts. Every time that Icarus receives low level perceptual information it constructs a representation of the current state of the world from this input. The current state representation includes higher level concepts that it infers from the primitive perceptual input. Icarus then uses this more abstract state representation to check whether or not its goals have been satisfied, to verify that preconditions hold for its rules, and so on. Icarus' concept hierarchy does some of the work that HOPPER and TADPOLE assume is done by the vision system.

2.3.2 Icarus achieves goals with hierarchical skills

As well as having a concept hierarchy, Icarus' domain knowledge includes a hierarchy of decomposition rules it calls "skills". A skill is a method for achieving a goal given a precondition, and it can be either primitive or compound. A primitive skill specifies one or more actions that, when

executed in order, will achieve the specified goal (given that the precondition holds before execution). A compound skill specifies one or more sub-goals or primitive skills that, when satisfied, will achieve its main goal (given that the precondition holds before execution). A compound skill may be decomposed into an ordered or unordered set of sub-goals, which Icarus treats differently during execution (see below). Skills may also have an expected value function that represents the utility expected if the skill is executed, and Icarus can learn this expected utility with reinforcement learning [46].

An example of a compound skill for achieving the goal of having a block be clear (not have any block stacked on top of it) in the Blocks World domain could be:

```
(clear (A)
   PRE: isBlock(A), isBlock(B), on(B,A), hand-empty →
      achieve unstackable(B),
      unstack(B)
```

The skill is applicable if there's a block (B) on top of the block that is to be clear (block A). To execute the skill, the agent should first achieve the sub-goal of having block B be unstackable. This means that block B is clear and the agent's hand is empty. Then the agent should execute the primitive skill of unstacking block B from A. If there are multiple blocks on top of block B, then achieving the sub-goal of having block B be unstackable will involve a recursive use of the clear skill to clear block B (by unstacking the block above it), and so on for every block above B. In this way, Icarus can use a single rule to unstack an indefinite number of blocks above block A.

2.3.3 Icarus uses its skill set reactively

As well as having long term domain knowledge that includes all of its decomposition rules, Icarus also maintains a short-term memory for more temporary knowledge. Included in this short-term memory is a set of skills currently under consideration for the task(s) at hand. If this set is empty, then Icarus goes through its entire set of skill knowledge, finds a skill that solves a goal it has and whose precondition is satisfied, and then adds this skill to the set of skills to be considered.

Icarus operates in cycles. It receives perceptual information, processes it and infers which higher level concepts hold, selects an appropriate skill to execute to achieve its goal(s), and returns the appropriate action. In each cycle Icarus finds an applicable skill (one whose precondition is satisfied and whose effect is not) from the set of skills to be considered and if it is primitive then it executes the corresponding primitive action(s), otherwise it decomposes the skill into sub-skills, adds them to the short-term memory of skills to be considered, and then considers each of these sub-skills. If the sub-skills are unordered, then out of the applicable ones Icarus selects the one whose expected value function gives the highest result; if the sub-skills are ordered, then Icarus selects the last applicable sub-skill (the rationale being that later sub-skills are closer to the parent skill's effects and should be preferred to earlier sub-skills). After selecting a sub-skill, Icarus decomposes it and recursively continues selecting sub-sub-skills until reaching a primitive skill and returning the appropriate action(s).

2.3.4 A reactive decomposition strategy has difficulty focusing on a single goal

A reactive decomposition strategy has the problem that if multiple goals (or sub-goals) are equally achievable in the current state, then there is no guarantee that the same goal (or sub-goal) will be focused on from one cycle to the next. An extreme example of this problem is when there are two

goals (or sub-goals) the agent is trying to achieve and it tries to achieve one goal for several cycles (by decomposing the appropriate skill and returning the appropriate action), and then it reactively switches to achieving the other goal for the next couple of cycles (possibly undoing any progress it had made with the first goal). It could then continue flipping between the two goals every couple of cycles and, like Buridan's donkey[1], not make any progress with either goal.

To get around this problem, Icarus uses a persistence parameter that can be used to bias its decomposition strategy [12]. When it is zero the decomposition strategy is completely reactive. The higher the persistence factor the more likely Icarus is to select the same skills and sub-skills it did in the previous cycle.

2.3.5 A reactive decomposition strategy depends on accurate rule preconditions

A reactive decomposition strategy decomposes sub-skills whose preconditions are satisfied in the current state. For many tasks, it is important that their sub-goals are satisfied in a particular order. To ensure this, the preconditions of the sub-skills have to specify when each sub-skill is inapplicable.

Requiring accurate preconditions for ordered decompositions makes the decomposition rules very difficult to learn. The agent has to know the exact rationale why each sub-goal should be achieved and in what order. If the agent learns incorrect preconditions for the rules, then it will execute the sub-goals in the wrong order and likely not achieve the main goal at all.

This problem is exacerbated if the task involves hidden properties — properties of the world that are not directly observable, that are important

[1]The hypothetical donkey that starved to death by not being able to decide between two equally sized stacks of hay.

to the successful completion of the task, and that are altered throughout the execution of the task's sub-skills. An example of such a property is whether or not a door is locked. In domains with hidden properties, the agent has to have comprehensive knowledge of the current state of the world and also of the physics of the world — what the important properties of the state are and what effects changing them will have.

In order for a reactive decomposition strategy to execute a task's sub-skills in the appropriate order, the agent has to know the reason for the ordering, why some sub-skills should be executed only after earlier sub-goals have been achieved. These reasons have to be encoded in the appropriate preconditions.

It is also clear that humans do not behave in this way. Humans will often learn sub-optimal, ritualistic behaviour, achieving a task in a particular, specified order for no rationale other than it has always been done that way.

2.3.6 A reactive decomposition strategy has difficulty optimizing plans through interleaving

An important way that a hierarchical decomposing system can optimize its plans is by interleaving the execution of one skill with the execution of another. If the agent can achieve the sub-goals of two (or more) different skills at the same time in parallel, then the agent will save time and effort, sometimes a great deal of time and effort depending on how high the shared sub-goal is in the decomposition hierarchy. For example, in a logistics domain, if the agent is tasked with delivering two packages from the same location to the same destination, then it should interleave the execution of the skills used to achieve these two goals and load and unload both packages into the same truck at the same time, rather than making a separate trip for each package.

However, interleaving the execution of separate skills is not trivial. The

sub-skills for achieving the sub-goals of different skills will often interfere with each other and undo their effects. Only when they are executed in a particular order will the interleaving successfully achieve all of the requisite goals. A persistence factor that causes the agent to switch with some random probability from the execution of one skill to another is too crude a mechanism to ensure that the appropriate sub-skills are executed in the correct order.

2.3.7 Icarus can achieve goals without applicable skills

In cases where there is no applicable skill to achieve a goal, Icarus uses means end analysis to achieve it [33]. During each cycle, when Icarus decomposes skills into sub-skills, it treats the heads of the skills as goals, sub-goals, sub-sub-goals, and so on, finally bottoming out at the atomic action that Icarus returns. This decomposition path from the top goal to atomic actions forms a goal stack. If a goal has no applicable skill then Icarus attempts to satisfy it in several different ways:

- Icarus selects any skill that achieves the goal and that does not clobber any of the goals achieved earlier in the goal stack.
- Icarus uses a default decomposition rule that decomposes the goal into a number of sub-goals for achieving the sub-concepts (see section 2.3.1) of the goal.
- Icarus looks for a primitive skill that achieves the goal but whose precondition (a single start condition) is unsatisfied. It then posts the precondition as a new goal to be achieved.

As soon as this process finds an appropriate action, Icarus immediately executes it. This means that Icarus' means end analysis does not backtrack which can be problematic because it can lead the agent into difficulties, especially if the agent can do un-undoable actions. However, if the agent

successfully achieves the goal, then Icarus can learn new decomposition rules from its plan.

2.3.8 Icarus learns new decomposition rules from successful plans

If Icarus manages to successfully achieve a goal using means end analysis, then it records the plan in a new skill decomposition rule:

- If Icarus achieved the goal by first achieving the precondition of a primitive skill, then Icarus learns a new skill whose head is the original goal and whose ordered sub-skills are the precondition and the primitive skill used to achieve the original goal. The precondition of this skill is the same as the precondition of the skill used to achieve the precondition.
- If Icarus decomposed the goal into a sequence of sub-goals for achieving the sub-concepts of the goal, then Icarus learns a new skill whose head is the original goal and whose sub-skills are an ordered list of the skills used to achieve the sub-concepts of the goal (the ordered sub-skills have a number of cumbersome guard conditions associated with them to try to make sure that they are achieved in the right order and not in reverse order as described in Section 2.3.3). The precondition of this skill is simply the sub-concepts of the goal that were true when Icarus first tried to achieve it.

Clearly, this skill learning is very limited. The first method for extending Icarus' skill set does make the skill for achieving the goal more general and makes it applicable in a new situation, but it depends on the precondition having only a single condition that is unsatisfied and the skill that achieves it has to be primitive.

The second method critically depends on the goal being achievable simply by achieving all of the sub-concepts of the goal. However, most

goals cannot be achieved in this way. For example, when making a cup of tea you have to boil water in a kettle, but the sub-goal of having a kettle of boiling water is in no way a sub-concept of the concept of a cup of tea.

Because Icarus' goals and skills and HTNs' tasks and methods are very similar, this method for learning skill decompositions has also been applied to learning HTN procedures [42].

2.3.9 Icarus can learn skills from flat reactive rules

There has also been some research into learning Icarus decomposition rules from the observation of the behaviour of other agents [22]. The input consists of sequences of states and the actions that the other agent took in those states. The algorithm ignores the order of the demonstrated states and generates a set of state-action pairs. From this set of state-action pairs, the CN2 algorithm is used to induce a set of flat, reactive rules.

Given a set of reactive rules of the form $(s_1, s_2, ...s_n) \rightarrow a_1$ where s_x represents a state condition and a_y represents an atomic action, the algorithm induces several decision trees by promoting shared preconditions. For example, given the rules $(s_1, s_2) \rightarrow a_1$ and $(s_2, s_3) \rightarrow a_2$, this method would promote the shared precondition s_2 into a parent node. Icarus decomposition rules were then handcrafted from these decision trees (though this could have been automated).

There are a number of limitations of this method of learning decomposition rules. The most significant one is that it ignores the order that the actions were executed in. However, in the HPD, the order that actions are executed in is almost always critically important. Furthermore, if one already has a complete set of reactive rules that effectively achieve the agent's tasks, then there is no point in learning equivalent hierarchical rules. If the rules are incomplete or ineffective, then the learned hierarchical rules will be as well.

2.4 Miscellaneous systems

The systems discussed in this section are less relevant but they have some notable features either in the planning or learning aspects of their algorithms that are related to HOPPER and TADPOLE.

2.4.1 Teleo-reactive agents use production rules to execute continuous actions

Teleo-reactive agents [43] are noteworthy because they are based on concepts from control theory and they use decomposition rules to generate and maintain the execution of continuous actions.

Later teleo-reactive agents were extended to be able to form teleo-reactive plans with STRIPS-like action-effect rules to achieve tasks for which they did not have appropriate rules [8]. These plans could be hierarchical by making use of macro-operators for achieving sub-goals. However, a human designer had to identify which sub-goals the planner should try to solve and store as macro-operators for use in the higher-level plan.

Another extension was the ability to learn the STRIPS-like action-effect rules from observations of the effects various atomic actions had on the environment. Using positive examples of a continuous action resulting in some effect and negative examples of the same action not producing the effect, the agent learned a disjunctive normal form of the predicates that form the precondition of the action-effect rule. Although the agent was able to learn action-effect rules in this way in very simple and limited domains, it is unlikely that this method would scale to a rich domain with many objects, properties, and relationships. This method is also not appropriate for learning higher-level decomposition rules for more complex tasks.

2.4.2 Soar is a cognitive architecture whose inherent learning mechanism is limited

Soar is a cognitive architecture that has been in development for a long time [29] and which has seen extensive use in modeling human cognition. Soar achieves tasks by making use of its central knowledge base which consists of production rules. When applicable production rules fire, they propose operators that either modify Soar's internal state or cause the system to execute an external action. If there are multiple possible proposed operators, an impasse occurs. This causes Soar to post a new sub-goal of resolving the impasse. It achieves this sub-goal in the normal way by finding and firing applicable production rules.

After resolving an impasse, Soar learns a new production rule whose precondition consists of what was true when the impasse occurred and whose effect is the resolution of the impasse. This process of storing the solution to an impasse in a production rule is known as "chunking". The next time Soar finds itself in a similar situation, no impasse will occur because the new rule will automatically fire and resolve it immediately. Although chunking will speed up the rate at which Soar achieves goals over time, it depends on Soar being able to achieve the impasse sub-goal in the first place. The chunking learning mechanism does not allow Soar to learn how to solve completely novel tasks.

Since its inception, Soar has undergone a number of extensions [28] including a mechanism for reinforcement learning. Although this mechanism allows Soar to tune the numerical preferences of operator selection rules from experience and so to learn to select the appropriate operator when multiple are applicable, it does not allow it to learn how to achieve novel tasks.

Properly hand-crafted production rules can allow Soar to solve a wide variety of tasks in a range of different domains. However, the issue of learning how to achieve novel tasks has not been unaddressed.

2.4.3 NOAH is a decomposition system that uses critics to resolve sub-task interactions

NOAH [45] is a very early decomposition system that, similarly to HTNs, decomposes completely and generates the whole plan before it begins execution. It repeatedly applies its hand-crafted decomposition rules to gradually deepen its hierarchy and produce the next (lower) level of abstraction.

At each level of abstraction, NOAH makes use of critics to identify and resolve conflicts between the sub-tasks in the plan. An example of a conflict is if the effect of one sub-task negates the precondition of another, and NOAH would resolve this conflict by enforcing an ordering constraint on these sub-tasks to ensure that the precondition-negating subtask was executed later. As well as general critics, NOAH also makes use of hand-crafted critics to resolve additional, domain-specific problems with its plans as they become apparent. For example, plans that instruct a human apprentice how to perform hardware maintenance have to take into account the fact that a human only has two hands and can only be in one place at a time.

A noteworthy aspect of NOAH is that it was designed to interact with people as a Computer-Based Consultant and provide assistance to humans doing hardware maintenance. It was useful not only in generating a plan for complex cases, but it could also present parts of the plan at different levels of abstraction depending on how confident the user was in achieving a given sub-task. Even more interestingly, if the user was curious, NOAH could provide the reason and motivation for achieving a particular sub-task. This highlights the fact that hierarchical decomposition rules are very comprehensible to people.

2.4.4 PRODIGY experiments to determine why its operators failed

PRODIGY is planning system that uses declarative operators to encodes its domain knowledge. It uses an operator refinement method to acquire new preconditions and postconditions for an operator when it observes unexpected effects after applying the operator [11].

If all the preconditions of an operator were satisfied but a postcondition of the operator does not apply after the operator was executed, then one of the properties of the current state that differs from the last time the operator was successfully executed must be an additional precondition of the operator. Similarly, if a precondition of an operator is unexpectedly undone since the last time the consistency of the plan was checked, then one of the previous operators must have an additional postcondition negating the precondition.

The system hypothesizes which state property or operator is responsible for the failure and devises experiments to verify its hypothesis. If its hypothesis is correct, then the system refines its operators and re-plans as needed.

This approach is useful for identifying hidden relationships between atomic operators, but it fundamentally depends on a deterministic model of the world. It cannot distinguish between unexpected events that occur because the domain knowledge was wrong or incomplete and those that occurred because of an unpredictable, random event. It would be interesting to extend this approach to verifying and refining the preconditions and effects of high-level decomposition rules. However, identifying which sub-task(s) or sub-sub-task(s) in a decomposition's sub-hierarchy have incorrect pre- or post-conditions is a much more challenging problem than refining completely atomic operators.

2.4.5 X-Learn extends its knowledge by solving increasingly difficult problems

X-Learn encodes its domain knowledge in the form of d-rules — rules that decompose goals into sub-goals [44]. This is similar to HOPPER and TADPOLE whose rules decompose tasks into sub-goals (this rule representation is described in chapter 3). X-Learn extends its rule set by solving increasingly difficult problems presented by the teacher.

The teacher begins with simple problems that can be solved by a short sequence of atomic actions. X-Learn searches for a solution to these problems by an iterative-deepening, depth-first search. After finding a sequence of operators that solves the problem, X-Learn learns a new d-rule whose goal is the problem to be solved, whose sub-goals are the operators it applied, and whose precondition is the initial state the problem was presented in. Because the newly learned d-rule has a precondition and it can be executed (by achieving its sub-goals) to produce an effect, X-Learn can make use of it in its forward-chaining, iterative-deepening, depth-first search to solve other, more complex problems. After solving this more complex problem, X-Learn again learns a new d-rule whose sub-goals are the operators it used. These operators may include previously learned d-rules. In this way, X-Learn gradually learns increasingly more abstract d-rules by using simpler d-rules to solve more complex problems.

This approach to extending X-Learns domain knowledge balances the burden of learning between the learner and the teacher. Requiring complete solutions to problems including a full plan trace places too high a burden on the teacher. On the other hand, giving no feedback to the learner and expecting it to learn everything by itself makes the learning problem too hard and unreasonable (even humans do much of their learning from teachers).

The fundamental limitation of X-Learns learning algorithm is that it can only learn search-control heuristics. Because the teacher only presents

problems to solve, the only information that can be gleaned from this is how operators can be grouped together to achieve sub-goals of higher-level goals. This is very useful for improving the efficiency of the planner, but it is insufficient for solving previously unsolvable problems. Every problem that X-Learn learns to solve can be solved (at least in principle) by searching for a sequence of atomic operators. This means that all of the domain knowledge for solving every kind of problem in every kind of state has to be encoded in the atomic operators that X-Learn takes for granted. If it lacks the domain knowledge to solve a given problem, there is no way for it to learn that knowledge; it can only learn to solve problems faster. This is akin to Soars chunking mechanism.

Because all of the side-effects of X-Learns d-rules are not specified, while planning, it has to decompose down to the atomic operators to determine what the effect of achieving the goal of a d-rule would be. This leads to the same brittle plans of atomic actions that plague classical planners in non-deterministic domains as described in chapter 1.

TADPOLE also learns from a structured lesson plan presented by a teacher; however, it receives complementary information: a sequence of states demonstrated by the teacher. TADPOLE has to infer what the problem being solved is. Because of this, it can learn useful rules for mimicking the teacher without always knowing the rationale for every atomic action (for example, it can learn that it should plug in a kettle to boil water without understanding how electricity works). TADPOLE also refines its operators and decomposition rules from experience, starting with incomplete and incorrect domain knowledge and gradually improving it over time (chapter 5 describes TADPOLE in more detail).

2.4.6 OBSERVER interleaves planning and learning of operators

OBSERVER learns the preconditions and effects of atomic operators from positive and negative examples. It can make use of these learned atomic operators when they are still imperfectly learned, being able to repair plans that fail due to incorrect operators [49].

OBSERVER uses a variant of the version space algorithm [35] to learn the preconditions of its operators. It maintains a most-specialized representation of the preconditions and a most-generalized representation that is consistent with the positive and negative examples it has seen for that operator. OBSERVER generalizes its most-specialized representation from positive examples of the operator being applied and it specializes its most-generalized representation from near-miss negative examples (the example differs by only one literal). The most-specialized and most-generalized representations eventually converge on the correct precondition.

However, it can take a lot of examples for the most-specialized and most-generalized representations to converge. Nevertheless, OBSERVER makes use of operators as soon as it has begun to learn about them. It uses the radical strategy of planning using the most-general precondition of the operator. Although this makes it likely that the plan will fail because of missing preconditions for various operators, it also gives OBSERVER opportunities to refine its domain knowledge.

When an operator fails to apply, OBSERVER tries to repair its plan. It repeatedly selects literals in the most-specialized precondition of the operator that are not in the most-generalized precondition and tries to achieve them to make the operator applicable again. If this fails to make the operator applicable, OBSERVER then hypothesizes negated preconditions that need to be achieved, and if this also fails, it tries a different operator for achieving the effects the original operator was trying to achieve.

Although this algorithm has been applied to atomic actions only, it

would be interesting to extend it to learning the preconditions of TAD-POLE's decomposition rules. Each rule would maintain both a most-specialized and most-generalized precondition, and when planning, HOPPER would use the most-generalized precondition to select applicable rules.

However applying this algorithm to learning preconditions for high-level decomposition rules introduces a number of challenges. When an atomic operator fails, it has little to no effect on the world. When a high-level decomposition rule is applied in an inappropriate state, then many actions may be executed by the agent before the failure becomes evident. Repairing a plan after many inappropriate actions have been executed may be difficult or even impossible. Another, problem is that the precondition of a rule may indicate not only whether a rule is applicable but also whether it is appropriate. It may be possible to execute a rule in a different state, but it may be so inefficient as to be impractical. Generalizing a precondition just because the rule is applicable in a different state without considering how efficient the decomposition is is too simplistic an approach when dealing with high-level decomposition rules.

2.4.7 LIVE learns operators from state changes

LIVE is a system that learns the preconditions and effects of prediction rules from examples of the operator being applied [47]. A prediction rule is effectively an operator or atomic action that LIVE uses to find plans to achieve goals. However, because the prediction rules are learned, they can be incomplete, leading to plan failure. In such cases, LIVE uses its exploration and experimentation modules to refine its operator knowledge.

If LIVE is unable to find a plan, then it activates its exploration module so that it can try various atomic actions to learn about their effects. It focuses on actions that it knows have some effects related to the goal in some way and on anomalous actions that apparently have no effect. If planning produces errors such as regression loop (the same sub-goal is re-

peatedly proposed) or regression deadlock (the proposed atomic actions conflict with each other resulting in no feasible plan) then this is an indication that some of the specific operators are incorrect. In such cases, LIVE uses its experimentation module to generate experiments for refining the faulty operators.

When learning a new operator, LIVE takes the difference between two states — one before the operator was applied and the one immediately following and notes what has changed. It gives more weight and attention to changes that it deems relevant — those that are closely related to the learner and to the goal objects. The additions and deletions become the operators effects, and their negation becomes the operators precondition. Clearly such a rule is overly general, but LIVE subsequently refines it after subsequent examples using complementary discrimination learning.

LIVE is also capable of learning new concepts using predefined relations between percepts. If an operator has different effects in two states that seem to be the same, then LIVE searches for relationships between predicates in the percept that can explain the difference. For example, it can learn the concept of bigger than using its $>$ relation and the predicates size(x, 3), size(y, 2).

LIVE has two main limitations. The first is that it assumes a purely deterministic world. The rule refining mechanism assumes that any unexpected events or noisy observations are caused by the operator being executed. The second limitation is that LIVE does not learn any search control knowledge or heuristics to make difficult planning problems tractable. This has kept LIVE restricted to simple, toy domains.

TADPOLE takes a similar approach to learning the tasks of its decomposition rules in that it uses the difference between earlier and later states to determine object relevance (chapter 5). However, it also maintains a count of how many times each property and relationship in its object graph occurred in the examples it observed. This makes the learned rules robust against non-determinism and noisy observations. TADPOLE

is also focused on learning hierarchical goal-decomposition rules as well as atomic operators. This greatly improves the efficiency of HOPPERs planning (chapter 4).

2.4.8 ARMS uses a weighted MAX-SAT solver to learn action models from observed plans

ARMS is another system that learns atomic action models from observed example plans [53]. It reinterprets the plans as a disjunctive set of conjunctive boolean constraints weighted by their importance. ARMS then uses a weighted MAX-SAT solver to find the maximum weighted number of clauses that can be satisfied. It derives the action models from the solution of this weighted propositional satisfiability problem.

The system first converts the instantiated actions observed in the plans into actions parameterized by variables of the same type. It then builds the action and plan constraints, converts them into clauses and associates them with weights depending on how important they are.

The generated constraints are: the intersection of the precondition and add lists of all actions must be empty, the deletions of all actions must be in their precondition, observed intermediate predicates in the plan must be in the add list of one of the preceding actions and it must not be in the deletions of the action immediately preceding it, every precondition of every action must be in the add list of a previous action and not in the delete list of any action in-between, and at least one predicate in the effect of every action must be in the precondition of a subsequent action. The final constraint defines possible preconditions for predicates that tend to occur frequently before a particular action. The weight of this constraint is determined by how frequently the predicate occurs in the state just before the action is executed.

Although a system like ARMS could be used to learn the preconditions and effects of tasks learned by TADPOLE, in practice this would be diffi-

cult because ARMS approach is antithetical to TADPOLEs.

The ARMS system requires a large number of examples, while TADPOLE is designed to learn useable decomposition rules immediately from a bare minimum of examples. This problem is exacerbated for tasks of higher-level decomposition rules because the more abstract a decomposition rule is, the fewer examples of it being executed will be presented to the agent.

Furthermore, ARMS constraints are too strong for an agent without a complete understanding of the domain. For example, one constraint requires that at least one effect of an action be in the precondition of a subsequent action (to ensure that the action is actually useful for something). However, for an agent with no understanding of electricity, this constraint would be violated by a plan involving flipping an electrical switch which has no visible effect other than changing the position of the switch itself. TADPOLE is designed to elucidate *what* the teacher was trying to achieve at each part of the plan without requiring the agent to understand exactly *how* the teacher achieved it.

2.4.9 Gap Solver

Gap Solver is an algorithm that is most closely related to TADPOLE and was developed concurrently with it [20]. It is a learning algorithm for learning new HTN methods and extending the domain coverage of HTN planners.

Standard HTN planners do not include preconditions of atomic operators and do not have goals describing what each method tries to achieve. This means that an HTN planner has no way of determining plan correctness, and it must rely on its decomposition rules to ensure that the generated plans are correct. However, this stringent requirement makes the rules prohibitively difficult to learn. To get around this difficulty, Gap Solver extends the standard HTN representation to include preconditions

on atomic operators and goals on methods. Because of these extensions to the standard HTN representation, a modified HTN planner can determine whether the plan it generated is correct and, if it is not, backtrack and try alternative methods or bindings. This flexibility in planning means that the planner no longer relies on perfectly crafted method rules, and Gap Solver can get away with learning minimal method preconditions (just enough to bind all of the variables in the method decomposition).

Gap Solver learns new method rules from demonstrated lessons of a teacher where each lesson consists of a goal the teacher achieved and the sequence of atomic actions that achieved it. Gap Solver does a top-down and a bottom-up parse simultaneously, and then fills any gaps between the top-down and bottom-up parses with newly learned methods. The top-down parse is just the resulting decomposition hierarchies of a backtracking HTN planner (as much as it is able to generate with its current rule set). The bottom-up parse results from applying inverted methods — rules for composing sub-tasks into a super-task which inherit the precondition of the inverted method. It is unclear whether this kind of search scales well (Gap Solver has only been partially implemented so this question remains unresolved). The top-down planner backtracks resulting in multiple top-down decomposition hierarchies. The bottom-up parser also results in multiple bottom-up hierarchies. Potentially, this may result in a large number of candidate combined hierarchies. Selecting the best candidate (the one which will result in the most reusable learned methods) is an important part of the algorithm, but this has only been partially addressed. Gap Solver currently uses only one heuristic: preferring candidates whose newly learned methods have more primitive tasks included in their method decompositions. After selecting a candidate decomposition hierarchy, Gap Solver uses explanation based generalization to learn the minimal preconditions of the new methods. Finally, it normalizes the learned methods, changing the common constants into variables.

It is not clear whether a backtracking HTN planner using minimalist

rules will scale. One of the driving motivations behind HTNs was to improve upon the inefficiency of classical planners by using domain-specific decomposition rules encoding search control knowledge. Using only minimal decomposition rules results in a loss of much of the search control knowledge. For any goal or sub-goal there may be multiple methods for achieving it most of which are not appropriate in the current state. Minimalist rules that do not guide the planner in the selection of appropriate methods force the planner to backtrack to try alternate methods and bindings. It remains to be seen whether this is tractable in rich domains.

Another important limitation of Gap Solver is that it relies on complete knowledge of how operators affect the domain. Gap Solvers modified HTN planner relies on complete operator knowledge to determine whether a generated plan is correct so that it knows when to backtrack, and Gap Solver relies on the planners ability to backtrack to get away with learning only minimal rules. However, complete operator knowledge is an unreasonable demand on a learner in rich, complex domains. For example, to completely understand the atomic action of turning on an electric kettle, the agent needs to have an understanding of electronic circuits.

TADPOLE avoids the limitations of Gap Solver by relaxing the requirement of ensuring plan correctness (a requirement that is impossible to satisfy in an unpredictable domain anyway). It does only a bottom-up parse, and it learns rich preconditions for both atomic and non-atomic rules based on what the teacher usually does. This results in sensible, useful rules, without requiring complete understanding of the domain.

Chapter 3

Representation

This chapter describes the way that HOPPER (Chapter 4) and TADPOLE (Chapter 5) represent states, goals, tasks, and why decoupling tasks and goals results in more re-usable decomposition rules.

Previous systems that have used decomposition rules, such as Icarus and HTNs (described in Chapter 2), have used a wide range of terms such as goals, tasks, skills, and methods to describe their decomposition rules and the components the rules are made up of. To add to the confusion, different systems use semantically different decomposition rules made up of different components. This has a marked impact on the algorithms that make use of the different decomposition rules.

The purpose of this chapter is to clearly define the terms that will be used in subsequent chapters of the thesis. The chapter also describes the task to sub-goal decomposition rules used by HOPPER and learned by TADPOLE, contrasts them with the decomposition rules used by other systems, and argues that task to sub-goal decomposition rules are most appropriate for the HPD.

Because TADPOLE learns in a rich domain, the states and decomposition rules it learns are extensive. For the sake of clarity, this chapter presents only very simplified examples of states, goals, tasks, and decomposition rules. A fully detailed exampled of a state and a decomposition

rule (including its head-task and sub-goals) can be found in the appendix.

Organization of the chapter

- Section 3.1 defines the terms *goal* and *task* and clarifies the distinction between the two.
- Section 3.2 explores the various ways that decomposition rules can be constructed using *goals* and *tasks*, what effect the types of decomposition rules have on algorithms that make use of them, and what kind of decomposition rule is most appropriate in the human planning domain.
- Section 3.3 describes how HOPPER and TADPOLE represent states, the passage of time, and state differences.
- Section 3.4 describes how HOPPER and TADPOLE represent goals, tasks, and goal-state differences; as well as describing their internal structure.
- Section 3.5 describes the representation of the decomposition rules used by HOPPER and learned by TADPOLE in terms of the task and sub-goals that comprise it.
- Section 3.6 concludes the chapter by describing limitations of the representation scheme used by TADPOLE and HOPPER and ways in which it will need to be extended if it is to scale to the Human Planning Domain.

3.1 Distinction between goals and tasks

Goals and tasks are similar to each other and it is easy to confuse the two. However, they are semantically distinct and they play different roles in decomposition rules and in the algorithms that make use of them. This section distinguishes the two terms.

3.1.1 A goal is a constraint on states

A goal represents an objective that the agent wants to achieve. It is a constraint on states, specifying the requirements that have to be met in a state for the goal to be satisfied. In particular, it specifies the properties and relationships in the state that must be true and the properties and relationships in the state that must not be true in order for the goal to be satisfied. Note that a goal may be the main objective that the agent is trying to achieve, or it could be a sub-goal, a means to an end. Achieving such a sub-goal helps to achieve the main objective of the agent.

For example, the goal of having a cup of tea is satisfied in any state where an object of type "cup" is related to an object of type "tea" by the "contains" relationship and the tea object has the property of being "hot". A possible sub-goal that, if achieved, would help to achieve this main objective would be having a kettle of boiling water.

3.1.2 A task is a description of a state change

A task represents a change in the state of the world. It is a constraint on pairs of states, an earlier state and a later state, specifying how the later state should differ from the earlier state. In particular, it specifies properties and relationships that are not true in the earlier state and must become true in the later state, and it specifies properties and relationships that are true in the earlier state and must become false in the later state in order for the task to be achieved.

For example, the task of moving a cup from a cupboard on to a table is achieved if the cup was in a cupboard in some initial state and then moves to the table in a later state. Buying a cup and then moving it from the shop to the table would satisfy the *goal* of having the cup on the table, but it would not achieve the same task.

In order to be useful, a task also needs to incorporate a precondition to constrain the state changes it specifies. In the example given above, the only aspect of the state that changes is that the cup stops being related with the cupboard via the "in" relationship and it becomes related with the table via the "on" relationship. However, these two state changes by themselves are not enough to specify a meaningful task. All the state changes express is that something will no longer be in something and that something will become on something.

To fully express the task in the example, the task must also have a precondition that: constrains the objects taking part in the deleted "in" relationship to have the types "cup" and "cupboard", constrains the objects taking part in the added "on" relationship to have the types "cup" and "cupboard", and constrains the two "cup" objects that take part in both state changes to refer to the same state object.

3.2 Decomposition rules are composed of goals and tasks

A decomposition rule encodes a way of breaking down a complex problem into a number of simpler sub-problems in such a way that solving all of the sub-problems will solve the complex problem. This section describes how decomposition rules use sub-goals or sub-tasks to solve goals or tasks.

3.2.1 Decomposition rules can encode how to decompose goals or tasks

The complex problem that a decomposition usually breaks down is either a goal or a task. Both kinds of decomposition rules are useful to an agent. A rule that decomposes a goal is directly useful to an agent that wants to satisfy that goal because it provides a way of simplifying the problem. A rule that decomposes a task is also useful, although less directly. It provides the agent with a way of achieving a particular state change which it can in turn use to transform the current state into one that satisfies the goal it is trying to achieve. A rule that decomposes tasks will tend to be more specific than one that decomposes goals. This is because a task will always include a precondition that will not only describe the state change but also constrain the state in which the task, and by extension the rule, is applicable; a rule that decomposes goals does not need to have a precondition and can be more general.

For any given goal or task that the agent knows how to achieve, the agent usually knows multiple different decomposition rules for achieving it. This is because the best way to achieve a goal or task changes in different states, and a different decomposition rule is most appropriate. To allow the agent to select the most appropriate decomposition rule for achieving a goal or task in a given state, each decomposition rule has a precondition specifying what states it is applicable in.

3.2.2 Goal decomposition rules are too constrained

A goal decomposition rule specifies a way of achieving a particular goal in a particular state. However, goals are very rarely completely unachieved. Usually, some aspects of the goal are already satisfied in the current state. For example, if the agent's goal is a hot cup of milk, then it most likely will not need to make a cup or create any milk because those objects will already exist; instead, it will only need to combine them and alter their

properties. Decomposition rules describe how to satisfy the unachieved parts of goals.

A goal decomposition rule describes how to achieve a particular goal, and so its precondition must specify not only the unachieved parts of the goal, but also all of the achieved parts that will remain unchanged. For example, the rule for heating a cup of coffee in the microwave would include in its precondition the constraints that the agent is trying to satisfy the goal of having a hot cup of coffee, and that the coffee is cold in the current state. This would make the rule inapplicable for any other goal. However, the change achieved by the decomposition rule could also be useful for satisfying other goals that are unachieved in the same way. For example, the method for heating wet laundry in the microwave is the same as for heating a cup of coffee.

Because goal decomposition rules are constrained to apply to particular goals, the agent would need to learn separate, redundant decomposition rules for each kind of object it would want to heat. Rules that decompose tasks specify only the change that occurs, and so they can be re-used for different goals that are unachieved in the same way.

3.2.3 Decomposition rules decompose complex problems into actions, sub-tasks, or sub-goals

The standard way that a rule decomposes a goal or a task is by breaking it down into a sequence (that is usually ordered) of actions, sub-tasks, or sub-goals.

The simplest kind of decomposition rule decomposes the problem of satisfying a goal or achieving a task into a sequence of directly executable atomic actions. In effect, they are almost identical to macro actions, with the only difference being that their sequence of atomic actions does not have to be completely ordered. Clearly, these rules by themselves do not scale very well to more complex problems. It is not feasible to store a

separate directly executable plan of atomic actions for every goal or task that the agent could be interested in and for every possible state the agent could find itself in.

A more sophisticated decomposition rule decomposes a goal or task into a sequence of sub-tasks. Rather than committing to a particular sequence of atomic actions to execute, the rule specifies the sequence of state changes (and their order) that the agent needs to achieve to achieve the main task or to satisfy the main goal. Because the decomposition rule only specifies what the required state changes are and not how to achieve them, the agent can make use of any other rules it may have that specify how to decompose tasks to achieve the necessary state changes. The agent can use further decomposition rules to achieve sub-tasks and sub-sub-tasks, breaking the original problem down into increasingly smaller and simpler parts.

A final kind of decomposition rule is one that decomposes a goal or task into a sequence of sub-goals. Such a decomposition rule makes even fewer commitments than one that decomposes into sub-tasks. Not only does it not specify particular atomic actions to execute, it also does not specify state changes. Instead, it merely specifies what the agent should make sure is true (or make sure is false) in intermediate states without assuming what sequence of changes would actually be performed.

3.2.4 Rules that decompose into sub-tasks over-commit to future states

Decomposition rules that break a problem down into a sequence of sub-tasks or state changes make more commitments to future states than rules that break a problem down into a sequence of sub-goals. Because tasks specify a state change, they inherently make assumptions about the state they will be achieved in. A rule that decomposes into sub-tasks will, in essence, make predictions about the states the sub-tasks will be achieved

in. However, the Human Planning Domain is fundamentally unpredictable: at any moment in time an unexpected and potentially disruptive event can occur. Furthermore, depending on the complexity of the problem being decomposed, the time between the decomposition of the parent rule and the beginning of the execution of its last sub-task can be arbitrarily long. This makes predictions about future states particularly unreliable.

If an agent strives to achieve a state change in a state that is different than the one it predicted when it generated the sub-task, then the state change will often no longer be the best way (or even a possible way) of achieving what the agent wants. For example, if the agent planned to achieve the sub-task of boiling some water with a kettle in order to make a cup of tea, but when it got to the kitchen it noticed that there was already a pot of boiling water on the stove, then to achieve the necessary state-change specified in the sub-task, it would needlessly boil water with a kettle rather than making use of the water in the pot.

3.2.5 Rules that decompose tasks into sub-goals are most appropriate to the Human Planning Domain

In the HPD the agent often needs to achieve rich, complex goals that may be partially unsatisfied in many different ways. This makes decomposition rules that decompose tasks preferable because they need to encode only a single non-redundant method for achieving one particular state change making them much easier to learn.

The HPD is also unpredictable and unexpected events can occur at any time. This makes decomposition rules that decompose into sub-tasks unreliable because they depend on the predictions of future states of their sub-tasks. Decomposition rules that decompose into sub-goals are preferable because they do not over-commit to the future and so are less likely to be derailed by unexpected events. It is better to compare a sub-goal to the current state when it comes time to satisfy it rather than trying to perfectly

predict the appropriate state change an arbitrary time in the future.

When an agent tries to satisfy a goal, it can use the difference between the current state and the goal to determine the desired state change. The agent can then search through all of its decomposition rules to find the most appropriate rule that will achieve this state change, and it is not limited to searching only through the decomposition associated with the goal it is trying to satisfy. The decomposition, in turn, does not make unnecessary assumption about the uncertain future states, and only specifies the sub-goals the agent should strive to satisfy. Both HOPPER and TADPOLE use rules of this form. Section 3.5 describes how these rules are represented in more detail.

3.2.6 Previous systems have decomposed goals and procedures

Previous systems that have used decomposition rules for planning have not made a clear distinction between tasks and goals.

HTNs [38] use rules that decompose procedures[1] into sub-procedures to generate plans. A procedure is a task decomposition rule that does not explicitly specify the task's effects. These rules are most appropriate for deterministic domains and are not suitable for TADPOLE and HOPPER.

As explained above, decomposing into sub-tasks and, by extension, into sub-procedures is not appropriate in an unpredictable domain because such rules over-commit to future states. However, this has not been an issue for HTNS because they have mostly been applied to deterministic and predictable domains where future states can be predicted perfectly.

The critical problem with decomposing procedures in a nondeterministic domain is that they do not specify their effects. This means that when it is achieving a sub-procedure, the agent has no way of knowing what

[1]HTNs refer to procedures as tasks, but I use the term procedures to distinguish them from my own definition of tasks described in Section 3.1.2

sub-effect it is trying to achieve. If an unexpected event were to derail the agent's plan, it would have no way of adjusting the plan to deal with the disruption. Not specifying what the effects of procedures are also means the agent cannot use such rules in novel situations to achieve a particular effect. Furthermore, these rules are very difficult to learn. The agent has no way of knowing whether a new decomposition it learns is a new method for achieving the same procedure, or whether the new decomposition corresponds to a completely new procedure.

Because TADPOLE learns its decomposition rules by observing a teacher and HOPPER executes them in an unpredictable domain, HTN rules are not the appropriate representation for them.

Icarus [31] has been applied in very reactive domains and the representation of its decomposition rules more closely matches that of the rules used by TADPOLE and HOPPER. However, although Icarus uses task decomposition rules to specify the effects of atomic actions, it decomposes goals into sub-goals at higher levels of abstraction. As explained above, goal decomposition rules are difficult to learn and contain redundant information making them an inappropriate representation for the decomposition rules used by TADPOLE and HOPPER.

3.3 Representation of states and time

Both HOPPER and TADPOLE interact with a simulated environment and they both periodically receive information about the current state of this environment. These states are represented symbolically in first order predicate logic. This is a standard approach for systems using decomposition rules, and it strikes a balance between expressiveness of the representation language and ease of learning.

3.3.1 The description of the state is qualitative

The state of the world is described by a qualitative representation of the objects, their properties, and the relationships that hold between them. The state representation does not describe any quantities or continuous variables. Although such a representation cannot completely capture the richness of the Human Planning Domain, humans can solve many problems without being aware of the values of exact quantities. Qualitative physics [18] is a branch of Artificial Intelligence that strives to duplicate humans' ability to reason about physical effects without knowing the exact quantities involved. Although a purely qualitative representation of the state is unlikely to be sufficient to express every problem in the HPD, it is likely to be expressive enough for many if not most of them.

3.3.2 States are represented by graphs of objects

HOPPER and TADPOLE represent a state as a graph whose nodes represent objects and whose links represent relationships between the objects. Each node has a set of properties, where each property consists of a pair of atoms: the property type and its value (*e.g.* [Colour blue]). Each link between two nodes in the object graph has a list of atoms that represent all of the relationships that hold between the two objects represented by the nodes.

For example, the state of a cup lying on a table would be represented by the following (simplified) graph of objects:

obj1([Type cup], [MadeOf ceramics], [Clean no], ...)
obj1 → (On) → obj2

obj2([Type table], [MadeOf wood], [Colour black], ...)
obj2 → (Supports) → obj1

Some property types can have multiple values. For example, an object can have multiple types, but only one weight: [Type cup, dish, container], [Weight light].

3.3.3 The agent receives limited sensory information about the world

It is important to note that the sensory information that HOPPER and TADPOLE receive about the world is limited. This means that they are unable to directly observe "hidden" aspects of the state. What aspects of the state are directly observable and which are hidden depends on the particular domain. For example, in a kitchen domain, whether or not a door is locked might be a hidden property of the door, and the contents of a closed drawer may not be included in the sensory information of HOPPER and TADPOLE.

3.3.4 Time is represented by a sequence of event-driven state changes

The state information that HOPPER and TADPOLE receive is a static "snap shot" of the state of the environment at a particular time. Whenever the state changes (for example, because of the result of an action being executed) HOPPER and TADPOLE receive updated state information representing the current state of the world as described above. A sequence of static states represent the state of the world changing over time. How fine-grained the atomic actions and therefore the time-slices are is completely dependent on the particular domain. In a kitchen domain, a time slice may be as long as the time it takes for the agent to lift up a cup. In a logistics domain, on the other hand, a time slice may be as long as the time it takes for a truck to drive from one location to another.

3.3.5 State-differences are represented by labelled graphs of objects

HOPPER and TADPOLE both need descriptions of the change that occurred between an earlier and a later state. A state-difference represents these changes using a graph of object nodes and relationship links in the same way that a state does, but some of the properties and relationships of the state-difference may be labelled as either added or deleted. The object graph of a state-difference is a combination of the object graphs representing the earlier and later states. Properties of objects or relationships between objects that are present in the earlier state but not in the later state are labelled as deleted, and those that are present in the later state but not in the earlier state are labelled as added. The unlabelled properties and relationships of the object graph are those that remained unchanged between the earlier state and the later state.

For example, if a cup was in a closed cupboard in an earlier state and then the cup was on a bench and the cupboard was open in a later state, then this state-difference would be represented by the following (simplified) graph of objects (where the additions are coloured green and the deletions are coloured red):

obj1([Type cup, container], [Clean yes], ...)
obj1 → (In) → obj2
obj1 → (On) → obj3

obj2([Type cupboard], [Open no, yes], ...)
obj2 → (Contains) → obj1

obj3([Type table, surface], [MadeOf wood], ...)
obj3 → (Supports) → obj1

3.4 Representation of goals, tasks, and goal-state differences

Goals and tasks are represented as graphs of objects similar to the graphs of objects that represents a state-difference. However, the properties and relationships of goals and tasks have counts associated with them indicating the importance and relevance of each property and relationship. Goals and tasks also have additional internal structure: their object graphs are divided into semantically distinct parts.

3.4.1 A goal specifies what must and must not be true

A goal specifies a state constraint by labelling some of its properties and relationships in its object graph as *must* or *must not*. A goal is satisfied in a state only if the goal is matched with the state in such a way that all of its *must* properties and relationships are matched successfully and all of its *must not* properties and relationships are not. This means that in order for the goal to be satisfied, the goal objects with *must* properties must be matched with state objects that have those properties, goal objects with *must not* properties must be matched with state objects that do not have those properties, the goal objects that are related by *must* relationships must be matched with state objects that are related by the same relationships, and the goal objects that span the *must not* relationships must be matched with state objects that are not related by those relationships.

The sub-goals of decomposition rules are learned by TADPOLE from lessons given by a teacher. In the HPD, the states and state-differences that make up the teacher's lessons contain a large number of irrelevant objects, properties, and relationships, and TADPOLE learns which properties and relationships are relevant based on experience from examples. To keep track of the properties and relationships that the agent is unsure are relevant to the goal, the goal representation includes unlabelled prop-

erties and relationships in the object graph. Because of the way goals are learned, these properties and relationships are candidate *must* properties and relationships — the agent is unsure whether or not they are important to the goal.

Because TADPOLE determines which properties and relationships are relevant based on experience from examples it sees, it will believe some candidate properties and relationships in the learned sub-goals to be more or less relevant depending on the examples they were learned from. To keep track of the agent's confidence in the different properties and relationships, the candidate properties and relationships that make up the sub-goals of a decomposition rule each have counts indicating their importance. The agent updates these counts as it refines its rules with subsequent examples.

The candidate properties and relationships do not directly determine whether or not a goal is satisfied in a state. Instead, they do so indirectly, by determining how the goal is matched with a state (this is covered in more detail in Chapter 4).

For example, the following (simplified) graph of objects would represent a goal that will be satisfied in a state where there is a hot cup of coffee (the *must* properties and relationships are coloured green and there are no *must not* properties and relationships):

Total: 10
obj1([Type cup(8), glass(2), container(10)], [Colour black(4), white(4), clear(2)], ...)
obj1 → (Contains) → obj2

obj2([Type liquid], [Taste coffee], [Temperature hot], ...)
obj2 → (In) → obj1

The goal in the example specifies that in order to be satisfied, an object in

the state must contain hot, liquid coffee. It also specifies that it is important that the containing object be at least a container, if not a cup, but that its colour is not as important.

3.4.2 A task specifies a state change and a precondition

Tasks form the head of decomposition rules learned by TADPOLE and used by HOPPER. They specify what state change will occur if the decomposition rule is executed successfully. Because these rules are usually only applicable in certain states, they have also have a precondition which is also encoded in the task.

A task specifies a state change by labelling some of the properties and relationships in its object graph as *additions* or *deletions* indicating what aspects of the state will change if the task is achieved. The remaining unlabelled properties and relationships form the precondition of the task constraining the labelled state changes. In other words, the labelled parts of the object graph specify how the state will change and the unlabelled parts specify what will change.

The preconditions of the decomposition rules used by HOPPER are learned by TADPOLE from lessons given by a teacher. Because TADPOLE determines which properties and relationships are relevant based on experience from examples it sees, it will believe some properties and relationships in the learned preconditions to be more or less relevant to their decomposition rules depending on the examples they were learned from. To keep track of the agent's confidence in the different properties and relationships, the properties and relationships that make up the precondition of a decomposition rule each have counts indicating their importance. The agent updates these counts as it refines its rules with subsequent examples.

For example, the following (simplified) graph of objects would represent a task specifying the state change of delivering a package from one location to another by truck (the additions are coloured green and the dele-

tions are coloured red):

Total: 10
obj1([Type location(10), airport(2)]
obj1 → (HasObject) → obj2
obj1 → (ConnectedTo) → obj3

obj2([Type package(10)], [MadeOf cardboard(8), wood(2)], ...)
obj2 → (AtLocation) → obj1
obj2 → (AtLocation) → obj3

obj3([Type location(10), airport(3)]
obj3 → (HasObject) → obj2
obj3 → (ConnectedTo) → obj1

The task in the example specifies that if the rule is successfully executed, the package object will no longer be related by an AtLocation relationship with location obj1 and it will become related by an AtLocation relationship with location obj3. The rest of the properties and relationships specify the precondition of the decomposition rule, what should be true in the state when the task is achieved. It is particularly important that the type of the object that changes location is package and that the two locations are related by a ConnectedTo relationship. This is because a truck can only be used to deliver packages, and it can drive only between locations that are close enough to each other.

3.4.3 Goals and tasks have a core and a context

Goals and tasks have a similar, though semantically distinct, structure: the labelled and unlabelled properties and relationships in their object graphs. Subsequent chapters of the thesis will also refer to the labelled properties

and relationships (additions and deletions for tasks and *must* and *must not* for goals) as the core, and to the unlabelled properties (the precondition for tasks and the candidates for goals) as the context.

3.4.4 Goal-state differences encode how the current state needs to be transformed to satisfy a goal

Although the decomposition rules of HOPPER and TADPOLE are ways of achieiving tasks, they can just as easily be used to satisfy goals. The way to use these decomposition rules to satisfy a goal is to first match the goal to the current state, find the difference between the two, and then find a decomposition whose head-task would transform the current state in the appropriate way. After the decomposition rule is applied, the same method can be used recursively to achieve each of the sub-goals in the decomposition.

A goal-state difference is represented by a graph of objects. It labels its properties and relationships in four different ways: what has to be made true, what has to be kept true, what has to be made false, and what has to be kept false. This depends on which aspects of the goal are already satisfied in the current state and which are not. If no properties or relationships have to be made true or made false then the goal is achieved, otherwise the agent searches for a decomposition rule whose head task will achieve the desired state changes without undoing those aspects of the goal that are already satisfied.

The remainder of the goal-state difference graph consists of the properties and relationships of the state objects that the goal objects are matched with in the current state. The agent uses these unlabelled properties and relationships to compare them with the preconditions of various decomposition rules to determine whether they are applicable in the current state.

For example, if the agent wants to satisfy the goal of having a hot cup

of coffee in a state where there is a cold cup of coffee, then the goal-state difference would be (where the *make true* properties and relationships are coloured green, the *keep true* are coloured dark green):

Total: 10
obj1([Type cup, container], [Colour blue], ...)
obj1 → (Contains) → obj2

obj2([Type liquid], [Taste coffee], [Temperature hot], ...)
obj2 → (In) → obj1

3.5 Representation of decomposition rules

As well as decomposing tasks into sub-goals, the representation of decomposition rules used by HOPPER and learned by TADPOLE includes constraints that apply to the decomposition rule as a whole.

3.5.1 The sub-goals of a decomposition are partially-ordered

For many tasks the order that the sub-goals are achieved in is important. For example, when mixing food, the cover of the blender should be closed *before* the blender is turned on. It would be sufficient for decomposition rules to store one total-ordering of their sub-goals, describing one known way of achieving their task. However, often some sub-goals have no dependencies between them and can be achieved in arbitrary order. For example, when making lunch it does not matter whether you make a sandwich or a cup of coffee first. Sub-goals that have no ordering constraints between them can often be achieved in parallel. Knowing which sub-goals can be achieved in parallel enables the interleaving of the achievement of sub-goals, and in general allows for more efficient plans.

Every decomposition rule has a set of sub-goal dependencies specifying for each sub-goal which other sub-goals need to be satisfied first. From these sub-goal dependencies it is straightforward to determine the partial ordering of the sub-goals. The sub-goal dependencies are also useful for determining how to interleave two decompositions (this is explained in Chapter 4).

3.5.2 Decompositions have variables constraining object matching in sub-goals

It is usually important for the goal objects in different sub-goals of the same decomposition to refer to the same state objects. For example, a simplified decomposition for making a cup of tea could look like:

Task: make a cup of tea
- sub-goal: cup on table
- sub-goal: tea-bag in cup
- sub-goal: kettle filled with boiling water
- sub-goal: cup filled with boiling water
- sub-goal: tea-bag in trash

In this decomposition rule, it is critically important that the cup referred to in the task and the sub-goals be the exact same cup object in the state and not different cups. When the head task is matched with a goal-state difference and when each of the sub-goals are matched with a state, it is important that the cup objects specified in their object graphs match with the same cup state object.

Decomposition rules keep track of these object matching constraints with variables. A variable in a decomposition rule constrains two or more object nodes in the head task and in the sub-goals to match to the same object in the state.

3.5.3 Decomposition rules can be recursive

Because the decomposition rules of HOPPER and TADPOLE decompose tasks into sub-goals, they cannot be directly recursive. However, in the right state, the best way to satisfy one of the sub-goals of a decomposition may be to use the same decomposition rule recursively. For example, the decomposition for the task of getting an object out of a container may be to first satisfy the goal of the container being open, then to satisfy the goal of the object not being in a container, and then to satisfy the goal of the container being closed. If the object inside is within another smaller container, then the the same decomposition rule can be used recursively to get it out of the smaller container and out of any other nested containers that may be inside.

3.6 Limitations

Although the representation used by HOPPER and TADPOLE is expressive, it has a number of shortcomings that have to be addressed if the algorithms are to scale to the Human Planning Domain.

3.6.1 The state representation cannot express quantities

The biggest limitation of the state representation used by HOPPER and TADPOLE is that it is fundamentally qualitative. Although this provides a good environment in which to use and learn decomposition rules, it cannot express quantitative properties of the world which are often critically important in the human planning domain.

There is no measure of similarity among property values and relationships that describe a state; two property values are either equal or unequal. For example, the two colours light red and dark red are as dissimilar as the colours red and blue.

Multivalued properties help to address this issue to some extent. For example, the similarity of a truck and a car would be expressed by their type properties having some property values in common: [Type truck, vehicle] and [Type car, vehicle]. However, this representation is by itself not adequate to express numerical quantities such as the distance between two places. Such numerical properties and relationships may be very important in the human planning domain. For example, distance is critical when deciding how to get from one location to another.

There are two main ways of addressing this problem. The first is to extend the state matching aspects of the HOPPER and TADPOLE algorithms to recognize that two unequal values for numerical properties may nevertheless be very similar. The second way is to transform a quantitative state description into a qualitative one and then present that to HOPPER and TADPOLE as normal. For example, the relationships [Distance 104metres], [Distance 193metres], [Distance 49kilometres], [Distance 158kilometres] could be transformed into [Distance near], [Distance near], [Distance far], and [Distance far] respectively. Which approach is best remains an issue to be explored.

3.6.2 The representation cannot express continuous time

An important limitation of the way that time is represented by HOPPER and TADPOLE is that it is qualitative in nature. Not only is time discrete, but the duration in absolute time between time steps can vary. HOPPER and TADPOLE only receive a state update when a state change occurs. They have no way of determining how long in absolute time the state has remained unchanged between time steps. However, in the HPD, actions and their resulting state changes can take a variable amount of time and this can be significant, especially when the agent is achieving multiple tasks at once.

One way of addressing this issue without modifying either HOPPER

or TADPOLE is to make sure that the time slices between states are very fine grained. In this way, actions that take a longer amount of time can be broken down into shorter sub-actions, resulting in a greater number of time slices to execute them. For example, the action of moving a cup from one place to another can be broken down into: moving the agent's hand next to the cup, grasping the cup, lifting the cup, moving the hand and cup to the desired location, releasing the cup, and moving the agent's hand away from the cup.

However, this does not address cases where the agent needs to wait a certain amount of time until a state change occurs (*e.g.* waiting until the water in a pot starts boiling). This is particularly important when the agent wants to achieve multiple goals at once. It is only worthwhile for the agent to begin another task while waiting if the waiting time is long enough. The event-driven state change representation that simulates time in HOPPER and TADPOLE is not rich enough to distinguish between a short and a long waiting time. In order to fully scale to the HPD, the representation of time as well as HOPPER and TADPOLE will have to be extended to properly handle executing and learning from behaviour that involves waiting.

3.6.3 The state representation cannot represent compound objects

The Human Planning Domain includes hierarchical compound objects and it is often important to deal with them at multiple levels of abstraction. This is especially true when the creation of compound objects is integral to the execution of a plan. For example, setting the table involves creating a set of place settings which themselves are compound objects consisting of a knife, fork, plate, cup, and so on. The state representation used by HOPPER and TADPOLE has no way of representing such hierarchically structured objects.

A related limitation of the representation is the inability of goals to represent any kind of quantification. However, this can be resolved by enriching the state description to include dynamically created sets of objects.

The kind of decomposition rules that HOPPER can use and TADPOLE can learn are determined by the state representation. This means that if the state representation is extended to include compound objects, then the goals, tasks, and decomposition rules used by HOPPER and learned by TADPOLE will be automatically extended as well.

3.6.4 Goals and tasks cannot represent disjunctive conditions

The properties and relationships that specify the state constraints of goals and the preconditions of tasks are all conjunctive conditions. Goals and tasks do not represent disjunctive conditions because this would unduly complicate TADPOLE's learning task. It may sometimes be useful to be able to represent a disjunctive condition within a single goal or precondition, although this is rare.

A decomposition rule with a disjunctive precondition can be expressed by dividing the disjuncts among multiple decomposition rules that all achieve the same task, with each disjunct in the precondition of a separate rule. However, this would involve a significant amount of redundant information because each of the rules encoding each disjunct would have to have a redundant copy of the same decomposition.

A disjunctive sub-goal can be expressed by a higher-level constraint that forms a superset of all of the disjuncts. The goal could then include additional constraints to exclude any undesired objects. For example, if the object to deliver a package should be a truck or a car, then this sub-goal can be expressed by constraining the type of the object to be a vehicle that can handle a payload at least as heavy as a the heaviest package (excluding bicycles). However, for some disjunctive goals it may be difficult (or even

impossible in some domains) to express them in this way.

The context of goals and tasks provide a way of approximating disjunctive conditions. Given enough examples to learn from, any disjunctive conditions will become more heavily weighted than the other properties and conditions in the context. The more alternatives in the disjunction the more examples are necessary to distinguish the weighting of each of the alternatives.

However the context can only at best roughly approximate disjunctive conditions because every property and relationship in the context is treated independently of every other, so disjunctive conditions involving more than a single property or relationship cannot be accurately represented.

3.6.5 Variables cannot constrain object properties

The variables in decomposition rules currently only constrain the identities of different task and goal nodes, making sure that they all match to the same state object. However, it is sometimes important to also constrain the properties of different objects in tasks and in sub-goals. For example, when two substances (liquids or powders) are mixed together, then the resulting mixture has the properties of both original substances. The current representation has no way of expressing this state change — a single task can only specify what happens when two specific substances are mixed together (*e.g.* if hot water and a tea-bag are mixed together, then the resulting liquid will taste like tea).

Property variables can also be important when expressing sub-goals. For example, when setting the table, a repeated sub-goal could be to place a plate and a cup of the same colour on the table. The current representation has no way of expressing this goal — it can specify that the cup and plate should be a particular colour, but it has no way of expressing the constraint that they be the *same* colour.

Chapter 4

HOPPER

This chapter describes HOPPER (Hierarchical Ordered Partial-Plan Executor and Re-planner), an algorithm that uses decomposition rules to generate and execute plans.

As well as being a planner, HOPPER also serves the secondary function of being a testbed for the decomposition rules learned by TADPOLE (Chapter 5). The only effective way of evaluating TADPOLE is by actually using the decomposition rules it learns to generate plans. Although HOPPER can stand alone and use hand-coded rules, it is specifically designed to use the decomposition rules learned by TADPOLE.

A tertiary function of HOPPER is to further refine the rules learned by TADPOLE. HOPPER learns from experience and refines its decomposition rules based on whether or not they successfully achieved their task when HOPPER executed them.

Organization of the chapter

- Section 4.1 presents an overview of the HOPPER algorithm. It explains how HOPPER creates and maintains a goal decomposition hierarchy by decomposing goals into sub-goals. Subsequent sections elaborate on important aspects of the algorithm.

- Section 4.2 covers the exceptions to the decomposition algorithm: when HOPPER is unable to decompose a goal or sub-goal, and when a goal has to be decomposed multiple times to be achieved.
- Section 4.3 delves more into the mechanics of how goals are matched to states and how tasks are matched to goal-state differences.
- Section 4.4 describes how HOPPER detects and reacts to unexpected events (both disruptive events and unexpected opportunities).
- Section 4.5 describes the requirements of a planning and plan-executing algorithm that achieves complex goals in the HPD. It then explains HOPPER's least commitment decomposition strategy, and how it is particularly suited to meet these requirements.
- Section 4.6 explains the importance of minimizing the side-effects of decompositions with "clean-up" sub-goals and how HOPPER deals with such sub-goals when it updates its decomposition hierarchy.
- Section 4.7 describes how HOPPER interleaves parts of its decomposition hierarchy in order to execute sub-goals in parallel and to produce a more optimal plan.
- Section 4.8 describes HOPPER's limitations and the kinds of problems it is not suited to handle well.
- Section 4.9 concludes the chapter by discussing how HOPPER could be extended to handle non-routine, novel tasks.

4.1 Overview of the Algorithm

This section presents a simplified overview of HOPPER's planning and plan-execution algorithm; subsequent sections provide further details about specific aspects of the algorithm. The pseudo-code for the HOPPER algorithm can be found in appendix A.1.

4.1.1 HOPPER acts in sense-action-sense cycles

Although HOPPER is a planning agent that produces and executes plans with look ahead, it interacts with its environment like a reactive agent. It achieves a goal by calling the achieveGoal function (A.1.1) which first initializes the decomposition hierarchy, and then repeatedly interacts with a simulated environment at a sequence of discrete time steps or cycles until the goal is achieved or HOPPER determines that it cannot achieve the goal.

At each time step, cycle (A.1.3) receives limited sensory information about the world, and from this it constructs a partial description of the current state. Using this information and HOPPER's decomposition rules, it selects and returns an atomic action to execute.

4.1.2 HOPPER generates a goal decomposition hierarchy

HOPPER achieves a goal by choosing an appropriate decomposition rule, using it to decompose the goal into a partial order of sub-goals and then recursively decomposing the sub-goals into sub-sub-goals and so on all the way down to atomic goals. Atomic goals are goals that are decomposed with atomic decomposition rules; rather than being decomposed into a partial order of simpler sub-goals, atomic goals are decomposed into a single atomic action that will achieve them in the current state.

In this way, HOPPER constructs a goal decomposition hierarchy by associating higher-level goals with partial orders of sub-goals. Each partial-order of sub-goals associated with a higher-level goal indicates which goals

have to be achieved (and in what order) in order to achieve the higher level goal.

While it is constructing the decomposition hierarchy by recursively decomposing sub-goals, HOPPER only decomposes the unconstrained sub-goals in each partial order: those sub-goals that are not ordered to come after any other sub-goal. HOPPER decomposes goals only when necessary; only the goals that are achievable by HOPPER in the current time step are decomposed further. Goals that are to be achieved in future time steps remain undecomposed. In this way HOPPER builds a goal decomposition hierarchy with the earlier goals and sub-goals more fully decomposed than the later ones.

Figure 4.1 shows a graphical representation of the decomposition hierarchy HOPPER would generate to achieve the goal of having lunch. HOPPER would choose a decomposition rule appropriate for achieving the goal in the current state and then use it to decompose the goal into a number of sub-goals that, if achieved in the given order, would achieve the main goal of having lunch. The sub-goals in each decomposition in the figure are strictly ordered except for those joined by double-headed arrows which can be achieved in either order.

The figure shows one valid decomposition for having lunch: making tea, making a sandwich, eating, and then cleaning up afterward. Of the four top sub-goals, making tea and making a sandwich are unconstrained and do not depend on earlier sub-goals, and so HOPPER decomposes both of these sub-goals into partial-orders of sub-sub-goals (note that for the sake of clarity and readability, only the decomposition for making tea is shown in Figure 4.1). The eating and cleaning up goals cannot be achieved until after the tea and sandwich have been prepared, so those goals remain undecomposed.

When HOPPER is first presented with a goal to achieve, it initializes an empty decomposition hierarchy with the initializeHierarchy function (A.1.2). It then calls the updateHierarchy function (A.1.6) to expand the

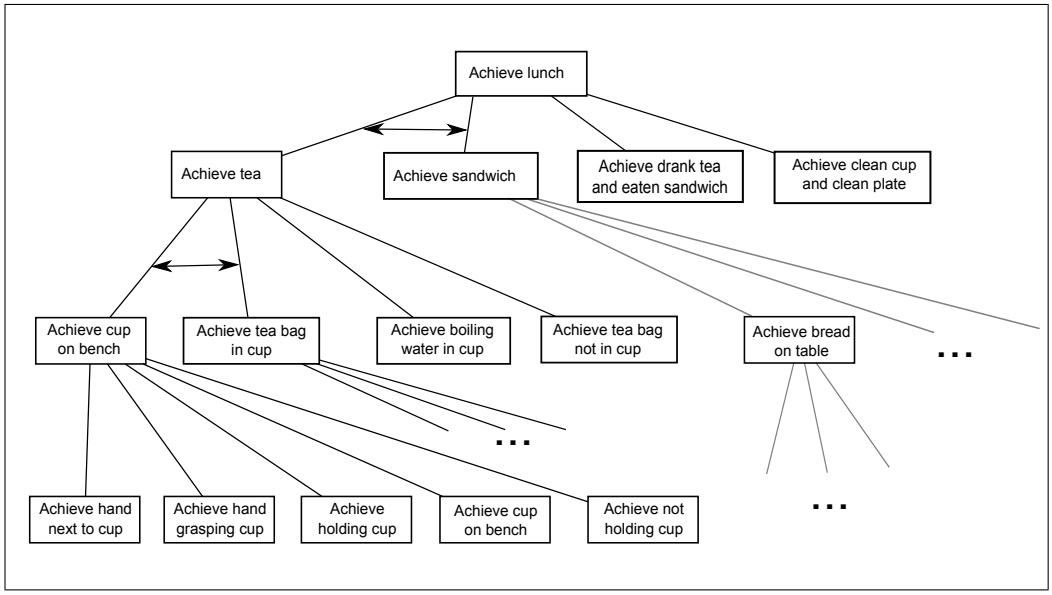


Figure 4.1: Decomposition Hierarchy for having lunch

newly intialized decomposition hierarchy. Each newly initialized goal in the decomposition hierarchy has no sub-goals, and for each such goal the updateHierarchy function calls the decompose A.1.8 function to find its appropriate sub-goals. HOPPER then recursively calls updateHierarchy on the sub-goals to generate the rest of the decomposition hierarchy.

4.1.3 HOPPER maintains its goal decomposition hierarchy

At each time step, after HOPPER receives sensory information and updates its state description, it calls the updateHierarchy function to go down the decomposition hierarchy and update the unconstrained sub-goals (the sub-goals not ordered to come after any other sub-goal), from the top-goal, down the left-hand side of the hierarchy to the atomic goals at the bottom.

updateHierarchy removes any unconstrained goal that is satisfied in the current state (whether by design or serendipity) from the decomposition hierarchy. The function removes a satisfied unconstrained goal even if it has unsatisfied sub-goals (unless they are labelled as clean-up, see Section

4.6): the goal and the entire sub-hierarchy below it is removed. The sub-goals of a goal in the hierarchy specify how to achieve the higher-level goal, and if the higher-level goal has already been achieved, then there is no reason to achieve the sub-goals.

Note that updateHierarchy does not remove any satisfied goals that are ordered to come after any other goal. This is because achieving the earlier goals in the decomposition may undo the future goals that are currently satisfied. For example, the decomposition to deliver a package by truck involves driving a truck to where the package is, loading the package into the truck, driving the truck to the destination, and unloading the truck. The final sub-goal of the decomposition is to make the package not be in the truck. Initially, this final sub-goal is satisfied (the truck is initially empty), but it is incorrect to remove this sub-goal from the decomposition hierarchy. It is important that the goals of a decomposition be achieved in their specified order, and so updateHierarchy removes only the unconstrained satisfied goals from the decomposition hierarchy.

After the decomposition hierarchy has been updated and the unconstrained satisfied goals removed, new goals in the hierarchy become unconstrained. When a goal is removed from the decomposition hierarchy, the subsequent goal (or goals) that were ordered to come after that goal become unconstrained. These newly unconstrained goals need to be updated as well. If they are satisfied in the current state, then they are also removed from the decomposition hierarchy and the next goal (or goals) in the decomposition become unconstrained. The newly unconstrained goals that are not satisfied are recursively decomposed down to atomic goals in the manner described in Section 4.1.2. updateHierarchy continues this process until there are no un-updated, unconstrained goals in the decomposition hierarchy.

As time progresses, subsequent time steps pass, the state changes, and HOPPER receives new sensory information; the unconstrained goals in the hierarchy become satisfied, they are removed, and the subsequent

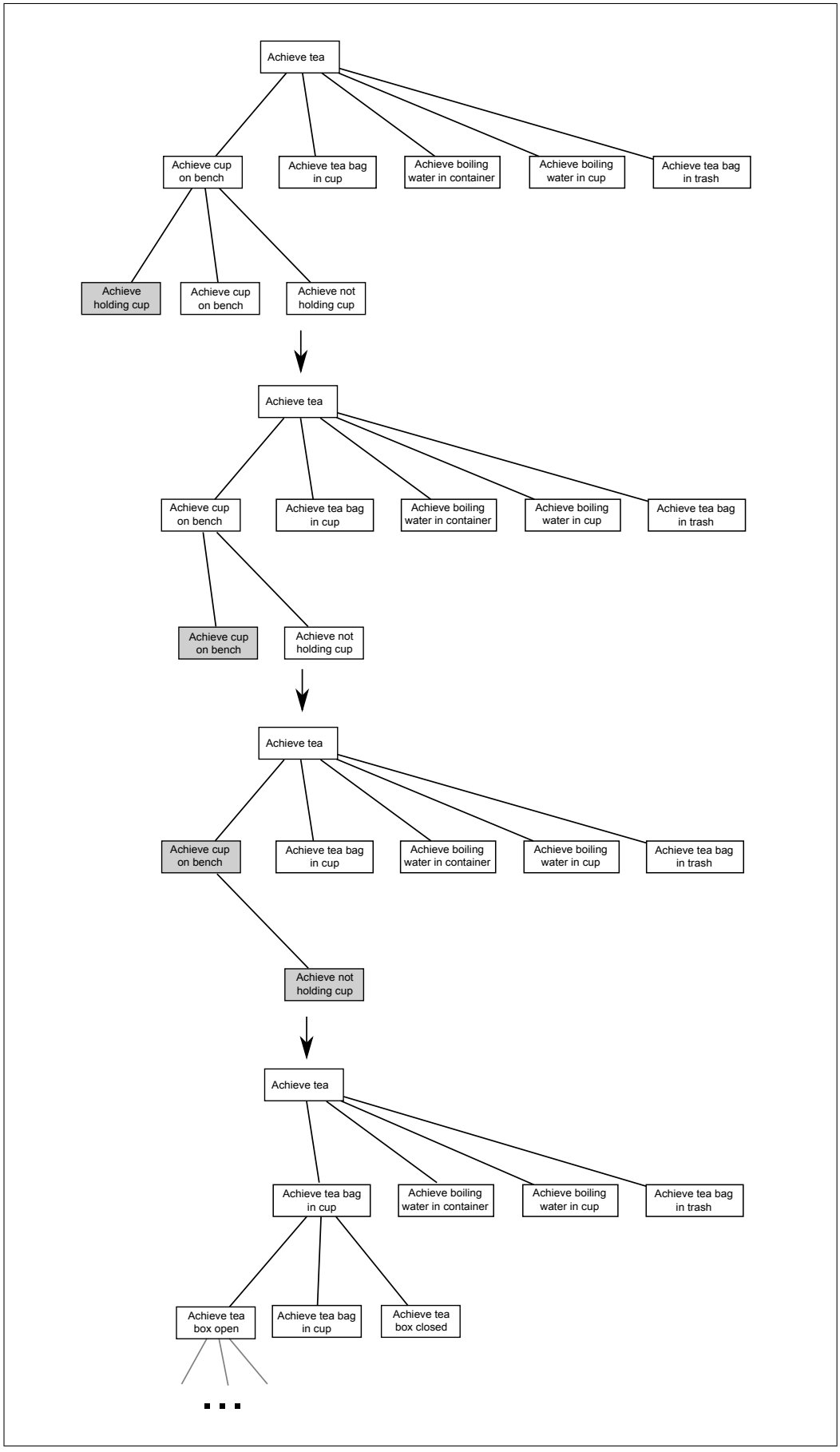


Figure 4.2: Achieving a cup of tea

goals are decomposed. In this way the decomposition hierarchy gradually shifts: earlier, satisfied goals are removed and future, unsatisfied goals are decomposed and achieved. This process continues until all the goals in the hierarchy have been decomposed and satisfied, culminating with the achievement of the top goal.

Figure 4.2 shows an example of how the decomposition hierarchy to achieve a cup of tea changes as the initial unconstrained sub-goals are achieved. The highlighted goals in the decomposition hierarchy indicate the unconstrained goals that are satisfied in the current state. The example decomposition for making a cup of tea is fully ordered so that sub-goals are decomposed one by one; Section 4.7 covers the more complex case of interleaving the plans of co-ordered sub-goals.

The first step in making a cup of tea is to get the cup on the bench, and the first step to achieve that is to grasp the cup. After this goal is achieved, HOPPER removes it from the hierarchy. In the next state, the goal of the cup being on top of the bench is achieved and, because it is now unconstrained, HOPPER removes it from the hierarchy as well. The parent goal remains unsatisfied because it still has an unsatisfied clean-up sub-goal of not grasping the cup (Section 4.6 explains the importance of clean-up sub-goals). In the next state, both the low-level goal of not holding the cup is achieved as well as the higher level goal of getting the cup on the bench. HOPPER removes both of these goals from the decomposition hierarchy. In general, achieving the last sub-goal of a decomposition will tend to achieve the associated higher-level goal. The next sub-goal of getting a tea bag into the cup now becomes unconstrained (it depends only on the cup being on the bench). Because this sub-goal is not atomic, HOPPER decomposes it into the appropriate partial order of sub-goals (note that for the sake of clarity the lowest level of the decomposition is not shown in the figure) and the process continues until the top goal is satisfied.

4.1.4 HOPPER achieves atomic unconstrained goals

Once the decomposition hierarchy has been completely updated in the current time step, HOPPER uses the hierarchy to select an atomic action to execute with the chooseAtomicAction function (A.1.14). It has to select an atomic unconstrained goal from the decomposition hierarchy and execute the corresponding atomic action that will achieve it in the current state.

Note that there may be two or more unconstrained atomic goals in different decompositions to choose from. This is because the ancestors of the unconstrained atomic goals are co-ordered in a single decomposition somewhere higher in the decomposition hierarchy.

For example, when achieving the goal of having lunch, the two sub-goals of making tea and making a sandwich can be achieved in either order and so HOPPER would decompose both at the same time. These two sub-decomposition hierarchies would bottom out with atomic goals resulting in at least two unconstrained atomic goals at the bottom: one for achieving tea, and one for achieving a sandwich. One of the unconstrained atomic goals could be to get the agent's hand next to the cup (in order to pick it up, in order to get it on the bench, in order to have it filled with hot water, in order to make tea), and the other could be to get the agent's hand next to the cupboard handle (to grasp the handle, to open the pantry, to get the bread on the table, to make the sandwich).

Because each goal in the decomposition hierarchy except for the root goal is a means of achieving a higher-level goal, choosing which unconstrained atomic goal to achieve (with an atomic action) is a choice of which of the two co-ordered ancestor goals to start achieving. When faced with such a choice, chooseAtomicAction calls the chooseActionFromCandidates function (A.1.15) which selects the one that is the most achieved already: the one whose highest level sub-decomposition has the most sub-goals achieved already.

For example, if the agent has to choose between achieving a cup of tea and achieving a sandwich in the current time step and it notices that

there is already a kettle of boiling water on the stove, then it will choose to begin achieving a cup of tea because the decomposition for that goal has a sub-goal that is already achieved.

If the co-ordered goals are equally unachieved; if they have the same number of sub-goals already achieved at the same level of the decomposition hierarchy (this will usually only happen when the goals are initially decomposed and they have no sub-goals already achieved), then chooseActionFromCandidates will choose the one that it thinks will be the easiest to achieve. This means the one with the with the shallowest sub-hierarchy. A shallower hierarchy will tend to have fewer leaf nodes, which means fewer atomic goals, which ultimately corresponds to fewer atomic actions, and thus implies that the parent co-ordered goal is simpler (easier to achieve). So when there are multiple unconstrained atomic goals in different decompositions, chooseActionFromCandidates selects the one that is highest in the decomposition hierarchy (it has the fewest number of ancestors between it and the root goal).

If the unconstrained atomic goals are at the same level of the decomposition hierarchy, and their ancestor co-ordered goals are equally unachieved, then chooseActionFromCandidates selects arbitrarily among them. However, once it has made a choice, HOPPER will in general continue achieving the same co-ordered goal in subsequent time steps. This is because after one of the co-ordered goals has been chosen, it will in general be more achieved (by one atomic goal) than the other alternative co-ordered goals in the next time step, and so HOPPER will choose to execute it again. The only way that HOPPER can switch to trying to achieve a different co-ordered goal is if an unexpected event were to serendipitously (see Section 4.4) achieve one of the high-level sub-goals of one of the other co-ordered goals. However, switching becomes increasingly unlikely as HOPPER achieves more of the co-ordered goal in subsequent time steps.

4.1.5 HOPPER refines its decomposition rules after successfully executing them

After the updateHierarchy function successfully executes a decomposition rule (the goal being achieved is satisfied in the current state and all of its sub-goals have been satisified) it refines the decomposition rule with the refineRule function (A.1.10). The function refines the core of the head-task and the sub-goals, updates the counts of the context properties and relationships based on the state objects they were matched with, it updates the decomposition variables, and updates the partial ordering of the sub-goals based on the order they were achieved.

The way that HOPPER refines a decomposition is identical to the way TADPOLE does after a successful parse, and Chapter 5 gives the details of this process.

However, updateHierarchy does not refine a decomposition rule if the goal it was achieving becomes satisfied serendipitously. If a goal becomes satisfied because of an unexpected event when the decomposition that was achieving it is still incomplete (it still has unachieved sub-goals), then updateHierarchy has no guarantee that the way the decomposition rule was matched to the current state was correct and so it does not refine it.

4.2 Decomposition Exceptions

The standard manner in which HOPPER achieves goals (as described above) is by decomposing them into a partial order of sub-goals, and then achieving the sub-goals one by one in the order specified by the decomposition. The higher-level goal is usually achieved at the same time as its last sub-goal. However, there are two exceptions to this rule: HOPPER may be unable to decompose a goal at all, and a goal may remain unsatisfied after all of its sub-goal are achieved. This section describes these two cases in detail.

4.2.1 HOPPER redecomposes the parents of failed sub-goals

When updateHierarchy attempts to decompose an unconstrained goal with the decompose function, it may be unable to do so. This is because of incomplete knowledge: HOPPER does not know an applicable decomposition rule to decompose the goal in the current state.

When a failure occurs in the decomposition of a goal in the hierarchy, updateHierarchy is conservative and tries to minimize any change to the decomposition hierarchy. It tries to resolve the problem as low in the hierarchy as possible, and it only redecomposes higher level goals when it becomes necessary.

To make progress, updateHierarchy backtracks one level up the decomposition hierarchy to the parent goal and tries to find an alternate decomposition. updateHierarchy continues looping until it is unable to find a viable alternate decomposition; only then does it propagate the failure up the decomposition hierarchy and tries to redecompose the grandparent goal. updateHierarchy continues to propagate the failure up the hierarchy until it either finds a valid alternate decomposition or it reaches the root goal and is unable to decompose it. If updateHierarchy cannot decompose the root goal in the current state, then it has no way of achieving the main goal and so it fails.

If updateHierarchy finds a viable alternate decomposition, then it decomposes the parent goal and generates its sub-hierarchy in the manner described in Section 4.1. Note that this redecomposition can also fail and be re-redecomposed if updateHierarchy is unable to decompose one of its sub-goals.

Figure 4.3 shows an example of HOPPER achieving the goal of having lunch with the decomposition of making a sandwich, making something to drink, and then cleaning up afterward. When achieving having a drink, a decomposition that the agent could use is making a cup of tea. If the agent subsequently discovers that there is no tea in the pantry, then its decomposition of making a cup of tea would fail and the agent would

have to find an alternate way of achieving having a drink. It would then backtrack up the hierarchy and redecompose the goal of having a drink with an alternate decomposition (in this example, buying a can of soda) while preserving the rest of the decomposition hierarchy. If there were no other viable alternative decompositions (*e.g.* the agent had no money), then the failure would be propagated higher, and the agent would have to find an alternate way of having lunch (*e.g.* a sandwich with no drink).

4.2.2 HOPPER redecomposes unsatisfied goals

Usually when the last sub-goal in a decomposition is achieved the parent goal is achieved as well. However, in some cases the parent goal remains unsatisfied even when all of its sub-goals have been achieved. This can be because of four causes:

- The decomposition rule used to decompose the goal is incorrect (achieving the sub-goals in the given order does not in fact achieve the higher-level goal).
- The agent's model of the world is incorrect. The perceived state of the environment is noisy and uncertain, and the true state may be different from what the agent believes it to be. In such a case the selected decomposition rule may be inappropriate. For example, if the agent incorrectly believed that a desired object was inside a closed drawer, then the decomposition of getting something out of a closed drawer would fail to acquire the object.
- The agent executed the decomposition incorrectly. Even with the correct decomposition it is possible that the agent executed the corresponding actions incorrectly. For example, if the agent tries to bake a souffle, then even if it uses the correct decomposition, it may not beat the mixture correctly and fail.

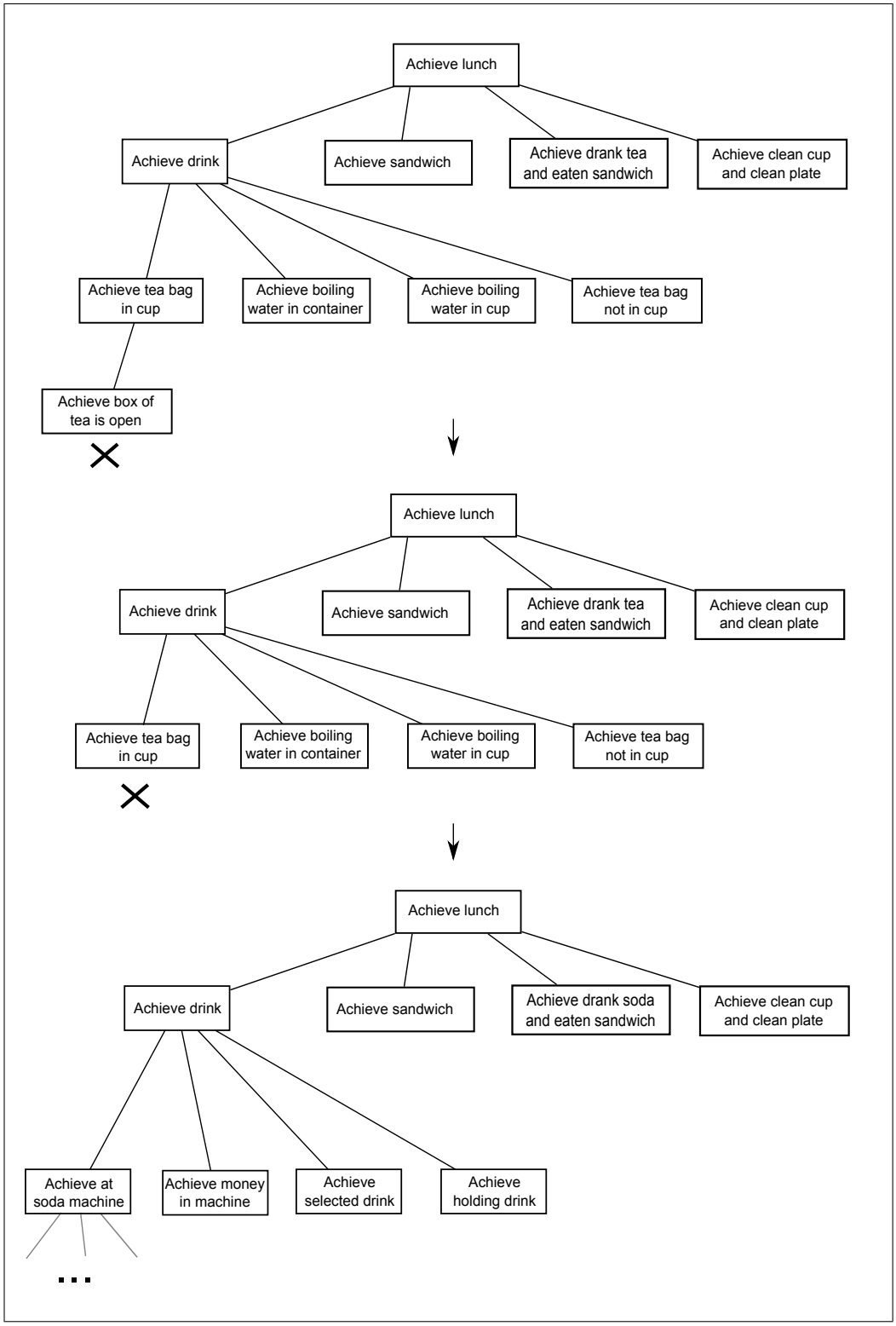


Figure 4.3: Redecomposing making a drink

- Multiple applications of the decomposition are required to satisfy the parent goal. If the agent is stirring its tea to make the sugar dissolve, then it may need to repeat this multiple times until the sugar is dissolved.

In the first two cases, HOPPER should try an alternative decomposition (or propagate the failure up the decomposition hierarchy if no alternative decomposition can be found, as described in the previous section). In the last two cases, HOPPER should try the decomposition again.

The updateHierarchy function has no way of knowing which of these cases was actually the cause, so it optimistically assumes that it was because of one of the two latter cases, and it redecomposes the parent goal. This is consistent with HOPPER's conservative decomposition update strategy: it tries to resolve problems in the decomposition hierarchy as locally as possible, and failures are propagated up the hierarchy only as a last resort.

4.2.3 HOPPER usually uses the same decomposition when redecomposing

When a goal is redecomposed, the decompose function will usually try the same decomposition again. The relevant parts of the state of the world that made the previous decomposition most appropriate to achieving the goal are likely to still hold, and so the previous decomposition will again be most appropriate. However, if the state has changed significantly since the last attempt, a different decomposition rule may be more appropriate to the new state and decompose will use it instead.

4.2.4 HOPPER attempts the same decomposition a limited number of times

A consequence of retrying decompositions is that if multiple attempts make no progress towards achieving the goal (the decomposition rule is incorrect or there is a hidden aspect of the state that makes the decomposition inappropriate), then HOPPER could get stuck in an infinite loop, repeatedly trying the same decomposition without success.

To prevent such infinite loops, updateHierarchy keeps track of how many times a decomposition rules has been tried for any given goal in the decomposition hierarchy. HOPPER tries the same decomposition only a limited number of times. After this limit is reached, updateHierarchy prevents decompose from trying this decomposition rule again to decompose the goal. If there are no other viable decompositions, it propagates the failure up the hierarchy and tries to redecompose the parent goal.

There is a trade-off between not attempting a decomposition enough times and failing to achieve the goal, and trying failed decompositions too many times and wasting time and effort. Different decompositions may have a different optimal number of times that they ought to be attempted. For example, starting an old car may require a large number of turns of the ignition key, while turning on a light switch should not be attempted more than a few times.

Currently, HOPPER uses a fixed limit of four failed attempts for every decomposition. However, extending it to be able to learn different counts based on experience is straightforward: every time that HOPPER successfully achieves a decomposition (or TADPOLE successfully parses a decomposition) it could note how many times in a row it attempted the decomposition, increase this number (*e.g.* by 50%), and set the new count for the decomposition rule to be the maximum of the current count and the new number. For example, if it took 10 stirs to completely dissolve the sugar in a cup of tea, then the next time HOPPER executed this decompo-

sition it would be willing to execute it up to 15 times.

4.2.5 Multiple redecompositions can encode indefinite behaviour

Redecomposing a decomposition multiple times is not only a useful way of retrying an incorrectly executed decomposition, it also makes decompositions more powerful, allowing them to encode indefinitely repeated behaviour in a single rule.

To accomplish some goals, it is necessary to repeat some behaviour multiple times, but it is often impossible to predict ahead of time exactly how many times the repetition will be necessary to achieve the goal. The behaviour involved can range from being very simple and low-level (*e.g.* repeatedly stirring sugar into a cup of tea until it is dissolved) to more complex and high-level (*e.g.* repeatedly unloading packages from a truck until the truck is empty).

It is important to note that a deliberative decomposition strategy has no way of making use of this feature. Instead, it has to predict the exact number of times each decomposition will be used when it generates its plan. This can be awkward when dealing with tasks where this kind of prediction is not possible (*e.g.* stirring sugar).

4.3 Matching Goals, States, and Tasks

As HOPPER maintains the goal decomposition hierarchy with the updateHierarchy function, it has to constantly check whether the unconstrained goals in the hierarchy are satisfied in the current state. To do this, the goals (a graph description of what is desired) have to be matched against the current state of the world (a graph description of what is believed to be true) to determine whether the desired goal properties and relationships hold in the state. If a goal is not satisfied, then this same matching process

determines the goal-state difference: a graph description of which goal properties and relationships have to be made true, kept true, made false, or kept false in order to achieve the goal.

HOPPER achieves goals by finding a task decomposition rule whose head task (a graph description of the properties and relationships that will change if the task is achieved) successfully matches with the goal-state difference, and then decomposing the goal into a partial order of sub-goals specified by the decomposition rule.

This section describes these two graph matchings, goals against states and tasks against goal-state differences, in more detail.

4.3.1 Goals are matched against states

A goal is represented by a graph whose nodes consist of objects with properties and whose links consist of relationships between the objects. Similarly, a state is represented by a graph of objects with properties and relationships (Chapter 3 provides more details about the representation of states, goals, and tasks).

Matching a goal against a state (to check whether it has been achieved or to determine what state changes are necessary to achieve it) involves searching for the best imapping of goal nodes to state objects. Figure 4.4 shows a possible mapping of goal nodes to state objects.

HOPPER searches for the best goal to state mapping by running a beam search where at each step an unmapped goal node is mapped to an unmapped state node until all of the goal nodes are mapped. The score of a mapping (or partial mapping) depends on how well the properties of the goal objects and their relationships match with the corresponding state objects.

Each property and relationship in the goal graph is classified as: *context*, *must*, or *must not* (together, the *must* and *must not* properties and relationships constitute the *core* of the goal). The properties and relationships

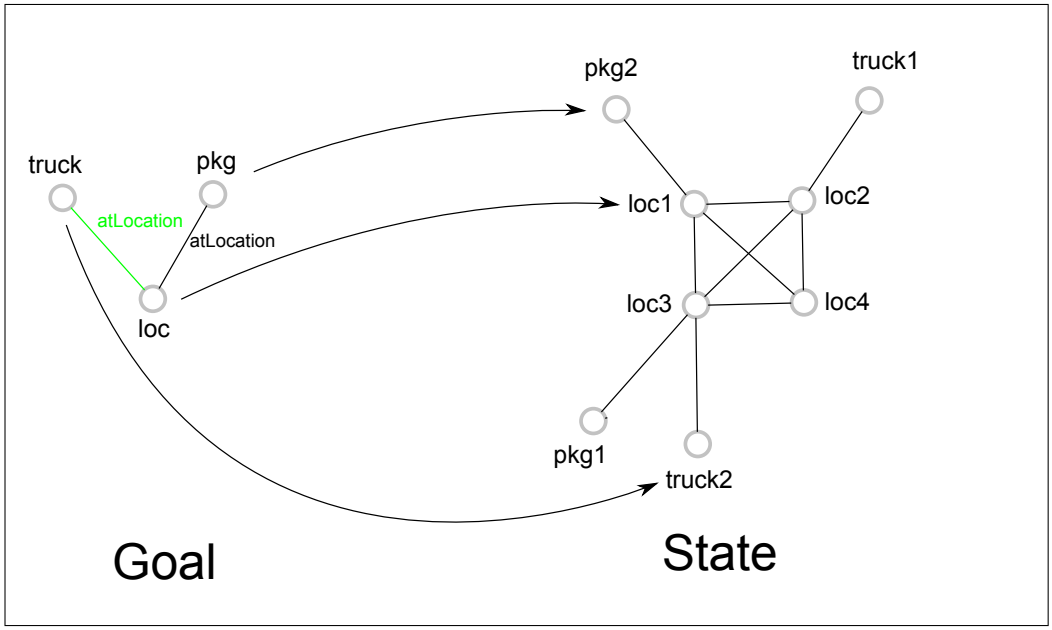


Figure 4.4: Goal-State mapping

in the *context* of the goal have a count associated with them indicating how many times they were present in successfully achieved decompositions (by HOPPER) or in successfully parsed decompositions (by TADPOLE).

The properties and relationships in the context of the goal are those that the agent believes may be important to the goal because they were present in decompositions successfully achieved by HOPPER or successfully parsed by TADPOLE, but the agent has not seen them enough times to justify assuming that they are. HOPPER gauges the importance and relevance of a property by how many times it has seen it before, and so matchings are scored based on how well the context matches the state, weighted by the relative frequencies of the properties and relationships.

The core properties and relationships specify what is necessary for the goal to be achieved. A goal is achieved only if every *must* property and relationship is present in the current state, and every *must not* property and relationship is not present. Otherwise, the goal is not achieved, and HOPPER constructs a goal-state difference graph of the necessary state

Total: 10
truck([Type: truck, vehicle], [Colour: red(4), blue(6)], ...) *must*
truck → (atLocation) → loc *must*

loc([Type: location, postOffice(3), warehouse(4), office(3)] *must*
loc → (hasObject) → pkg
loc → (hasObject) → truck *must*

pkg([Type, package(10)], [Colour: yellow(4), brown(6)], ...)
pkg → (atLocation) → loc

Figure 4.5: Goal for getting a truck to the location of a package

changes to achieve, and then it finds a task to achieve them (see below).

Figure 4.5 shows an example of what a goal to move a truck to the location of a package could look like (the only *core* parts of this goal are the *must* relationships marked green). The numbers in parentheses indicate how many times a particular context property or relationship was seen in a successfully achieved (by HOPPER) or parsed (by TADPOLE) goal. These numbers are what determine the score of a goal-state match. In the given example, the truck node would match with a blue truck object slightly better than with a red truck, the pkg node would match with a brown package slightly better than with a yellow package, etc.

HOPPER and TADPOLE use the same scoring mechanism for determining how well a goal node matches with a state node, and this is described in more detail in Chapter 5.

The goal in Figure 4.5 would be satisfied if: the truck node were mapped to a state object of type truck and vehicle, the loc node were mapped to a state object of type location, and these two state objects were related to each other by the atLocation and hasObject relationships.

It is important to note that only the context properties and relationships of the goal contribute to the score of a mapping. When a goal is matched

with a state, having more or fewer core properties matched does not indicate a better or worse match but a goal that is more or less achieved. A goal-state mapping that has fewer unachieved core properties will still require a task to achieve it and it is unclear whether tasks that make fewer changes are shorter or easier to achieve than tasks that make more changes. For example, changing your bank account from $0 to $500,000 (a small change) can require an enormous amount of time and effort to achieve. On the other hand, changing your house into a smoking, charred ruin (a large change) requires no more effort than setting the curtains on fire[1].

Any unmatched core properties will be achieved by a task. Unmatched context properties will not. This is why only the context is used to calculate how well a goal has matched (or partially matched) to a state, while the core determines which properties and relationships have to be added or deleted to achieve the goal. How well the context properties and relationships match determines how good a goal-state mapping is. Whether or not the core properties and relationships match determines whether or not the goal is achieved in a given goal-state mapping.

4.3.2 HOPPER constrains unachievable goal elements to be true in the state

Some of the core elements of a goal (the *must* and *must not* properties and relationships) are sometimes unachievable by any of the agent's decomposition rules. This is because none of the head tasks of the decomposition rules specify the corresponding additions to add the *must* elements or the corresponding deletions to delete the *must not* elements. This means that if such a goal were matched to a state in such a way that these unachievable core elements were not already satisfied, then the goal itself would be unachievable because there would be no decomposition rule that could

[1]Of course, if you set up your insurance appropriately, you can achieve both goals simultaneously.

make them true or make them false as required.

For example, a goal to get a truck to a certain location may have as part of its core that the truck node *must* be related to the location node by the atLocation relationship and the truck node *must* have the type truck. If the agent knew of no way of changing the type of an object and the truck node were matched with a state object that was not a truck, then this goal would not be achievable because the agent would have no way of making it become a truck. The purpose of this method is to prevent ridiculous goal-state matchings (for example, matching a red package to the truck object of a goal just because the colour of the truck had always been red, and depending on the decomposition rules to also change the type of the package object to "truck").

Usually when matching a goal against a state, HOPPER only considers how well the goal's context matches with the state. However, this assumes that any part of the goal's core that does not match will be achieved by a task. This assumption does not hold for unachievable core goal elements, and so when matching a goal against a state HOPPER constrains the unachievable core goal elements to already be true or false in the state as required.

In the example above, HOPPER would constrain the truck node to have to match with a state object whose type was truck. Similarly, if the goal had any unachievable *must not* properties or relationships, then it would constrain the goal to match against the state so that these elements were already not true in the state.

The way that HOPPER determines whether a core goal element is unachievable or not is by keeping a set of all of the property and relationship additions and deletions specified by all of the head tasks of its decomposition rules. If HOPPER is given any new decomposition rules (*e.g.* new rules learned by TADPOLE), then it adds their additions and deletions to the appropriate sets. A *must* property or relationship is unachievable if it is not in the set of property additions or relationship additions respectively,

and a *must not* property or relationship is unachievable if it is not in the set of property deletions or relationship deletions respectively.

Note that this method does not guarantee that a goal-state difference will always be achievable. There may be separate rules that achieve each of the state changes required to satisfy the goal, but no single rule that can achieve all of them. In such cases, HOPPER would fail to find an applicable rule, it would propagate the failure up the hierarchy, and redecompose the parent goal as normal. As long as the required state changes are not ridiculous, it is better for HOPPER to fail and try an alternative decomposition rule than to use a poor goal-state matching and end up achieving an irrelevant goal just because it can.

4.3.3 HOPPER constructs a goal-state difference from the best goal-state mapping

Once HOPPER has found the best goal-state mapping, it constructs a goal-state difference graph where the properties and relationships are classified as: *keep true* (the *must* properties and relationships that are present in the current state), *make true* (the *must* properties and relationships that are not present in the current state), *keep false* (the *must not* properties and relationships that are not present in the current state), *make false* (the *must not* properties and relationships that are present in the current state), and *context* (all the other properties and relationships)[2]. Together, the *make true*, *keep true*, *make false*, and *keep false* properties and relationships constitute the core of the goal-state difference.

For example, if the goal in Figure 4.5 were mapped to the state depicted in Figure 4.6 by the mapping:

[2]For efficiency reasons, HOPPER does not actually construct an explicit goal-state difference graph, instead it matches tasks to states and goals directly. However, the effect is the same so for clarity I describe this process as if an explicit goal-state difference graph were generated.

truck2([Type: truck, vehicle], [Colour: red], ...)
truck2 → (atLocation) → loc2

loc1([Type: location, warehouse])
loc1 → (hasObject) → pkg1

pkg2([Type, package], [Colour: brown], ...)
pkg2 → (atLocation) → loc1

loc2([Type: location, postOffice])
...

Figure 4.6: State description

truck → truck2
loc → loc1
pkg → pkg2

then the *must* relationship atLocation between truck2 and loc1 (and the inverse hasObject relationship between loc1 and truck2) would not be satisfied in the state, and the goal-state difference would be the one depicted in Figure 4.7.

To achieve the goal, HOPPER would have to find a task decomposition rule that would add the appropriate atLocation and hasObject relationships and maintain the type of truck2 as truck and vehicle and the type of loc1 as location.

4.3.4 HOPPER uses the best task that matches the goal-state difference

After constructing a goal-state difference for goals not satisfied in the current state, updateHierarchy calls the decompose function to search through HOPPER's decomposition rules to find one that best matches the goal-

truck2([Type: truck, vehicle], [Colour: red], ...) *keep true*
truck2 → (atLocation) → loc1 *make true*
truck2 → (atLocation) → loc2
...
loc1([Type: location, warehouse]) *keep true*
loc1 → (hasObject) → pkg2
loc1 → (hasObject) → truck2 *make true*
...
pkg2([Type, package], [Colour: red], ...)
pkg2 → (atLocation) → loc1
...
loc2([Type: location, postOffice])
...

Figure 4.7: Goal-state difference for getting a truck to the location of a package

state difference and therefore achieves the goal.

The changes that will occur if a decomposition rule is executed are specified by its head task, a graph whose nodes consist of objects with properties and whose links consist of relationships between them. Matching a task against a state-difference involves searching for the best mapping of task nodes to goal-state difference nodes.

decompose searches for the best task to goal-state difference mapping by running a beam search where at each step an unmapped task node is mapped to an unmapped goal-state difference node until all of the task nodes are mapped. Whether the mapping (or partial mapping) is successful depends on how well the properties of the task objects and their relationships match with the corresponding goal-state difference objects.

The properties and relationships in the task graph are classified as either: *precondition*, *additions* or *deletions*. The *additions* and *deletions* specify what properties and relationships will be added and deleted respectively if the decomposition rule is achieved. The properties and relationships in

the *precondition* specify what should be true in the state when the decomposition is executed.

A task to goal-state difference mapping is viable if the task additions and deletions form a superset of the *make true* and the *make false* properties and relationships respectively. The task may have extra additions or deletions that do not match the *make true* or *make false* elements of the goal-state difference. The extra additions have to match with nodes or relationships where the property or relationship to be added is not already present. The extra deletions have to match with nodes or relationships where the property or relationship to be deleted is present. However, these extra additions or deletions must not undo any aspects of the goal that are already satisfied: they must not delete any *keep true* properties or relationships that are already satisfied and they must not add any *keep false* properties or relationships that are already not present.

The remaining *precondition* properties and relationships determine how good the match is. The properties and relationships in the *precondition* of the task are those that were present when the task was successfully achieved by HOPPER or successfully parsed by TADPOLE. A property or relationship may be a crucial part of the *precondition* or it may be irrelevant. HOPPER gauges the importance and relevance of a property or relationship by how many times it has seen it before, and so matchings are scored based on how well the *precondition* matches the goal-state difference, weighted by the relative frequencies of the properties and relationships.

To find the best matching task, decompose first searches through HOPPER's known decomposition rules to find all the applicable rules: all the the rules whose head task to goal-state difference mapping is viable. This corresponds to the set of rules for achieving the goal in question. From this set, decompose then selects the rule whose head task to goal-state difference mapping has the highest score as determined by the matching of the task's *precondition* properties and relationships. This corresponds to

Total: 7
truck([Type: truck(7), vehicle(7)], [Colour: red(5), green(2)], ...)
truck → (atLocation) → destination *add*
truck → (atLocation) → start *delete*

start([Type: location(7), postOffice(5), warehouse(2)])
start → (connectedTo(7)) → destination
start → (hasObject) → truck *delete*

destination([Type: location(7), airport(6), warehouse(1)])
destination → (connectedTo(7)) → start
destination → (hasObject) → truck *add*

Figure 4.8: Task for moving a truck

the rule that is most applicable in the current state.

Note that dividing the task to goal-state difference mapping into two phases, matching the additions and deletions first and only then the precondition, greatly reduces the cost of the mapping. This is because the additions and deletions must satisfy the stringent applicability constraints given above. This greatly reduces the number of candidate nodes they can match with and it can lead to early failure. For example, if there is a *make true* property in the goal-state difference and there is no corresponding *addition* property anywhere in the task, then the decomposition rule cannot be used to achieve the goal and there is no need to match any other node.

Figure 4.8 shows an example of a head task for the decomposition of moving a truck from one location to another. A viable mapping of the task to the goal-state difference in Figure 4.7 would be:

truck → truck2
start → loc2
destination → loc1

As soon as decompose successfully finds a decomposition rule whose task matches the desired goal-state difference, udpateHierarchy applies the corresponding decomposition. If the decomposition is atomic then udpateHierarchy associates the appropriate atomic action with the parent goal, otherwise it associates the partial order of sub-goals specified by the decomposition rule with the parent goal.

4.3.5 Decomposition variables restrict sub-goal matchings

The head task and the sub-goals of a decomposition rule often refer to the same objects, and it is critically important that the state objects the head task matched with are the same objects that the sub-goals match with.

Figure 4.9 shows a simplified decomposition rule (not showing any irrelevant parts of the context) that specifies how to deliver a package: get a truck to the location where the package is, get the package into the truck, get the truck to the destination location, and then get the package out of the truck. It is important that the package, truck, and location objects referred to by the head task and sub-goals are constrained to match to the same state object when the decomposition is being achieved. It is no good trying to deliver a package by loading it into one truck and then driving another very similar truck to the destination location.

To ensure that the objects in the head task and in the sub-goals of a decomposition rule match with the same state objects, each decomposition rule can have decomposition variables that refer to objects in the head task and in the sub-goals. If after a matching (task with goal-state difference or goal with current state) one of the objects referred to by a decomposition variable is bound to a particular state object, then all subsequent matchings involving that decomposition variable are constrained to match with the bound state object.

For example, the decomposition rule to deliver a package has four decomposition variables referring to: the package, the initial location of the

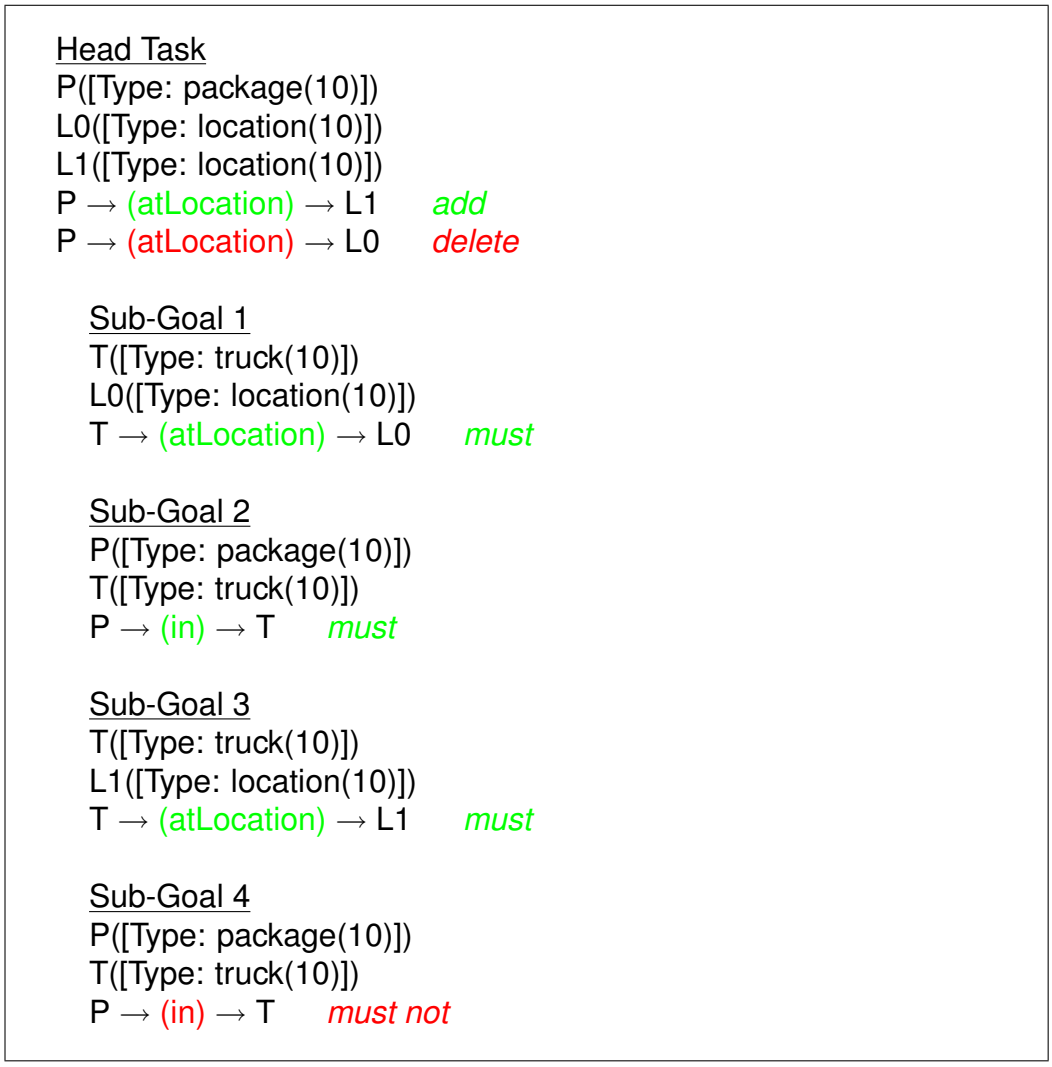


Figure 4.9: Decomposition rule for delivering a package

package, the destination of the package, and the truck used to deliver the package. When the head task is matched, the package and both location variables will be bound to particular state objects. Any future sub-goal state matchings that refer to the package or either location will be constrained to match to the same state objects. Similarly when the first sub-goal is matched, the truck variable will be bound to a specific state object, and in future sub-goal state matchings the truck variable will be constrained to match to the same state object.

In practice, most of the nodes in sub-goal graphs are associated with decomposition variables. This means that when goals are matched with the current state most if not all of their nodes will often be bound to particular state objects. This greatly reduces the size of the graph to be matched and makes matching goals with states a much more tractable problem.

4.4 Unexpected Events

As described in Chapter 1, one of the major challenges in the Human Planning Domain is its nondeterminism, and more generally its unpredictability. In a nondeterministic domain, future states cannot be determined with certainty. In an unpredictable domain, not only can future states not be determined with certainty, but also the *possible* future states cannot be determined. In other words, in an unpredictable domain there is always the possibility of a completely unexpected event occurring at any time. This can be because of incomplete or incorrect knowledge about the current state of the world, incomplete or incorrect knowledge about how the world works, or the actions of other agents.

An unexpected event can disrupt the decomposition hierarchy at any level in a number of ways: it can undo a previously achieved goal, achieve a previously unachieved goal, invalidate the precondition of a decomposition, and satisfy a previously unsatisfied precondition of an alternative. The unexpected events can be disruptive to the agent's plan, or they can

present opportunities to achieve the plan faster. Most unexpected events are handled automatically by HOPPER's least-commitment decomposition strategy (covered in Section 4.5); those that are not must be handled directly by HOPPER.

When responding to an unexpected event there are a number of possible ways of updating the decomposition hierarchy, ranging from ignoring the event completely and keeping the decomposition hierarchy unchanged, to redecomposing completely at every step. Both of these extremes are undesirable.

4.4.1 Ignoring unexpected events leads to plan failure

Ignoring unexpected events is problematic in non-deterministic domains and especially in ones that have un-undoable actions. It is important that plan failure is detected and handled as early as possible, otherwise the agent can end up producing and executing long futile plans. Furthermore, the agent would be unable to take advantage of any unexpected opportunities that arose that could make the plan shorter.

For example, if the agent were planning a trip to another country and it did not react to the information that its flight had been cancelled, then it would end up going to the airport only to find when it go there that there was no plane to board.

4.4.2 Redecomposing at every time step is too expensive

On the other extreme, unexpected events can be handled by dropping the entire goal decomposition hierarchy and rebuilding it at every time step. This is similar to Icarus' decomposition strategy described in Chapter 2. However, there are two main problems with this approach.

The first problem is that there is no guarantee that the decomposition hierarchy generated in the next time step will be the same as the one in the previous time step. There may be two or more decompositions for achiev-

ing the same sub-goal that are equally applicable in the current state. If the decomposition hierarchy is dropped and rebuilt at every time step, then it is possible that alternate decompositions will be used in different time steps. Alternate decompositions will often disrupt each other because they are selected to achieve the same goal and so deal with the same objects in the state. The approach of constantly rebuilding the decomposition hierarchy can lead to the agent trying one decomposition to achieve a goal or sub-goal, partially achieving it, then switching to an alternate decomposition, partially achieving that decomposition, and in the process interfering and undoing the other decomposition. The agent can end up switching back and forth between alternate decompositions while making no progress towards achieving the goal. This problem applies not only to which decomposition of a number of viable alternatives is used to achieve a particular goal, but also which goal of a number of co-ordered goals to achieve first.

For example, if the goal were to deliver two packages to two different destinations, then there would be two co-ordered sub-goals: delivering one package and delivering the other one. It does not matter in what order these two sub-goals are achieved, but one sub-goal should be achieved completely before the other. If the decomposition hierarchy were dropped and rebuilt at every time step, then this could lead to the agent driving halfway to one destination, switching to the other co-ordered sub-goal, driving partway to the other destination, switching to the other sub-goal, and so on, resulting in the agent not making any progress towards either goal.

The second problem is that redecomposing the entire hierarchy at every time step is prohibitively expensive. Although this approach is viable for relatively small hierarchies, coarse atomic actions, and a small number of known rules, it is clear that it will not scale to larger, more complex domains. If this approach is to be scaled to the HPD, then it will have to deal with deep hierarchies (used to solve complex tasks), a very large rule

set (representing the agent's knowledge about how to solve a wide variety of different tasks), and fine-grained atomic actions (the actions have to be fine-grained enough that the agent can be reasonably confident of executing them successfully without unexpected interruptions). The larger the decomposition hierarchy and the larger the agent's rule set, the more expensive it is to construct the goal decomposition hierarchy. With fine-grained atomic actions, the entire goal decomposition hierarchy would have to be rebuilt within very short intervals, which, given the large cost involved, would be problematic.

A further difficulty with dropping and rebuilding the decomposition hierarchy is that it precludes any reasoning about future sub-goals (covered in Sections 4.7 and 4.9), because this reasoning depends on the relative stability of the decomposition hierarchy.

4.4.3 HOPPER only redecomposes after unexpected events

Rather than redecomposing at every time step, HOPPER instead assumes that its decomposition rules are generally correct and that redecomposition is warranted only when an unusual and unexpected event occurs.

To detect unexpected events, HOPPER makes predictions about future states. When cycle finishes updating the decomposition hierarchy and selecting the next action to return for the current time step, it makes a prediction about what the subsequent state will be.

Because every atomic action executed by HOPPER is associated with an atomic decomposition rule used to achieve an atomic goal, cycle can make use of the atomic decomposition rule's head task to make a prediction about what the result of executing the action will be. It predicts that the subsequent state will differ from the current state by the additions and deletions specified by the head task of the atomic decomposition rule. If the subsequent state differs in any way from the predicted state, then cycle treats this as an unexpected event and it redecomposes the decomposition

hierarchy accordingly by calling updateHierarchyUnexpected (A.1.5).

Clearly this prediction mechanism is limited, and it cannot account for changes to the state not caused directly by the agent. HOPPER's prediction mechanism can be extended to arbitrarily sophisticated physics-learning and state-predicting systems. However, it is important to note that being able to accurately predict a state change is not enough to guarantee that it is non-disruptive. It is only safe to assume that the changes directly specified by the decomposition are non-disruptive.

4.4.4 HOPPER only redecomposes affected goals

Rather than dropping and then rebuilding the entire goal decomposition hierarchy, updateHierarchyUnexpected only redecomposes the goals directly affected by the unexpected event that prompted the redecomposition.

When an unexpected event occurs, cycle first determines which state objects changed unexpectedly. It labels a state object as changed if any of its properties or relationships have been added or deleted unexpectedly (the changes were not described by the head task of the last atomic decomposition). updateHierarchyUnexpected then goes down the left hand side of the decomposition hierarchy and verifies that the decompositions used to decompose the unconstrained goals are still valid.

In keeping with HOPPER's conservative decomposition strategy, updateHierarchyUnexpected tries to maintain the current decomposition hierarchy and it redecomposes goals only when necessary. This means that updateHierarchyUnexpected will not redecompose the current decomposition if it was not directly affected by the unexpected event even if the unexpected event made a new decomposition possible and even if the new decomposition would match better than the current decomposition.

If the head task of a decomposition has any objects that were affected by the unexpected event (the state objects the task objects are matched

with were affected), then updateHierarchyUnexpected drops the entire sub-hierarchy below it. When cycle subsequently calls updateHierarchy, the goal-state difference will be recalculated and the goal will be redecomposed with a new decomposition rule (note that this can be the same decomposition rule that decomposed it previously).

For example, the decomposition for delivering a package P1 from L1 to L2 is to get a truck to L1, get P1 into the truck, get the truck to L2, and get P1 out of the truck. If upon arriving at L1 the agent noticed that P1 was not in fact at location L1 but at location L3, then HOPPER would register this as an unexpected event (the package was not where it was expected to be). The unexpectedly modified state objects would be P1, L1, and L3, and HOPPER would rematch any decomposition whose task objects were matched to these state objects. The matching of the head task of the decomposition used to achieve the deliver package goal would change (the initial location object would now match with L3 rather than L1), and so HOPPER would drop the sub-hierarchy of the deliver package goal and rebuild it using the new task matching. The new decomposition would now have as the first sub-goal getting a truck to L3. The result of this would be that the agent would now drive to location L3 rather than trying to load the non-existent package into the truck at location L1.

The way HOPPER handles opportunities is described in Section 4.7.

4.5 Decomposition by Least Commitment

When achieving goals, HOPPER decomposes only the unconstrained goals in the decomposition hierarchy and leaves future goals undecomposed. There are often multiple different decomposition rules that will achieve a given goal, but HOPPER does not commit to a particular decomposition rule until it is necessary. HOPPER waits until the goal becomes unconstrained and it is time to decompose it before selecting a decomposition rule. It can then make a more informed decision and pick the decomposi-

tion rule most appropriate to the current state.

This least-commitment goal-decomposition strategy is particularly well suited to an unpredictable domain. By not committing to particular decomposition rules for future goals until necessary, it can automatically handle many of the unexpected opportunities and disruptions that would affect the plan.

4.5.1 The higher a goal is in the hierarchy the more abstract it is

The main goal of the goal decomposition hierarchy makes no constraints on what state it should be achieved in. Whatever the initial state is when HOPPER begins achieving the goal, HOPPER will find the decomposition rule whose precondition best matches that state in order to decompose the goal. However, the sub-goals will tend to reflect the precondition constraint of their parent decomposition rule. In general, it will only be appropriate to achieve them in a state where the precondition holds. The sub-sub-goal will inherit the state constraints of the sub-goals as well as their own precondition constraints from their own parent decompositions and so on down the hierarchy. In this way, the lower a goal is in the decomposition hierarchy the more state constraints it inherits from its parent decomposition, the narrower the range of states in which it is appropriate to achieve the goal, and the more concrete it is.

4.5.2 The leaves of the decomposition hierarchy correspond to a plan of increasing abstractness

The way to achieve a goal in the decomposition hierarchy is to achieve its sub-goals in the specified order. Because this applies to all of the decomposed goals in the decomposition hierarchy, the way to achieve a goal in the decomposition hierarchy is to achieve its leaf goals in order. An in-

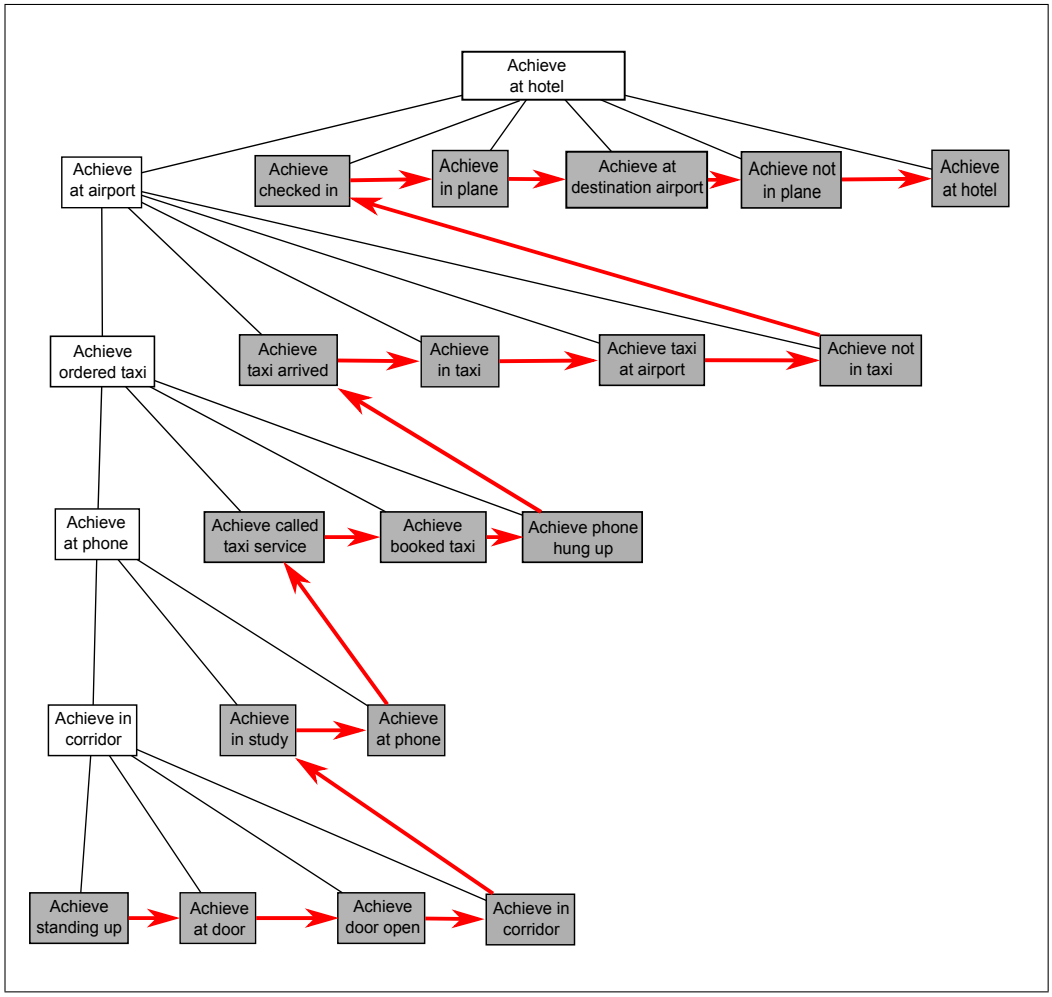


Figure 4.10: Decomposition hierarchy for flying to another country

order traversal of a decomposition hierarchy's leaf nodes corresponds to a plan where each step in the plan is a goal that needs to be achieved.

Because HOPPER decomposes only the unconstrained and therefore earliest goals and sub-goals, it produces a left-heavy decomposition hierarchy, and an in-order traversal of the leaf nodes results in a sequence of sub-goals where the initial sub-goals are at the lowest level of the decomposition hierarchy and subsequent sub-goals are at higher (or equal) levels of the hierarchy. Because goals lower in the decomposition hierarchy are

more concrete and higher-level goals are more abstract, the hierarchy's leaf nodes correspond to a plan that begins with the lowest level atomic goals and continues with increasingly higher-level and more abstract goals to achieve. As it executes its plan, HOPPER receives updated information about the state of the world which it uses to gradually fill in the details.

For example, Figure 4.10 shows the decomposition hierarchy for flying to another country. An in-order traversal of the leaf nodes gives the sequence of goals that the agent needs to achieve in order to achieve the main goal. The initial goals of this plan are very low-level and very strongly constrain the state in which it is appropriate to achieve them. Standing up, getting to the door, opening the door, and getting to the corridor are only appropriate goals to achieve if the agent is sitting down in a room connected by a corridor to a study that contains a phone. On the other hand, the final goals in the plan are high-level and abstract, and make relatively few constraints on the state in which they are achieved. The goals of disembarking from the plane and then getting to the hotel do not constrain the state to any particular layout of the airport or even any means of transportation (*e.g.* taxi, bus, walking) of getting to the hotel. HOPPER would not commit to a specific means of getting to the hotel until it had arrived at the destination airport and it could make a more informed decision based on the most up-to-date state information.

4.5.3 A plan of increasing abstractness is well suited to an unpredictable domain

In an unpredictable domain, an unexpected event can occur at any time. Planning requires the agent to predict future states given various actions that the agent could take. However, the more distant a state is in the future, the more likely that an unexpected event will occur between the current state and that future state, possibly invalidating any prediction made. One of the main problems with classical planning algorithms applied to

an unpredictable domains (described in Chapter 2) is that they depend on reliable state prediction from the initial state to the final state where the goal is satisfied.

The advantage of a plan of increasing abstractness like the one generated by HOPPER is that it makes fewer and fewer constraints on predicted future states which corresponds well with their increasing uncertainty in an unpredictable domain. A high-level decomposition rule that has been learned and applied in a wide variety of different states will have a generalized precondition. The constraints it makes on the state will be unlikely to be undone by unexpected events. Most of the unexpected events, whether they are opportunities to facilitate the plan or disruptions to impede it, will occur below the level of the undecomposed goal, and so they will be handled automatically when HOPPER selects the most appropriate decomposition rule to the then current state.

For example, in the example of flying to another country, if the agent were currently checking in with the airline, then HOPPER would ignore any unexpected events that have implications for getting from the destination airport to the hotel until the agent actually arrived at its destination and assessed the situation. If there were unexpectedly no taxis available, then because HOPPER had not actually planned out how to get to the hotel from the airport, it would not have to modify its plan at all. Instead, it would use its knowledge about the current situation to select the appropriate rule (*e.g.* taking the bus). Similarly, if an unexpected opportunity arose, like a free shuttle service being available, then HOPPER could make use of this opportunity immediately when it refined its plan for getting to the hotel.

HOPPER's least-commitment strategy decomposes a sub-goal only when it is actually time to achieve it and it has as much relevant information as possible.

4.5.4 Least-commitment goal decomposition is only justified if future goals are achievable

HOPPER's least-commitment goal-decomposition strategy has clear advantages in an unpredictable domain like the HPD. However, not decomposing a future goal until it is necessary is only justified if HOPPER can be confident of having at least one decomposition rule for achieving it when it comes time do so.

HOPPER can be confident of being able to achieve a future goal if the problem it is solving is routine. A routine problem is one that it has achieved before, possibly several times. This means that HOPPER should have at least one decomposition rule for every goal and sub-goal it will need to decompose, and possibly multiple rules with different preconditions appropriate for different states. When solving a routine problem, HOPPER is unlikely to face future goals or sub-goals for which it will not have applicable decomposition rules. The more routine the problem is the less likely that HOPPER will be unable to achieve the corresponding future goals.

4.6 Clean-up sub-goals

Many decomposition rules have "clean-up" sub-goals at the end of their decomposition whose only purpose is to minimize the superfluous side-effects of their decomposition rule. Clean-up sub-goals are important to make decomposition rules more modular: using one decomposition as opposed to another to achieve a goal in the decomposition hierarchy makes no difference (or as little difference as possible) to the rest of the hierarchy.

4.6.1 Many decomposition rules achieve their head task part-way through their decomposition

Decomposition rules often achieve a superset of the effects specified by their head task part-way through their decomposition. The remaining sub-goals of their decomposition are clean-up sub-goals that serve to undo any significant side-effects, so that after all the clean-up sub-goals have been achieved, the effects of the decomposition are limited as much as possible to those specified by their head task.

For example, the (simplified) decomposition rule to deliver a package could be:

Task: -atLocation(P, L0), +atLocation(P, L1)
  Goal1: atLocation(T, L0)
  Goal2: in(P, T)
  Goal3: atLocation(T, L1)
  Goal4 (clean-up): ¬in(P, T)

The task of changing the location of the package is achieved as soon as the truck reaches its destination, before it is unloaded. The final sub-goal in the decomposition ensures that the final effect of executing the decomposition will not include the package being in the truck.

Clean-up sub-goals occur at all levels of the decomposition hierarchy. A low-level decomposition for moving a cup to a table would involve grasping the cup, lifting it, placing it on the table, and then releasing it. This decomposition would be achieved as soon as the cup is placed on the table but before it is released by the agent's hand. The final sub-goal ensures that the only effect of this decomposition is that the cup's location changes and not its relationship to the agent's hand.

High level decompositions also have clean-up sub-goals. For example,

the decomposition of having dinner could be to acquire the necessary ingredients, set the table, cook the meal, eat the food, and clean the plates and cutlery. Again, the task of having dinner is accomplished as soon as the food has been eaten, but before the plates have been washed. The sub-goals for cleaning the plates and cutlery minimize the side-effects of the decomposition for making dinner. Clean-up sub-goals in general derive their name from these specific examples of sub-goals for cleaning up after making dinner.

4.6.2 Clean-up sub-goals make decomposition rules modular

The reason that it is important that clean-up sub-goals are achieved is that they minimize the side-effects of their decompositions. Minimizing side-effects minimizes the interactions between different decompositions and keeps them as modular as possible.

If a decomposition does not undo its side-effects, then those side-effects may interfere with future decompositions. For example, not cleaning the dishes after a meal will interfere with the making of future meals. Uncleaned up side-effects may also make it easier for the current goal to be interfered with by future decompositions. For example, when delivering a package, if the package is not unloaded from the truck that delivered it, then any movement of the truck by future decompositions will undo the original goal. This is particularly important for HOPPER because it does not keep track of goals that it has already achieved.

If side-effects are not undone, then every decomposition rule that could be affected would need to account for the side-effects in their sub-goals and preconditions. For example, every decomposition rule that made use of a truck would have to consider the case of the truck already being loaded with another package, and every such rule would need to include as its first sub-goal the truck being empty. While it is possible to carefully

handcraft such rules, learning such rules automatically is problematic because the information for dealing with the side-effect has to be re-learned for every affected rule. Rather than redundantly duplicating the information for handling side-effects at the beginning of many decomposition rules, it is more economical to encode this information at the end of the rule that creates them.

Minimizing the side-effects also makes decomposition rules more predictable. High-level tasks may be decomposed into a large number of different decomposition hierarchies depending on the specific state in which each sub-goal is decomposed (different states will satisfy different preconditions for alternate decomposition rules for achieving the same sub-goal). Nevertheless, if the side-effects of the decomposition rules are minimized, then, despite the variety of possible decomposition hierarchies, the effects of achieving the high-level task will be limited (as much as possible) to those specified by the task itself. Minimizing the side-effects of decompositions means HOPPER can use alternative decompositions to achieve a particular goal without affecting the rest of the hierarchy.

The predictability of decomposition rules allows HOPPER to reason about the effects of achieving future sub-goals because the effects decomposition rules used to achieve it will be minimized to achieving the goal and nothing else. This is important for interleaving the execution of different decompositions (described in Section 4.7), and also for high-level planning with decomposition rules (discussed in Section 4.9).

4.6.3 HOPPER ensures that goals with clean-up sub-goals are not removed prematurely

A consequence of HOPPER's reactive decomposition updating algorithm is that a goal is removed from the decomposition hierarchy as soon as it is satisfied. This is so that HOPPER can notice and take advantage of goals that are serendipitously achieved by unexpected events. However,

it is important for HOPPER to distinguish between goals that have been serendipitously achieved, and goals that have not been "cleaned up" yet. Because clean-up sub-goals undo superfluous side-effects after the head task has been achieved, they always come after their parent goal has been satisfied. Note that it is not enough to distinguish uncleaned-up goals from those serendipitously achieved by noticing unexpected events, because the unexpected events may be irrelevant to the goal being considered.

To make sure that a decomposition's "clean-up" sub-goals are achieved, updateHierarchy does not remove a goal from the decomposition hierarchy if it has any "clean-up" sub-goals. If a goal is satisfied, then HOPPER removes all of its non-clean-up sub-goals. Only when the satisfied goal has no "clean-up" sub-goals left does updateHierarchy remove it from the decomposition hierarchy.

4.6.4 HOPPER achieves sub-goals only when necessary

The only purpose of clean-up sub-goals is to reduce the side-effects of their decomposition, otherwise they are not critical to achieving the parent goal. Occasionally, clean-up sub-goals are achieved and then immediately undone by the initial goals of the next decomposition. For example, a high-level description of a decomposition to put sugar into a cup of tea could be:

Task: +cup of tea is sweet
- Goal1: hand grasping spoon
- Goal2: spoon over sugar container
- Goal3: spoon contains sugar
- Goal4: spoon over cup
- Goal5: sugar in cup
- Goal6: spoon in sink, spoon not in cup (clean-up)
- Goal7: hand not grasping spoon (clean-up)

and a high-level description of a decomposition to stir a cup of tea until the sugar is dissolved could be:

Task: -tea has visible sugar in it
  Goal1: hand grasping spoon
  Goal2: spoon in cup
  Goal3: sugar dissolved
  Goal4: spoon not in cup
  Goal5: spoon in sink (clean-up)
  Goal6: hand not grasping spoon (clean-up)

Note that the first two goals of the decomposition to dissolve sugar will undo the two clean-up sub-goals of the decomposition to put sugar into the tea in the reverse order that they were achieved. Clearly, it is wasteful to achieve clean-up goals that will be immediately undone; there is no point putting the spoon down only to pick it up again.

HOPPER addresses this issue by not achieving clean-up sub-goals immediately. If a goal in the decomposition hierarchy has only clean-up sub-goals remaining to be achieved, then updateHierarchy searches through the subsequent sub-goals in the decomposition hierarchy (sub-goals that would become unconstrained if this sub-goal were removed) to find any that are satisfied in the current state but would be undone if any of the pending clean-up sub-goals were achieved. If a pending clean-up sub-goal would clobber one of these already satisfied sub-goals, then achieving it would be counter-productive, and updateHierarchy removes it from the decomposition hierarchy. After the decomposition hierarchy has been updated, HOPPER then achieves the remaining pending clean-up goals in the order specified by their decomposition.

4.7 Interleaving Decompositions

Decomposition rules that undo their side-effects with clean-up sub-goals are modular. Because they achieve only the effects specified by their head task, HOPPER can use alternative decomposition rules to achieve a particular sub-goal without affecting the rest of the decomposition hierarchy. However, the same modularity which minimizes how much different decompositions interfere with each other also limits how much they can facilitate each other.

Naively achieving one goal after another in the decomposition hierarchy will tend to result in sub-optimal behaviour. To remedy this problem, HOPPER decomposes co-ordered goals (goals that are part of the same decomposition where neither is constrained to come before the other) in parallel, and searches for a way of interleaving the execution of both goals in a way that results in a shorter plan.

4.7.1 Naively achieving goals in order results in sub-optimal plans

Unlike classical planners, HOPPER does not search the space of possible sequences of atomic actions that will lead to a desired state, so it can make no guarantees about the optimality of the number of atomic actions it will end up performing. This is especially true because HOPPER begins executing atomic actions before its plan is completely specified.

Although HOPPER cannot guarantee or even search for optimal plans, there are ways of improving the performance of goal decomposition planning within the framework of the goal decomposition hierarchy. This depends on finding and exploiting ways of achieving multiple sub-goals at the same time.

For example both the plans for doing the shopping and dropping off a letter in a mailbox require getting in the car, driving to the required lo-

cation, and then driving back home. If the two instances of driving could be combined and executed at the same time, then the plan could be made much shorter and more efficient (e.g. dropping off the letter in the mailbox on the way to the shop).

Naively achieving one of these goals completely and then achieving the other results in an inefficient plan (*e.g.* driving to the shop, doing the shopping, driving back home, driving to the mailbox, dropping off the letter, and then driving home again).

4.7.2 Identical sub-goals achieved at the same time improve plan efficiency

The plans to achieve different goals often have identical sub-goals lower in the decomposition hierarchy. If the execution of these plans is carefully interleaved in such a way that the identical sub-goals are achieved at the same time, then the identical sub-goal needs only to be achieved once to be satisfied in all of the separate plans. Not having to redundantly re-achieve the same sub-goal in other decompositions will reduce the length of the agent's plan to a greater or lesser extent depending on how high the sub-goal is in the decomposition hierarchy. The higher a sub-goal is in the hierarchy, the deeper its sub-hierarchy, the more atomic actions are needed to achieve it, and the more atomic actions will be saved if it does not need to be achieved.

4.7.3 HOPPER tries to interleave the sub-hierarchies of co-ordered goals

At each time step, cycle calls the updateInterleavings function (A.1.11) to maintain the interleavings it has found in previous time steps. updateInterleavings removes any interleavings that were affected by any unexpected events, and if no interleavings remain, then it calls the interleave-

Hierarchy function (A.1.12) to search for a way of interleaving the sub-decomposition hierarchies of co-ordered goals.

The interleavings generated by interleaveHierarchy do not affect the structure of the decomposition hierarchy. Rather, they are a separate index into the structure, indicating to chooseAtomicAction and chooseAction-FromCandidates which sub-goals in the hierarchy should be achieved and in what order. An interleaving has the same structure that a decomposition hierarchy does, and it can be used in place of the decomposition hierarchy when chooseAtomicAction and chooseActionFromCandidates select the appropriate atomic action to execute.

interleaveHierarchy interleaves the sub-decomposition hierarchies of co-ordered goals in such a way that any identical sub-goals they share are achieved at the same time (co-ordered goals are goals in the decomposition hierarchy that belong to the same decomposition and there are no ordering constraints between them: it does not matter which of the goals is achieved first).

It only makes sense to interleave the plans of goals in the decomposition hierarchy if there are no ordering constraints between them. Two goals cannot be achieved simultaneously if one is constrained to be achieved before the other. For example, an international trip could involve taking a taxi to the airport, flying to the destination, and then taking a taxi to the hotel. Despite the fact that the two taxi rides share many identical sub-goals (ordering a taxi, getting in and out of the taxi, paying the driver, etc.), they cannot be interleaved because they are strictly ordered to occur one after the other.

There are two ways in which two (or more) goals in a decomposition hierarchy can have no ordering constraints between them:

- The goals are top-level goals that the agent needs to achieve. For example, the agent is tasked with delivering a number of packages to various different locations. Because these top-level goals are not within the same decomposition and have no parent goals, HOPPER

assumes they have no ordering constraints between them, and that it does not matter what order they are achieved in.

- The goals are part of the same decomposition but have no ordering constraints between them. An example of such a decomposition is achieving relatively independent parts of a goal: making dinner involves making a salad, boiling some potatoes, and frying a steak; because these components are relatively independent of each other, it does not matter what order they are achieved in.

Any goals that are not constrained to come after any other goal can potentially be interleaved. This is one of the main reasons why HOPPER recursively decomposes all of the unconstrained goals in the decomposition hierarchy during the construction and updating phases of the algorithm: so that it can search through the sub-hierarchies for shared sub-goals.

4.7.4 HOPPER searches the sub-hierarchies of co-ordered goals for matching sub-goals

After interleaveHierarchy finds the candidate co-ordered sub-goals, it calls interleave (A.1.13) to find a way to interleave their sub-hierarchies into a single, consistent plan.

interleave searches the sub-hierarchies of co-ordered goals for matching sub-goals. Two sub-goals match if their respective goal nodes are mapped to state objects in such a way that their *must* and *must not* constraints on state objects and relationships are identical. The context of the sub-goals does not have to be identical for them to match, because the only purpose of a goal's context is to help match it against a state, and the sub-goals in question have already been matched to the current state.

There may be multiple groups of co-ordered goals at different levels of the decomposition hierarchy, and there may be multiple pairs of matching sub-goals in a pair of sub-hierarchies of co-ordered goals. Because

achieving higher-level sub-goals in parallel results in a greater efficiency gain, interleave searches the decomposition hierarchy with a breadth-first search. It examines the sub-hierarchies of co-ordered goals higher in the decomposition hierarchy before examining those lower in the hierarchy, and it searches the sub-hierarchies themselves with a breadth-first search, searching for matching sub-goals higher in the sub-hierarchies before those lower down. Note that the matching sub-goals need not be at the same depth in the decomposition hierarchy.

Figure 4.11 shows part of HOPPER's decomposition hierarchy if the agent had the goals of delivering two packages, one locally by truck and one to a more distant location by plane. These two goals are co-ordered; the agent could achieve them in either order, and so HOPPER would decompose both and search for matching sub-goals. Delivering a package locally requires picking it up and dropping it off with a truck. Delivering a package to a more distant location requires first delivering it to the airport and then flying it to the appropriate destination. In this case, both the packages are at the same initial location, and so both sub-hierarchies have the same initial sub-goal (and sub-sub-goal) of getting a truck to that location.

Once interleave has found two matching sub-goals in the sub-hierarchies of two co-ordered goals it calls interleaveDecomps to search for a way to interleave the execution of their two parent decompositions in such a way that the sub-goals are achieved at the same time, but making sure that both parent goals are still achieved.

4.7.5 HOPPER preserves the goal dependencies when interleaving decompositions

The sub-goals of a decomposition are partially ordered, and a sub-goal generally depends on all of the sub-goals that are ordered to come before it: the sub-goals that are ordered to come before the sub-goal in ques-

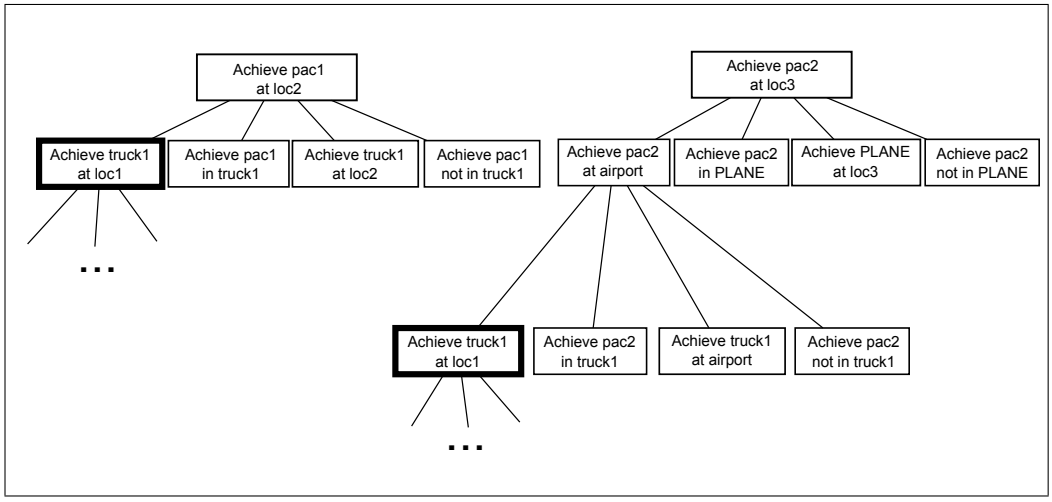


Figure 4.11: Two matching sub-goals at different depths in the hierarchy

tion should be satisfied before the sub-goal in question is achieved. However, some sub-goals are mutually incompatible with other sub-goals in the same decomposition; they cannot both be satisfied at the same time. Such sub-goals "clobber" each other; achieving one unachieves the other. interleaveDecomps assumes that a sub-goal depends on all of the sub-goals that are ordered to come before it except those that are clobbered by intervening sub-goals or the sub-goal itself.

For example, the (simplified) decomposition to deliver a package:

Task: -atLocation(P, L0), +atLocation(P, L1)
  Goal1: atLocation(T, L0)
  Goal2: in(P, T)
  Goal3: atLocation(T, L1)
  Goal4 (clean-up): ¬in(P, T)

is fully ordered (Goal1 → Goal2 → Goal3 → Goal4). Goal1 is the first sub-goal and so has no dependencies. Goal2 depends on Goal1. Goal3 comes after both Goal1 and Goal2, however it clobbers Goal1 (it is not

possible to achieve Goal1 and Goal3 at the same time because the same truck cannot be at two different locations at the same time), and so it depends only on Goal2. Goal4 is ordered to come after Goal1, Goal2, and Goal3, but because Goal3 clobbers Goal1, Goal4 depends only on Goal2 and Goal3.

When interleaving two decompositions interleaveDecompsfirst determines their respective goal dependencies, and then searches for a way to order all the sub-goals so that the matching sub-goals are achieved at the same time and all of the goal dependencies of both decompositions are preserved.

4.7.6 HOPPER fixes the matching sub-goals and then interleaves the remaining sub-goals

When interleaving two decompositions, interleaveDecomps first fixes the matching sub-goals to be achieved at the same time, and then searches for a valid ordering of the remaining sub-goals that preserves all of the goal dependencies of both decompositions. However, two decompositions may have more than one matching sub-goal in common. It may not be possible to interleave the decompositions so that all of the matching sub-goals are achieved in parallel, but it may be possible to achieve some of the sub-goals in parallel.

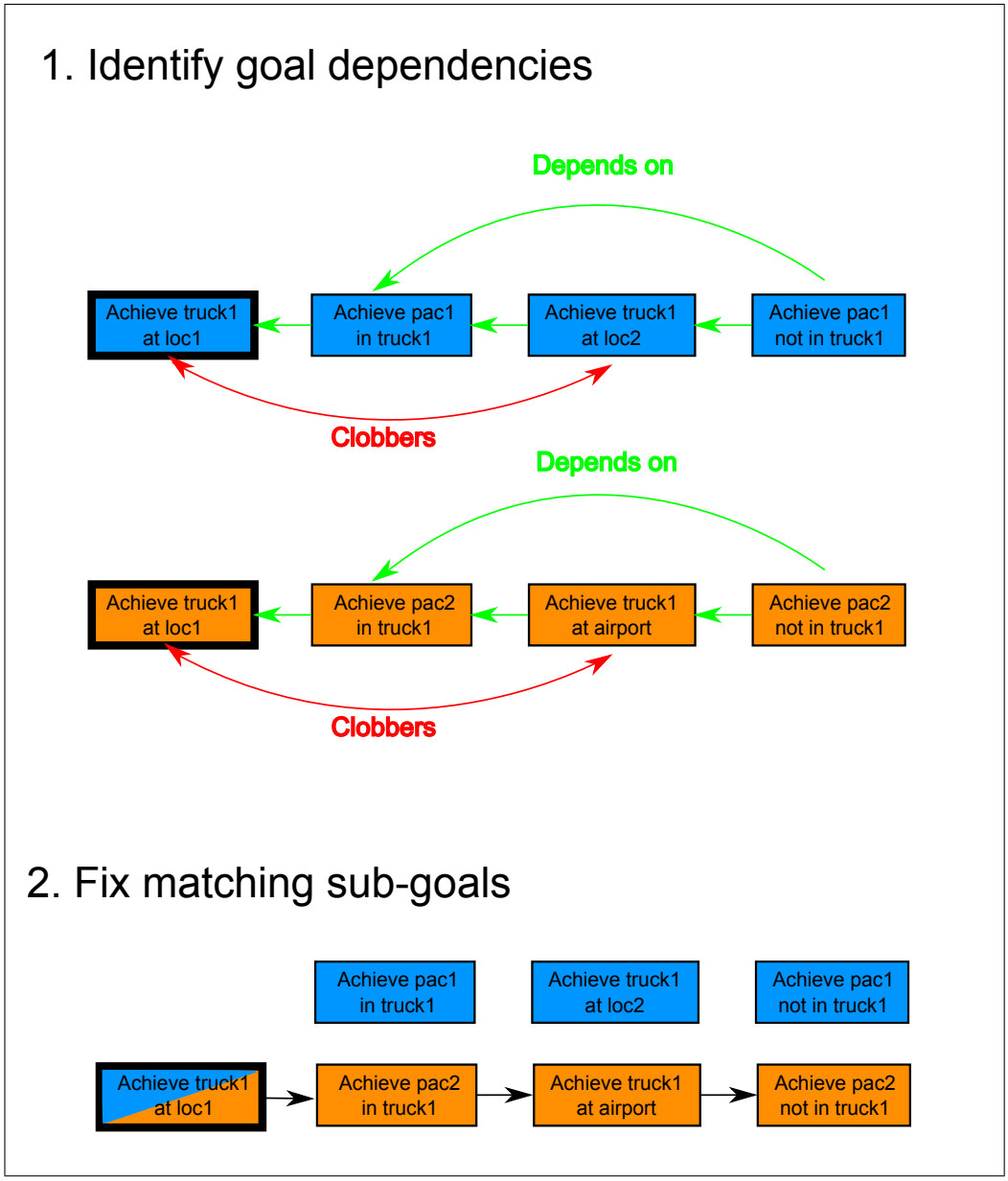
interleaveDecomps accounts for this by first trying to find a viable interleaving with all of the matching sub-goals constrained to be achieved at the same time, and then successively relaxing these constraints. If two decompositions have n matching sub-goals, then after failing to find a viable interleaving where all of them are fixed, it tries to find a viable interleaving for all cases where $n-1$ are fixed. If it can find no viable interleavings, then it continues searching in all cases where $n-2$ are fixed, and so on, until it finally tries to find a viable interleaving in all cases where only one of the matching sub-goals is fixed.

For example, the decomposition to deliver package1 from location1 to location2 with truck1 and the decomposition to deliver package2 from location1 to location2 with truck1 have two matching sub-goals in common: getting truck1 to location1, and getting truck1 to location2. HOPPER would first try to find a viable interleaving where both of the shared sub-goals are achieved at the same time. If it were not successful, then it would search for an interleaving where first one and then the other pair of matching sub-goals was achieved at the same time. HOPPER stops searching for a way to interleave the two decompositions only when all three attempts to find a viable interleaving fail. In the worst case, if two decompositions share n matching sub-goals, HOPPER will attempt to find an interleaving $2^n - 1$ times (the number of possible subsets of an n-sized set, excluding the empty set), but this is not an issue because two decompositions will only very rarely share more than 2 matching sub-goals.

4.7.7 HOPPER searches for valid insertion points for the remaining sub-goals into the interleaving

Once it has fixed the matching sub-goals, interleaveDecomps progressively searches for appropriate places to insert the remaining sub-goals of one decomposition into the other. Figure 4.12 shows an example of interleaveDecomps interleaving two decompositions for delivering two different packages from the same initial location.

Each goal in the interleaving has a range over which it is satisfied beginning just after the goal itself and ending with the first goal that clobbers it. If no goal clobbers it, then the range extends across the entire interleaving. The sub-goals of one decomposition will often interfere with or clobber the sub-goals of the other. This means that as interleaveDecompsinserts goals into the interleaving, it has to update the ranges over which each goal in the interleaving is satisfied because each new goal may clobber one of the goals already in the interleaving.


1. Identify goal dependencies
Depends on
Achieve truck1 at loc1
Achieve pac1 in truck1
Achieve truck1 at loc2
Achieve pac1 not in truck1
Clobbers
Depends on
Achieve truck1 at loc1
Achieve pac2 in truck1
Achieve truck1 at airport
Achieve pac2 not in truck1
Clobbers
2. Fix matching sub-goals
Achieve pac1 in truck1
Achieve truck1 at loc2
Achieve pac1 not in truck1
Achieve truck1 at loc1
Achieve pac2 in truck1
Achieve truck1 at airport
Achieve pac2 not in truck1

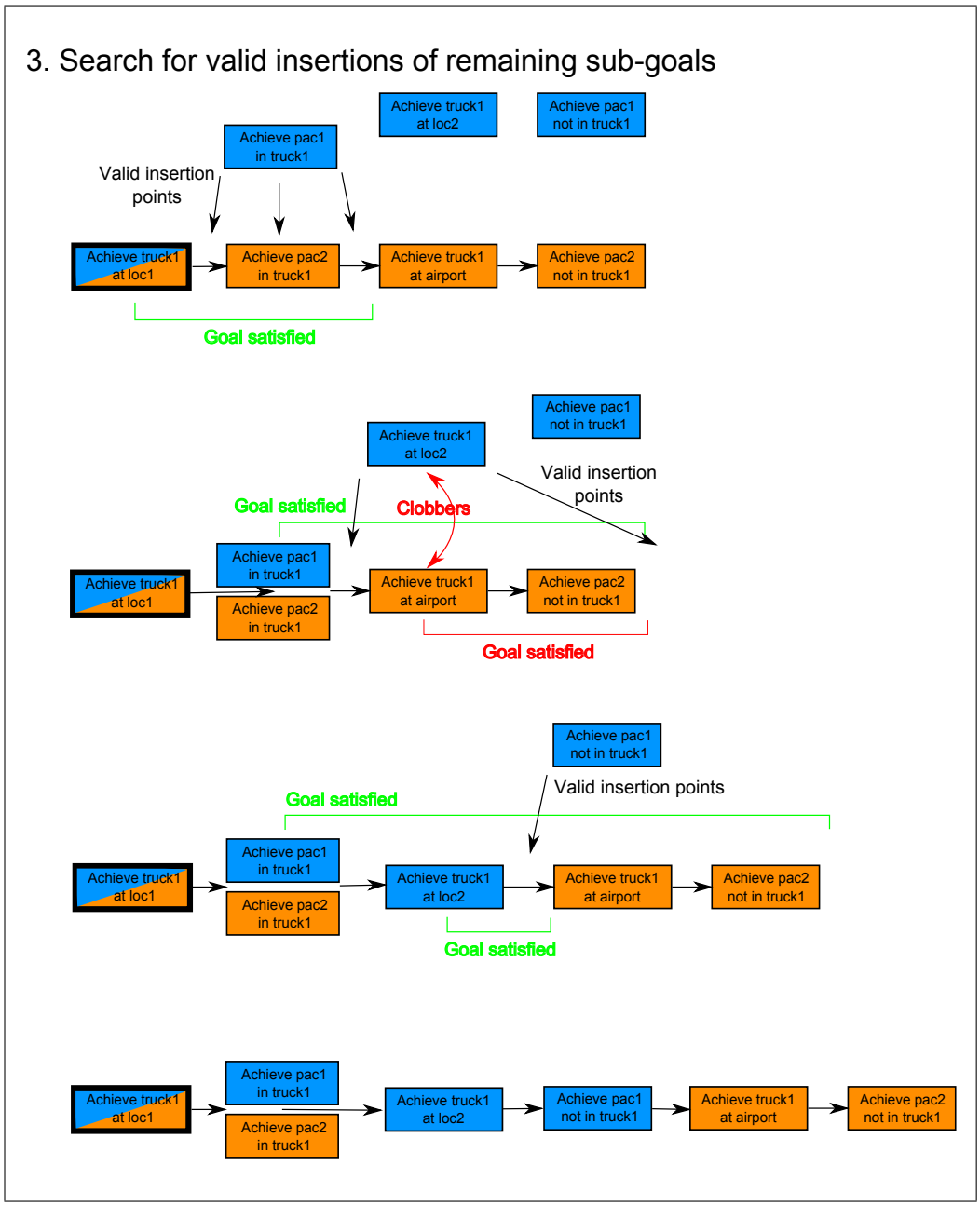


Figure 4.12: Interleaving two decompositions for delivering packages

When searching for an insertion point for a goal into the interleaving, interleaveDecomps makes sure that its goal dependencies are satisfied: the goal can only be inserted at a point in the interleaving where all of the goals it depends on are satisfied. interleaveDecomps also makes sure that the goal being inserted does not itself violate the goal dependencies of other goals in the interleaving: the goal cannot be inserted within the satisfied range of a goal that it clobbers.

A goal will be inserted only when all of the goals it depends on are already in the interleaving. This means that, in general, when a goal is inserted into the interleaving there will not be any goals that depend on it already inserted. An exception to this rule is the matching sub-goals which are inserted at fixed points at the beginning of the algorithm. To account for this, interleaveDecomps enforces a further constraint that if any matching sub-goals depend on the goal being inserted, the goal must be inserted at a point where its satisfaction range includes all the matching sub-goals that depend on it.

4.7.8 The interleaving is a partial order of sub-goals

A decomposition consists of a partial order of sub-goals. When interleaveDecomps inserts sub-goals from one decomposition into another, it partially orders them (while preserving their respective goal dependencies). If a goal can be inserted directly before a goal in the interleaving and it can be inserted directly after it, then there are no ordering constraints between the two goals. As well as inserting the goal before or after the other goal in the partial order, a third option is to co-order the two goals. When searching for a viable interleaving, interleaveDecomps runs a priority search where it prefers to co-order the inserted goals with the interleaved goals if possible. This is because the more co-ordered goals there are, the more flexible the interleaving, and the more opportunities interleaveDecomps has for subsequent sub-interleavings.

4.7.9 HOPPER interleaves interleavings with co-ordered goals

If a group of co-ordered goals consists of three or more goals, then interleave looks for shared sub-goals in their sub-hierarchies and, if it finds them, tries to interleave two of the co-ordered goals. If it finds a viable interleaving for two of the co-ordered goals, then the rest of the co-ordered goals remain co-ordered with the interleaving itself. This means that they can be interleaved with the interleaving. In such a case, interleave searches through the sub-hierarchies of the remaining co-ordered goals and looks for matching sub-goals they share with the interleaving or any of the sub-goals of its unconstrained goals. If it finds such matching sub-goals, then interleave tries to interleave the decomposition in which they occur with the interleaving by first fixing the matching sub-goals, and then searching for valid insertion points for the remaining sub-goals as normal.

For example, if the agent had the following three co-ordered goals:

Deliver pkg1 from loc1 to loc2 using truck1
Deliver pkg2 from loc1 to loc3 using truck1
Deliver pkg3 from loc2 to loc3 using truck1

HOPPER would first find an interleaving for the first two co-ordered goals:

(truck1 at loc1) → (pkg1 in truck1, pkg2 in truck1) → (truck1 at loc3) → (pkg2 not in truck1) → (truck1 at loc2) → (pkg1 not in truck1)

HOPPER would then decompose the third co-ordered goal and search its sub-hierarchy for any matching sub-goals it had in common with the interleaving (or the sub-goals of its unconstrained goals). In this case, there are two matching goals in common: (truck1 at loc3) and (truck1 at loc2). HOPPER would then find the following interleaving:

(truck1 at loc1) → (pkg1 in truck1, pkg2 in truck1) → (truck1 at loc3) → (pkg2 not in truck1, pkg3 in truck1) → (truck1 at loc2) → (pkg1 not in truck1, pkg3 not in truck1)

Note that HOPPER does not guarantee optimal interleaving for more than two co-ordered goals. In this example, if in the first interleaving the agent planned to go to loc2 first and then to loc3, then the best interleaving HOPPER would be able to find for the third co-ordered goal would be:

(truck1 at loc1) → (pkg1 in truck1, pkg2 in truck1) → (truck1 at loc2) → (pkg1 not in truck1) → (truck1 at loc3) → (pkg2 not in truck1, pkg3 in truck1) → (truck1 at loc2) → (pkg3 not in truck1)

However, since HOPPER does not begin achieving any sub-goals until it has completed interleaving all of the co-ordered goals, it should be relatively straightforward to extend HOPPER to backtrack to earlier interleavings during its search so that it can find the optimal interleaving for all of the co-ordered goals.

4.7.10 HOPPER treats the resulting interleaving as an index into the decomposition hierarchy

The interleaving of two co-ordered goals is a partial order of sub-goals that are already present and fully decomposed in HOPPER's decomposition hierarchy. The interleaved goals can themselves be viewed as the top nodes of a partial decomposition hierarchy, making it possible for HOPPER to treat an interleaving like any other decomposition hierarchy. However, it is important to note that each interleaving is dependent on the main decomposition hierarchy. Any changes to the decomposition hierarchy such as a sub-goal being serendipitously achieved are also reflected in the interleavings involving those sub-goals or any of their descendants.

4.7.11 HOPPER finds sub-interleavings for the co-ordered goals within an interleaving

When HOPPER is partway through achieving the goals of an interleaving, updateInterleavings treats the co-ordered goals of an interleaving like the co-ordered goals of the decomposition hierarchy and searches for a way of sub-interleaving their sub-hierarchies. If it finds a sub-interleaving, then it pushes it on to a stack of interleavings and recursively applies the interleaving algorithm to it. This can lead to further sub-sub-interleavings as HOPPER finds ways of optimizing its plan at ever finer levels of detail. When chooseActionFromCandidates determines what action to execute in a particular time step, it only makes use of the lowest-level interleaving at the top of the interleavings stack.

For example, after satisfying the first goal of the interleaving given above of having truck1 at location loc1, HOPPER would then decompose the two goals of (pkg1 in truck1) and (pkg2 in truck1). Because these two goals are co-ordered, HOPPER would search their sub-hierarchies for shared sub-goals and try to interleave them if it found any. If the decomposition to load a package into a closed truck was to first open the truck, load the package, and then close the truck again, then the sub-interleaving HOPPER would find would be:

(truck1 is open) → (pkg1 in truck1, pkg2 in truck1) → (truck1 is not open)

HOPPER would again treat the sub-interleaving as any other decomposition: it would interleave it with other decompositions if they shared matching sub-goals, and it would search for sub-sub-interleavings when it decomposed and achieved any co-ordered sub-goals that it may have.

4.8 Limitations

HOPPER has a number of limitations that have to be addressed if it is to fully scale to the HPD. These limitations relate not only to HOPPER but to TADPOLE as well. This is because any modification to the decomposition rules used by HOPPER that extends its power adds an additional burden on TADPOLE that has to learn the rules in the first place.

4.8.1 HOPPER has no way of keeping a goal satisfied

As soon as a goal is satisfied, HOPPER removes it from its goal-decomposition hierarchy and moves on to achieving the next goal. This can be problematic if a sub-goal needs to remain satisfied throughout the execution of a decomposition. Once a sub-goal is removed from the hierarchy, HOPPER has no way of detecting let alone handling an unexpected event that would disrupt it. However, extending HOPPER to handle this should be relatively straightforward. HOPPER's decomposition rules already keep track of sub-goal dependencies, and before executing a sub-goal, HOPPER could insist that all of its dependencies are already satisfied in the current state, and if they are not, then it could redecompose the parent goal and execute the decomposition again (skipping the already-achieved sub-goals).

4.8.2 HOPPER has a fixed limit for the number of failed decomposition attempts

HOPPER has a fixed limit of four failed attempts before it gives up and tries an alternative decomposition. This crude, general limit needs to be extended to a separate, specific limit for each rule that is learned from experience. An additional factor that HOPPER should take into consideration is the height of the decomposition being redecomposed. The higher a failed decomposition rule is in the hierarchy, the more atomic rules it

will take to achieve, and the more expensive it is to reapply it. HOPPER should be less eager to repeatedly execute high-level decompositions than cheap, low-level ones.

4.8.3 HOPPER does not manage resource use

An important limitation of HOPPER is that it in no way keeps track of resource use. For example, driving a truck from one location to another would use up some fuel which could affect the execution of subsequent decompositions (without fuel, the truck cannot be reused to deliver anything else). It is quite straightforward to add resources to the preconditions and effects of tasks (HTNs have done this); however, it is completely unclear how to learn such rules without additional domain knowledge. Furthermore, it is unclear whether intensive resource management is even part of the HPD. People are not particularly effective at managing logistics operations, and instead rely on specialized algorithms to optimize such tasks.

4.8.4 HOPPER does not deal with action duration and waiting

HOPPER's most important limitation is that it does not deal with action duration and waiting. This is an important aspect of HPD, and to address it, the representation would have to simulate time more realistically (*e.g.* provide sensory information about the current state at fixed time intervals rather than when a state change occurs). The representation of the decomposition rules would also have to be extended so that tasks would have an estimated duration and waiting a certain amount of time was a possible action the agent could execute. HOPPER's interleaving mechanism would also have to be modified so that it could make efficient use of waiting time by achieving other sub-goals during that time. A good cook can achieve multiple different sub-goals while waiting for others to complete. To make

full use of the ability to wait, especially in novel situations, the agent will also need to be extended with a physics learning system that can predict how the state will change over time.

4.8.5 HOPPER does not learn from negative examples

HOPPER only refines a decomposition rule after it successfully executes it. It cannot learn from a failed attempt to execute a decomposition because it is difficult for it to identify the reason the rule failed. A rule can fail to achieve its task (even when it is reapplied repeatedly) because its precondition is incorrect and it was applied in an inappropriate state, or because the rule's variable or sub-goal constraints are wrong and they matched to the wrong state objects, or it could be because the rule's ordering constraints are wrong and the sub-goals were achieved in the wrong order, or the rule could be completely correct but a hidden, unexpected effect disrupted the plan.

4.8.6 HOPPER is only applicable for routine tasks

HOPPER can only achieve tasks that it is familiar with — tasks for which it has the appropriate rules. Although it can achieve such tasks in new situations, it has no way of achieving truly novel tasks that it has never seen before. The next section discusses several ways to extend HOPPER to handle novel tasks.

4.9 Achieving Goals without Applicable Decomposition Rules

HOPPER's goal decomposition hierarchy is useful for achieving relatively simple goals and routine tasks; however, it could be extended to be a framework for more sophisticated reasoning about the agent's plan in

new or unusual situations where the agent has no applicable decomposition rule. This section describes two interesting ways that HOPPER could be extended to handle non-routine situations: backward-chaining and forward-chaining. These planning methods are flexible enough to solve arbitrary problems given very little domain knowledge, but they have an exponential cost and are utterly intractable for achieving the agent's main task. However, the modularity of the decomposition rules could be used to constrain the search to only the relevant sub-goals and the relevant level of abstraction of the decomposition hierarchy.

4.9.1 Backward-chaining could be used to satisfy the precondition of a decomposition rule

While planning with HOPPER, if the agent came across a sub-goal in the goal decomposition hierarchy for which it had no applicable decomposition rule, then rather than failing and trying to find an alternative decomposition rule for one of the ancestor goals, the agent could instead try to satisfy the precondition of a decomposition rule for achieving the goal. The agent would find the difference between the precondition of this decomposition rule and the current state, and post this as a sub-goal to achieve. Once this sub-goal is achieved, the decomposition rule should become applicable and the agent should be able to achieve the goal normally. Note that the new sub-goal could itself be unachievable. In such a case, the agent would then recursively apply the same algorithm and try to satisfy the precondition for a decomposition rule for achieving the new sub-goal. Of course, at each step there could be multiple different decomposition rules for achieving the goal/sub-goal in question, and the agent would have to search backward for the appropriate precondition to satisfy. The agent would continue backward-chaining in this way until it found an achievable sub-goal.

When performing this search, an important factor to consider is that

the cost of executing one decomposition rule can be radically different from the cost of executing another decomposition rule. For example, it is far more expensive and time-consuming to travel to another city by walking there compared with the cost of executing the decomposition of driving there. A short plan of expensive actions may be worse than a longer plan of cheaper actions. The backward-chaining search should take note of the cost of doing each decomposition "action". A good way of estimating the cost of a decomposition rule is by decomposing only its unconstrained sub-goals and determining the length of their longest branch. The longer the branch, the deeper the decomposition is likely to be, and the more expensive it will be to execute.

The advantage of embedding the backward-chaining algorithm within HOPPER is that it is very tightly constrained. Rather than trying to achieve the agent's entire main goal with this kind of search, the agent only searches for a way of achieving one particular sub-goal within the decomposition hierarchy. And rather than back-chaining with atomic actions, the agent back-chains with goals and sub-goals and it stops when it reaches one that it knows how to achieve (it has an applicable decomposition rule for it). The rest of the planning would proceed as normal.

4.9.2 Forward-chaining could be used to search with decomposition rules

Decomposition rules can be viewed as high-level, abstract actions. They have preconditions and they cause effects in the state. Because they try to minimize their side-effects, they are also predictable which means that it is possible to plan with them directly, whether to solve the top-level goal, or a sub-goal in the decomposition hierarchy that the agent cannot otherwise achieve.

If the agent has enough of an understanding of what the effects of its rules are, including their various hidden effects that are not directly ob-

servable, if it has enough knowledge about the physics of the domain and it can predict the consequences of the direct effects, then the agent can search for a sequence of decompositions that will have the desired effect on the state, whether to satisfy the precondition of a decomposition to achieve some sub-goal, or to achieve the sub-goal directly.

An important problem that needs to be overcome if this approach is to be feasible is pruning the search space. An agent that can solve a wide range of tasks in the HPD is likely to have a large set of decomposition rules. This means that the branching factor for the search tree of a forward search will be large and the search could end up being prohibitively expensive even for short plans. One possibility is to prefer decomposition rules that manipulate the same kinds of objects that the agent wants to affect. For example, there is not much point considering a decomposition for changing a spare tire if the agent wants to bake a cake.

Chapter 5

TADPOLE

This chapter describes TADPOLE (TAsk Decomposition Parser and Online LEarner), an algorithm that uses decomposition rules to parse the demonstrated behaviour of a teacher and then learns new decomposition rules and refines its existing decomposition rules from the resulting parsed decomposition hierarchy.

The TADPOLE algorithm is the inverse of HOPPER. HOPPER uses decomposition rules to generate a decomposition hierarchy in a top-down manner by decomposing a top-level goal (or goals) into sub-goals, sub-sub-goals, and so on down to atomic actions. TADPOLE uses decomposition rules to generate a decomposition hierarchy in a bottom-up manner by parsing an atomic sequence of states into a sequence of achieved tasks, and then parsing the sequence of achieved tasks into a sequence of higher-level tasks, and so on up to the top-level task that was achieved by the original sequence of atomic actions.

The primary function of TADPOLE is to parse a teacher's demonstrated actions, reconstruct the decomposition hierarchy used to achieve the top-level task, and then to use the parsed decomposition hierarchy to learn and refine its decomposition rules. TADPOLE can also be used to parse and interpret the observed behaviour of any agent, not just a teacher.

When TADPOLE learns from a teacher, it makes a number of felicity

assumptions about the teacher's behaviour and the lessons they provide. These felicity constraints on the teacher ease TADPOLE's learning task and allow it to parse the teacher's behaviour when TADPOLE's rule set is still incomplete and not fully learned. Because TADPOLE cannot make the same felicity assumptions about the behaviour of arbitrary agents that it can for the behaviour of a teacher agent, it is best for it not to attempt to parse the behaviour of non-teacher agents until it has correctly learned a robust set of decomposition rules.

Organization of the chapter

- Section 5.1 provides an outline of what TADPOLE does. It describes the kind of input that TADPOLE expects from the teacher and the structure of the parsed decomposition hierarchy that it generates. The section also describes the constraints on the teacher, and the felicity condition assumptions that TADPOLE makes about the teacher and the lessons it provides.

- Section 5.2 presents an overview of the TADPOLE algorithm, and describes how TADPOLE interacts with its simulated environment. This section covers TADPOLE's basic parsing techniques and how it gradually builds its goal-decomposition parse as it sees subsequent states of the teacher's demonstration.

- Section 5.3 describes how TADPOLE addresses the problem of identifying the relevant features of a large, complex state. It goes on to explain in more detail the mechanics of how TADPOLE matches goals and tasks to state-differences.

- Section 5.4 describes how TADPOLE scores the matchings between a decomposition rule and a set of demonstrated state-differences, distinguishing between good matches and poor matches, and guiding its beam search for the best parse of the teacher's demonstration.

- Section 5.5 covers how decomposition rules are refined once the teacher's demonstration has been successfully parsed.
- Section 5.6 covers TADPOLE's advanced parsing techniques: how TADPOLE parses demonstrations with decompositions that have subgoals that are already achieved, how it parses demonstrations that involve interleaved decompositions, and how it parses a repeated decomposition into a single rule.
- Section 5.7 explains how TADPOLE uses gaps in the parsed goal-decomposition hierarchy to learn new decomposition rules.
- Section 5.8 concludes the chapter by describing the limitations of TADPOLE and ways in which the algorithm could be improved and extended.

5.1 Input and Output of TADPOLE

TADPOLE is an algorithm that produces a decomposition hierarchy by parsing the demonstrated behaviour of a teacher agent. This decomposition hierarchy is an interpretation of the teacher's behaviour, and TADPOLE uses it to learn new decomposition rules and refine old ones. This section describes the assumptions TADPOLE makes about the nature of its input and the output it generates.

TADOPLE learns and refines its decomposition rules by observing a teacher agent that provides lessons of how to achieve various goals. The purpose of each lesson is to teach TADPOLE a way of achieving a goal (in other words, each lesson demonstrates a task). The demonstration consists of a sequence of states generated by a teacher executing a sequence of atomic actions in the given domain that result in the goal being achieved.

5.1.1 A teacher is necessary in order to learn complex rules in the HPD

One of the fundamental motivations for both TADPOLE and HOPPER is to duplicate at least to an extent humans' ability to rapidly learn in very complex domains. The domains TADPOLE learns in are modelled after the Human Planning Domain to be large, complex, and unpredictable. In such domains humans can learn very quickly given only a few examples, even when they have only a rudimentary and incomplete understanding of the physics of the domain. For example, humans can successfully make a cup of coffee after seing only one example, and they can successfully operate electrical devices in the kitchen without needing any understanding of electronic circuits. However, in order to learn how to achieve a novel task in a poorly understood domain, humans require a teacher to first demonstrate how to do it. Trial-and-error experimentation with electricity and fire is dangerous, and may never result in the discovery of key

steps such as the use of yeast to make dough rise.

The vast majority of skills that humans have are learned by observing others, and a key learning ability that humans have is the ability to learn from a teacher. It would not be sensible to attempt to duplicate humans' ability to learn quickly in complex domains and do it without a teacher

5.1.2 Having a human teacher places constraints on TADPOLE

As described in Chapter 1, TADPOLE and HOPPER are inspired by the problem of creating an autonomous agent that acts intelligently in the real world, specifically in the Human Planning Domain. In such a domain, TADPOLE would run as a program in a robotic body and the role of the teacher would fall to a human instructor.

Note that TADPOLE could also learn from a non-human teacher, as long as the system used decomposition rules to generate its lessons and it abided by the felicity conditions described here.

TADPOLE can only observe the external effects of the teacher's actions

TADPOLE would have no access to the internal state of a human teacher, and in order to scale to the HPD, TADPOLE cannot depend on such additional information. Because of this, TADPOLE cannot observe what actions the teacher decided to perform, or any of the reasoning the teacher went through to select the actions. TADPOLE can only see the sequence of states that resulted from the teacher's actions. Note that depending on the particular domain, actions can have hidden effects that modify the state but are not directly observable by TADPOLE.

The teacher will not always undo unimportant side-effects

Some side-effects of executing a decomposition are not important and the teacher may not bother to undo them before concluding the lesson. For example, when delivering a package by truck, the fact that the truck's position changes as well is an unimportant side-effect that can be ignored, so when the teacher demonstrates how to deliver a package by truck, they may not bother returning the truck to its original location in the demonstrated lesson especially if they go on to demonstrate the delivery of more packages.

TADPOLE must be able to recognize that some of the effects of the same decomposition may or may not be present in different instances, and TADPOLE must be able to match and update its rules appropriately.

The teacher provides only a limited number of lessons for each task

Generating lessons for TADPOLE can be a time-consuming process. This is especially true for more complex tasks. The more complex a task, the deeper its goal-decomposition hierarchy, the more atomic actions need to be executed to achieve it, and the more actions the teacher needs to incorporate in its lesson demonstration. Because of this, TADPOLE cannot depend on the teacher providing a large number of examples for each decomposition rule, and one of the criteria for evaluating TADPOLE is the number of lessons it requires to learn effective decomposition rules (see Chapter 6 for more details).

5.1.3 TADPOLE makes felicity assumptions about the teacher

It can be very difficult to learn from a bad teacher, especially when the teacher provides no feedback as is the case for TADPOLE. A learner may need to use much more cautious learning techniques to learn from bad teaching. TADPOLE makes several assumptions about the teacher's demonstrations in order to simplify its learning task. If the teacher does not sat-

isfy these assumptions, then TADPOLE may fail to learn or may learn incorrectly.

TADPOLE assumes that the teacher uses decomposition rules

An important assumption that TADPOLE makes is that the teacher uses decomposition rules to achieve the goals it demonstrates in its lessons.

There are many ways of planning to achieve goals: finding an ordering of atomic actions by reasoning about the preconditions and effects of atomic actions, using a learned reactive policy from states to actions, using learned macro-actions, and so on. However, TADPOLE assumes that the teacher uses decomposition rules to generate a goal-decomposition hierarchy (in a similar manner to HOPPER) and determines the appropriate atomic actions to execute from it. Note that it would not matter if the teacher actually used a different planning method, as long as that method generated behaviour that could have been generated from decomposition rules.

TADPOLE assumes that the teacher demonstrates simple tasks before complex ones

TADPOLE assumes not only that each lesson is implicitly structured by a goal-decomposition hierarchy, but also that the sequence of lessons itself is structured.

Learning a large number of new, interacting rules, or even correctly parsing a teacher's demonstration that involves a large amount of novel information is very difficult. This is because without knowing enough of the rules used by the teacher, the parse of the teacher's demonstration becomes increasingly ambiguous, and the more novel information there is in the lesson, the more uncertain and ambiguous the parse is. If the teacher presents too many new decompositions at once, then the learner may not be able to separate the new decompositions correctly and will

either combine them into a single, very complex, and ultimately useless decomposition or it will fail to learn at all.

Good human teachers will often present new information and new ways of doing things in the context of things the student has already mastered, with subsequent lessons building on previous lessons.

TADPOLE exploits this assumption by building on the knowledge it gained from earlier lessons. It parses the teacher's demonstration in a bottom-up manner, using rules it learned in previous lessons to parse the low-level decompositions and form the context in which it can learn new, higher-level decomposition rules. In this way, TADPOLE only has to deal with only a limited amount of novel information and new rules at a time.

In order for TADPOLE to make use of knowledge from earlier lessons, the sequence of lessons presented by the teacher must generally build from simpler tasks to more complex tasks in such a way that each new rule will be demonstrated within the context of rules that have already been taught. If this felicity condition is violated, TADPOLE will either become too confused and will simply reject the lesson and refuse to learn from it, or it will learn incorrect or overly large rules.

TADPOLE assumes that the teacher does not make mistakes

TADPOLE receives only a limited amount of information from the teacher: apart from the demonstration itself, the only other information it receives is a signal when the lesson begins and when it ends. Because of this, TADPOLE must assume that the teacher's demonstration is completely correct with no unnecessary actions or mistakes. If the teacher does make a mistake, then TADPOLE treats the mistake as just another part of the demonstration.

TADPOLE has no *a priori* way of determining what is intentional and what is a mistake in the lesson it observes. Although mistakes do share some characteristics with each other, they are by themselves not enough to uniquely identify mistakes in the teacher's demonstration.

If a teacher accidentally does something wrong they usually immediately undo the mistake. For example, when demonstrating how to make an omelette, if the teacher accidentally drops an egg on the floor, then they will immediately undo this unwanted effect by cleaning up the mess before proceeding with the rest of the demonstration. However, immediately undoing a sub-goal is not enough to distinguish mistakes from intentional changes to the state. In many tasks, sub-goals achieved earlier in the task are undone later (with clean-up sub-goals). For example, when making a cup of coffee, the teacher will first open the cupboard containing a cup, and then close the cupboard after taking out the cup. There is no *a priori* way for TADPOLE to determine that opening and closing a cupboard with the side-effect of having a cup on the bench is intentional, whereas creating a mess on the floor and cleaning it up with the side-effect of having egg shells in the trash is a mistake. Without having an understanding of the physics of the domain and the various hidden effects and relationships, TADPOLE cannot determine which side-effects are important and which are not. In the example given above, for all TADPOLE knows there could be an omelette spirit that needs to be appeased with egg shells in the trash in order for the omelette to come out right.

Note that the demonstrations TADPOLE observes may contain irrelevant state changes caused by unpredictable events occurring in the domain that are beyond the control of the teacher. However, TADPOLE requires that the teacher continually makes progress towards achieving the task it is demonstrating so that at least some of the state changes are always caused and intended by the teacher.

A requirement for identifying mistakes is more and richer information from the teacher. Human teachers often provide significantly more information that just what they demonstrate. They use body language to show emotions such as frustration which help to show when things have gone wrong. Human teachers also use verbal communication with their students and inform them when something unexpected occurs that dis-

rupts the demonstration. Future work on TADPOLE could exploit this additional information to relax the assumption that the teacher makes no mistakes during a lesson.

TADPOLE assumes the teacher will demonstrate a range of examples

When learning decomposition rules, TADPOLE learns which properties and relationships are significant by observing demonstrated instances of the teacher using decomposition rules to achieve tasks. TADPOLE assumes that properties and relationships that tend to recur in different examples are significant so it is important for the teacher to diversify its examples. For example, if the teacher were demonstrating how to deliver a package and it showed 1000 instances where it always delivered a package with a red truck, then TADPOLE would assume that the red colour of the truck was significant to the decomposition rule.

5.1.4 Each lesson consists of a sequence of states separated by atomic time slices

Both TADPOLE and the teacher agent interact with a simulated environment in sense-action-sense cycles (in similar way that HOPPER does). At each time slice, TADPOLE receives a sensory description of the world, and the teacher agent selects an atomic action and executes it.

Immediately before beginning a lesson, the teacher informs TADPOLE that the lesson is about to begin. The teacher then proceeds to select and execute a sequence of atomic actions that result in a goal being achieved in the state. Once the goal has been achieved and the lesson is concluded, the teacher informs TADPOLE that the lesson is finished. The teacher is then free to demonstrate a new lesson of how to achieve another (or the same) task.

5.1.5 The agent and the teacher may control an avatar

Depending on the particular domain, the agent (consisting of both TADPOLE and HOPPER) and the teacher may or may not control avatars represented in the state. An avatar consists of an object (or several closely related objects) in the state. In domains with avatars, every agent that interacts with the simulated environment has a corresponding avatar. An agent in such a domain can only execute atomic actions to directly affect its corresponding avatar. Only by manipulating its avatar can an agent affect other objects in the world.

In a domain such as a logistics domain there is no need for the agent or the teacher to have any "physical" presence in the simulated world, since the actions are not performed directly (*e.g.* the trucks are assumed to have drivers). In other domains that assume actions are performed directly, it is necessary to give both the agent and the teacher avatars with which to execute their actions.

In a domain involving avatars, the agent and the teacher each control separate avatars. When TADPOLE observes the demonstration of a teacher in such a domain, it maps its own avatar to the teacher's avatar in its mental representation of the world. For example, if the teacher demonstrates how to make a cup of coffee and in one state the robotic arm of the teacher's avatar is next to a cup and in the next state the arm is grasping the cup, then TADPOLE will remap the states in the demonstration so that the robotic arm of TADPOLE's avatar is next to and then grasping the cup. TADPOLE does this so that when it learns decomposition rules from the demonstration HOPPER will apply them to its own avatar, rather than trying to control the teacher's avatar (when learning how to pick up a cup, HOPPER should use its own arm and not try to grab the teacher's arm and use it to lift objects!).

Children's ability to mimic observed behaviour at an early age suggests that humans may have such a mechanism built in [17].

5.1.6 TADPOLE assumes it knows about all of the atomic actions the teacher may execute

In order to completely parse the teacher's demonstrated lessons, TADPOLE must able to learn the appropriate atomic decomposition rules. However, because TADPOLE can only observe the effects of the teacher's actions but not the actions themselves, it must have a way of determining what atomic action the teacher executed at each time step based on the state changes that occurred. TADPOLE assumes such a mechanism, and though the mechanism for learning how to perform the atomic actions themselves is outside the scope of this thesis, TADPOLE does learn the atomic appropriate decomposition(s) for each atomic action from the teacher's demonstrations.

5.1.7 TADPOLE generates a hierarchical decomposition parse of the teacher's demonstration

The end result of TADPOLE's parse is a reconstructed goal-decomposition hierarchy that TADPOLE believes the teacher used to achieve the task demonstrated in the lesson. Note that there may be a number of possible alternative parses, in which case TADPOLE selects the one that best matches its current rule set.

The decomposition hierarchy generated by TADPOLE is a DAG of state-differences achieved by the teacher during the demonstration, where each state-difference describes the state changes that the teacher achieved between an earlier time and a later time in the lesson. Each node in the DAG (except for the atomic nodes at the bottom of the hierarchy) has an ordered sequence of children.

Each atomic node at the base of the hierarchy corresponds to the difference between two consecutive states in the teacher's demonstration, the result of the teacher executing a single atomic action. The state-differences higher in the decomposition hierarchy correspond to the difference be-

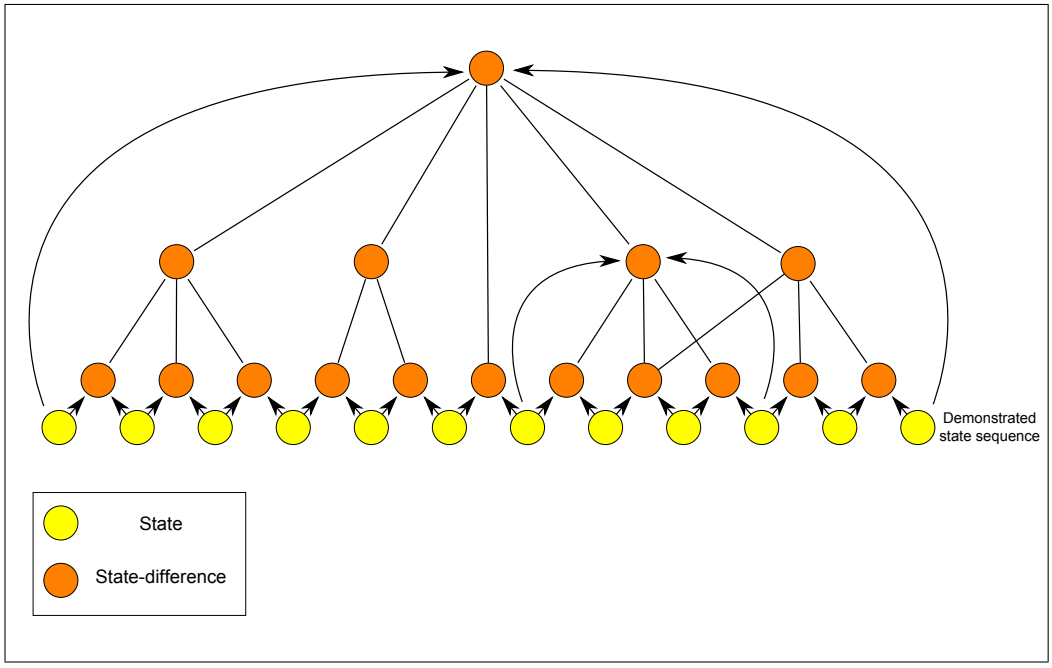


Figure 5.1: State-difference decomposition hierarchy

tween the initial state of their earliest descendant atomic node and the final state of their latest descendant atomic node. The root node of the hierarchy corresponds to the difference between the initial state and the final state of the demonstration. Figure 5.1 shows a graphical representation of such a state-difference hierarchy, and indicates with arrows which states the root state-difference node and a mid-level state-difference node are the difference of.

For any demonstration, there are a very large number of ways of parsing it, with each parse corresponding to a unique DAG of state-difference nodes. TADPOLE justifies its final parse by matching its decomposition rules to the state-difference nodes of the parse. Every state-difference node and its direct children in the decomposition hierarchy are matched against a decomposition rule. A node's state-difference corresponds to the head-task of a decomposition rule that achieves it, and the child nodes correspond to the sub-goals of the decomposition rule. Note that a node that

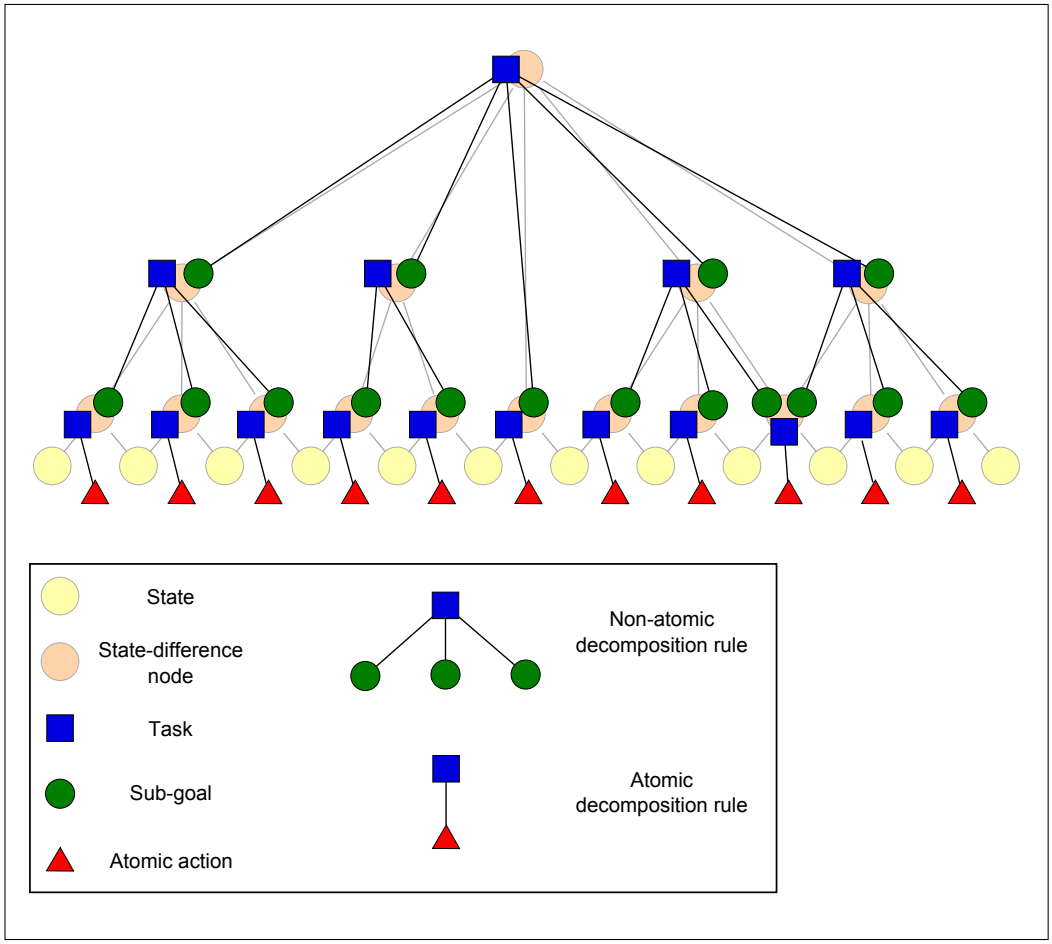


Figure 5.2: Decomposition rules matched with a state-difference hierarchy

corresponds to the head-task of a decomposition also corresponds to a sub-goal of a parent decomposition, and a node that corresponds to a sub-goal also corresponds to the head-task of a sub-decomposition. This is true for all nodes in the hierarchy except for the atomic nodes at the base of the hierarchy which correspond to atomic decompositions, and the root node(s) at the top of the parsed hierarchy. Figure 5.2 shows a graphical representation of how decomposition rules would be matched with the state-difference hierarchy in the previous example.

The entire decomposition hierarchy is scored depending on how well

the state-differences calculated from the teacher's demonstration match with the decomposition rules used to parse them. Every node in the decomposition hierarchy keeps track of its decomposition rule matching so that the overall score of the partial parse can be calculated and so that the decomposition rules can be refined if the partial parse is completed and selected.

TADPOLE is capable of parsing interleaved decompositions resulting in a goal-decomposition hierarchy where the sub-goals of one node are interleaved with the sub-goals of another. In such cases, two nodes in the goal-decomposition hierarchy can share some of the same sub-nodes. These correspond to sub-goals of two different decompositions that the teacher achieved in parallel.

A parsed hierarchy may have nodes that are not matched with any decomposition rules. These represent "holes" that TADPOLE cannot explain with its existing rules. These holes are opportunities to learn new decomposition rules as described in Section 5.7.

5.2 Overview of the Algorithm

This section presents a simplified overview of TADPOLE's parsing algorithm; subsequent sections provide further details about specific aspects of the algorithm. The pseudo-code for the TADPOLE algorithm can be found in appendix A.2.

5.2.1 TADPOLE runs a beam search over the space of partial parses

For any demonstration longer than the bare minimum there are almost always multiple different possible ways of parsing it. TADPOLE searches for the best parse with the highest score where the score of a parse is calculated from how well the decomposition rules used in the parse match with

the demonstrated sequence of states (and the resulting state-differences). Because the number of possible parses can be intractably large for demonstrations of even moderate length, TADPOLE cannot simply enumerate all the possible parses and select the best one. Instead, TADPOLE searches for ways of extending partial parses into complete parses.

A partial parse, like a complete parse, consists of a hierarchical DAG of state-difference nodes. However, a complete parse will have only one root node at the top level, while a partial parse can have multiple such nodes. A completely unparsed partial parse will consist of a single level of state-difference nodes which will correspond to the state-differences achieved by the teacher's atomic actions.

TADPOLE runs a beam search where each partial parse is scored in (almost) the same way that a complete parse is (Section 5.3 provides more details about the scoring mechanics). Once it has found a number of possible complete parses, TADPOLE simply selects the one with the highest score.

The beam needs to accommodate at least 10 to 20 partial parses in order to correctly parse the demonstrations in the kitchen and logistics domains (these two domains are described in Chapter 6), but it should be as wide as practically possible (the wider the beam, the slower the algorithm, but the greater the chance of TADPOLE finding the correct parse).

5.2.2 TADPOLE begins to parse the teacher's demonstration as soon as it observes a new state

TADPOLE is an online algorithm: it begins to parse and interpret the teacher's demonstration as soon as it begins. The learnLesson function (A.2.1) begins to parse a new lesson after the agent has observed the first two states (so that it can determine the initial state change caused by the teacher), and it extends its partial parses with every new observed state of the teacher's demonstration.

When it observes a new state, learnLesson first calculates the difference between the new state and the last state it observed. It then makes use of this state-difference to determine what atomic action the teacher just executed. learnLesson then calls findBestAtomicHierarchy (A.2.2) to search through TADPOLE's atomic decomposition rules (decomposition rules that have a head-task and an atomic action instead of a partial order of sub-goals) to find the best matching rule. After finding the best rule, learnLesson calls extendParse (A.2.3) to extend every partial parse on the beam with a new node that has the calculated state-difference and its matching to the head-task of the best matching atomic rule.

It is important to note that as the parse progresses, TADPOLE matches new decomposition rules only to the root state-difference nodes. How the state-difference nodes are parsed lower down in the parsed hierarchy does not affect how the parse progresses except for their contribution to the overall score of the partial parse. Because the new atomic state-difference node will always be the difference between the last observed state and the newly observed state, matching different atomic decompositions will always result in the same sequence of root-state difference nodes, so there is no reason not to simply use the best matching atomic decomposition for every partial parse.

After extendParse has extended a partial parse with the new state-difference parsed by an atomic decomposition rule, it searches through TADPOLE's non-atomic decomposition rules for ways of matching them to the root nodes of the newly extended partial parses. learnLesson then calls getNeighbouringParses (A.2.8) to find the possible ways of extending the partial parse with the newly matched decomposition rules. learnLesson continues to search for ways to match decomposition rules to the possible partial parses and then to extend them until it can no longer make progress (when none of its non-atomic decomposition rules match completely to any part of any of the partial parses). It then keeps the best partial parses on its beam and waits to observe the next state in the teacher's

demonstration.

5.2.3 TADPOLE can make use of additional domain knowledge to help match atomic rules

Atomic actions are "black box" procedures that affect the state when executed. Atomic decompositions show how to achieve a particular task by executing an atomic action. If learnLesson can determine or at least narrow down the possibilities of what atomic action the teacher executed in the last time slice, it can greatly narrow down what task the teacher was trying to achieve and which decomposition rule they used to achieve it. The viable atomic actions are those that, if executed in the previous state, will generate a subset of the state changes noted in the state-difference.

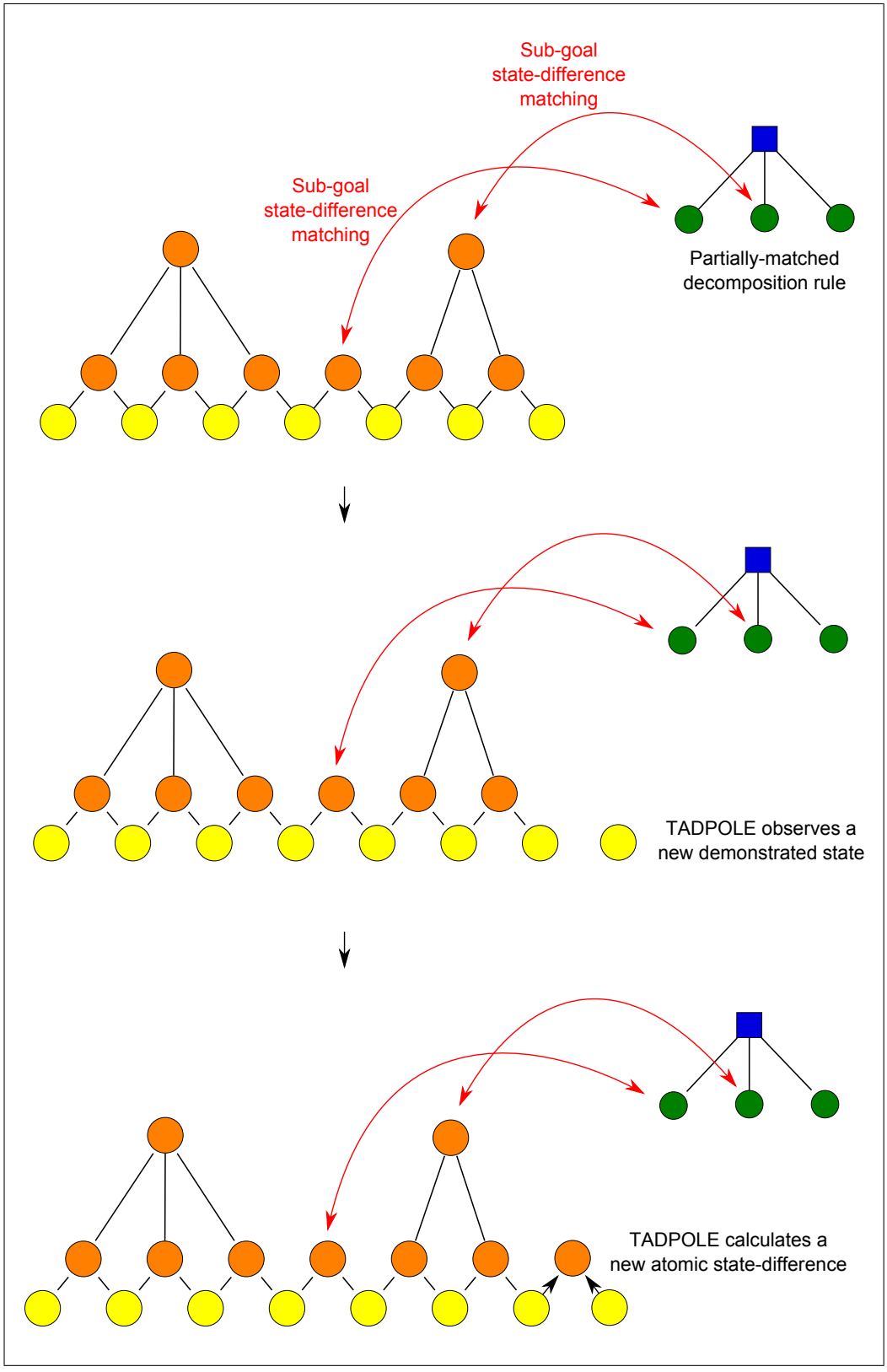
Note that even if TADPOLE knows which atomic action the teacher executed, there can still be multiple possible candidate atomic decompositions, and findBestAtomicHierarchy will still have to find the best matching one. This is because there could be multiple atomic decompositions with the same atomic action, reflecting the fact that the same atomic action can have very different effects in different states. For example, the atomic action of twisting one's hand can result in a door becoming unlatched or water being released from a tap depending on whether the hand is grasping the knob of a door or the spigot of a tap. To determine what task the teacher was trying to achieve (unlatching a strange, tap-shaped door or turning on the water of a tap), findBestAtomicHierarchy has to match the head-tasks of the candidate atomic decompositions with the observed effect demonstrated by the teacher.

5.2.4 TADPOLE matches the sub-goals of decomposition rules to the state-differences of partial parses

TADPOLE extends the decomposition hierarchy of a partial parse by bottom up parsing. extendParse searches for non-atomic decomposition rules that match a consecutive sequence of root state-difference nodes (nodes without parent nodes at the top of the decomposition hierarchy). A decomposition rule matches a consecutive sequence of state-difference nodes if every sub-goal of the decomposition rule matches with one of the state-difference nodes and if the head-task of the decomposition rule matches with the difference between the first state of the state-difference sequence and the last state of the sequence.

If a rule matches, then getNeighbouringParses calls the getNeighbour (A.2.10) function to add a new node to the top of the decomposition hierarchy. The new node has the newly calculated state-difference between the first and last state of the state-difference sequence, and its child nodes are the consecutive sequence of state-difference nodes that the decomposition rule's sub-goals matched. In this way, the node becomes a new root node in the decomposition hierarchy of the partial parse in place of the consecutive sequence of nodes which now become its sub-nodes. Figure 5.3 shows a graphical representation of TADPOLE extending a partial parse by matching a decomposition rule against demonstrated state-differences.

Parsing the decomposition hierarchy with a decomposition rule in this way alters the root level of the partial parse, replacing a consecutive sequence of nodes with a new node. A new decomposition rule could potentially match to a consecutive sequence of root nodes that includes this newly parsed node, and so after successfully parsing part of a decomposition hierarchy with a decomposition rule learnLesson continues to search for decomposition rules to further parse the decomposition hierarchy until it can make no more progress (it can find no decomposition rule that matches any consecutive sequence of root nodes).


Sub-goal
state-difference
matching
Sub-goal
state-difference
matching
Partially-matched
decomposition rule
TADPOLE observes a
new demonstrated state
TADPOLE calculates a
new atomic state-difference

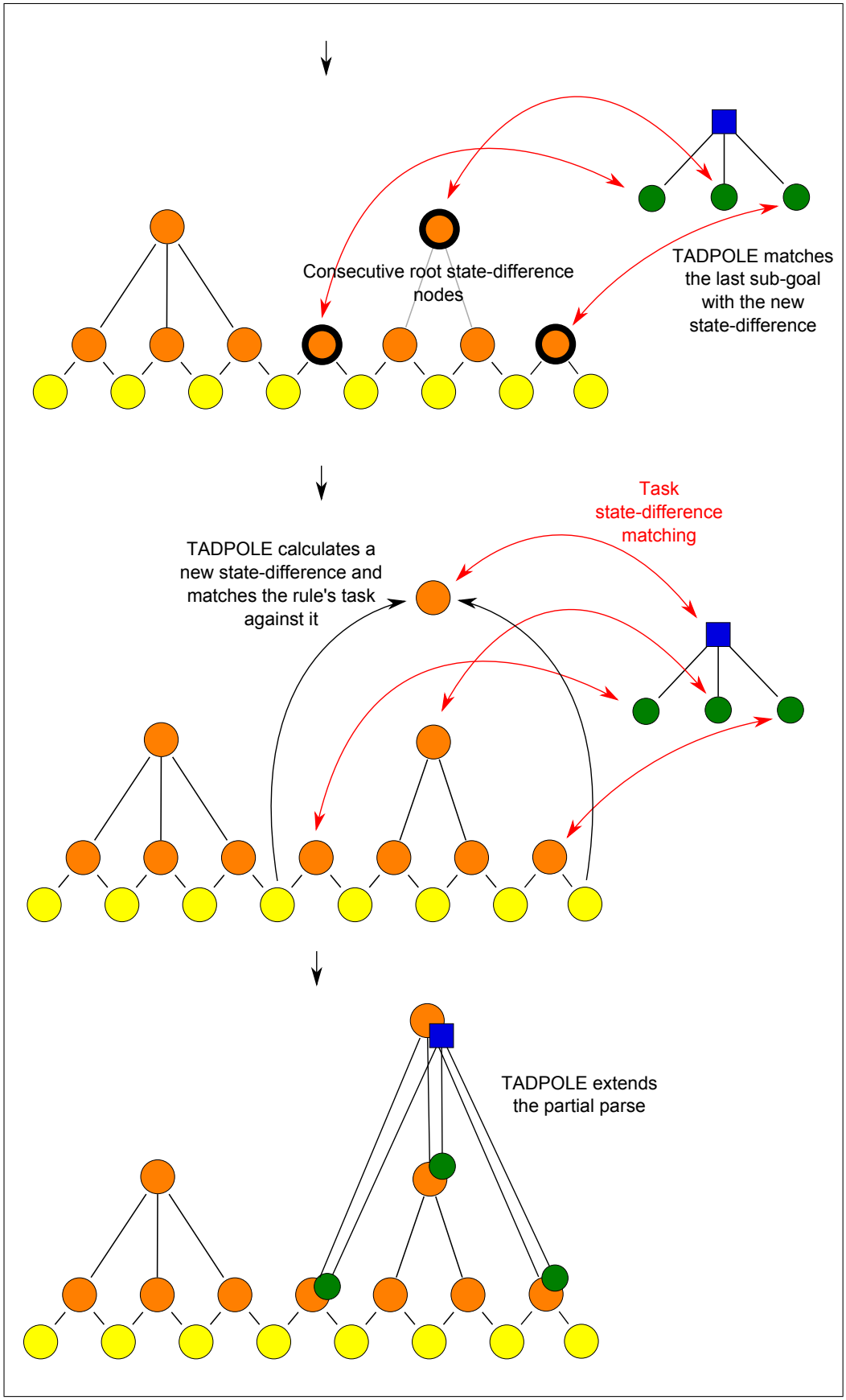


Figure 5.3: TADPOLE parsing with a decomposition rule

For any partial parse, there may be multiple decomposition rules that match to different consecutive sub-sequences of root nodes. If these sub-sequences overlap, then they will, in general, be mutually exclusive: they will correspond to different ways of parsing the teacher's demonstration. Parsing with one of the decomposition rules will remove the entire sub-sequence of root nodes it is matched to (making them all sub-nodes of a new root node) and thus make the overlapping nodes unavailable to be used in the parse of the other decomposition rule(s).

TADPOLE cannot determine which way of parsing is correct based on how well their respective decomposition rules match to the state-differences because a locally worse matching decomposition can end up parsing the decomposition hierarchy in such a way as to enable much better matching decomposition rule parses at higher levels of the hierarchy. TADPOLE cannot reliably predict how good the score of the final parse will be based on how well one of the decomposition rules matches to a part of that parse. Instead, whenever getNeighbouringParses finds a new way of parsing a decomposition hierarchy, it adds the new partial parse to the beam of partial parses and continues searching.

In effect, when parsing, TADPOLE searches the space of partial parses where the neighbouring partial parses result from all the possible ways of matching a decomposition rule to a partial parse and then parsing it.

Note that just because there are some decomposition rules that match does not mean that it is correct to immediately use any of them to parse a partial parse. It may instead be correct to wait for future input and then use a decomposition rule that also matches with the newly added nodes. Parsing with a decomposition prematurely may preclude the parsing of a future, better-matching decomposition. To avoid this problem, when learnLesson adds all of the possible extensions of a partial parse to the beam, it also adds the partial parse itself back into the beam.

5.2.5 TADPOLE maintains a beam of partially-matched decompositions

It is inefficient to constantly try to rematch decomposition rules every time that a partial parse is modified (either by being further parsed by a decomposition rule, or by being extended with a newly observed state-difference). Matching a sub-goal of a decomposition rule to the same state-difference more than once is wasteful. The partial parses on the beam have many of the same root state-difference nodes. All of the partial parses are related to each other, whether because they are direct siblings (in the search space) or because they are more distantly related, and so most of them will share some (but never all) of the same root nodes. At every time step they are also all extended by the same atomic state-differences which means that at least at the beginning of each time step all of the partial parses share at least one common root state-difference node.

To avoid rematching decomposition rules, for every partial parse, TADPOLE maintains a set of partially-matched decompositions: decomposition rules that have matched some but not all of their sub-goals to some of the root state-difference nodes. When getNeighbouringParses finds extension of a partial parse, it copies the still valid partially matched decompositions. When it parses a partial parse with a decomposition rule and replaces the matched root nodes with a new node, it removes only the partially-matched decompositions involving the now demoted state-difference nodes.

5.2.6 TADPOLE extends partially-matched decompositions

Whenever a new root node is added to a partial parse, whether by being extended with a new atomic state-difference, or by being parsed with a decomposition rule, extendParse calls matchToNew (A.2.5) to search through TADPOLE's known decomposition rules to find sub-goals that match to the new node. If it finds a decomposition rule with a sub-goal that matches

the new node, then it adds the new partially matched decomposition rule (having only one of its sub-goals matched) to the set of partially matched decomposition rules of that partial parse.

extendParse also calls extendPartialMatches (A.2.4) that finds any partially matched decomposition rules with an unmatched sub-goal matching the new node, and extends them to include the new node. If this extension completes the partial decomposition rule matching (all of its sub-goals are matched, and the head-task of the decomposition rule matches with the state-difference between the first and last state of the state-difference node sequence) then getNeighbouringParses parses the partial parse with the decomposition rule as described above.

Matching decomposition rules in this way prevents TADPOLE from needlessly re-matching the same sub-goals to the same state-difference nodes, and it also allows it to prune poorly matched partially matched decompositions. If TADPOLE knows a large number of decomposition rules, then it is possible that there will be an excessive number of partially matched decomposition rules, many of which score quite poorly. How well a decomposition rule matches overall is determined from how well the sub-goals and task matching match to their respective state-differences (this is described in more detail below). If most of a partially matched decomposition matches poorly, then it is very likely than any complete decomposition matching that derives from it will also match poorly, and so it is unlikely to be worth searching for a way to complete the partial matching. To prune such poor-scoring partial decomposition matchings, TADPOLE maintains a secondary beam of partial matchings for each partial parse. TADPOLE keeps only the n best scoring partial decomposition matchings for each partial parse on the main beam, where n is a parameter for the width of the secondary beam. The optimal value for this parameter depends on the domain, but it should in general be as large as is practically feasible.

5.2.7 TADPOLE only considers parsing sequences that include a new state-difference node

When TADPOLE searches for a way to further parse a partial parse, it searches for a sequence of root state-difference nodes to match against a decomposition rule. However, many of these sequences will have already been parsed in other partial parses on the beam. This is because whenever getNeighbours parses a partial parse during a time step, it adds all of the possible extensions of the partial parses to the beam. These extensions will share many of their root nodes. Reparsing the same sequence of state-difference nodes with the same decomposition rule as another partial parse on the beam would result in duplicate partial parses. Such duplicate partial parses would not only take up unnecessary room on the beam, but would also greatly slow down the algorithm.

To avoid this problem, getNeighbours only parses partial parses in a way that does not produce duplicates. getNeighbours considers only sequences that include either the new root state-difference node every partial parse was extended with in the current time step or a root state-difference node that is an ancestor of the new node.

The only possible way to create duplicate neighbouring partial parses from two different partial parses is if the two sequences of root state-difference nodes used to parse each partial parse are completely disjoint from each other. As long as both parsed sequences have at least one state-difference in common, then the resulting neighbouring partial parses will have to be different from each other. The only way that a root state-difference node can be an ancestor of the new node that every partial parse was extended with in the current time step is if it was created in the current time step as well. This means that the sequences including this node are safe from the danger of creating duplicate partial parses as well.

5.3 Matching decomposition rules to state differences

This section and the subsequent one describes the mechanics of how task and sub-goal graphs are matched with the state-difference graphs of the teacher's demonstration. This section describes how TADPOLE deals with the problem of distinguishing the relevant and irrelevant objects in a large and complex state so that the state-difference graph can be pruned to a tractable size. The next section delves into the details of how the matchings are scored.

5.3.1 Nodes are matched with state objects and links are matched with state relationships

When parsing the lessons demonstrated by the teacher, TADPOLE matches the task and sub-goal graphs of its decomposition rules to the state-difference graphs demonstrated by the teacher. It matches the nodes of a task or sub-goal graph with objects of the corresponding state-difference graph. As a result, the links of the task or sub-goal are automatically matched with the appropriate relationships in the state-difference. A link between two nodes in a task or a sub-goal graph matches with the relationship between the two objects the two nodes are matched with. If there is no relationship between the two objects, then the link is matched with an empty relationship.

5.3.2 Complex domains have many irrelevant objects

The HPD is extremely complex with a very large number of objects, each with many properties and relationships with other objects. If TADPOLE is to scale to the HPD, it must be able to handle such domains and learn rules in very complex states that have a large number of objects. For example,

the kitchen domain can have hundreds of objects each with many properties and many relationships between objects. Determining what aspects of the state are relevant to the lesson and which are not is a significant challenge that needs to be overcome. Different aspects of the state are relevant depending on the specific goal that is being achieved. For example, if the task is to put a cup on the table, then the fact that the cupboard containing the cup is closed is important while the specific colour of the cup is irrelevant. However, if the task is to paint the cup, then the colour of the cup is critically important.

5.3.3 Decomposition rules should have as few irrelevant objects as possible

Decomposition rules that have a large number of irrelevant objects in their task and sub-goal graphs are problematic for both TADPOLE and HOPPER. They are more difficult for TADPOLE to parse and more difficult for HOPPER to execute. The irrelevant objects act as noise when the decomposition rule is matched to the state, acting to obscure the significant objects in the precondition of the rule and the import aspects of the rule's sub-goals. This makes it difficult to determine when the precondition of the decomposition rule is satisfied and when its sub-goals are achieved. Also, because matching decomposition rules involves matching the task and sub-goal graphs to state graphs, the larger the object graphs of the task and sub-goals of the rule, the more computationally expensive the algorithm is (if the goal/task graph has m nodes and the state graph has n nodes, then there are n^m possible matches of goal/task nodes to state nodes). It is therefore important for TADPOLE to limit the task and sub-goal graphs to contain the relevant objects and as few irrelevant objects as possible.

The problem of identifying the important aspects of the state is exacerbated by the fact that, as described in Section 5.1.2, TADPOLE cannot

rely on the teacher to provide a large number of lessons and examples. It cannot gradually distinguish the relevant and irrelevant objects from experience, but must instead use heuristic rules to prune most of the objects in the state immediately.

5.3.4 TADPOLE identifies important aspects of the state by what has changed

Central to TADPOLE's task is identifying the teacher's intention - what sub-goals the teacher wanted to achieve throughout the demonstration. TADPOLE does this by finding the difference between earlier and later states and noting which aspects of the state changed. It in fact matches the tasks and sub-goals of its decomposition rules with these state-difference graphs. A state-difference graph is like a state graph, but some of the properties and relationships in the graph are labelled as additions and deletions. Added properties and relationships are those that were present in the later state but not the earlier state, and deleted properties and relationships are those that were present only in the earlier state.

At least some of the differences between two states in the teacher's demonstration are intentional, caused by the teacher trying to further the task being demonstrated, and they help greatly in identifying both the important aspects of the state and what sub-goal the teacher was trying to achieve. TADPOLE therefore makes the assumption that things that have changed are relevant components of the state difference.

However, not all of the differences between two states will be relevant. Some of the differences that occurred between two states will be unintended changes that were not caused by the teacher and are not relevant to achieving the goal being demonstrated. This can include both predictable changes that occur in the state (for example, the passage of time), and completely unexpected and unpredictable changes. Part of TADPOLE's task is distinguishing between the state differences that were intended by the

teacher and those that are irrelevant to the task being demonstrated.

As well as the properties and relationships that changed directly, some of the properties and relationships in the state that did not change could also be important. This includes properties and relationships in the precondition of the decomposition rule encoded in its head task, and properties and relationships of sub-goals that were already true and did not need to be achieved by the teacher.

For example, when a package is delivered from one location to another, only the location of the package changes from the initial state to the final state, but the fact that the object being moved is a package does not. Nonetheless, this property is critically important as a precondition to the decomposition rule. The decomposition rule for moving an object from one location to another by loading it in a truck and then delivering it is only applicable for packages and other similar objects. If the object to be moved is not a package but a car, for example, then the delivery decomposition rule is not appropriate (it is better to simply drive the car to the desired location rather than first loading it into a truck).

Similarly, when TADPOLE matches the sub-goal of a decomposition rule to a state difference, some of the properties and relationships in the state that did not change may nevertheless be important aspects of the sub-goal. For example, when making a cup of coffee it is important that the object that will contain the coffee is at least a container if not a cup even though this property of the object never changes in the teacher's demonstration.

5.3.5 TADPOLE prunes away state objects that are only distantly related to state changes

When matching goals and tasks to state differences, TADPOLE first prunes the state difference graphs to get rid of as many irrelevant nodes as possible. To do this, TADPOLE makes use of the heuristic that the farther away

an object node is from any state changes in the state difference graph (distance between nodes in the graph is measured normally as the length of the shortest path between them), the less relevant it is. Objects whose properties changed in the state difference graph are the most relevant, the objects directly related to the objects whose properties changed are the next most relevant, the objects directly related to the objects that are directly related to the objects whose properties changed are the next most relevant, and so on. The same heuristic also holds for relationships that changed: the objects that directly take part in the changed relationship are most relevant, the objects directly related to these objects are next most relevant, and so on.

The threshold distance from a state change beyond which a node is pruned is a parameter that can be set for TADPOLE and whose optimal value depends on the nature of the domain. In general, however, for every unit of distance that the threshold is increased, the size of the state-difference graph can potentially increase exponentially (*e.g.* if every object node on the fringe of the graph is related to at least two new, unique object nodes, then by including these new object nodes, the fringe of the state-difference graph would double), so it is important to keep this threshold distance as small as possible.

In both the kitchen and the logistics domains (see Chapter 6), the threshold distance was set to 2, which means that state-difference graphs included the objects that changed, and all of the objects they were directly related to. Larger threshold distances did not produce better matchings and just made the matching more expensive.

5.3.6 TADPOLE prunes away state changes that are not matched in sub-nodes

In the HPD, unpredictable events occur and the world changes of its own accord or due to the actions of other agents. This means that when ob-

serving the teacher's demonstrations, TADPOLE can expect some of the state changes it observes to not be caused by the teacher and be irrelevant to the lesson. The greater the difference in time between an earlier state and a later state, the more irrelevant state changes there are likely to be between the two states.

Each node in a parsed decomposition hierarchy represents the state-difference between the first and last states of the base of the node's sub-hierarchy as described above. The higher a node is in the decomposition hierarchy, the wider the base of the sub-hierarchy, the greater the time difference between the first and last state, and the more irrelevant state changes there will be in the state-difference.

Irrelevant state changes are a minor concern at low levels of the decomposition hierarchy. However, at higher levels they can introduce a large number of irrelevant nodes to the state-difference graphs reducing both the efficiency and accuracy of the algorithm. TADPOLE therefore uses additional heuristics to further prune away irrelevant state changes and hence the objects associated with them.

After matching the sub-goals of a decomposition rule to a sequence of root state-difference nodes, TADPOLE constructs a new state-difference node by finding the difference between the first and last states of the sequence. Before matching this new state-difference against the head task of the decomposition rule, TADPOLE eliminates irrelevant state changes by pruning away any that were not matched in any of the rule's sub-goal matchings. A state change that was not important in any of a rule's sub-goals is unlikely to be important in the rule's head-task or the in the goal that the rule achieves.

While this may sometimes eliminate an important object from the head-task, it is nevertheless a useful heuristic for limiting the size of the state-difference graphs at higher levels of the decomposition hierarchy. Section 5.8 discusses ways of addressing the limitations imposed by this heuristic.

5.3.7 At least some of the core elements of a task must be successfully matched

Because a task is learned, it may include some spurious additions and deletions, and TADPOLE allows some of these core elements to mismatch. However, if none of the additions or deletions specified by the head-task of a rule occurred in the state-difference, then the teacher could not have used that rule to achieve the state-difference. To account for this, TADPOLE disallows any matching between a task and a state-difference that does not successfully match at least some of the core elements of the task, no matter how well the context of the task matches or how well the sub-goals of the rule matched. TADPOLE would resolve such a situation by trying alternate parses instead.

5.3.8 At least some of the core elements of a sub-goal must be achieved

Sub-goals are also learned, so they may include spurious *must* and *must not* properties and relationships, and TADPOLE allows some of these core elements to mismatch. Similarly to a task, if none of the sub-goal's core elements match successfully with the demonstrated state-difference, then the teacher could not have been achieving this sub-goal, and TADPOLE disallows the match. TADPOLE also imposes an additional constraint on sub-goal to state-difference matchings: at least one of the *must* and *must not* properties and relationships of a sub-goal must be directly achieved by the state-difference it is matched with.

A sub-goal can only match with a state-difference if the state-difference is a plausible instance of the teacher satisfying that sub-goal. Because some parts of the sub-goal may initially be satisfied, TADPOLE allows the core elements of a sub-goal to successfully match with the context (elements that have not changed in the state and so were initially true) of the state-difference. A sub-goal's *must* properties and relationships match success-

fully if the corresponding objects have the corresponding properties and relationships in the context, and a sub-goal's *must not* properties and relationships match successfully if the corresponding objects do not have the corresponding properties and relationships in the context.

However, at least one *must* property or relationship of the sub-goal must match successfully with an addition or at least one *must not* property or relationships must match successfully with a deletion specified by the state-difference, indicating that the sub-goal was not completely satisfied in the initial state of the state-difference and only became satisfied when the state-difference was achieved by the teacher. If all of the core elements (and therefore the goal itself) were initially already satisfied, then the state-difference could not be an instance of the teacher satisfying the sub-goal, so TADPOLE would disallow the match.

5.4 Scoring the decomposition rule matchings

This section describes how the matchings between decomposition rules and sets of state-differences are scored. The scoring differentiates good matches from poor matches and guides TADPOLE's beam search for the best parse of the teacher's demonstration. The details of the scoring mechanism are a minor part of TADPOLE and are not a focus of the thesis. I believe that many alternative weightings and alternative scoring mechanisms would work equally well as long as they took into account the same factors.

5.4.1 The task, sub-goals, variables, and sub-goal dependencies contribute equally to the overall score

The score of the matching between a decomposition rule and a set of demonstrated state-differences depends on how well the rule's head-task matches with its corresponding state-difference and how well each of the

rule's sub-goals match with each of their corresponding state-differences. The score also depends on how consistent the matched state-differences are with the rule's variables, and how consistent the ordering of the sub-goals is with the sub-goal dependency constraints of the decomposition rule.

The task matching and sub-goal matchings all involve matching a graph with a context and a core against a state-difference graph and so their scores are all comparable. However, these graphs can be of differing sizes. A larger graph does not indicate that the sub-goal is more important than sub-goals with smaller graphs, only that it contains more detail, and so it should not contribute more to the final score than the other sub-goals. TADPOLE takes the differing sizes of the sub-goal and head-task graphs into account by scaling their matching scores to be in the range of 0 to 1.

The consistency of the rule's variables and sub-goal dependencies are also important for determining how well the rule matches the demonstrated state-differences. To compute the rule's overall score, TADPOLE calculates an average of the head task score, the sub-goal scores, the variable consistency score and the sub-goal dependencies score so that the last two scores each have a weighting equivalent to a sub-goal or task matching.

5.4.2 The matching score is a weighted average of the frequency counts

The properties and relationships in the head-task and sub-goal graphs of decomposition rules have counts indicating how often they have been present in the instances the rule is a generalization of. The score of a matching of a task or sub-goal to a state-difference is a weighted average of how often the properties and relationships of the state-difference appeared in the task or sub-goal expressed as a fraction between 0 and 1. If a property or relationship has never appeared in any of the instances

the rule is a generalization of, then it gets a frequency count of 1, and so its score will be 1/(the total number of instances).

For example, a node in the head-task of a decomposition rule (generalized from 10 instances) for opening a box could look like:

node1(10)
Type: box(10), container(10)
MadeOf: wood(3), cardboard(7)
Colour: red(2), blue(3), green(3), yellow(2)
Open: yes no

If this task node were matched to the following state-difference node:

box1
Type: box, container
MadeOf: cardboard
Colour: black
Open: yes no

then the frequencies of the properties would be 1.0 twice because the object is a box and a container, 0.7 because it is made of cardboard, 0.1 because its colour is black (a colour the box has never been before), and 1.0 twice because it was closed and then it became open.

The scores for the matchings of different rules that are generalizations of different numbers of instances can be directly compared and combined. It does not matter whether a rule is the generalization of 10 instances or 100, because the score reflects the frequency with which properties and relationships occurred.

5.4.3 The weight of the score of core attributes is double the weight of ones in the context

The core properties and relationships of a task and sub-goal are much more important and much more indicative of a good match than the properties and relationships in the context. To reflect this, the weight of the score of matching a core property or relationship is double that of context properties and relationships.

Note that the frequency score for core properties and relationships will always be either 1.0 or 0.0 because a core property or relationship has to have been present in every instance that the rule is a generalization of. If it is present in the state-difference then the frequency score is 1.0; if it isn't then it is 0.0.

5.4.4 TADPOLE gauges the importance of properties and relationships by their maximum possible score

Not all of the properties and relationships of a task or sub-goal are equally important and they do not contribute equally to the final score. Properties and relationships that do not favour a single value strongly but have had a broad range of different values appear in the different instances the rule is a generalization of are less likely to be important than those that have had the same value in the majority of the instances.

For instance, in the example node given above, the type of the object being a box and a container is likely to be important because each of the 10 instances has had these same values for the Type property. On the other hand, the colour of the box is less likely to be important because in the 10 instances, the object has been 4 different colours without any real consistency.

TADPOLE determines the importance of a property or relationship and the weight of its frequency score (and therefore how much it contributes to the overall score) by the frequency of its most common value. In the

example node given above, the weight of the MadeOf property would be 0.7 because the most common value (cardboard) has a frequency of 0.7, and the weight of the Colour property would be 0.3.

Note that some of the properties of an object are multivalued. The Type property, for example, can have multiple values for a single object (the node given above is both a box and a container). The Colour property, on the other hand, can only have a single value for any given object. TADPOLE treats multivalued properties as if they were separate, independent properties whose weight is equal to their frequency.

In the node to state-difference object matching given above, the score for the node would be: (1.0*1.0[box] + 1.0*1.0[container] + 0.7*0.7[cardboard] + 0.1*0.3[black] + 1.0*1.0*2[yes] + 1.0*1.0*2[no])/(1.0 + 1.0 + 0.7 + 0.3 + 2.0 + 2.0) $\approx$ 0.93 (a very high score).

5.4.5 The variable matching score is the fraction of consistent variables

Decomposition rules usually have at least one variable constraint. When HOPPER uses a decomposition rule to achieve a goal, the rule's variable constraints constrain how the head-task and sub-goals are matched with the state. Nodes in different task and sub-goal graphs that belong to the same variable are constrained to match to the same object in the state.

When TADPOLE parses the demonstration of a teacher, this restriction on having variable nodes all match the same state object is relaxed. The fact that in all the previous instances two nodes in two different subgoals matched to the same state object may have just been a coincidence rather than a requirement, and by parsing a demonstration where such a constraint does not hold, TADPOLE has a way of unlearning an incorrect variable constraint.

Variable constraints form an integral part of a decomposition rule and whether or not a matching with a set of state-differences is consistent with

these constraints affects the final score strongly. In a rule matching, the score for all of the variable constraints is the fraction of variables that are consistent. This score contributes to the final overall score and has a weight that is equivalent to that of a sub-goal matching or task matching. Because decomposition rules tend not to have a large number of variables, this means that the score of a matching with any inconsistent variable constraints will tend to be lowered significantly.

5.4.6 The sub-goal ordering score is the fraction of consistent sub-goal dependencies

Each decomposition rule has a set of sub-goal dependency constraints that constitute a partial order for its sub-goals. This set of sub-goal dependencies consists of those that held in every instance the rule is a generalization of. If the rule matching involves a new ordering of sub-goals not previously seen in any of the instances the rule is a generalization of, then at least some of its sub-goal dependencies will be violated.

The order the sub-goals of a decomposition rule are achieved in is often critically important, and instances that have a very different ordering compared to what TADPOLE has seen before will decrease the matching score significantly. The score for the sub-goal ordering in a decomposition rule is the fraction of sub-goal dependencies that are consistent with the example. This score contributes to the final overall score and has a weight that is equivalent to that of a sub-goal matching or a task matching.

5.5 Refining decomposition rules

At the end of the teacher's demonstration, if the highest scoring parse on the beam has a high enough score, then TADPOLE uses it to refine the decomposition rules it used to construct the parse.

5.5.1 Matching state-differences are instances of decomposition rules

A decomposition rule used by TADPOLE and HOPPER is a generalization of the set of instances of when that decomposition rule was successfully achieved. Examples of the decomposition rule being successfully achieved come from either HOPPER or from the examples of a teacher that are parsed by TADPOLE. When TADPOLE matches a decomposition rule with a set of demonstrated state-differences, the state-differences are assumed to be an instance of the decomposition rule. When TADPOLE refines the rule, it generalizes the rule to take account of the new instance.

5.5.2 TADPOLE drops the non-matching core elements of a refined task

The core of a head-task specifies what was added and deleted from the state in every instance when the decomposition rule was achieved. When the task is refined with a new instance, any addition or deletion that does not appear in the state-difference must have been noise or an unimportant side-effect in previous instances and is dropped from the core of the task.

Every property and relationship deletion in a task had to have been present in the initial state of every state-difference instance the rule is a generalization of. The deletions TADPOLE determined were noise or unimportant side-effects and so did not belong in the core of the task could nevertheless be important parts of the precondition that happened to have been undone by the time the decomposition was completely achieved. To account for this, TADPOLE moves these former deletions to the context of the task and they become part of the task's precondition. Because these properties and relationships appeared in every instance so far, they each receive a count equal to the number of instances the rule is a generalization of.

Because the core additions of a task are never present in the initial state

of the state-difference instance of the rule, they do not belong in the pre-condition of the task. So if TADPOLE determines that these additions are noise or side-effects, it simply drops them from the core of the rule.

5.5.3 TADPOLE drops the non-matching core elements of a refined sub-goal

The core of a sub-goal specifies what had to be true and what had to be false in the state in every instance when the sub-goal of a decomposition was achieved. When the sub-goal is refined with a new instance, any core property or relationship that is not consistent with the state-difference is not a definite requirement and is dropped from the core of the sub-goal.

When the teacher achieves a sub-goal, they only need to achieve the aspects of the sub-goal that are not already true. The changes achieved by the teacher in the state-difference the sub-goal is matched against only have to make the unsatisfied parts of the sub-goal true without affecting the satisfied parts. This means that a core property or relationship is consistent with a state-difference if it is achieved by the additions or deletions of the state-difference, or if it already holds in the context of the state-difference.

A *must* property or relationship in the sub-goal's graph holds if it appears in the context of the matched state object or relationship (as appropriate) or if it is added by an addition specified in the state difference. If a *must* property or relationship does not hold, then TADPOLE removes it from the sub-goal's core and moves it into the context. Because it held in every previous instance, the property or relationship gets a count equal to the number of instances the rule is a generalization of.

A *must not* property or relationship in the sub-goal's graph holds if it does not appear in the context of the matched state object or relationship (as appropriate) or if it is deleted by a deletion specified in the state difference. If a *must not* property or relationship does not hold (*i.e.* if the

property or relationship appears in the context or in an addition), then TADPOLE removes it from the sub-goal's core. TADPOLE does not move the property or relationship in question to the context, because the context of a sub-goal consists only of the properties and relationships that potentially should also be true and not those that should potentially not be true.

5.5.4 TADPOLE updates the counts of properties and relationships in the context

After refining the core of a task or sub-goal and after moving any requisite core properties and relationships into the context, TADPOLE refines the context. For every node in the task or sub-goal graph, TADPOLE increases the count of every property that also appears in the state object the node is matched with and it adds a new property with a count of 1 for every property in the state object that does not appear in the node. Similarly, for every link in the task or sub-goal graph, TADPOLE increases the count of every relationship that also appears in the state relationship it is matched with, and adds a new relationship with a count of 1 for any new state relationship that does not appear in the link. If the link is not matched with a state relationship (there exists no relationship between the two state objects the two task or sub-goal nodes at either end of the link are matched with), then TADPOLE does not modify the link. Note that this will implicitly lower both the score of the link and its importance in future matchings, because the number of instances of the rule has increased by 1 while the counts of the relationships of the link have not, decreasing their frequency and importance scores as described in Section 5.4.

5.5.5 TADPOLE drops any sub-goal dependencies not consistent with the instance's sub-goal ordering

Each decomposition rule has a set of dependency constraints between its sub-goals specifying for each sub-goal which sub-goals ought to be achieved before it. When TADPOLE refines a decomposition rule with new instances, TADPOLE keeps only the sub-goal dependencies that are consistent with all of the instances the rule is a generalization of. The reason that TADPOLE does not maintain a count of how many instances each sub-goal dependency is consistent with is to maintain a single consistent set of sub-goal dependencies. This information is necessary to generate a partial ordering of the rule's sub-goals which is necessary for HOPPER to interleave its plan as described in Chapter 4.

A sub-goal dependency constraint on two sub-goals specifies that the constraining sub-goal should be satisfied before the dependent sub-goal is achieved. A sub-goal dependency is inconsistent with a decomposition matching if the constraining sub-goal is not satisfied in the initial state of the state-difference the dependent sub-goal is matched with. Furthermore, any instance that does not satisfy the ordering constraint is considered to be sufficient evidence that the ordering is not a necessary constraint, so when TADPOLE refines a decomposition from a decomposition matching, it drops any inconsistent sub-goal dependency constraints.

5.5.6 TADPOLE splits inconsistent variables into two or more variables

Every decomposition variable constrains at least two nodes in the head-task and sub-goals of a decomposition rule to bind to the same state object. If this constraint is violated in the teacher's demonstration, then the constraint must really have been just a coincidence where the nodes happened to bind to the same state object and TADPOLE drops the variable. However, TADPOLE must account for the fact that two or more variables

may be hidden beneath a single variable. If these variables, happened to have been bound to the same state object in previous instances, then they would have been indistinguishable from a single variable. TADPOLE addresses this issue by splitting any inconsistently matched variable into two or more new variables depending on how many groups their nodes form, where a group of nodes consist of those that are matched to the same state object.

For example, if a rule had a variable that constrained 5 nodes to bind to the same state object, and in the teacher's demonstration two of the nodes matched to one state object, two matched to another state object, and the fifth matched to a third state object, then TADPOLE would split the original variable into two new variables each constraining one of the two groups of nodes. TADPOLE does not create a new variable to account for any group of nodes of size 1 because a decomposition variable must constrain at least two nodes.

5.5.7 TADPOLE modifies its decomposition rules gradually

TADPOLE can modify its decomposition rules significantly by refining them from observed examples demonstrated by the teacher. The refinements include:

- adding new properties and relationships and modifying the importance of the old properties and relationships in the tasks and subgoals of its decomposition rules
- removing incorrect core properties and relationships
- generalizing the dependency constraints of the sub-goals of its rules
- dropping old variables and learning new ones.

However, any rule matching with an instance that would result in any significant change if the rule was to be refined would necessarily have a significant penalty in its matching score as described in Section 5.4. This means that such a rule matching is unlikely to end up in TADPOLE's final parse and is therefore unlikely to be refined. This ensures that TADPOLE will modify its decomposition rules only when it is confident that it matched the correct rule and that it matched it correctly to the state-differences demonstrated by the teacher.

In general, TADPOLE will significantly modify only one aspect of a decomposition rule at a time, leaving the rest of the decomposition unchanged. This is consistent with TADPOLE's assumption about the teacher that they demonstrate only a limited amount of novel information at a time and in the context of familiar rules as described in Section 5.1.

5.6 Parsing interleaved, partially-achieved, and repeated decompositions

When the teacher demonstrates novel rules, TADPOLE can expect the teacher to abide by the felicity conditions described in Section 5.1 and demonstrate lessons that are easy to parse. In such lessons, TADPOLE can expect each novel decomposition to appear in its entirety and have all of its sub-goals achieved in the parsed hierarchy. TADPOLE can also expect novel decompositions to appear individually and independently of the other decompositions in the parse.

However, it is not reasonable for TADPOLE to expect the teacher to abide by such stringent constraints when demonstrating rules that make up the context surrounding the novel rules, or when the teacher presents reinforcing lessons that do not contain any novel rules at all. Though it is reasonable for the teacher to take care in being clear when demonstrating novel rules, novel rules generally make up only a small part of

the parsed decomposition hierarchy, and requiring the same kind of care when demonstrating the remaining, familiar decompositions is difficult and tedious. This problem is exacerbated in a deeper decomposition hierarchy because the deeper the hierarchy, the more familiar decomposition rules it will contain.

The teacher should not be restricted in this way when demonstrating familiar rules for three reasons:

- It is much more efficient to achieve decomposition rules in parallel whenever possible. This is the reason that HOPPER interleaves the execution of its decomposition rules, and it is reasonable for the teacher to also interleave the execution of any familiar decompositions in the demonstration.

- Whenever decomposition rules are used to generate a goal decomposition hierarchy, many of the decompositions within the hierarchy will have some of their sub-goals initially achieved and so these sub-goals will be absent from the hierarchy. It is very tedious for the teacher to make sure that every decomposition in a lesson has all of its sub-goals present in the decomposition hierarchy. To ensure this, the teacher would have to verify that none of the sub-goals of any decomposition is satisfied when the teacher begins to execute it. Not only is this difficult, but it also greatly limits the kinds of lessons the teacher can demonstrate to those that take place in states satisfying such a rigid constraint.

- There are decompositions that have to be executed multiple times before their head-task is achieved. The difficulty with parsing such a decomposition lies in the fact that the number of times the decomposition has to be executed can vary. The teacher may not be able to predict how many times they will need to execute the decomposition, and if they could somehow fix the number of times they execute

the decomposition in every lesson, TADPOLE would end up learning a decomposition rule that had a fixed, repeated set of sub-goals rather than the more flexible rule that keeps achieving its sub-goals until it is itself achieved.

TADPOLE must therefore be able to parse familiar rules that are interleaved with each other, familiar rules that are initially partially achieved and do not have all of their sub-goals represented in the decomposition hierarchy, and familiar rules that are repeatedly executed.

Being able to parse these kinds of decompositions not only eases the burden of the teacher, but it also has the important benefit of allowing TADPOLE to parse the behaviour of other, non-teacher agents. If TADPOLE cannot depend on the teacher being extra helpful when parsing familiar rules, then it definitely cannot make such assumptions about other agents that are not constrained to follow any felicity conditions at all. The parsing techniques described in this section are also useful for parsing and learning from the behaviour of non-teacher agents so long as they do not utilize novel rules TADPOLE is not yet aware of.

5.6.1 TADPOLE parses contiguous sub-goals of interleaved decompositions

When extendParse searches for a way to match the root nodes of a partial parse to the sub-goals of decomposition rules, it allows multiple sub-goals from different decomposition rules (and different sub-goals from the same decomposition rule) to match with the same state-difference node forming separate partial decomposition matchings that have nodes in common. extendParse also does not require that the matched sub-goals of one partially matched decomposition be contiguous with each other: they can be separated by one or more state-difference nodes not matched to any of the sub-goals of the decomposition.

When getNeighbouringParses looks for extensions of a partial parse, it goes through all of the completed decomposition matchings (those that do not have any sub-goals left to bind to a root state-difference node) and tries to find a combination of them that together form a viable parse. For every combination, if the completed decomposition matchings form a viable parse, then getNeighbouringParses adds the resulting partial parse to the beam and continues its search.

In order to be a viable parse, a set of decomposition matchings must satisfy the following conditions:

- Every decomposition matching must be complete.
- The decomposition matchings must be contiguous with each other. The set of root state-difference nodes the sub-goals of the decomposition matchings are matched with must not have any other root nodes between them.
- Each decomposition matching must match at least one of its sub-goals with a root state-difference node that none of the other decomposition matchings have matched with. A decomposition whose sub-goals are all achieved in parallel with the sub-goals of other decompositions is completely superfluous.
- Each decomposition matching must match one of its sub-goals with at least one root state-difference node that is matched by at least one other decomposition matching and therefore is achieved in parallel with the sub-goal(s) of another decomposition. If none of the sub-goals of a decomposition are achieved in parallel, then there is no reason to interleave that decomposition. Note that if TADPOLE learns in a domain where decompositions are interleaved without having any of their sub-goals achieved in parallel, then this constraint can be relaxed.

- Each decomposition matching must be matched in such a way that its variables and all of its sub-goal dependency constraints are consistent. If a decomposition matching violates any of the rule's variable or sub-goal dependency constraints then this an indication that the wrong rule has been matched (or the right rule has been incorrectly matched) or that TADPOLE has not learned the rule completely; in either case, TADPOLE should not proceed with the parse.

- As with a single completed decomposition matching, to prevent duplicate partial parses being added to TADPOLE's beam, at least one of the decomposition matchings must match one of its sub-goals with the latest root state-difference node.

These constraints not only ensure that getNeighbouringParses parses interleaved decompositions correctly but it also prunes the search space of partial parses. If getNeighbouringParses tried to interleave every set of contiguous partial parses, then the parse search space would greatly increase. The constraints on parsing interleaved decompositions keeps the search space tractable so that getNeighbouringParses tries interleaving only promising decompositions.

Once getNeighbouringParses calls getNeighbour with a viable set of interleaved decomposition matchings (note that in most cases, the viable set of decomposition matchings will correspond to a single, uninterleaved decomposition matching), getNeighbour parses the matched root state-difference nodes in the same manner as described in Section 5.2.4 except that it adds a new root state-difference node for each interleaved decomposition matching. getNeighbour adds the new root state-difference nodes in an arbitrary order but it notes that the order is in fact arbitrary. If these new root state-difference node are later parsed to be sub-goals of the same higher-level decomposition, then there will be no ordering constraint present between these sub-goals.

5.6.2 TADPOLE identifies already achieved sub-goals if their variables are bound

When a decomposition is executed (either by the teacher or another agent) often some of its sub-goals will already be true in the state and will not need to be achieved. In order to be able to parse the teacher's lessons and especially the behaviour of other agents, TADPOLE must be able to parse decompositions that have some sub-goals already achieved.

It is not feasible for extendParse to search for every way that every sub-goal of every decomposition could have been initially achieved. This is especially true in states with hundreds of objects. Sub-goals that are already true in the state are not achieved by the teacher (or other agent) and so there are no state-differences to guide the search making the matching very expensive.

extendParse resolves this problem by calling matchToAlreadyAchieved (A.2.7) that takes advantage of a decomposition's variables. A decomposition's sub-goal will have many if not all of its nodes constrained by variables that also constrain the nodes of the other sub-goals. If the other sub-goals have already been matched with their respective state-difference nodes, then they have bound the variables that constrain them. TADPOLE can then make use of these bindings to constrain the matching of any unmatched sub-goals with the state to determine whether it was initially already achieved.

matchToAlreadyAchieved checks for initially achieved sub-goals only in partial decomposition matchings that are consistent with the decompositionl's variables and the decomposition ordering constraints. This is to ensure that the rule in question is well-learned and matchToAlready-Achieved can correctly match any potentially already achieved sub-goals with the state. It is especially important that the decomposition's variables are correct because the sub-goal to state matching depends on this. It is also important that the sub-goal's context is also accurate because it

guides the matching of the sub-goal's nodes that are not constrained by the variables.

When matchToAlreadyAchieved searches for a way to extend partial decomposition matchings it goes through all of the consistent partial decomposition matchings and matches any of their unmatched sub-goals that have all of their variables completely bound (all of the variables that constrain any of the sub-goal's nodes are bound by other sub-goals that are already matched with state-difference nodes). Note that this does not include variable bindings between the sub-goal and the task (which has not been matched yet). matchToAlreadyAchieved matches any remaining unbound objects in these sub-goals with the state the decomposition was initially executed in (the first state of the first state-difference node the decomposition is matched with).

If matchToAlreadyAchieved determines that a sub-goal of a partial decomposition matching is initially satisfied, then it adds a new partial decomposition matching to the beam of the partial parse. Just because a sub-goal is satisfied at the beginning of the execution of a decomposition does not mean that it is not reachieved later within the decomposition. The sub-goals of decompositions often interact, achieving and unachieving each other. Therefore extendParse keeps track of both possibilities on its partial decomposition matching sub-beam, one in which the sub-goal is initially achieved, and one in which the sub-goal remains unachieved to await the rest of the teacher's demonstration.

For example, the decomposition for loading a package into a truck involves opening the truck, loading the package, and closing the truck. The final sub-goal of ensuring that the truck is closed will usually be initially achieved (when the truck is closed). However, because this sub-goal is undone when the first sub-goal of having the truck open is achieved, the final sub-goal will have to be re-achieved despite the fact that it was satisfied in the initial state.

If matching a sub-goal to the initial state of a partial decomposition

matching completes the matching (because the sub-goal was the last unmatched sub-goal of the matching), then getNeighbouringParses generates an extension of the partial parse and adds it to the beam as normal.

Note that matchToAlreadyAchieved only checks for initially achieved sub-goals in new partial decomposition matchings that have at least one of their sub-goals matched with the latest state-difference node to prevent duplicate partial parses on its beam as described in Section 5.2.7.

5.6.3 TADPOLE combines identical adjacent decompositions into a single repeated one

There are decompositions that need to have their sub-goals achieved (in a given order) a number of times before their head-task is achieved (for example, stirring a cup of tea until the sugar is dissolved). The exact number of times the decomposition will have to be executed is unpredictable and cannot be determined beforehand. This means that parsing such decompositions is not straightforward because they will be executed a different number of times in different states.

When the same decomposition is parsed multiple times in sequence, and each decomposition is matched to the same state objects, then getNeighbour treats this as a single repeated decomposition. It parses all of the matched state-difference nodes and creates a new root state-difference node that is the difference between the initial state of the entire sequence and the final state of the entire sequence. It is important that the corresponding nodes in each of the decomposition matchings match with the same state objects, because decompositions that are matched with different state objects achieve different goals and should not be collapsed into a single repeated decomposition.

getNeighbour will not parse a repeated decomposition if the decomposition matchings in the sequence have any inconsistent sub-goal dependency constraints and especially not if they have any inconsistent variable

constraints. getNeighbour must be confident that the rule it has learned is correct and that all of the decomposition matchings in the sequence indeed refer to the same decomposition rule before it will commit to parsing them as a single repeated decomposition.

5.7 Learning New Rules

When parsing the teacher's demonstration, TADPOLE will sometimes be unable to parse parts of the goal-decomposition hierarchy. This is either because it has no rules to match a given sequence of state-differences or because any rules that do match, match so poorly that it is better off not parsing the sequence at all. At the end of the lesson, if the best scoring partial parse has these kinds of holes in the parsed goal-decomposition hierarchy, then TADPOLE learns new decomposition rules to fill these holes.

There are two kinds of holes that TADPOLE could have in its goal-decomposition hierarchy: a hole at the very top of the hierarchy which exists if the hierarchy has more than one root state-difference node, and a hole at a lower level of the hierarchy where TADPOLE can determine from the surrounding context that a sequence of state-differences achieve a sub-goal but TADPOLE does not have an appropriate rule to parse them.

5.7.1 TADPOLE learns a new rule if the final parse has multiple root nodes

The easiest and most logical way for the teacher to instruct TADPOLE and teach it a new decomposition rule is to simply demonstrate an instance of the decomposition rule being executed. Because the decomposition is at the highest level of the decomposition hierarchy, this results in a partial parse that at the end of the teacher's lesson remains incomplete, with a number of unparsed, root state-difference nodes.

If at the end of the teacher's lesson, the best partial parse (the one

with the highest score) has more than one root node, then learnLesson calls completePartialParse (A.2.13) to construct a new rule from these top nodes. It constructs the rule by inferring both the head-task and the sub-goals of the rule.

Each state-difference node and the sub-hierarchy below it is a way of achieving a sub-goal of the new decomposition. The sub-goal was not satisfied before the sub-hierarchy of the state-difference node was executed and it was satisfied afterward, and the state changes specified by the state-difference node are useful for determining what the sub-goal being achieved was. At least some of the changes specified in the state-difference node were caused by the teacher and at least some of those changes were necessary to satisfy the sub-goal.

completePartialParse learns a first approximation of a new sub-goal from its corresponding state-difference node. It determines what *must* be true from the additions and what *must not* be true from the deletions. Note that completePartialParse only considers the relevant changes, those pertaining to objects that were matched with nodes in the tasks matched with the state-difference nodes of the decompositions one level lower in the decomposition hierarchy as described in Section 5.3.6.

completePartialParse generates the sub-goal's context from the remaining properties and relationships that remained unchanged in the state-difference node, but it includes only those that are within a threshold distance from the relevant state changes. It is even more important to limit the size of sub-goals than it is the size of state-differences they match with. Because of this, completePartialParse uses an even stricter threshold of 1 to limit the size of newly learned sub-goal graphs, so that it includes only the objects that changed and their direct relationships. It initializes the properties and relationships in the newly generated context with a count of 1.

completePartialParse calculates the head-task of the new decomposition rule by finding the difference between the initial state and the final

state. Again, it includes only the relevant state changes and it generates the head-task's context in the same manner as described above.

completePartialParse learns the initial sub-goal dependencies by determining, for each sub-goal, which other sub-goals were already satisfied in the state just before the teacher achieved the sub-goal (the initial state of the state-difference node that the sub-goal was calculated from).

completePartialParse learns the initial variables of the new decomposition rule cautiously. It groups all of the nodes of the newly learned task and sub-goals and generates a variable constraint for each group of nodes that matched the same state object. Although this creates an initial decomposition rule that is very constrained, the variable constraints will be rapidly relaxed when the rule is refined from subsequent examples.

completePartialParse learns a new decomposition rule in this way when the best scoring partial parse on the beam is incomplete and has multiple root nodes. Note that there may be other partial parses on the beam that are completed, but TADPOLE considers only the highest scoring partial parse once the teacher's lesson is concluded. Figure 5.4 shows a graphical representation of TADPOLE learning a new rule in this way.

5.7.2 The context of sub-goals is extended to include important relationships

When the agent learns new sub-goals it drops objects and relationships farther than a threshold distance from the observed state changes. Though this is an effective way of dropping irrelevant parts of the goal, sometimes relevant relationships with important objects in the decomposition may be dropped.

For example, to pick up a cup from within a closed cupboard the agent must first open the cupboard, but it is critically important that the cupboard the agent opens is the same one that the cup is in! The first sub-goal of the appropriate decomposition involves opening a container. However,

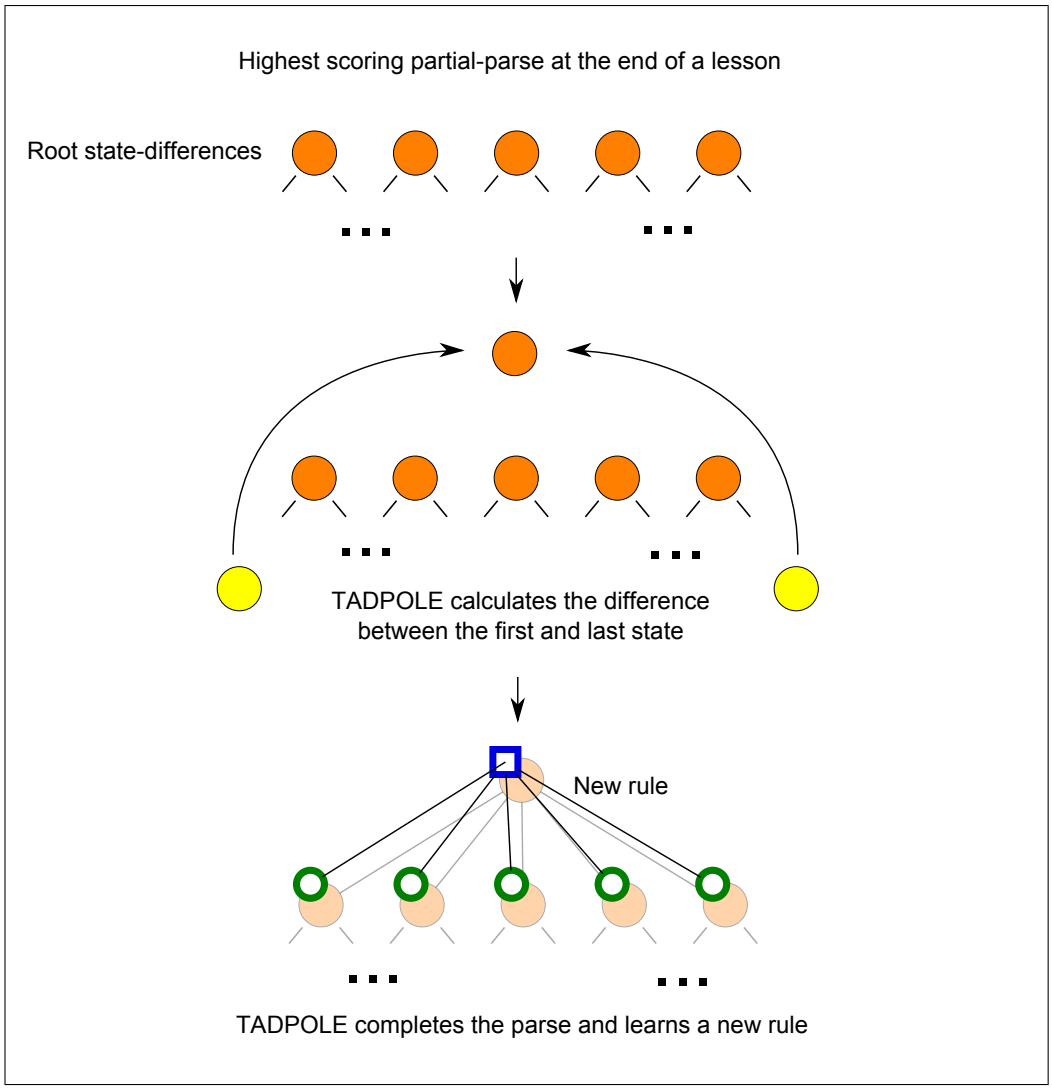


Figure 5.4: TADPOLE learning a new rule by completing a parse

the state change this goal was learned from does not directly involve the cup that is about to be picked up, because only the cupboard changed from being closed to being open.

In order to capture the critical relationship that the cupboard being opened must contain the cup being picked up, the threshold distance would have to be increased to include all the objects directly related to the core components and all the objects directly related to those objects. In this example, in order to include the critical Contains relationship between the cupboard and the cup by increasing the threshold, all the other relationships and objects involved with the cupboard and cup would also have to be included. Such an extension of the threshold would include all of the other irrelevant items within the cupboard as part of the sub-goal. Increasing the threshold distance exponentially increases the number of (mostly irrelevant) objects included in the goal which is extremely undesirable as discussed above.

To include important relationships when learning a new rule without extending the threshold, completePartialParse extends each sub-goal graph with all of the relationships between its nodes and state objects that correspond to decomposition variables. In the example above, the cupboard would be matched in all three sub-goals of the decomposition (it is opened, emptied, and then closed), and the cup would be matched in the second sub-goal and the head-task of the new rule, and so both of these objects would correspond to decomposition variables. completePartialParse would extend the first sub-goal of the newly learned rule to include the Contains relationship between the cupboard and the cup (which corresponds to a decomposition variable). Note that it would not extend the second or third sub-goals because the second will already include a core relationship between these two nodes (the cup is no longer In the cupboard) and there is no direct relationship between the cupboard and the cup in the third state-difference graph.

If completePartialParse drops a variable after refining a decomposition

rule with a new example, then it also drops any of its additional relationships that the rule's sub-goals were extended with.

5.7.3 TADPOLE does not learn if it is not confident about its final parse

Updating incorrectly matching rules is clearly detrimental. Not only will the context be skewed; but because TADPOLE drops mismatching variable constraints, sub-goal dependency constraints, and task and sub-goal core constraints; such critical information may be lost completely if TADPOLE refines an incorrectly matching rule. Learning a new rule incorrectly is even more detrimental: the incorrectly learned rule will not only not help HOPPER achieve its tasks, but it will also make it more difficult for TADPOLE to parse future lessons correctly.

To prevent this, TADPOLE learns cautiously, and it requires that the best matching parse it finds at the end of the teacher's lesson have a score that is above a set threshold. The threshold value is a parameter that can be set depending on how reliable the teacher is. Note that this also applies to the learning of new rules (as described in Section 5.7): if the score of the best parse does not exceed the threshold, then TADPOLE will also not learn any new rules from the parse.

TADPOLE also does not learn new rules if the final parse has too many root state-difference nodes. Decomposition rules are unlikely to have more than 7 or 8 sub-goals, so a final parse that has 15 or more root state-difference nodes is a strong indication that TADPOLE did not parse the demonstrated lesson correctly. In such cases, TADPOLE is too confused about the teacher's lesson, and it does not learn any new rules or refine any of its old ones.

TADPOLE gives partial parses with fewer root nodes higher scores

TADPOLE parses demonstrated behaviour online. It begins to parse as soon as a lesson begins and it does not know when it will end. For any partial parse on the beam, at any point in the parse, the lesson could potentially end in the next time slice. If that were to happen, then TADPOLE would generate a new decomposition rule from the root state-difference nodes (assuming the best partial parse had a high enough score). New decomposition rules have a low score to discourage TADPOLE from learning spurious rules as described below in Section 5.7.4. The introduction of a new decomposition rule to the top of parse would decrease the score of the overall matching. The more unparsed root state-difference nodes, the larger the new decomposition rule, the greater the weights of its score, the greater the decrease of the overall score would be.

TADPOLE takes account of the fact that the lesson could end at any point and decreases the score of every partial parse on its beam to what it would become if the lesson ended in the next time slice. This has the effect of lowering the score of partial parses with more root nodes by a greater amount, and biasing the search toward partial parses that are more completely parsed and have fewer root nodes.

TADPOLE greatly benefits from this because at any time slice, no matter how deep the parsed hierarchies are, the partial parses on its beam will tend to have a small number of root nodes. Because at each time slice, TADPOLE only works on the root level of the partial parses and there are never more partial parses than the width of the beam, the cost of the algorithm will tend to be uniform for each time slice, and the overall cost of the algorithm is linear in the length of the demonstration. Note that this does not consider the increasing cost of searching for matching decompositions as TADPOLE learns more rules (this limitation is discussed in Section 5.8).

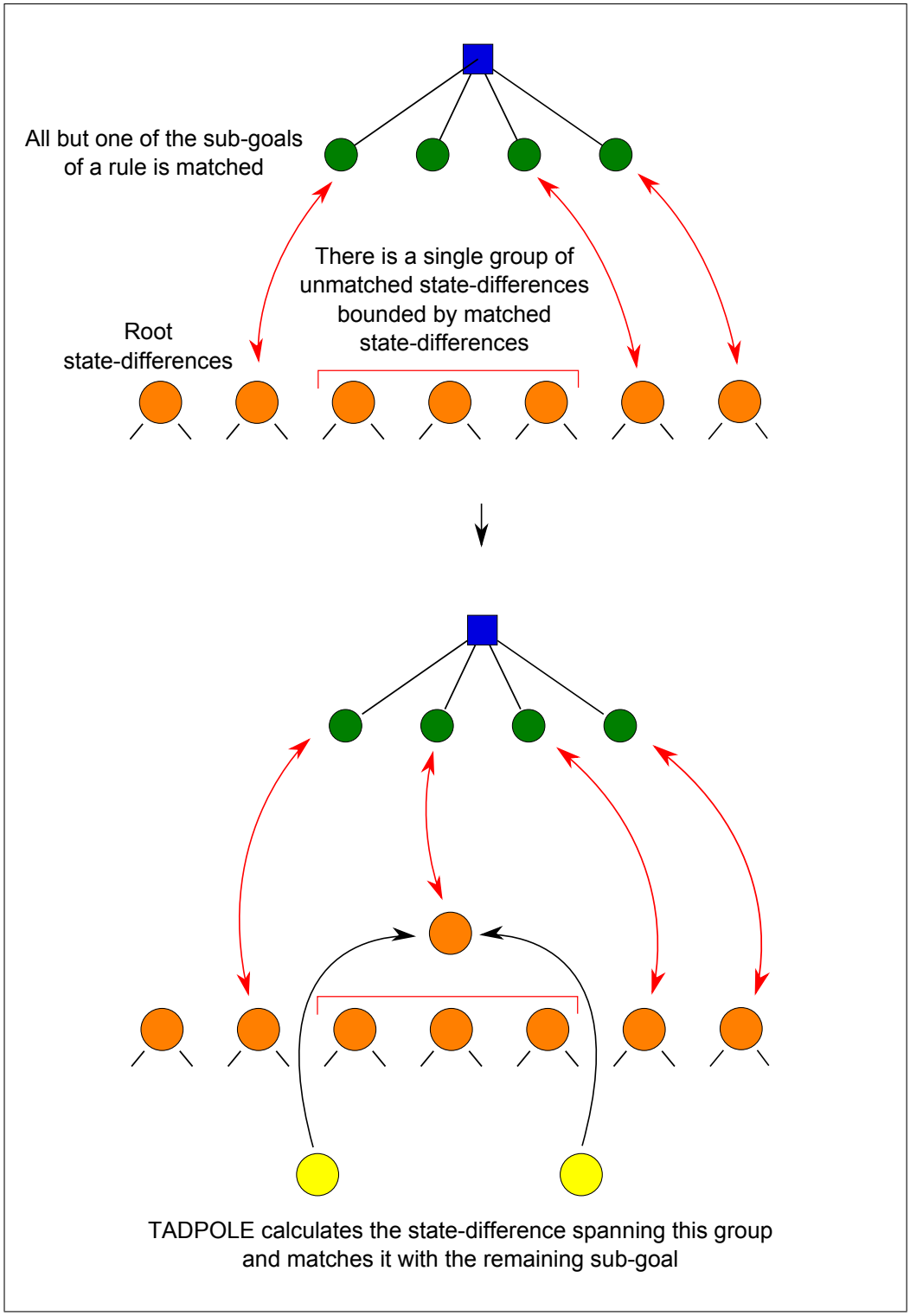
5.7.4 TADPOLE learns a new rule that would parse a new way of achieving a sub-goal

A second way that TADPOLE learns a new rule is if it observes a new way of achieving a sub-goal lower down in the hierarchy. Because the decomposition for achieving the sub-goal is a new one (TADPOLE has not learned it before), TADPOLE will not be able to parse it, but it can deduce the new rule from the surrounding context that it has parsed. If the parent decomposition has only a single remaining unparsed sub-goal and the other parsed sub-goals form two groups separated by unparsed state-difference nodes, then TADPOLE assumes that the head-task of the new decomposition achieves the unparsed sub-goal, and it assumes that the sub-goals of the new decomposition correspond to the unparsed state-difference nodes separating the two groups of parsed sub-goals of the parent decomposition.

TADPOLE can complete a partial decomposition matching by learning a new rule to parse its sub-goal

getNeighbouringParses incorporates this learning mechanism by calling getHoleFillingNeighbours (A.2.11) which completes a partial decomposition matching by bridging two groups of the decomposition's parsed sub-goals with a new decomposition rule that would parse its remaining sub-goal. As described in Section 5.2.4, a completed partial decomposition matching results in a new root state-difference node and an extension to a partial parse that TADPOLE adds to its beam.

getHoleFillingNeighbours constructs a new rule from the state-difference nodes the same way completePartialParse does for top-level new rules described above. Note that the boundaries of the new decomposition have to be delineated by the parsed sub-goals of the parent decomposition. This means that getHoleFillingNeighbours cannot learn new decompositions if the missing sub-goal is the first or the last one of the parent decomposition.


All but one of the sub-goals
of a rule is matched
There is a single group of
unmatched state-differences
bounded by matched
state-differences
Root
state-differences
TADPOLE calculates the state-difference spanning this group
and matches it with the remaining sub-goal

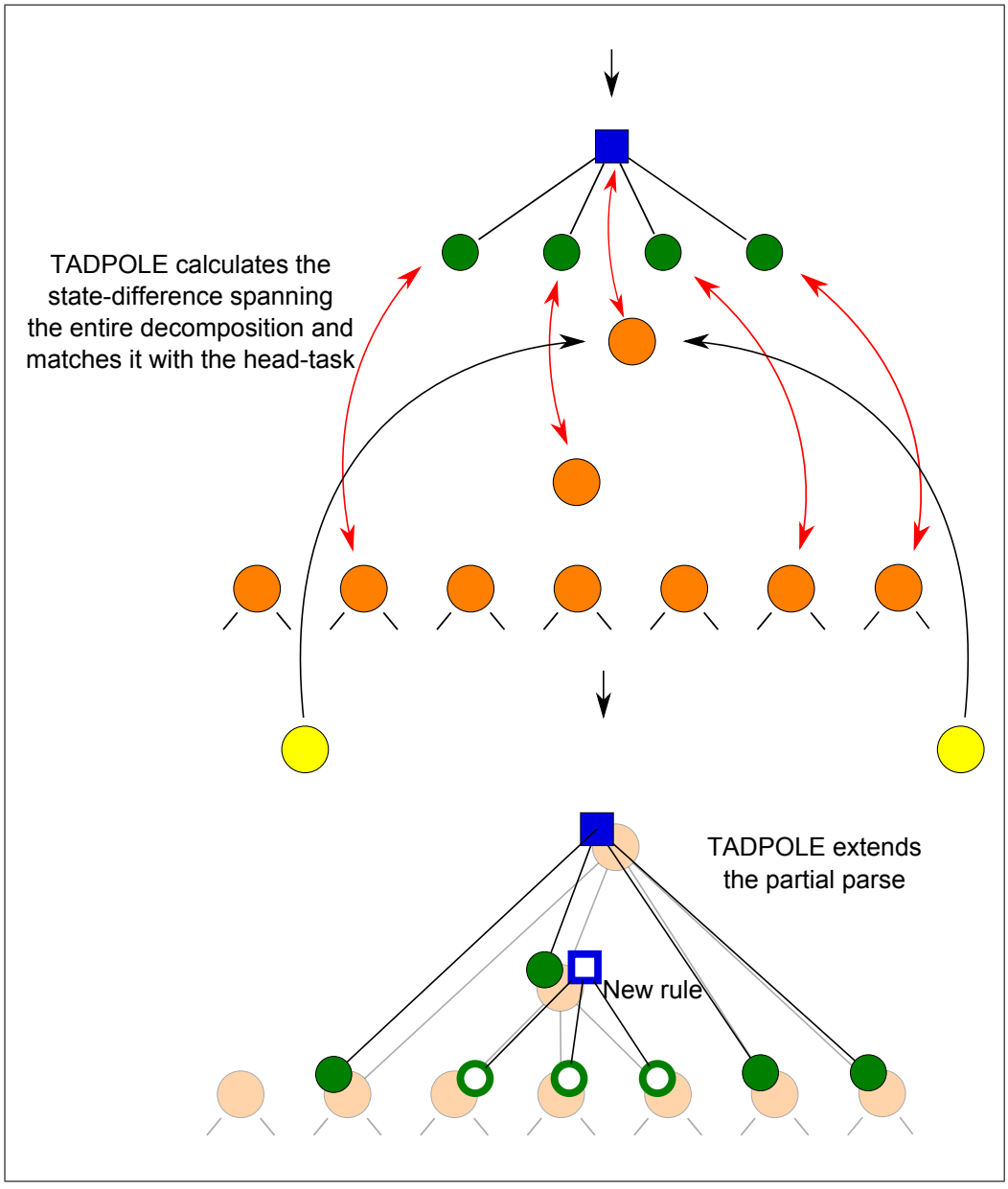


Figure 5.5: TADPOLE learning a new rule by filling a hole in the parse

Section 5.8 covers this in more detail.

Spuriously learned rules are detrimental to TADPOLE

If TADPOLE misparses the teacher's demonstration and learns an incorrect rule, then not only does it not learn the correct new rule(s) intended by the teacher, but the new incorrect rule will tend to disrupt future lessons as well.

Spurious rules negatively affect TADPOLE directly because every additional such rule is another rule TADPOLE has to search through in future matchings. Spurious rules that are similar enough to another rule TADPOLE has learned, may also be matched and refined with some of the future lessons intended for the other rule. TADPOLE would end up splitting its knowledge of the decomposition between two rules resulting in two moderately learned rules instead of one well learned one. Splitting knowledge about a decomposition among multiple rules is an issue that needs to be addressed (see Section 5.8), but it is particularly likely to happen with spuriously learned new rules because they do not represent any real decomposition used by the teacher and so they cannot make use of any examples to distance themselves from the rule they are mimicking.

Parsing holes within the hierarchy is most useful for learning from non-teacher agents

Demonstrating a new decomposition rule by embedding it within the goal-decomposition hierarchy is a cumbersome way to teach it. If the teacher wishes to demonstrate a new decomposition rule, then demonstrating it at the top of the hierarchy is a better approach because it requires a much smaller decomposition hierarchy (there is no context above the hole) and much less effort on the part of the teacher. TADPOLE is most likely to encounter a new decomposition rule in the middle of a demonstrated parse if the demonstration was not primarily intended to teach that rule.

This makes this method particularly useful for learning new decomposition rules from non-teacher agents whose primary purpose is not to instruct TADPOLE. However, in such cases, because TADPOLE cannot rely on any felicity conditions, it needs to be confident that it has used the correct rules to parse the agent's observed behaviour correctly so that it does not learn spurious rules.

The parent decomposition must be well-parsed

To minimize the risk of learning spurious rules, getHoleFillingNeighbours will only learn a new decomposition rule from a hole in the middle of a parsed decomposition hierarchy if it is confident that it has parsed the parent decomposition correctly. All of the parent decomposition's variable constraints, sub-goal dependency constraints, and all of the task and sub-goal core properties and relationships must be consistent with the demonstrated state-differences. The parent decomposition must also not have more than one unparsed sub-goal. Although these are very stringent constraints, getHoleFillingNeighbours cannot rely on the teacher's felicity conditions to facilitate learning because the observed behaviour may not have been generated by a teacher, and getHoleFillingNeighbours must ensure that its rules are correct and it has parsed the behaviour correctly before committing to learning a new rule.

Newly learned rules have a low score in the parse

New decomposition rules that fill holes in the middle of the parsed decomposition hierarchy contribute to the score of the partial parse they are part of. The score of other decomposition matchings in the partial parse reflect how well TADPOLE's rules match the demonstrated state-differences; because the new decompositions are new, their score cannot be determined in this way and is instead a parameter of TADPOLE that determines how confident TADPOLE is in learning new rules. However, initially, newly

learned rules will tend to be over-specialized to the particular instance they were learned from, so the score for newly learned rules should be relatively low. In the evaluation of TADPOLE covered in Chapter 6, this parameter was set to 0.5. Such a low score ensures that the rest of the decomposition hierarchy is parsed very well before new rules are included in the parse. This reduces the chances of learning a spurious rule.

5.8 Limitations and Future Extensions

This section describes the various limitations of TADPOLE, the issues that will need to be addressed if it is to scale completely to the Human Planning Domain, and various ways of extending and improving the algorithm.

5.8.1 TADPOLE cannot refine its rules to add additional relevant objects

The primary reason that TADPOLE learns and refines its rules by matching them with state-differences is so that it can identify the relevant and irrelevant objects in the state. Although this heuristic of determining object relevance by distance from properties and relationships that changed is very effective, it does not guarantee that all of the irrelevant objects will be eliminated from the learned rules or that all of the relevant objects will be included. As TADPOLE sees more demonstrated examples it can gradually learn which of the rule's properties and relationships are significant. However, if an important object is so distant from the state changes that it is not even included in the rule's task or sub-goal graphs, then TADPOLE has no way of learning that it is significant to the rule.

For example, whether or not an electrical switch is turned on is critically important to the working of a microwave, even though the state of the switch does not change and it is only distantly related to the contents of the microwave which do change in the state graph. But there is no *a*

priori way for TADPOLE to know that the electrical switch is important, so any rules that TADPOLE learns about heating objects in the microwave will not include the switch in their task and sub-goal graphs no matter how many examples of heating TV dinners it sees.

Extending the size of the task and sub-goal graphs is not a viable solution because important objects like a switch or a fuse-box may be arbitrarily distant from the state changes being learned, and extending the graphs to include all of the distantly related objects would make the graphs so large that the algorithm would become intractable. It would also make the algorithm inaccurate because most of the added object nodes would be irrelevant and their score would only swamp the score of the relevant nodes.

There are two ways of extending TADPOLE to address this problem:

- Include a more in depth understanding of the physics of the domain. If TADPOLE knows that electrical appliances are plugged into electrical sockets and the switches controlling those sockets are important, then it can include them in every rule that deals with turning on an electrical appliance.

- Have TADPOLE include objects in its rules that are directly pointed out by the teacher to be critically important. This would place a heavier burden on the teacher who would have to first determine which important objects TADPOLE would be likely to ignore and then directly draw its attention to them. However, good human teachers already do this during their lessons.

5.8.2 The set of decomposition rules will need to be indexed

When TADPOLE searches through its decomposition rules to find one that matches a sequence of root state-difference nodes, it simply iterates

through them one by one. This has a cost that is linear in the total number of decomposition rule, and it is manageable when the agent has a relatively small set of rules. However, to be effective in the Human Planning Domain the agent will need to have a very large set of rules and iterating through the entire set every time that the agent needs to use a rule to further parse a small part of a demonstration will not be feasible. TADPOLE will need to use a rule indexing scheme to narrow the set of rules to match to help it find the appropriate rule faster.

5.8.3 TADPOLE can only parse repeated decompositions once it has learned the underlying rule

There are some decomposition rules that need to executed and re-executed an indefinite number of times before their head-task is achieved. However, in order to learn such a rule, TADPOLE needs to see at least one example of the rule being successfully executed once. Once it has learned the fundamental decomposition, it can then apply it repeatedly an indefinite number of times and correctly parse indefinitely repeated sequences of the rule.

If the first time TADPOLE sees a decomposition, the decomposition is in a repeated sequence, then it will learn a single long decomposition rule with a repeating sequence of sub-goals rather than recognizing that this is an example of a short decomposition rule being repeated. To ease the burden of the teacher, future extensions of TADPOLE should be able to recognize repeated sequences of sub-goals and break them apart into repetitions of a single decomposition.

5.8.4 TADPOLE cannot properly interpret behaviour that is a response to unexpected events

In unpredictable domains, unexpected events can occur which dramatically alter the plan being executed. Although TADPOLE can parse demonstrations that include unexpected state changes, it assumes that all of the sub-goals of the parse are steps towards achieving a particular decomposition. It has no way of parsing a demonstration that includes a state change that causes the teacher to backtrack and try a different decomposition halfway through.

Although it is reasonable to expect the lesson of a teacher to be a single coherent plan for achieving a task in one particular way, if TADPOLE is to be used to parse the behaviour of arbitrary agents, then it must have a way of parsing plans that involve backtracking and re-planning mid-way through the plan.

5.8.5 TADPOLE cannot combine multiple rules into a single rule

TADPOLE is designed to avoid learning redundant rules that describe the same decomposition. However, given enough widely varying examples, it is inevitable that at least some redundant rules will be learned. If TADPOLE is to scale to the Human Planning Domain it will need to have a mechanism for dealing with redundant rules that will accumulate over time. It will need to have a way of identifying similar, redundant rules, matching them, and generalizing them together into a single rule. However, this would be a mechanism for generalizing the agent's set of decomposition rules, and this thesis focuses mainly on learning and generalizing the rules themselves.

Chapter 6

Evaluation

This chapter covers the evaluation of TADPOLE and HOPPER. It discusses the challenges of evaluating systems like TADPOLE and HOPPER that operate in general domains like the HPD. Because what TADPOLE and HOPPER do is qualitatively different from other systems, directly comparing them (*e.g.* by efficiency) is inappropriate. Instead, this chapter presents concrete examples of TADPOLE parsing and learning from a teacher's demonstrations and HOPPER achieving various tasks in two different domains. Every rule learned by TADPOLE is tested by being executed by HOPPER, and HOPPER uses only rules that have been previously learned by TADPOLE so that the two systems are evaluated together.

Organization of the chapter

- Section 6.1 describes the challenges of evaluating systems that learn and plan in general domains like the HPD.
- Section 6.2 describes the kitchen domain and the logistics domain, the two domains TADPOLE and HOPPER are evaluated in. The section also provides the motivation for selecting these two domains in which to evaluate the two systems.

- Section 6.3 provides a series of examples of TADPOLE parsing the demonstration of a teacher and HOPPER using the learned rules to achieve related tasks.
- Section 6.4 concludes the chapter by analyzing the examples presented in the previous section in more detail.

6.1 Evaluating General Intelligence Systems

Evaluating Artificial Intelligence systems that are focused on solving specific problems in narrow, well-defined domains is well-understood and relatively straightforward. However, the problem of duplicating the generality and versatility of human cognition has received far less attention, and the methodology for evaluating such Human-Level Intelligent (HLI) systems is also less developed. John E. Laird *et al.* have pointed out that it is not even clear what criteria are appropriate for evaluating HLI systems [30], and the evaluation of HLI systems has been limited.

HOPPER and TADPOLE neither separately nor together constitute an HLI system, but only the planning component of such an agent. A full HLI system would need additional capabilities including learning and executing low-level, continuous actions; identifying different types of objects and relationships in the environment; learning the physics of the world and predicting how objects interact and change over time; and communicating with other agents. However, because the HPD involves a wide range of different tasks in a wide range of different domains, HOPPER and TADPOLE have to show the same kind of generality and versatility required of an HLI system, and many of the same challenges of evaluating HLI systems also apply to the evaluation of HOPPER and TADPOLE in the HPD.

6.1.1 Directly comparing HOPPER and TADPOLE to human behaviour is inappropriate

HOPPER and TADPOLE are not trying to model exactly how humans learn to solve various tasks and problems. Instead, their purpose is to solve the same kinds of planning problems that people excel at. Comparing response times, accuracy, and error rates with those of people is appropriate only for evaluating systems directly modeling human cogni-

tion, such as ACT-R, so such evaluation techniques are not applicable to HOPPER and TADPOLE.

6.1.2 HLI systems satisfice different tasks rather than optimizing a single task

One of the primary characteristics that distinguishes HLI systems from other systems in Artificial Intelligence is their generality. AI systems that focus on achieving a single task in a single domain can be evaluated straightforwardly: better systems can solve the given task more efficiently than others. HLI systems, on the other hand, achieve many different tasks in many different domains. Optimal behaviour is not an important part of the evaluation. This is because humans can solve a wide range of tasks but rarely do so optimally, and it is unreasonable to expect HLI systems to duplicate humans' versatility and to solve each individual task optimally as well. Because of this, standard machine learning performance comparisons are not appropriate.

It is important to note that it is difficult to compare the generality of different HLI systems. It is unclear which system is better: a system that can achieve a large number of simple tasks or a system that can achieve a small number of difficult tasks. A further problem is that it is unclear how to quantify the difficulty of a task. Just because a system can handle tasks that humans find difficult is no guarantee that it will be able to handle tasks that humans find easy. For example, being able to quickly and accurately perform large mathematical calculations and playing chess at world-class level does not help in making a cup of coffee or even identifying a cup of coffee from a picture.

Because creating systems that can solve intricate problems (such as chess) that humans find challenging has not led to a generally intelligent system, HOPPER and TADPOLE take the opposite approach. They focus on learning how to solve straightforward (for humans), routine tasks so

that they can be used in the future as a foundation for mechanisms and algorithms for solving more difficult problems.

To give confidence that HOPPER and TADPOLE can scale to the wide range of tasks present in the HPD, they are evaluated on a number of different tasks in two different domains: the Kitchen Domain and the Logistics Domain. Section 6.2 describes the two domains in detail, and Section 6.3 presents detailed examples of HOPPER and TADPOLE learning and solving different tasks.

6.1.3 It is important to distinguish learned knowledge from hand-crafted knowledge

HLI systems have a fixed architecture that remains constant from task to task and from domain to domain, and they have domain knowledge that they acquire and which is only appropriate to a particular domain. Domain knowledge is critical to the system being able to successfully complete the corresponding tasks in a domain. The two critical issues that have to be addressed for any HLI system is how to use domain knowledge to robustly solve problems and how to learn and acquire the domain knowledge in the first place.

In hierarchical decomposing systems like HOPPER and TADPOLE, the domain knowledge is encoded in decomposition rules. As discussed in Chapter 2, the bulk of the research into these systems has focused on the problem of applying domain knowledge to solve tasks robustly and efficiently. The issue of learning decomposition rules has hardly been addressed. In most cases, the entirety of the domain knowledge is hand-coded. In systems where learning has been addressed, the vast majority (if not all) of the rules have been hand-coded and the system's task has been to learn the appropriate preconditions of the rules, their weights, utilities, or probabilities of success. The issue of learning new decomposition rules for solving completely novel tasks has not been addressed.

When a system's decomposition rules and its domain knowledge in general is hand-crafted, evaluating the architecture of the system becomes problematic. This is because it is difficult to tell whether the system solved a task successfully because of a robust architecture or because of cleverly engineered decomposition rules. An architecture can be directly evaluated in terms of how expressive it is and how easy it is to program new rules to solve novel tasks — how quickly and how accurately one can craft new decomposition rules. However to evaluate the problem-solving power of such a system requires a standard benchmark of tasks to be solved and a standard set of rules to be used. However, if all of the rules are hand-crafted, then it is difficult to come up with a justifiable set of benchmark rules.

A rule-execution system, such as HOPPER, cannot be independently evaluated using hand-crafted rules because there is no way of distinguishing the architecture from the hand-crafted rules. On the other hand, a rule-learning system, such as TADPOLE, cannot be independently evaluated because there is no way of determining the quality of the rules learned. These difficulties can be overcome by evaluating HOPPER and TADPOLE simultaneously: HOPPER uses only rules learned by TADPOLE, and every rule learned by TADPOLE is used by HOPPER to achieve related tasks.

6.2 The Kitchen and Logistics Domains

HOPPER and TADPOLE have been evaluated in a kitchen domain and a logistics domain because these two domains are both part of the HPD but have different characteristics and different kinds of tasks to achieve. Being able to effectively learn how to achieve different tasks in both domains serves to illustrate the generality of HOPPER and TADPOLE.

An important distinction between these two domains is that the distribution of the burden between the executor HOPPER and the learner TADPOLE is different. In the kitchen domain, the bulk of the burden is

borne by TADPOLE. Once TADPOLE has learned the necessary rules, using them to achieve tasks is relatively straightforward for HOPPER. In the logistics domain on the other hand, the bulk of the burden is borne by HOPPER. There are relatively fewer rules that need to be learned by TADPOLE, but it is important for HOPPER to apply those rules appropriately to produce efficient plans.

The two domains evaluate both TADPOLE and HOPPER, but the kitchen domain evaluates TADPOLE more intensely, and the logistics domain evaluates HOPPER more than TADPOLE.

6.2.1 HOPPER and TADPOLE are evaluated in a kitchen domain

The kitchen domain is a simplified representation of a typical household kitchen. HOPPER and TADPOLE have been tested in this domain because there are a large number of varied tasks to achieve requiring a large number of different decomposition rules. Furthermore, the domain is rich and complex with many objects and relationships, and it requires a large number of atomic actions to achieve even simple tasks. One benefit of evaluating HOPPER and TADPOLE in such a domain is that it tests their ability to deal with a large number of irrelevant objects in the state.

The kitchen domain has a large number of richly described objects

The kitchen domain is characterized by a large number of objects with rich properties and relationships. Figure 6.1 shows a diagrammatic representation of a typical kitchen world state. Note that the absolute placement of the objects and their dimensions are not included in the state representation that HOPPER and TADPOLE take as input. They are present in the diagrammatic representation only to help give an intuitive understanding of the state. The full, first-order description of the state, which includes all of the properties and relationships of every object (including relative

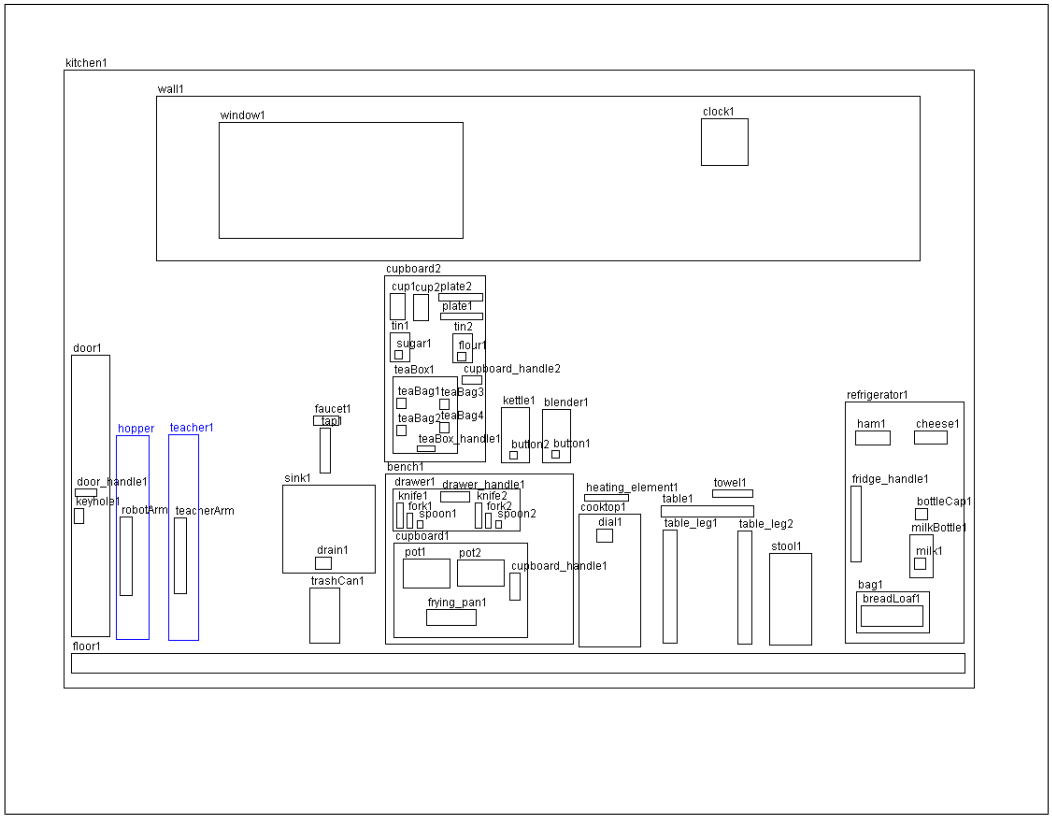


Figure 6.1: Graphical representation of a typical kitchen world state

positions), is extensive and it is not included here for the sake of clarity; however, it can be found in the appendix.

Note that the state has two avatars controlled by a teacher and by the agent. The agent is called hopper, because HOPPER interacts directly with the environment while TADPOLE only passively observes the actions of the teacher. The teacher and HOPPER interact with the environment by controlling their avatar's teacherArm and robotArm respectively. When observing the teacher, TADPOLE maps its own robotArm to any changes it observes of the teacherArm so that the rules it learns will be applicable to it (hopper) and not the teacher as described in Section 5.1.5.

HOPPER and TADPOLE assume a vision system that represents the world in first-order logic. A typical object in the kitchen domain is de-

scribed as:

kettle1: Type=[electricalKettle, container], MadeOf=[metal],Colour=[grey], Open=[yes], Running=[no], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
kettle1 → [On] → bench1
kettle1 → [Supports, Contains] → water1
kettle1 → [GraspedBy] → robotArm

For any particular task, most of an object's properties will be irrelevant; however, which properties are irrelevant will depend on the task being achieved. For more complex tasks where an object is repeatedly manipulated by the agent, which properties are relevant will change as the agent achieves different sub-tasks. For example, when making a cup of tea, the agent will manipulate the kettle object repeatedly as it achieves the task. The fact that the kettle is graspable and movable is important when the agent wants to move it around; the fact that it is a container that is open is important when the agent wants to fill it with water or pour water out of it; and the property of whether or not the electrical kettle is running or not is important when the agent is heating water with it. Other properties are almost never relevant, such as the colour of the kettle or what it is made of; however, for any property, a task can always be invented where that property is critically important. For example, if the agent wanted to heat the kettle in a microwave or on a stove, then the fact that the kettle is made out of metal would be very important.

Every object in the kitchen domain is also related to at least one other object in the state. These relationships describe quantitatively the relative spatial positions of different objects. For example, an object can be above another object, support it, be next to it, be within it, contain another object, and so on. As with an object's properties, only some of its relationships will be important for any particular task.

The kitchen domain is a great simplification of an average kitchen and ignores many important details. The domain does not deal with electricity or circuits, so the kettle does not have to be plugged in to work properly. However, it does have to be sitting on the bench in order to be able to boil water.

Another important aspect of real kitchens which the state representation generally leaves out is the internal structure of objects (substructure is included only when necessary for a task). For example, the kitchen domain does not distinguish a cup with a handle from a cup without a handle. This is so that TADPOLE does not have to work out for every set of state changes how to cluster the affected objects in the world and to determine what are the primary state changes and what are merely derived ones. For example, if the state description of the world were to include the internal structure of a clock as separate objects (the clock hands, the numbers on the clock face, the batteries of the clock, and so on) and the agent (or the teacher) moved the clock from one place to another, then the state change would include not only the clock changing position, but also all of its components as well (*e.g.* the base of the clock was on top of the table and now it is on top of the floor). The significant state change that TADPOLE should learn from is that the clock changed position and what is important is that it was graspable, movable, and not supporting another object. The components of the clock, most of which are not graspable or movable, changing position as well is an unimportant state change that is derivable from the fact that they are components of the clock. The state representation keeps the structure of objects relatively simple so that TADPOLE does not have to bear the additional burden of determining which state changes are derivable and which are not.

Clearly, requiring the state representation to be flat and without a hierarchical structure is a significant limitation of HOPPER and TADPOLE that will have to be addressed if these algorithms are to scale to more complex domains with more complex tasks. Ideally, HOPPER and TADPOLE

should use a hierarchical state description such as GRAM [6] and then change the "resolution" at which they view the world appropriately as the execution of a task progresses. For example, when achieving the task of moving a clock, the representation of the state should describe only the clock object; when achieving the sub-task of moving the agent's hand to grasp the clock, the representation should change to include a more detailed description of the clock so that the agent can grasp a particular part of the clock. The representation of the state should describe the numbers on the clock face and the clock hands only if the agent is trying to determine the time or setting the clock. Dynamically changing the resolution at which objects are described based on the task being achieved is an interesting avenue for future research but it is outside the scope of this thesis.

The kitchen domain has low-level atomic actions

The atomic actions of the kitchen domain are at a low level of abstraction (*e.g.* move hand to cup, grasp cup, lift). Evaluating HOPPER and TADPOLE in a domain with atomic actions at such a low level of abstraction has two main advantages:

- Low-level atomic actions better approximate continuous time.
- When expressed in terms of low-level actions, the tasks are longer and/or more difficult, which tests whether HOPPER and TADPOLE will scale to long tasks

Different tasks in the kitchen domain require different decomposition rules

The tasks in the kitchen domain deal mainly with food preparation and general cleaning up. These tasks share a set of common sub-tasks such as opening and closing containers, moving objects around, and pouring and

mixing liquids that can be re-used by the agent. However, most decomposition rules generate different effects, and without a complete representation of time that allows waiting, there is little opportunity for interleaving and optimizing plans in the kitchen domain. Once TADPOLE has learned the requisite decomposition rules, achieving tasks with HOPPER is relatively straightforward.

The kitchen domain has un-undoable actions

Because the kitchen domain has actions that cannot be undone (for example, you cannot unscramble an egg) it is important that the agent execute its tasks and sub-tasks in the correct order. This property of the kitchen domain makes it risky for a reactive agent with an imperfect policy.

TADPOLE makes use of a heuristic about the importance of grasping objects

In the kitchen domain, the agent and the teacher interact with the world by moving their avatar's arms next to objects in the world, grasping them, and then manipulating them in some way. TADPOLE identifies the relevant objects in the state by noting which objects have changed from time slice to slice (as described in Sections 5.3.5 and 5.3.6) and it ignores the rest. However, because grasping objects is so important to how the teacher (and agent) interacts with the world, whenever the teacher is grasping an object, TADPOLE automatically considers the grasped object and the teacher's arm relevant and includes it in the state-difference it constructs even if the grasped object has not changed since the last time slice. This reasonable heuristic allows TADPOLE to learn about buttons, switches, and faucets — objects that change other objects in the state but do not change themselves.

6.2.2 HOPPER and TADPOLE are evaluated in a logistics domain

The logistics domain is much simpler than the kitchen domain. It has fewer objects with fewer properties. The primary tasks in the logistics domain involve transporting packages from one location to another. Locations are grouped into separate "cities", and packages can be transported locally (within a city) by loading them into a truck and driving them to the appropriate location. To deliver packages between cities, they have to be transported to the city's airport, loaded on to a plane, flown to the destination city's airport, and then delivered to the appropriate location by truck. Figure 6.2 shows a graphical representation of a typical logistics state.

The atomic actions that can be performed in the domain are at a higher level of abstraction than those in the kitchen domain. Because of this, the state description does not include avatars, and the agent interacts with the environment directly instead. The atomic actions include loading a package into a truck (or plane), driving a truck from one location to another, and unloading a package, all without reference to an avatar.

Different tasks in the logistics domain re-use the same decomposition rules

The logistics domain has only a limited range of different *types* of tasks to achieve, the most important of which are delivering packages by truck and by plane. This limited range of task types require an equally limited number of decomposition rules, making TADPOLE's job relatively straight-forward because it has fewer decomposition rules to learn. However, this also means that the different tasks that HOPPER has to achieve will re-use the same decomposition rules, which increases the number of shared sub-goals and the number of opportunities HOPPER has to interleave their execution.

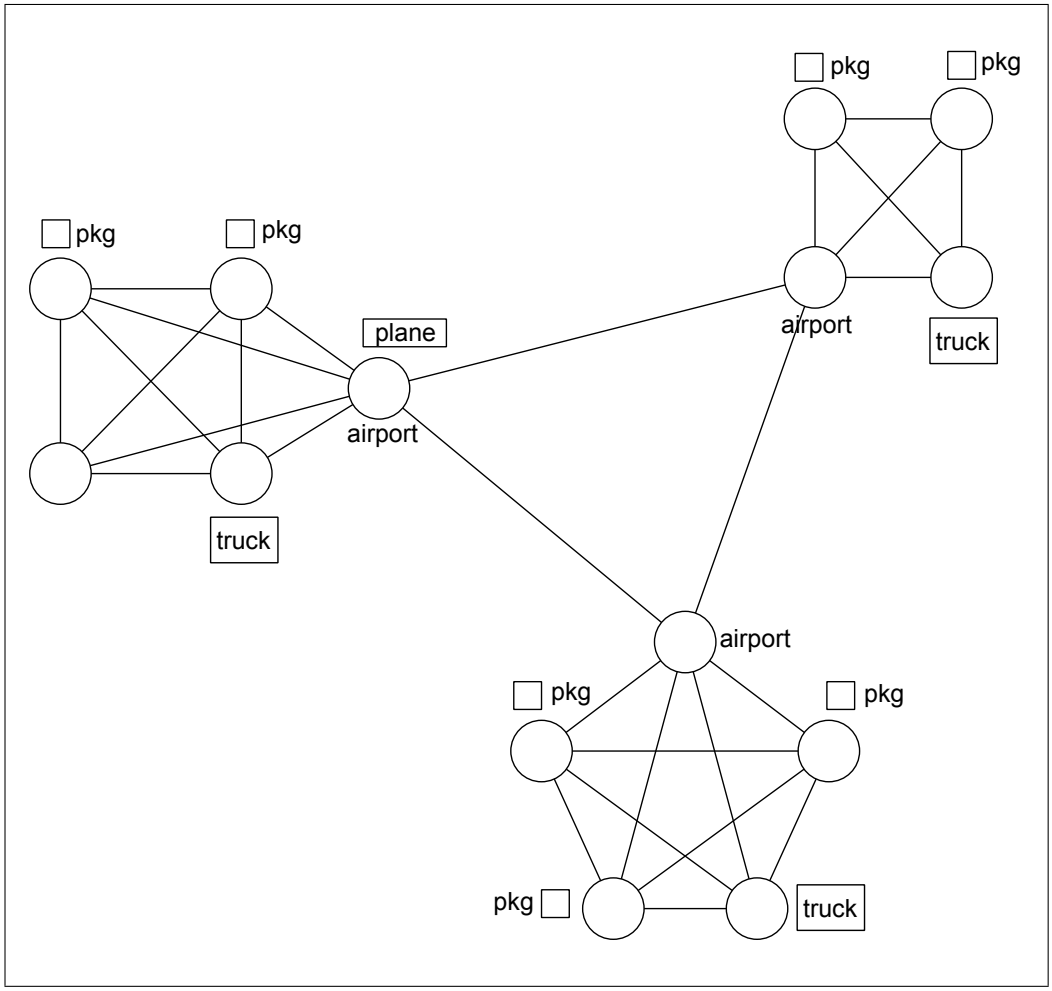


Figure 6.2: Graphical representation of a typical logistics state

Efficiency of plans is important in the logistics domain

The logistics domain does not include any pathfinding — every location in a city is connected and equidistant from every other location in the city. However, the order in which a truck visits various locations to pick up and drop off packages can greatly affect the final length of the plan. This makes the efficiency of the plan generated to deliver the various packages an important consideration, especially because every extraneous atomic action is so expensive, and needless driving and especially needless flying

from location to location should be avoided.

As discussed in Chapter 4, HOPPER cannot guarantee that the plan it generates will be optimal, however, it can strive to shorten its plans by interleaving its sub-tasks and satisfying multiple sub-goals in parallel. The logistics domain is ideal for testing HOPPER's interleaving mechanism as well as TADPOLE's ability to parse such interleaved plans. To this end, after TADPOLE has learned the fundamental decomposition rules, HOPPER was tasked with achieving multiple concurrent goals — delivering multiple packages to different locations. The multiple package deliveries are chosen so that the optimal way of satisfying all of the goals is to interleave their deliveries.

6.3 Example Tasks

This section provides a short description of the example tasks that TADPOLE learned and HOPPER executed. The next section considers some of the more interesting examples in more detail.

6.3.1 TADPOLE learned the necessary atomic rules of the kitchen domain

In order to parse high-level decomposition rules, TADPOLE first has to learn the lower-level rules they utilize, beginning with the atomic rules. TADPOLE learns new atomic rules by observing a one action demonstration by a teacher or by noticing a new atomic state-change within a larger demonstration for a higher-level rule.

TADPOLE learned the following atomic rules in the kitchen domain by observing single action demonstrations by the teacher:

- Moving the robotArm next to an object by seeing a demonstration of the teacher moving their hand next to a cup, a door, and a towel. The

wide range of different objects in the examples emphasized that the agent can move its arm next to *any* object no matter what its properties are.

- Grasping an object from two demonstrations of the teacher grasping a cup and a door handle.
- Pulling open a container from two demonstrations of the teacher pulling open a cupboard and a drawer.
- Releasing an object from two demonstrations of the teacher releasing a cup and a door handle.
- Moving the robotArm away from an object from two demonstrations of the teacher moving their arm away from a cup and a towel.
- Pushing a container closed from two demonstrations of the teacher pushing a cupboard and a drawer closed.
- Lifting an object from three demonstrations of the teacher lifting a cup, spoon, and towel.
- Putting down an object from two demonstrations of the teacher putting down a cup on a towel and a spoon on a bench.

It is very tedious for the teacher to teach such low-level actions by demonstrating them directly. However, because these actions are so simple, it would be straight-forward to extend the agent to make it explore its environment and learn such rules independently by trying various actions randomly.

TADPOLE learned the remaining atomic rules in the kitchen domain by observing the teacher demonstrate how to achieve higher-level tasks:

- Turning on a tap from two demonstrations of the teacher filling a cup and a pot with water.

- Turning off a tap from the same two demonstrations.
- Turning a kettle on from a demonstration of the teacher boiling some water with a kettle.
- Pouring boiling water on to a tea-bag from two demonstrations of the teacher making a cup of tea.
- Turning on a stove from a demonstration of the teacher boiling water with a stove (in order to make a cup of tea).
- Turning off a stove from the same demonstration.

6.3.2 TADPOLE learned the higher-level rules of the kitchen domain

TADPOLE learned its higher-level decomposition rules by observing and parsing demonstrated lessons of the teacher where each lesson consisted of nothing more than a sequence of states. It is important to note that the order in which the lessons were presented to TADPOLE is important. Later lessons rely and build on earlier lessons (the parsed hierarchies for higher-level rules require sub-parses of lower-level rules). For example, TADPOLE cannot learn how to make a cup of tea if it does not understand how to use the sink to fill the kettle with water, and it cannot learn how to use the sink if it does not know how to transfer an object into the sink.

TADPOLE learned the following decomposition rules by observing the teacher's demonstrations (in the given order):

- Opening a container from two demonstrations of the teacher opening a cupboard and a drawer. Note that this rule is not atomic and it is different from the atomic rule of pulling open a container. To open a container in the kitchen domain, the agent has to move its arm next to the handle of the container, grasp it, pull on it, release it, and finally move its hand away.

- Closing a container from two demonstrations of the teacher closing a cupboard and a drawer. This rule is also different from the atomic rule of pushing a container closed.
- Transferring an object from one location to another from two demonstrations of the teacher moving a spoon from a bench to a plate and a towel from a table to a bench.
- Transferring an object out of a closed container from a demonstration of the teacher taking out a cup from a cupboard and placing it on the floor and a demonstration of the teacher taking out a knife and placing it on the table.
- Filling a container with water from two demonstrations of the teacher filling a cup and a pot with water.
- Boiling water with a kettle from a demonstration of the teacher doing this.
- Pouring water from one container to another when the two containers are next to each other from a demonstration of the teacher pouring water from a kettle into a cup that is next to it.
- Pouring water from one container to another when the two containers are not next to each other from a demonstration of the teacher pouring water from a pot into a cup.
- Making a cup of tea from a demonstration of the teacher doing this in a state where the cup and the tea-bag are inside a cupboard and from a second demonstration where the cup and tea-bag are already on the bench.
- Boiling water with a pot on the stove from a third demonstration of the teacher making a cup of tea, but this time boiling the water with the stove.

6.3.3 HOPPER successfully made a cup of tea

To evaluate TADPOLE's learned decomposition rules as well as HOPPER itself, HOPPER was tasked with making a cup of coffee in three different scenarios:

- All cups and the box of tea are in the cupboard. HOPPER properly interleaved three applications of the decomposition for taking an object out of a container — taking a cup and a tea-bag out of the cupboard which itself involves taking the tea-bag out of the box of tea — so that it opened and closed each container (the cupboard and the box of tea) only once.

- A cup and the box of tea are already on the bench. HOPPER properly took advantage of the fact that the first two sub-goals of the decomposition for making a cup of tea were partially achieved and it did not waste time opening or closing the cupboard.

- The kettle breaks unexpectedly when HOPPER tries to use it. HOPPER first tried to get the kettle to work by pushing its "on" button repeatedly. After failing four times (the maximum number of times that HOPPER is willing to attempt a decomposition) to make the kettle work, HOPPER gave up and used its rule for boiling water with a stove instead without having to modify any other part of its plan.

In the kitchen domain, HOPPER was assigned only tasks for making a cup of tea. However, because this is such a complex task, HOPPER had to make use of every other decomposition rule TADPOLE learned in order to successfully achieve the task's sub-goals and sub-sub-goals, and so the task of making a cup of tea tested all of the decomposition rules learned by TADPOLE in the kitchen domain.

6.3.4 TADPOLE successfully learned the decomposition rules of the logistics domain

When learning decomposition rules for the logistics domain, for the sake of efficiency, TADPOLE did not take into account the rules it learned in the kitchen domain. The kitchen and logistics rules are sufficiently different from each other that it is unlikely that a mismatch would occur when parsing the teacher's demonstrations, and so ignoring the kitchen rules did not affect the resulting TADPOLE's parses. TADPOLE's and HOPPER's performance degrades as the total number of rules they know increases. So they will have to be extended with a rule indexing system that can determine what the appropriate rules are for a given domain and limit the number of rules they have to consider.

TADPOLE learned the following atomic decomposition rules in the logistics domain:

- Driving a truck from one location to another from a demonstration of driving a blue truck and a red truck.
- Flying a plane from one location to another from a demonstration of this.
- Loading a package into an open vehicle from demonstrations of loading a package into an empty truck and loading a package into a non-empty truck. TADPOLE learns an overly specific rule from the first demonstration, and it needs the second demonstration to realize that a truck will not necessarily become non-empty after loading a package into it, and so it drops this state-change from the rule's task.
- Unloading a package from an open vehicle from demonstrations of unloading a package from a truck with one package in it and unloading a package from a truck with multiple packages in it.
- Opening a vehicle from a demonstration of opening a truck.

- Closing a vehicle from a demonstration of closing a truck.

TADPOLE learned the following higher-level decomposition rules in the logistics domain:

- Loading a package into a closed vehicle from demonstrations loading a package into a closed empty truck and loading a package into a closed non-empty truck.
- Unloading a package from a closed vehicle from a demonstration of unloading a package from a closed truck with multiple packages in it.
- Emptying a vehicle from a demonstration of unloading a package from a closed truck with a single package in it.
- Delivering a red and then blue package from one local location to another local location by truck from two demonstrations one where the truck ended up in a new location and one where the truck ended up back at its start location.
- Delivering a package from one location to a non-local one by truck and by plane from a demonstration of this.

6.3.5 HOPPER successfully interleaved the plans for delivering packages to their destination

Although TADPOLE did not see a single example of the delivery of multiple packages being interleaved, HOPPER can nonetheless make use of the decomposition rules TADPOLE did learn to successfully interleave such plans by identifying common sub-goals in different decompositions.

HOPPER correctly interleaved the execution of the following tasks:

- HOPPER delivered two packages from two different start locations and to two different destinations (no interleaving was possible).

- HOPPER delivered two packages from the same start location to two different destinations. It correctly interleaved the two plans so that it drove to the start location only once, and it also correctly formed a sub-interleaving so that it only opened and closed the truck once when it was loading the two packages into it.
- HOPPER correctly interleaved the plans for delivering two packages from two different start locations to the same destination.
- HOPPER correctly interleaved the plans for delivering two packages from the same start location to the same destination.
- HOPPER correctly interleaved the plans for delivering three packages — two packages were at the same start location and the third was at the destination of one of the other packages. However, HOPPER finds the optimal plan only half of the time and generates a sub-optimal plan otherwise. Section 6.4.10 covers this in more detail.
- HOPPER correctly interleave the plans for delivering two packages from the same start location to two different destinations where one of the destinations was local and the other was non-local (and so required a plane to deliver it).

HOPPER was also able to adjusts its plan to handle unexpected events:

- HOPPER began executing the decomposition for delivering a package from one location to another, but when it got to the initial location of the package it noticed that the package was in fact at a different location. It redecomposed its decomposition for delivering the package and then immediately drove to the correct initial location rather than trying to load a non-existent package into the truck.
- HOPPER began executing one of the two decompositions for delivering two packages from two different start locations and to two different destinations, but when it got to the initial location of the first

package it noticed that the second package was unexpectedly also at this location. It redecomposed and then correctly interleaved the two decompositions so that it took advantage of the opportunity to load both packages into the truck.

6.4 Discussion of Example Tasks

The concrete examples of HOPPER and TADPOLE being applied to various tasks that are discussed below are in graphical form for the sake of clarity. HOPPER and TADPOLE operate in rich, complex domains (this is especially true for the kitchen domain) and the decomposition rules TADPOLE learns from demonstrations within those domains reflect that. Although TADPOLE cuts out a large number of objects it deems irrelevant from the state-differences it observes, each task and sub-goal of its decomposition rules still has several objects in its graph, and each object has a number of properties associated with it. This, combined with the fact that a large decomposition rule may have six or seven sub-goals, and that tasks such as making a cup of coffee require a dozen or more rules to successfully parse and accomplish, means that it is not possible to present all of the rules in their entirety (let alone all of the state descriptions after each atomic action) without including an overwhelming amount of detail. Instead, each example presents TADPOLE's final parse or HOPPER's complete decomposition hierarchy. Different examples evaluate different aspects of TADPOLE's and HOPPER's algorithms, and they highlight and omit the details of their corresponding hierarchies appropriately.

6.4.1 TADPOLE and HOPPER scale to large decomposition hierarchies

As discussed in Chapter 5, TADPOLE begins parsing the teacher's demonstration before the demonstration is completed. It maintains a beam of

partial parses and extends them gradually as it views additional states (and state-differences) of the teacher's demonstration. This allows it to successfully parse extensive decomposition hierarchies.

As discussed in Chapter 4, HOPPER begins executing its plan before it is fully formed. It uses a least-commitment decomposition strategy and decomposes only the unconstrained sub-goals of its decomposition hierarchy. Although HOPPER cannot guarantee that its plan is sound, by keeping most of its decomposition hierarchy undecomposed, it can handle even large decomposition hierarchies in nondeterministic domains.

Figure 6.3 shows a graphical representation of the final parsed decomposition hierarchy inferred by TADPOLE after observing the teacher's first lesson about how to make a cup of tea. The "makeTea" decomposition rule that TADPOLE learned from this lesson is extensive, and a complete description can be found in the appendix.

Note that because the demonstration to make a cup of tea consists of a sequence of 70 states (a result of 69 atomic actions) the graphical representation omits the lowest levels of the parsed hierarchy for the sake of clarity. Each node in the decomposition hierarchy represents three things: a state-difference, a task, and a goal. The state-difference is a difference between an earlier and a later state in the teacher's demonstration; which states are used to generate the state-differences of a parse depends on the structure of the decomposition hierarchy. The task is the head of a rule that achieves the additions and deletions specified by the state-difference. The goal is the sub-goal of the node's parent decomposition rule that is satisfied as a result of the state-difference being achieved by the task. For the sake of clarity, the graphical representation of the parsed decomposition hierarchy includes only a textual representation of the tasks in each node. The tasks are described with action verbs to emphasize the fact that tasks specify changes (additions and deletions).

To be able to properly parse this demonstration, TADPOLE had to first learn the necessary decomposition rules such as opening and closing con-

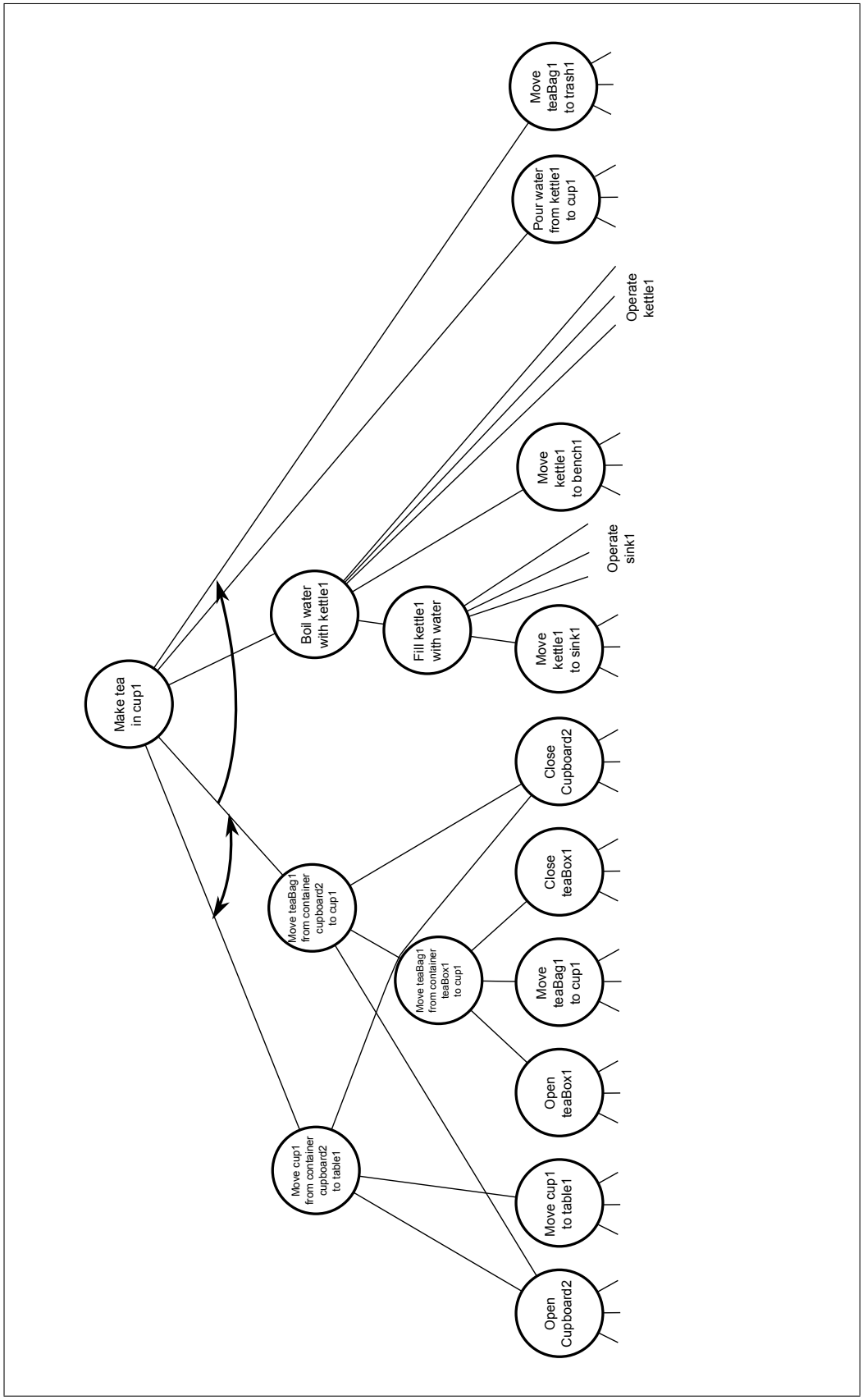


Figure 6.3: TADPOLE's parse for making tea

tainers, filling containers with water, boiling water in a kettle, pouring liquids, and so on. For each of these decomposition rules, TADPOLE had to observe and parse a separate lesson from the teacher (each lesson consisted only of a sequence of states and TADPOLE was not informed what decomposition rule is being demonstrated).

It is feasible for an agent to learn basic decomposition rules such as picking up and putting down objects through experimentation and trial and error. However, in order for this to be feasible, the agent would need to have a way of determining and identifying interesting changes in the world so that it would learn only useful decomposition rules and not swamp its rule set with superfluous rules learned from random combinations of atomic actions that had some unimportant effect on the world.

Although TADPOLE observed a lesson and learned every decomposition rule directly only once or twice, in the course of learning its rule set, TADPOLE observed multiple examples of lower-level rules. This is because the parsed decomposition hierarchies for higher-level rules tend to make use of lower-level decompositions. In the set of lessons that TADPOLE observed in the kitchen domain, it only observed two direct demonstrations for how to open a container; however, this decomposition rule was re-used in the two lessons demonstrating how to transfer an object out of a closed container, and the rule for transferring objects out of a closed container was itself re-used twice in two of the lessons demonstrating how to make a cup of tea (the cupboard was opened to take out the cup, and the box of tea was opened to take out the tea-bag). So after observing all the lessons in the kitchen domain, TADPOLE refined the "make-tea" decomposition rule based on three examples, the "transfer-object-from-closed-container" rule based on six examples, and the "open" decomposition based on eight examples.

The lessons TADPOLE learned in the kitchen domain demonstrate that the lower parts of TADPOLE's parsed decomposition hierarchies will tend to be parsed with decomposition rules that are used more often. TAD-

POLE will have observed the teacher use such low-level rules more often. Similarly, HOPPER will also have used low-level rules more often. This means that the lower-level a rule is, the more it will have been refined, and the more certain the parse making use of it will be.

The property of lower-level rules being more refined is important in allowing TADPOLE to scale to large parses for complex, high-level decompositions. The lower a decomposition is in TADPOLE's parse, the less constrained it is by the sub-parse below it (making it more difficult to parse it correctly), and the larger the proportion of the parse above it that depends upon it (making it important to parse correctly). The extra refinement of low-level rules offsets the fact that they are less constrained, and allows TADPOLE to generate the correct parse.

TADPOLE co-orders interleaved decompositions

The initial part of the demonstrated lesson for making a cup of tea involved the teacher taking a cup and a tea-bag out of the same cupboard and putting the tea-bag into the cup. Because the teacher took these two objects out of the same container, the two decompositions are interleaved, sharing the sub-goals of opening and closing the cupboard. However, the tea-bag was itself in a closed box which had to first be opened and then closed once the tea-bag was removed. TADPOLE had to first parse the decomposition removing the tea-bag from the box, and then recognize that removing the tea-bag from the box and placing it into the cup *also* satisfies the sub-goal of not having the tea-bag in the cupboard anymore and so makes it a viable part of the decomposition for removing the tea-bag out of the cupboard.

When TADPOLE parses two sub-goals whose decompositions are interleaved, it assumes that neither sub-goal is constrained to be before the other and that they can be satisfied in arbitrary order. The reasoning is that since the teacher began executing the plan for satisfying the second sub-goal before it had finished executing the plan for the first sub-goal

(the two plans were interleaved), the second sub-goal does not depend on the first sub-goal being already satisfied. Figure 6.3 depicts this arbitrary ordering of the first two sub-goals of the "make-tea" decomposition rule with a two-way arrow, and the total-ordering of the remaining sub-goals with a one-way arrow.

The heuristic of co-ordering interleaved sub-goals is correct most of the time; however, it can be too quick in dropping sub-goal ordering constraints, especially if the sub-goals that are shared are clean-up sub-goals. In the case of making a cup of tea, putting the tea-bag into the cup first and then moving both on to the kitchen bench would be a bit unusual but not incorrect. On the other hand if the teacher were making a birthday cake and opened a cupboard, took out and applied some icing to a cake, then took out some candles and put them on, and then closed the cupboard; then TADPOLE would incorrectly conclude that the sub-goals of getting icing and candles on a cake could be achieved in arbitrary order. Because TADPOLE drops the ordering constraints completely, it has no way to recover from this kind of mistake, and so without additional heuristics to determine when sub-goals can be co-ordered, TADPOLE should be more cautious and not make use of this heuristic.

Decompositions do not have to match clean-up sub-goals that match with already achieved sub-goals of subsequent decompositions

Figure 6.4 shows a lower part of TADPOLE's parse for the lesson of making a cup of tea in detail. Specifically, it shows the sub-parse for pouring water from the kettle to the cup in full, down to the sequence of demonstrated states from which TADPOLE infers the hierarchy of state-differences. The decomposition for pouring a liquid from one container to another involves moving the container containing the liquid to the other container, pouring, and then moving the now empty container back where it came from. The figure focuses on this part of the parse because it illustrates an instance of interleaving where clean-up sub-goals remain unachieved

because they are the inverse of already achieved sub-goals. The figure highlights already achieved sub-goals in green and unachieved clean-up sub-goals in red. TADPOLE correctly parsed the demonstrated states even though some clean-up goals remained unachieved, and the three sub-decompositions were all only partially complete.

When TADPOLE parses a decomposition, some of the rule's sub-goals may already be achieved in the decomposition's initial state and so are not made true by the teacher. TADPOLE also allows a decomposition to leave some of its clean-up sub-goals unachieved. But while already achieved sub-goals depend only on their decomposition's initial state, TADPOLE only allows a clean-up sub-goal to remain unmatched and be unachieved if it is the inverse of an already achieved sub-goal of an adjacent decomposition.

In the example shown in Figure 6.4, the clean-up sub-goals of releasing the kettle and moving the agent's hand away from it after placing it next to the cup are not achieved because they would be immediately undone by the first two sub-goals of the decomposition for pouring from the kettle to the cup. Similarly, the clean-up sub-goals of releasing the kettle of the pour decomposition are also not achieved because they are the inverse of the already achieved sub-goals of the decomposition for moving the kettle on to the bench. This leads to the unusual parse where the second sub-decomposition has only a single sub-goal that is satisfied (its two initial sub-goals are already satisfied when the decomposition is executed, and its last two clean-up sub-goals remain unachieved.

TADPOLE correctly parsed the demonstration for pouring in this way and learned the decomposition rule that breaks down the lesson into the three sub-goals of moving an object, pouring, and moving an object rather than as a sequence of 9 atomic decompositions.

Although in practice HOPPER will execute these same 9 atomic actions, breaking the rule into these three sub-goals helps it to achieve the task in unusual circumstances as well. For example, if HOPPER wants to

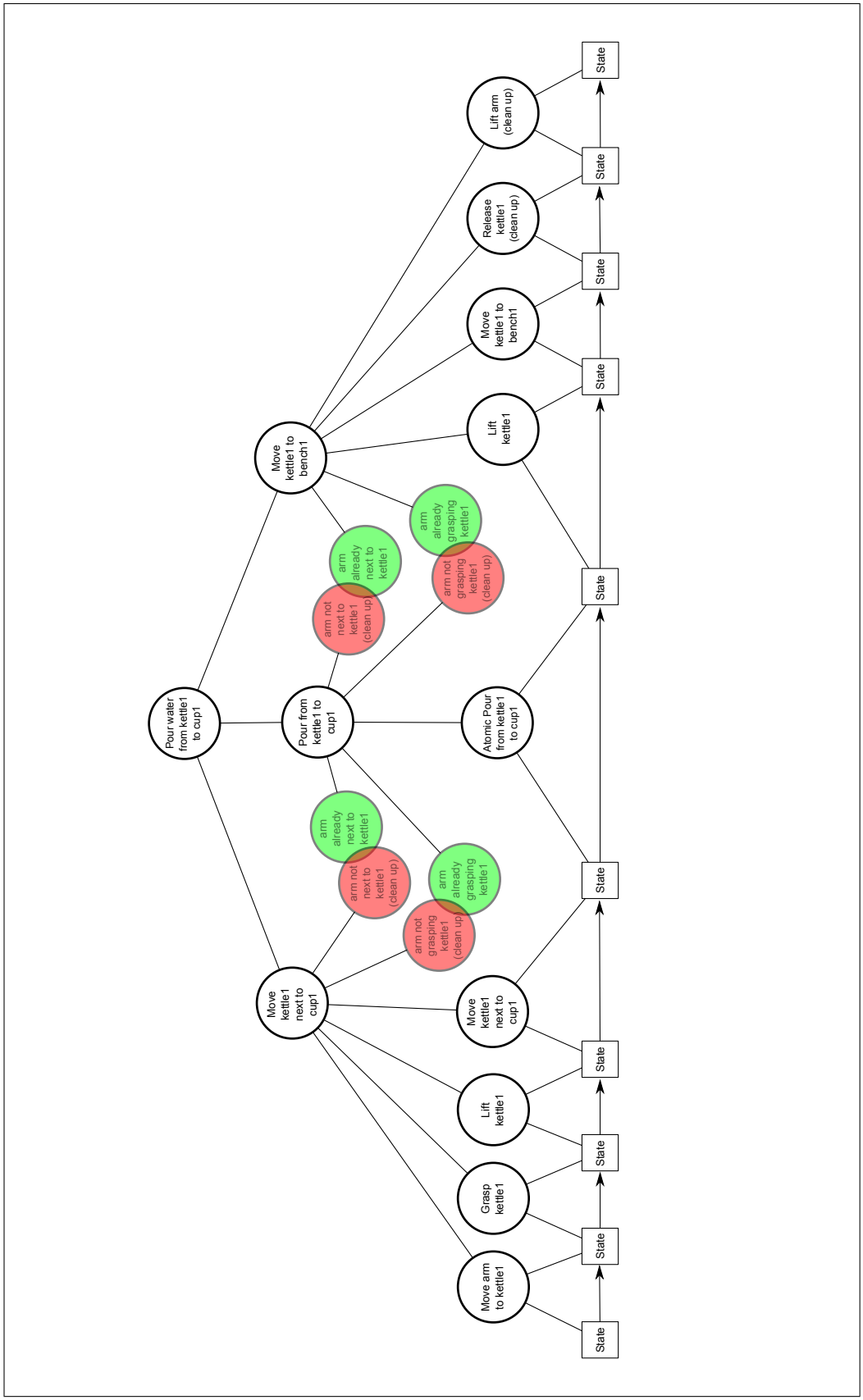


Figure 6.4: TADPOLE's parse for pouring

pour a liquid from a jug that is in a closed container such as the refrigerator, then HOPPER can make use of the decomposition rule for transferring objects out of closed containers (open the container, move the object, then close the container) to satisfy the first sub-goal of getting the jug next to the cup. Note that in this case, the clean-up sub-goals have to be achieved because HOPPER has to release the jug in order to close the refrigerator.

6.4.2 TADPOLE learns new rules from holes in its parsed decomposition hierarchy

TADPOLE learns new rules by filling in gaps in its parses of the teacher's lessons. Usually, the gap is at the very top of the parse when the teacher demonstrates a new lesson, and TADPOLE constructs the head-task of its new rule from the difference between the first and the last state, and it constructs the sub-goals from the top state-difference nodes of its partial parse. However, a gap in TADPOLE's parse can also occur within the decomposition hierarchy, when the teacher satisfies a sub-goal in novel way.

Figure 6.5 shows an outline of TADPOLE's parse for making a cup of tea and is focused on the decomposition for boiling water with an electrical kettle. In a subsequent lesson, the teacher instead used the stove to boil water in a pot. Because the decomposition for boiling water with a stove is quite different from the decomposition for boiling water with a kettle, TADPOLE was unable to directly parse the example. However, although TADPOLE could not parse the new way of boiling water, it *could* parse around this decomposition and recognize that the teacher was making a cup of tea. This allowed it to infer from the surrounding context that the teacher had to have achieved the sub-goal of having boiling water, and that the hole in its parse had to correspond to a new rule for achieving boiling water as shown in Figure 6.6. In this way, TADPOLE learned an alternative method for boiling water from a hole within the parse of the

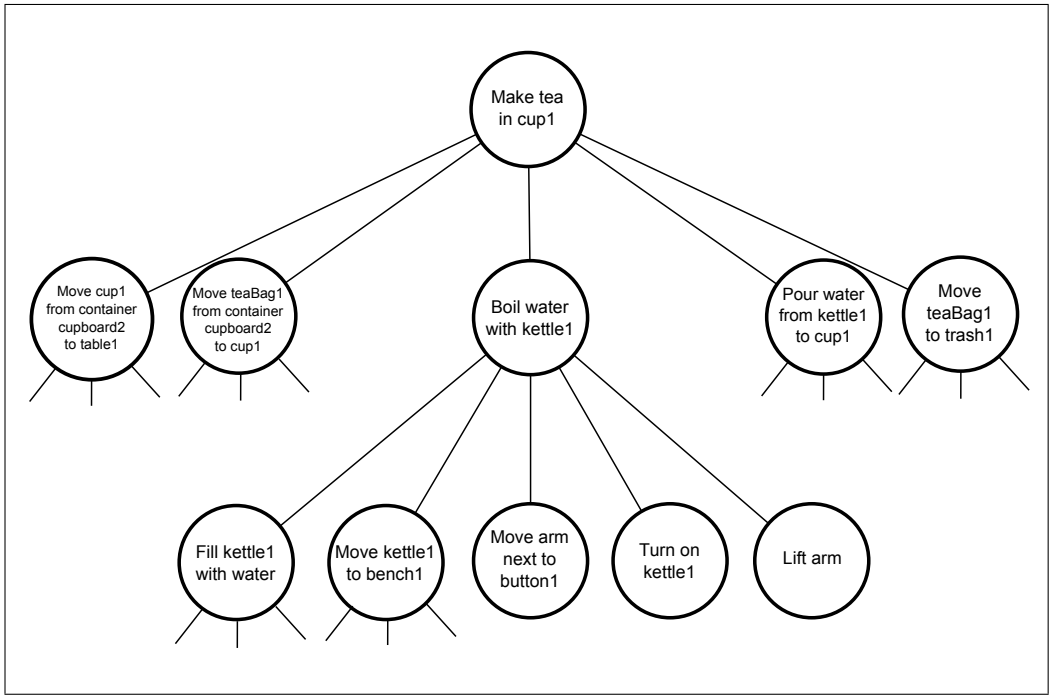


Figure 6.5: TADPOLE's parse for boiling water

teacher's lesson.

6.4.3 HOPPER adjusts its plan with alternate decomposition rules when faced with unexpected events

The more rules HOPPER has for achieving various goals, the more robust its plans are, and the less HOPPER has to modify them when they are disrupted by unexpected events.

If a decomposition fails (HOPPER satisfies the sub-goals but the parent goal remains unsatisfied), then HOPPER re-decomposes the parent goal with the same decomposition and tries again. HOPPER tries the same decomposition four times before giving up and trying an alternative decomposition rule. If HOPPER does not know an alternate decomposition rule, it keeps going up the decomposition hierarchy until it can find a parent

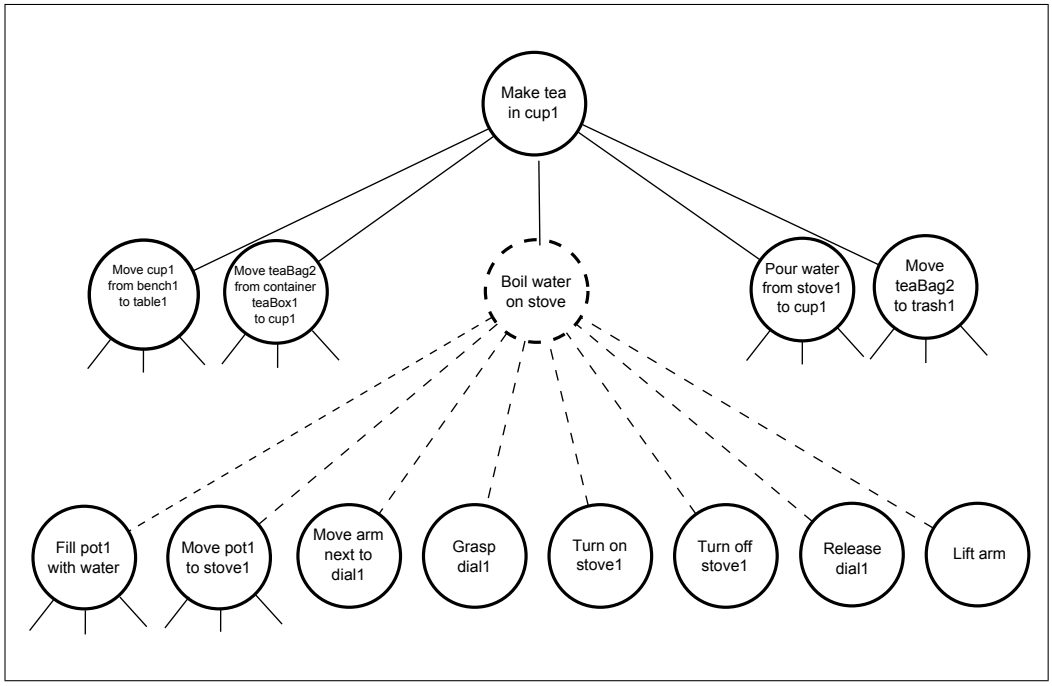


Figure 6.6: TADPOLE learning a new rule for boiling water from a hole in its parse

goal for which it has an alternate decomposition that it can use. The more decomposition rules HOPPER knows, the lower down in the hierarchy it redecomposes, preserving more of its original plan.

When HOPPER attempted to make a cup of tea and found that the kettle was broken (turning on the kettle had no effect), it first redecomposed the sub-goal with the same atomic rule for turning on the kettle (pushing the button on the kettle) and tried the same decomposition four times. After failing to turn on the kettle after pushing the button four times, HOPPER gave up. Since HOPPER knew no other rule for turning on the kettle, it went up the hierarchy and redecomposed the goal for boiling water with its newly learned rule for boiling water on the stove. The rest of HOPPER's plan remained intact, and it proceeded to successfully make a cup of tea.

6.4.4 TADPOLE can be prone to learning ritualistic behaviour

The way in which HOPPER used the stove to boil water was unusual. It took the kettle (which was already filled with water from the failed attempt to use the kettle to boil water) and moved it to the sink, and only then did it move the kettle on to the stove to heat it and the water inside it. The reason HOPPER did this is because the first sub-goal of the decomposition for boiling water with the stove is to get a container filled with water in the sink. The container that HOPPER matched to this sub-goal was the kettle, and the best way to achieve this first sub-goal was to pick up the kettle and transfer it into the sink.

This bizarre behaviour is a result of the agent's lack of knowledge about the domain. It has no way of knowing which state-changes the teacher achieved are significant and which are irrelevant — that filling a container with water is significant, but the fact that it is in the sink is irrelevant. For all it knows, there could be a hidden pressure switch under the sink that has to be activated by a heavy object (for example, a kettle filled with water) in order to be able to then turn on the stove.

This example highlights the fact that TADPOLE is vulnerable to learning rituals because of a lack of deeper understanding of the task at hand. Extending TADPOLE with a system that learns about the physics of the domain is a possible way of addressing this issue, but it is beyond the scope of this thesis.

6.4.5 HOPPER ought to use a different scoring mechanism than TADPOLE

The example of HOPPER boiling water with the stove identifies a second problem with HOPPER: it insists on boiling water on the stove in a kettle rather than in a pot. The reason for this is that the third sub-goal of the "make-cup-of-tea" decomposition is to get boiling water in a container that has been a *kettle* three times (twice from teacher demonstrations and

once from HOPPER successfully achieving the task) and a pot only once. HOPPER thus preferentially matches the kettle as the container to use even though the rule for boiling water with a stove would match better if it used a pot. The reasoning is once again, that for all HOPPER knows, having boiling water in a kettle specifically is important to making a cup of tea. However, TADPOLE had already seen a complete example of the teacher successfully using a pot to boil water when making a cup of tea, and so HOPPER should have no qualms about using a pot in the same way.

The problem lies with the fact that HOPPER uses the same scoring mechanism to select its rules as TADPOLE does for parsing demonstrations. This works well in most cases, but the contexts of rarely used alternative decompositions may be swamped by the default decompositions in the sub-goals they achieve. For example, if TADPOLE observed the teacher making a cup of tea with a kettle 100 times and with a pot only once, then HOPPER would insist very strongly that a kettle *always* be used.

The fundamental difference between TADPOLE and HOPPER is that TADPOLE is trying to determine what decomposition rule the teacher is using while HOPPER already knows which rules it is using. The fact that the teacher is using a *kettle* to boil water, though irrelevant for HOPPER, is useful evidence for TADPOLE that helps it determine that the teacher is making a cup of tea. Because HOPPER does not try to determine what decomposition rule is being used, it should focus only on the properties and relationships that have always been present and so may be necessary to the sub-goal being achieved (*e.g.* the object has to be a metal container but not necessarily a kettle or a pot).

This difference between indicative and necessary properties and relationships needs to be reflected in HOPPER's scoring mechanism.

6.4.6 TADPOLE refines the context of its decomposition rules

The way that packages are delivered to different locations (at least at the local level) in the logistics domain is by truck. The "deliver" decomposition rule specifies how to do this by driving a truck to where the package is, loading the package into the truck, driving the truck to its destination, and unloading the package. A truck can be moved from location to location by simply driving it there, and this is specified by the "drive" decomposition rule.

The "drive" and "deliver" decomposition rules are similar in that, once executed, they change the location of an object. However, the "drive" rule is only applicable to trucks and the "deliver" rule is only applicable to packages, and this is reflected in the preconditions of these two rules. Because TADPOLE learns these decomposition rules from demonstrated lessons and because it does not employ any heuristics about the relative importance of different object properties, it requires multiple examples to learn that the type of an object being relocated is much more important than its colour (for example).

After TADPOLE observed a single example of a red truck being driven from one location to another, and then a single example of a blue package being delivered from one location to another, because it had nothing more to go on, the two decomposition rules it learned weighed the type and colour of the objects being relocated equally in the rules' preconditions. If HOPPER were tasked with relocating a red package, then the preconditions of the head-tasks of both rules would match equally well (one would match the type of the object correctly and the other would match its colour), and HOPPER would have no basis for deciding which decomposition rule to use. HOPPER would have a 50% of choosing the incorrect (and ridiculous) decomposition of attempting to drive the red package to its destination (given what TADPOLE observed, for all HOPPER knows,

red things are driven and blue things are delivered).

Only after TADPOLE had seen additional examples of different coloured trucks being driven and different coloured packages being delivered would it adjust the relative weightings of the type and colour of the object being relocated to the correct value where the type is critically important and the colour is irrelevant. Having poorly learned rules and selecting the wrong decomposition rule for execution is not disastrous, however. This is because if HOPPER selects the wrong decomposition and attempts to drive a package (or deliver a truck), then that decomposition will fail, and HOPPER will try the alternative (correct) decomposition. When HOPPER successfully executes that decomposition it refines it in the same way that TADPOLE does and gradually reweights the properties and relationships of the rule's precondition appropriately.

6.4.7 TADPOLE has more difficulty finding the correct parse in a less detailed domain

The example of moving red and blue packages and trucks highlights the fact that it is often more difficult to learn in simpler, less detailed domains, than in richly detailed domains. Although richly detailed domains have a lot more irrelevant information that has to be pruned away, that same detail helps to disambiguate between similar decomposition rules. If enough detail is included in the state description, the difference between packages and trucks would become obvious enough that even after only a single example, HOPPER would recognize that red packages are more similar to blue packages than they are to red trucks.

TADPOLE has a similar problem with correctly learning the rule for delivering a package by plane. This is because the decomposition involves flying the plane to one airport, loading the package into the plane, flying the plane to the destination airport, and unloading it; and this decomposition is structurally very similar to the decomposition for delivering a

package by truck. The only difference between these two decompositions is that the Type of one delivery vehicle is [truck, vehicle] and the other is [plane, vehicle], and that the local locations are directly related by the ConnectedTo relationship while the two airports are not. Given the lack of detail in the state description and the complete lack of any background domain knowledge, TADPOLE has no way of knowing that the difference in vehicle type and location distance is any more significant than the vehicle's colour.

In this example, TADPOLE would normally interpret the demonstration of delivering a package by plane as a slightly unusual instance of delivering by truck. In order for TADPOLE to learn the correct new rule, it required an additional heuristic that flying a plane from one location to another is not an unusual form of driving a truck from one location to another. The example shows that in order to learn the correct set of decomposition rules from the lessons provided, TADPOLE requires a minimum of domain knowledge, either in the form of a more detailed state description or as additional heuristics.

6.4.8 TADPOLE refines the variables and the core of its decomposition rules

As well as refining the precondition of a decomposition rule's head-task and the contexts of its sub-goals, TADPOLE also refines the additions and deletions specified by the task, the *must* and *must not* properties and relationships specified by the sub-goals, and the variables of the rule. It is important to note that *only* TADPOLE makes these refinements; HOPPER does not, and in fact, cannot. This is because the core parts of decomposition rules are integral to how HOPPER uses them when planning. HOPPER uses the *must* and *must not* properties and relationships of subgoals to determine when they have been satisfied, it uses the additions and deletions of tasks to determine which decomposition rule is appropriate to

achieve desired state-changes (state-changes that will satisfy a particular goal in the current state), it uses the sub-goal dependencies to interleave decompositions, and it uses the variables of a decomposition rule to constrain the matchings of its sub-goals to the current state which is critical in determining what HOPPER needs to achieve. HOPPER can only refine the context of a successfully executed decomposition rule.

Refining the core of a decomposition rule is important because when it is first learned, TADPOLE may inadvertently pick up extraneous and irrelevant state changes. TADPOLE drops any addition or deletion of a task that is not present in the state change the task is successfully matched with. Similarly, it drops any *must* property or relationship of a sub-goal that is not true (or made true) in the state-difference the sub-goal is matched with, as well as dropping any *must not* property or relationship that is true (or is made true) in the state-difference. If there are significant differences between the core of a decomposition rule and the state-differences it is matched with, then this is usually and indication that the match is incorrect and a better (or completely new) rule is appropriate. This is why core mismatches are strongly penalized in the scoring of the matching. Similarly two task or goal objects may be matched with the same state object coincidentally and not because they *have* to be, and TADPOLE discards such spurious variables so as not to unduly constrain HOPPER when it uses the rule.

When TADPOLE first learned the decomposition rule for delivering a package locally, the effect of the demonstrated lesson was not only that the package changed locations, but also that the truck used to deliver the package did as well. TADPOLE noted both of these effects and specified the appropriate deletions in the head-task of the learned rule given below. Important variables that are included in the task are not only the package and its initial location and final destination, but also the truck and *its* initial location (its final destination is the same as the package's).

loc3: **VAR311** {Type={location=2}}
loc3 → link(1L): [HasObject] → package1
loc3 → link(0L): [HasObject] → truck1
loc3 → link(13L): ConnectedTo=2 → loc2
loc3 → link(10L): ConnectedTo=2 → loc1

loc2: **VAR312** {Type={location=2}}
loc2 → link(8L): [HasObject] → truck1
loc2 → link(5L): ConnectedTo=2 → loc1
loc2 → link(9L): ConnectedTo=2 → loc3

truck1: **VAR314** {Type={truck=2}, Colour={red=2,blue=1}, Open={no=2}}
truck1 → link(11L): [AtLocation] → loc3
truck1 → link(7L): [AtLocation] → loc2

package1: **VAR315** {Type={package=2}, Colour={blue=2,red=1}}
package1 → link(3L): [AtLocation] → loc3
package1 → link(12L): [AtLocation] → loc1

loc1: **VAR316** {Type={location=2}}
loc1 → link(2L): ConnectedTo=2 → loc3
loc1 → link(4L): [HasObject] → package1
loc1 → link(6L): ConnectedTo=2 → loc2

However, TADPOLE later observed and parsed a lesson where the truck's initial location was the same as the destination of the package, and at the end of the lesson, the truck returned to its starting location. This meant that the truck's location did not change after the lesson was complete and this was reflected in the top state-difference.

This was a significant mismatch with the head-task of the "deliver" rule because the additions and deletions of the truck's location are not

present in the state-difference it was matched with. However, because all four sub-goals of the decomposition rule matched well with the sub-state-differences, the overall score was high enough for a successful rule matching.

Note that in this case it does not matter how many times TADPOLE observes the truck's location change after delivering a package before seeing the special case of the truck delivering a package to its own initial location. The greater number of examples where the truck's location has changed will increase the weight of this change in the rule's task, and so increase the penalty for this change not being present. However, the greater number of examples will also increase the weight of the sub-goals' *must* and *must not* properties and relationships and their successful matching will outweigh this penalty.

After successfully matching the head-task of the "deliver" decomposition rule, TADPOLE recognized that the delivery truck will not always change location after the execution of the rule, and it successfully refined the decomposition rule to reflect this by removing these additions and deletions from the task graph:

loc3: **VAR311** {Type={location=3, airport=1}}
loc3 → link(1L): [HasObject] → package1
loc3 → link(10L): ConnectedTo=3 → loc1

package1: **VAR315** {Type={package=3},Colour={blue=2,red=1,green=1}}
package1 → link(3L): [AtLocation] → loc3
package1 → link(12L): [AtLocation] → loc1

loc1: **VAR316** {Type={location=3}}
loc1 → link(2L): ConnectedTo=3 → loc3
loc1 → link(4L): [HasObject] → package1

Similarly, TADPOLE initially learned an overly specific rule for making a cup of tea when it observed a demonstration of the teacher moving a cup and tea-bag out of the cupboard. The head-task of the initial rule specified that among the state-changes that would result when the "make-cup-of-tea" decomposition was executed would be that the cup and tea-bag would not be in their starting containers. After parsing a second lesson where the cup and tea-bag were already on the bench, TADPOLE successfully refined the "make-cup-of-tea" rule to drop these extraneous state changes.

6.4.9 HOPPER interleaves and sub-interleaves decompositions with shared sub-goals

When HOPPER has two goals to achieve neither of which is constrained to be satisfied before the other, then it decomposes both of them and searches for shared sub-goals. If it finds any shared sub-goals, then it attempts to find a way to interleave the execution of their decompositions in a way that does not violate the sub-goal dependencies of either decomposition (Chapter 4 describes the interleaving algorithm in more detail). The interleaving itself can also include co-ordered sub-goals, sub-goals that have no ordering constraints between them, and the decompositions of such sub-goals can themselves be interleaved if they share any sub-sub-goals.

Figure 6.7 shows an example in which HOPPER had two co-ordered goals: to deliver two packages from the same location to two different locations. The initial sub-goal of both decompositions was to get a truck to the location loc2 so HOPPER found a way of interleaving the execution of both decompositions so this sub-goal was only satisfied once. In the interleaving, the sub-goals of getting pkg1 and pkg2 into the truck did not interfere with each other or with either of the adjacent sub-goals, so HOPPER did not make any ordering constraints between them. As a result, once HOPPER achieved the first sub-goal of the interleaving by driving truck1

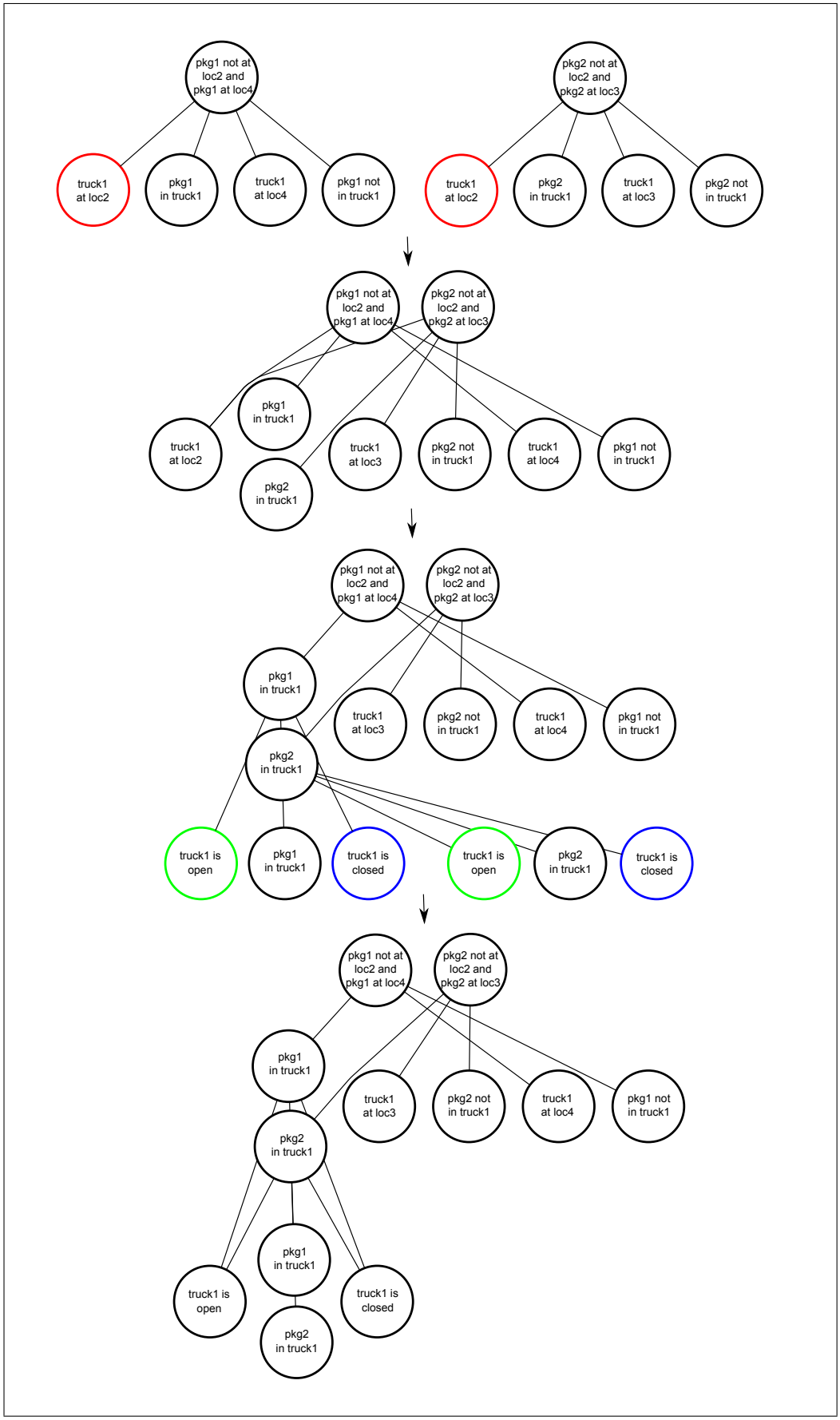


Figure 6.7: HOPPER interleaving the delivery of two packages

to loc2, it decomposed both sub-goals of getting pkg1 and pkg2 into the truck and it searched for any shared sub-sub-goals. Because they shared two sub-goals each (making truck1 open and then closed), HOPPER interleaved these two sub-decomposition to produce a sub-interleaving so that the truck was open and closed only once. Note that in this sub-interleaving, the sub-goals for loading the packages into the truck were again co-ordered, so if there had been a way for interleaving their decompositions as well (*e.g.* by using the same forklift to load both packages), HOPPER could have found a sub-sub-interleaving to further optimize the plan.

6.4.10 HOPPER interleaves multiple decompositions with shared sub-goals

For any two decompositions with shared sub-goals there are often multiple different ways of interleaving them. In the example above, after driving to the initial location of the two packages and loading both into the truck, the agent can deliver the packages in either order, and both interleavings are equally valid. When searching for an interleaving, HOPPER uses the first one it finds. However, this is not always ideal because the ordering of sub-goals in the interleaving can be important when HOPPER interleaves additional decompositions into the interleaving.

Figure 6.8 shows an example of HOPPER interleaving a third decomposition into the two-decomposition interleaving from the previous example. The third package's initial location is the destination of one of the interleaved packages and the destination of the third package is the destination of the other interleaved package. The best plan is to drive a truck to loc2, load pkg1 and pkg2 into the truck, then drive to loc3, the initial location of pkg3, and load pkg3 into the truck while unloading pkg2, and then to drive to loc4 and unload the two packages there; and this is the plan shown in Figure 6.8.

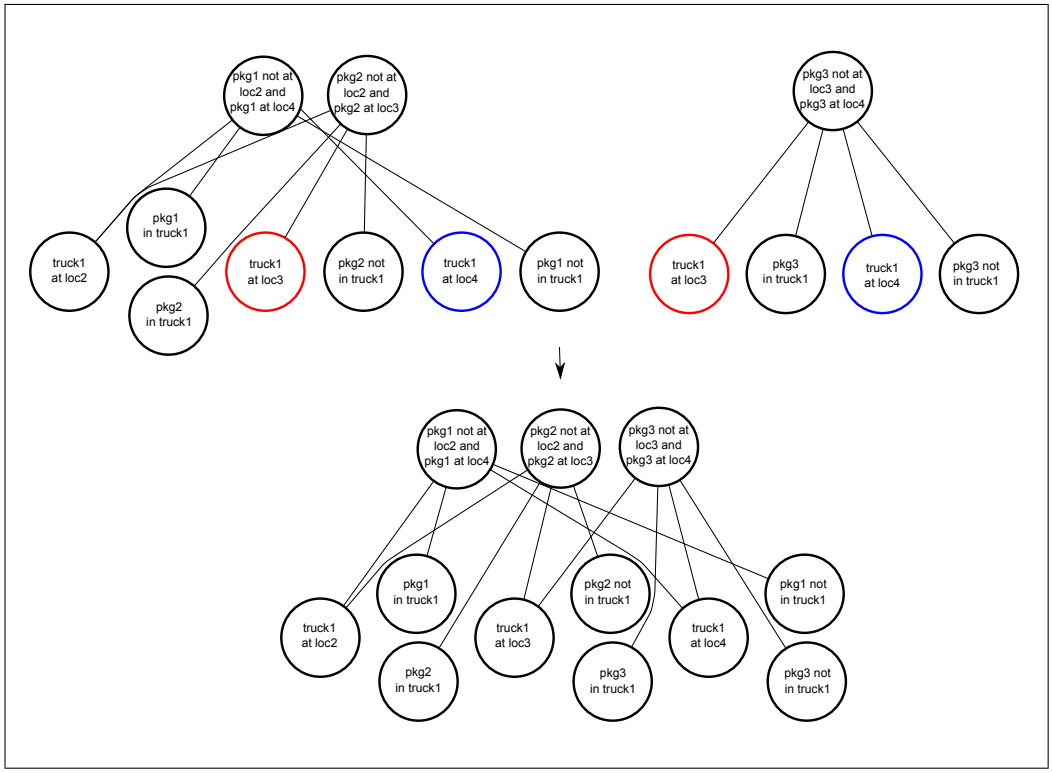


Figure 6.8: HOPPER interleaving the delivery of three packages

However, this optimal plan depends on HOPPER having selected the appropriate interleaving when it first interleaved the decompositions for interleaving the deliveries of pkg1 and pkg2. If, after loading both packages at loc2, HOPPER planned to unload pkg1 first at loc4 and only then to go to loc3 to unload pkg2, then it would only be able to produce a suboptimal plan to also deliver pkg3: drive to loc2, load pkg1 and pkg2 into the truck, drive to loc4, unload pkg1, drive to loc3, unload pkg1 and load pkg3 into the truck, then drive *again* to loc4, and unload pkg3.

A possible future extension for HOPPER is to keep track of multiple interleavings if it is interleaving more than two decompositions so that it can find the more optimal interleaving for multiple decompositions.

6.4.11 HOPPER interleaves decompositions that are on different levels of abstraction

When HOPPER searches for shared sub-goals of co-ordered goals, it not only looks through their immediate decompositions, but also through sub-decompositions at every level of the decomposition hierarchy. Because HOPPER follows a least-commitment decomposition strategy and decomposes only unconstrained sub-goals, if two goals are co-ordered with each other, then all of the decomposed (and therefore unconstrained) sub-goals and sub-sub-goals of one co-ordered goal are also co-ordered with the decomposed (and therefore unconstrained) sub-goals and sub-sub-goals of the other co-ordered goal. This means that the decompositions of any shared sub-goals at any level of the decomposition hierarchy are valid candidates for interleaving.

Figure 6.9 shows an example of interleaving two decompositions at different levels below their co-ordered parent goals. HOPPER had two co-ordered goals of delivering two packages, one locally, and one non-locally. The rule for delivering a package non-locally (too far to deliver it by truck) specifies that the package needs to first be delivered to a local airport before it can be flown by airplane to its destination. This local delivery of the package to the airport used a truck like any other local delivery, and since the second package had the same initial location as the first package, HOPPER interleaved these two plans as normal. Once HOPPER had satisfied all of the sub-goals in the interleaving, it had in the process achieved the first top goal, and the first sub-goal of the decomposition of the second top goal. HOPPER then went on to satisfy the rest of the sub-goals of the decomposition for flying the second package to its destination.

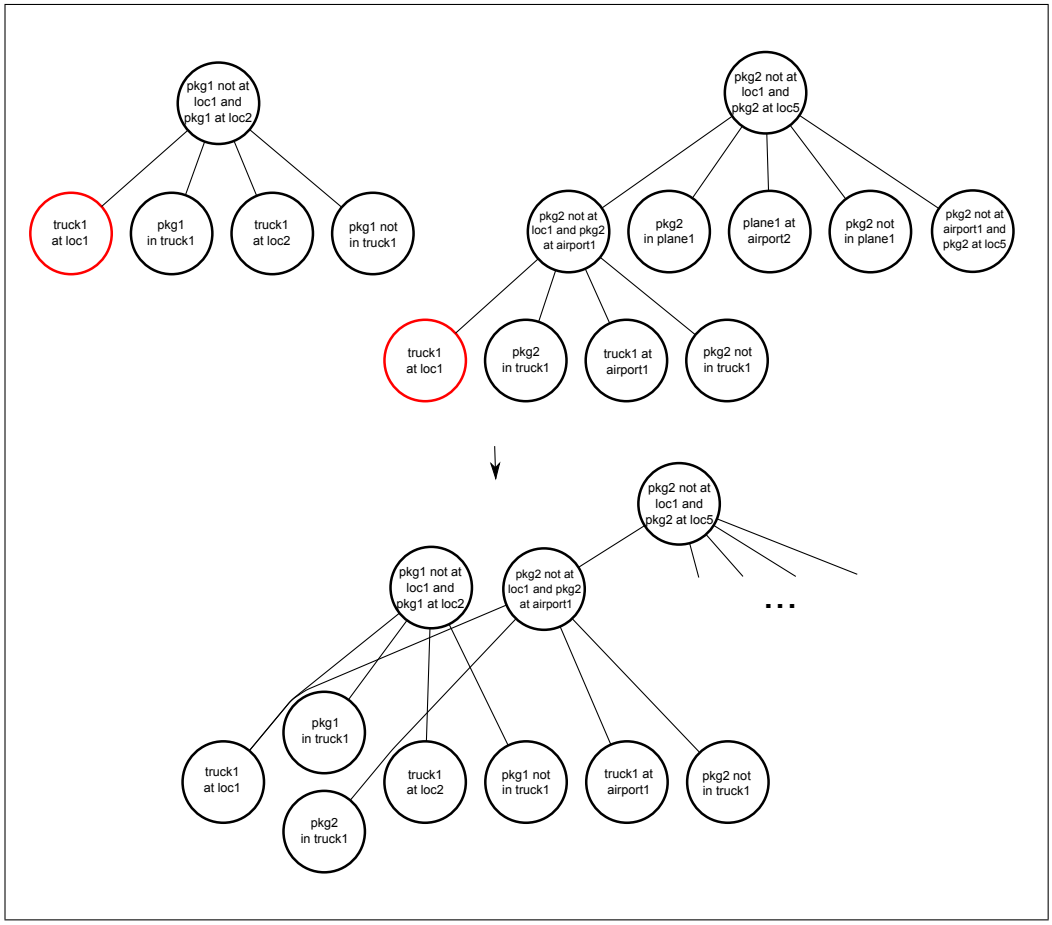


Figure 6.9: HOPPER interleaving a local and non-local delivery

6.4.12 HOPPER can approximate quantifiers with repeated decompositions

The decomposition rules used by HOPPER and TADPOLE are not expressive enough to represent quantifiers in their tasks or sub-goals. However, HOPPER can approximate some aspects of quantifiers with repeated decompositions. A decomposition can potentially be redecomposed an unlimited number of times (although HOPPER is configured to give up after 4 failed attempts), and HOPPER exploits this to achieve "for all" quantified goals.

Although repeated decompositions allow HOPPER to achieve "for all" quantified goals, it still cannot express such goals or tasks, and to compensate for this, the state description must be modified with explicit properties or relationships specifying the goal. For example, to express the goal that a truck should contain nothing (the truck should not be related to any object via the Contains relationship), the state description had to be modified to include an additional isEmpty property for the truck object.

The lesson for emptying a truck made use of the fact that HOPPER redecomposes failed decompositions to specify that the way to empty a truck (make its isEmpty property equal to yes) is to remove a single package from it. If the truck is not empty after the first package is removed, then removing more packages from the truck will eventually make it empty.

Figure 6.10 shows a graphical representation of HOPPER's decomposition hierarchy as it achieved the goal of emptying a truck. It is important to note that HOPPER does not immediately achieve clean-up sub-goals, but instead it keeps track of them on a stack and only achieves them if they do not conflict with the already-achieved sub-goals of subsequent decompositions (chapter 4 covers this in more detail). In this example, it allowed HOPPER to reload multiple packages from the truck without needlessly closing and re-opening the truck.

The limit of 4 failed attempts before giving up still applies to such quantified goals, which means that HOPPER can successfully empty a truck only if it has no more than 4 packages. This highlights the fact that a numerical limit is too crude a heuristic. Instead of a general limit, each decomposition could have its own learned limit based on the number of times it tends to get executed (*e.g.* once or twice to open doors, ten or twenty times when stirring a cup of tea).

A second way of overcoming this limitation is for HOPPER to have some way of determining whether or not it is making progress towards the goal. For example, as it is unloading packages it can notice that the truck is becoming steadily emptier, or when stirring a cup of tea it can no-

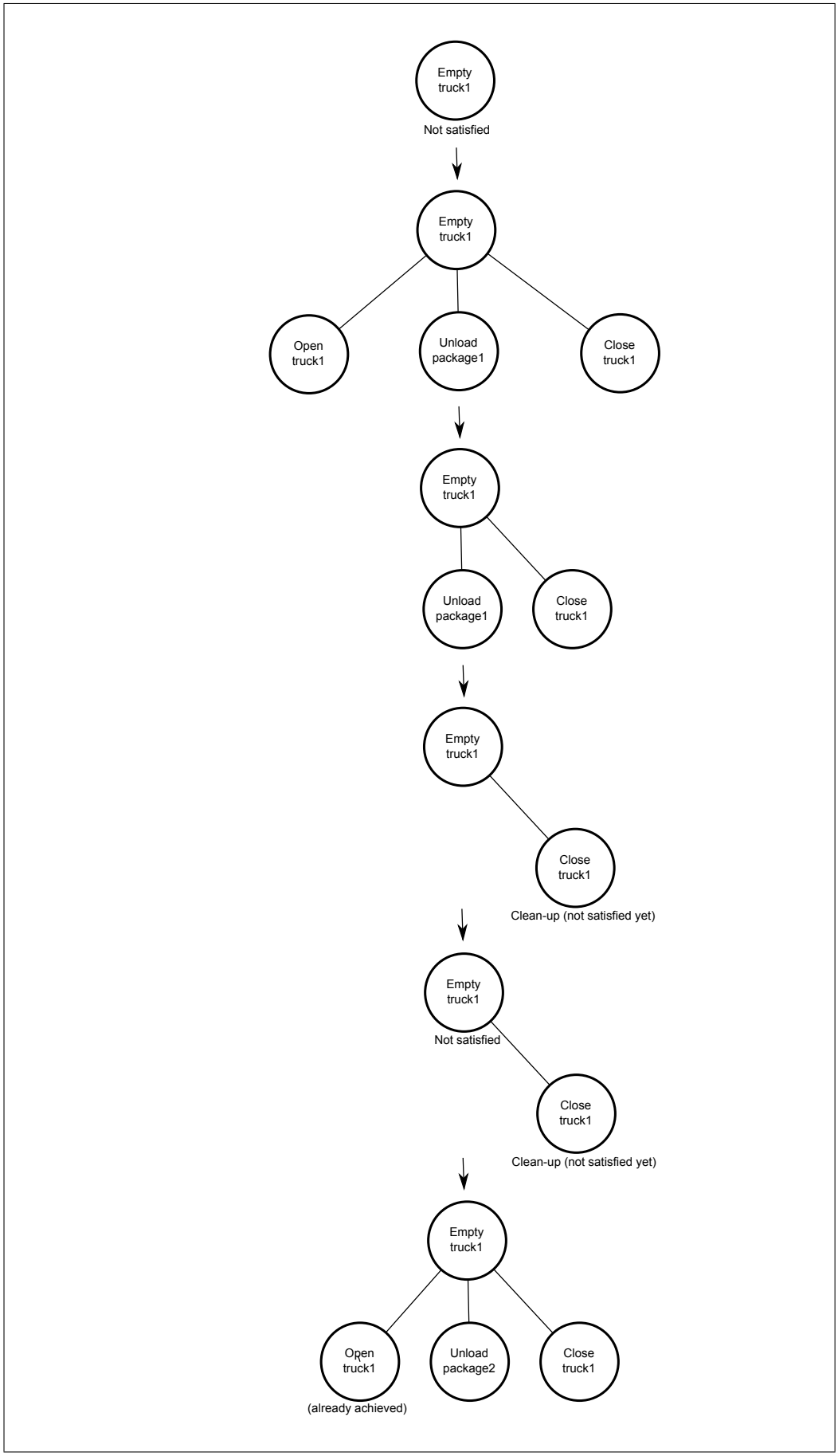


Figure 6.10: HOPPER emptying a truck

tice that the sugar is steadily dissolving. As long as HOPPER believes it is making progress towards satisfying the goal, it could continue redecomposing with the same decomposition. This approach has the advantage of allowing HOPPER to redecompose indefinitely as long as it thinks it is making progress. It would allow HOPPER to empty a truck of hundreds of packages even if TADPOLE had only ever seen examples of two or three packages being unloaded.

6.4.13 TADPOLE has to see a single successful use of a repeated decomposition to learn the correct rule

To learn a repeated decomposition rule in the first place, TADPOLE needs to see the task being achieved with a single execution of the decomposition. To learn the decomposition rule for emptying a truck, TADPOLE had to observe a lesson where the teacher emptied a truck by taking out a single package. However, once TADPOLE has learned the rule, it can parse redecompositions of it as shown in Figure 6.11. If TADPOLE parses multiple, adjacent instances of the same rule being applied to achieve the same task, then it collapses these instances into a single decomposition.

The restriction that TADPOLE places upon the teacher to initially demonstrate a repeated decomposition being successfully achieved after a single execution of the decomposition can make the initial lesson awkward, especially for tasks that almost always require multiple executions of their decomposition. For example, it would be difficult for the teacher to contrive an example where it dissolved the sugar in a cup of tea after stirring it only once. In the example of learning how to empty a truck there is the further difficulty that the decomposition rules for emptying a truck and unloading it are almost identical to each other, and the only way TADPOLE was able to learn two separate rules from these two lessons is because it was forced to by the teacher.

Ideally, TADPOLE should be able to learn a new repeated decomposi-

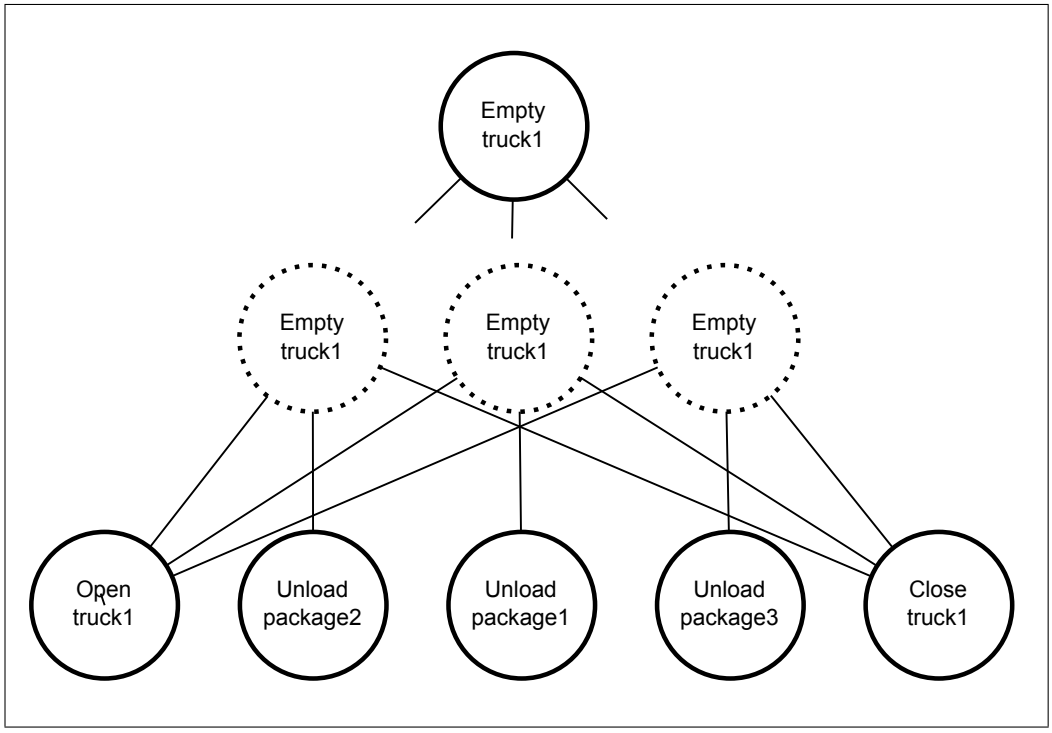


Figure 6.11: TADPOLE's parse for emptying a truck

tion from an example of the teacher repeatedly applying the same decomposition. However, it would need a way to distinguish between a repeated decomposition and one that had multiple identical sub-goals which were all achieved with the same sub-decomposition (*e.g.* stirring the tea until the sugar is dissolved and stirring the tea exactly 11 times). With a lack of in-depth knowledge about the domain and a lack of additional guidance from the teacher, a cautious method would be to learn learn multiple different rules from different examples of the decomposition being applied a greater or lesser number of times, and only then to combine these rules into a single, repeated decomposition rule.

6.4.14 HOPPER can take advantage of unexpected opportunities to interleave

When an unexpected event occurs while HOPPER is executing its plan, it identifies any affected decompositions and redecomposes them in light of the new state information. Usually when an unexpected event occurs it affects HOPPER's plan negatively (if it affects it at all) — it means that a rule has been misapplied and HOPPER has to redecompose at least part of its decomposition hierarchy to adjust its plan. The examples of HOPPER repeatedly trying to use a broken kettle and of it driving to the wrong location to pick up a package are examples of wasted effort. However, occassionally an unexpected event can present an opportunity for HOPPER to shorten its plan.

Figure 6.12 shows an example of HOPPER taking advantage of an unexpected opportunity to interleave and thus shorten its plan for delivering two packages. HOPPER was initially tasked with delivering pkg1 to loc2 and pkg2 to loc4. After decomposing both goals, because it initially believed that pkg1 was at location loc1 and pkg2 was at location loc3, it saw no way of interleaving these two plans, and so it began achieving the first one by driving truck1 to loc1. Upon arriving at loc1 it noticed that it had been mistaken about the initial location of pkg2 and that it was also at loc1. This unexpected state change prompted HOPPER to redecompose the decomposition dealing with pkg2. The first sub-goal of the newly redecomposed decomposition of getting truck1 to loc1 was already achieved in the current state, and so HOPPER removed it from the decomposition hierarchy and proceeded to decompose the next sub-goal of getting pkg2 into truck1. At this point, HOPPER noticed that it could interleave its plan for loading pkg1 and pkg2 into truck1 and it did so in the standard way, proceeding to successfully executing the rest of the plan and delivering both packages to their appropriate destinations.

In this way, HOPPER makes use of unexpected opportunities (that it

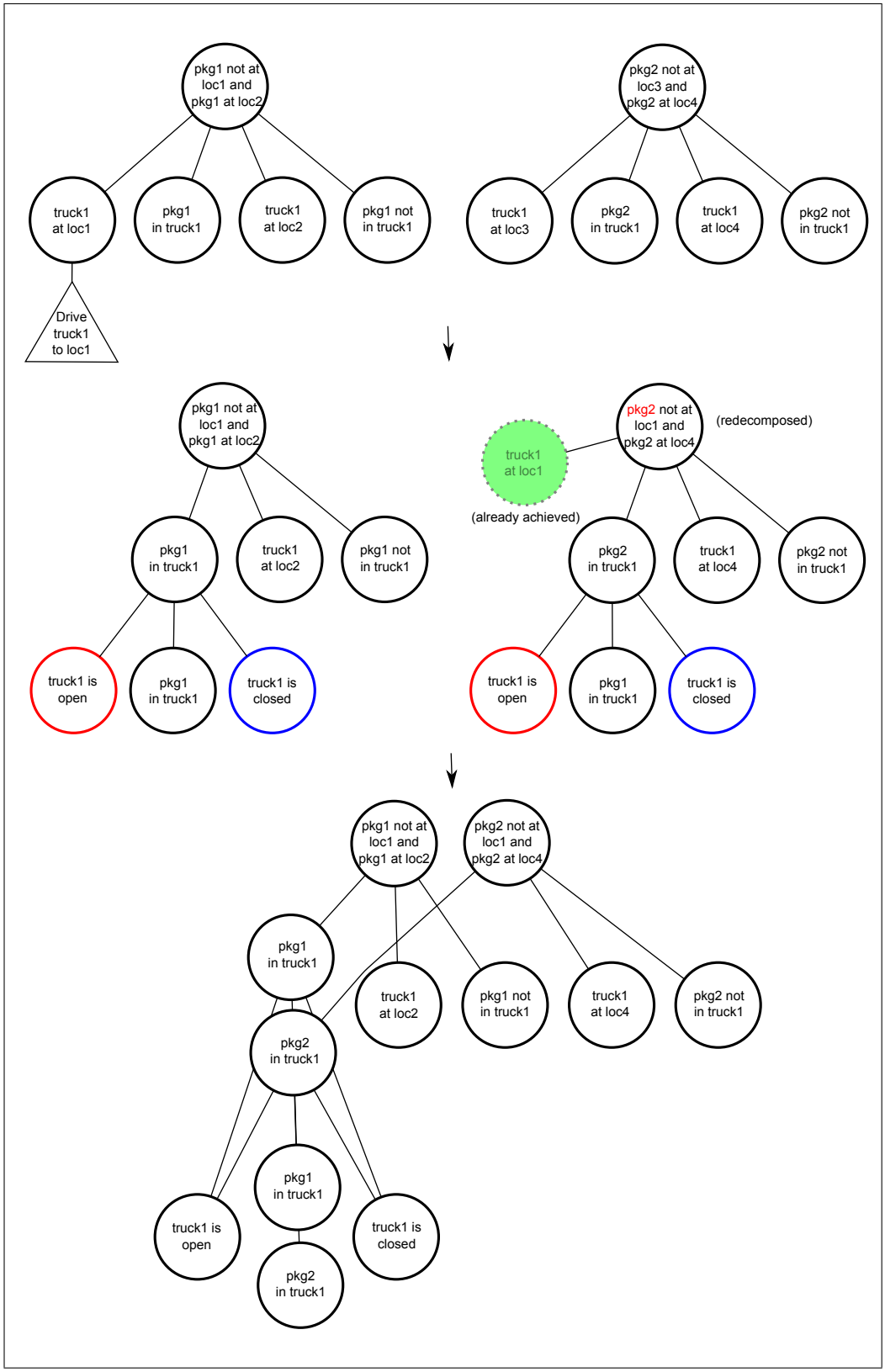


Figure 6.12: TADPOLE takes advantage of an opportunity to interleave

could not plan for in the initial state) while executing its plan, modifying only the necessary parts of its decomposition hierarchy.

Chapter 7

Conclusion

This thesis has presented three main contributions:

- a rule framework that decouples the concepts of tasks and goals resulting in more re-usable decomposition rules.
- HOPPER, an implemented planning system whose least-commitment decomposition strategy allows it to handle and recover from unexpected disruptive events, and whose novel interleaving algorithm allows it to take advantage of opportunities to shorten its plan by executing multiple sub-goals in parallel.
- TADPOLE, an implemented learning system that can parse and interpret the behaviour of other agents, learning both the structure and the preconditions of new decomposition rules by filling in the holes in its parse.

Rules decomposing tasks into sub-goals are most appropriate to the HPD

Rules that decompose tasks into sub-goals make up a flexible toolset for an intelligent agent. Because tasks specify only the preconditions and effects of their decomposition rules, the agent can determine from the cur-

rent state what changes it requires, and then re-use the rules to achieve multiple different goals.

The weakly constrained partial-ordering of sub-goals is particularly appropriate to a least-commitment decomposition strategy where rules are applied and decomposed only when it comes time to execute them. A rule's task tightly constrains the states to which the rule is applicable, helping to ensure that when it comes time to apply a rule, the right one will be used. A rule's sub-goals may be decomposed a significant time after their parent decomposition was and so their weak constraints ensure that the rule remains flexible in the face of possible future unexpected events.

However, the current representation will have to be extended for it to be applicable to the HPD. The decomposition rules will have to be able to deal with conditional effects within a single rule rather than inefficiently spreading them across multiple rules. They will also have to be able to represent resources and especially time which can be very important in the HPD.

HOPPER is able to react and to deliberate

HOPPER's least-commitment decomposition strategy strikes the right balance between deliberative and reactive planning. It makes future aspects of the plan increasingly abstract, only filling in the details as they become apparent. This allows it to react to unexpected events by only minimally modifying its plan. But it generates enough of the plan to allow it to look ahead and optimize it by interleaving the execution of different decompositions with shared sub-goals.

However, HOPPER does not address important aspects of the HPD. It has no mechanism for optimizing the use of resources. It will have to be extended to either handle resource management itself or to make use of a specialized system that does. A particularly important resource to manage is time. HOPPER will have to be extended to be able to wait for sub-goals to be achieved, and its interleaving algorithm will have to be extended to

make use of this time to achieve other sub-goals.

TADPOLE learns by first understanding demonstrated behaviour

The most significant contribution of this thesis is TADPOLE. It is an algorithm for learning complete decomposition rules while placing only a minimal burden on a teacher. TADPOLE extends its knowledge base by parsing, interpreting, and understanding the teacher's behaviour in terms of the decomposition rules it already knows.

When parsing a teacher's lesson, TADPOLE's biggest challenge is dealing with the enormous amount of irrelevant state information without eliminating any important objects from its consideration. TADPOLE currently uses a variety of heuristics to identify relevant objects, but there is a great deal of scope for extending TADPOLE to use additional domain knowledge to facilitate its parsing by helping it identify important, relevant objects. TADPOLE's interaction with the teacher is also extremely restricted, and it could be extended to make use of additional communication with the teacher.

No matter how effective TADPOLE is in learning decomposition rules or how careful the teacher is in presenting appropriate lessons, it is inevitable that TADPOLE will learn redundant rules describing different applications of the same rule. The most important way that TADPOLE needs to be extended is a mechanism allowing it to detect redundant rules into a single rule.

Because TADPOLE's efficiency degrades steadily as the number of rules it knows increases, a rule indexing system for determining which rules are appropriate to the current domain will have to eventually be included. For example, it is unreasonable for TADPOLE to consider rules for mending fences when it is observing the teacher make an omelette.

HOPPER and TADPOLE form the foundation of a generally intelligent agent

HOPPER and TADPOLE together form an agent that is capable of learning and achieving routine behaviour in a large, complex, unpredictable domain. However, this is not enough for an independent agent to be able to operate effectively in the HPD. No matter how extensive and good its rule set, the agent will have to be able to achieve completely novel tasks it has never seen before, and it will have to be able to interpret the behaviour and motivation of non-teacher agents behaving in unusual ways.

Although such problems exceed the capabilities of HOPPER and TADPOLE, both of them provide a solid foundation upon which to build an agent that *can* solve such challenges.

Appendix A

A.1 HOPPER pseudo-code

A.1.1

achieveGoal(goal, variableConstraints, knownRules)
    hierarchy ← **initializeHierarchy**(goal, variableConstraints)
    action ← **cycle**(<>, <>, hierarchy, <>, <>, knownRules)

    while(action ≠ SUCCESS **and** action ≠ FAIL)
        executeAction(action)
        action ← **cycle**(<>, <>, hierarchy, <>, <>, knownRules)
    endwhile

    return action

A.1.2

initializeHierarchy(goal, variableConstraints)
     matchedGoal ← <goal, <>, variableConstraints, <>>
    rootNode ← <matchedGoal, <>, {}, <>, {}>
    hierarchy ← <rootNode, {}>
    return hierarchy

A.1.3

```
cycle(prevState, predictedDiff, hierarchy, interleavingStack, knownRules)
    currentState ← sense()
    stateDiff ← calcStateDiff(prevState, currentState)
    unexpectedlyChangedNodes ← calcUnexpected(stateDiff, predictedDiff)

    for interleaving in interleavingStack do
        if interleavingAffected(interleaving, unexpectedlyChangedNodes) = TRUE then
            interleavingStack ← interleavingStack \ {interleaving}
        endif
    endfor

    hierarchy ← updateHierarchyUnexpected(hierarchy, unexpectedlyChangedNodes)
    hierarchy ← updateHierarchy(hierarchy, hierarchy, currentState, knownRules)
    interleavingStack ← updateInterleavings(hierarchy, interleavingStack)

    if hierarchy = FAIL then return FAIL
    endif

    if hierachy is empty then return SUCCESS
    endif

    if interleavingStack is empty then
        atomicAction ← chooseAtomicAction(hierarchy)
    else
        interleaving ← peek from interleavingStack
        atomicAction ← chooseActionFromCandidates(interleaving)
    endif

    predictedDiff ← makePrediction(currentState, atomicAction, matchedTask)
    prevState ← currentState

    return atomicAction
```

A.1.4

interleavingAffected(interleaving, unexpectedlyChangedNodes)
    for hierarchy in interleaving **do**
        <node, subHierarchies> ← hierarchy
        <matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules> ← node
        <goal, goalToStateMap, decompVarConstraints, state> ← matchedGoal

        if (stateNodes of goalToStateMap) ∩ unexpectedlyChangedNodes ≠ {} **then**
            return TRUE
        endif
    endfor
    return FALSE

A.1.5

updateHierarchyUnexpected(hierarchy, unexpectedlyChangedNodes)
    <node, subHierarchies> ← hierarchy
    <matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules> ← node
    <goal, goalToStateMap, decompVarConstraints, state> ← matchedGoal

    if (stateNodes of goalToStateMap) ∩ unexpectedlyChangedNodes ≠ {} **then**
        matchedGoal ← <goal, <>, decompVarConstraints, <>>
        node ← <matchedGoal, <>, {}, <>, {}>
        hierarchy ← <node, {}>
    else
        for each subHierarchy in subHierarchies **do**
            subHierarchy ← **updateHierarchyUnexpected**(subHierarchy,
                                                unexpectedlyChangedNodes)
            subHierarchies ← replace old subHierarchy with new subHierarchy
        endfor
    endif

    return hierarchy

A.1.6

```
updateHierarchy(root, hierarchy, currentState, knownRules)
     <node, subHierarchies> ← hierarchy
    <matchedGoal, rule, matchedTask, satisfiedSubGoals, d, prevRules> ← node

    unsatisfiedSubGoals ← {}
    for each unconstrained subHierarchy in subHierarchies do
        <subNode, ssh> ← subHierarchy
        <matchedSubGoal, mst, sssg, d, pr> ← subNode
        <subGoal, gtsm, vc, s> ← matchedSubGoal

        subHierarchy ← updateHierarchy(root, subHierarchy, currentState,
                                                knownRules)

        if subHierarchy = FAIL then
            return FAIL
        endif

        if subHierarchy is empty then
            subHierarchies ← subHierarchies \ {subHierarchy}
            satisfiedSubGoals ← satisfiedSubGoals ∪ {matchedSubGoal}
        else
            unsatisfiedSubHs ← unsatisfiedSubHs ∪ {subHierarchy}
            subHierarchies ← replace old subHierarchy with new subHierarchy
        endif
        hierarchy ← <node, subHierarchies>
    endfor

    allUnsatisfiedAreCleanup ← TRUE
    noUnsatisfiedAreCleanup ← TRUE
    for each subHierarchy in unsatisfiedSubHs do
        <subNode, ssh> ← subHierarchy
        <matchedSubGoal, mst, sssg, d, pr> ← subNode
        <subGoal, gtsm, vc, s> ← matchedSubGoal
        if subGoal is clean-up then
            noUnsatisfiedAreCleanup ← FALSE
        else
```

```
            allUnsatisfiedAreCleanup ← FALSE
        endif
    endfor

    if allUnsatisfiedAreCleanup = TRUE then
        rootCopy ← root with node removed
        satisfiedGoals ← getUnconstrainedSatisfiedGoals(rootCopy, currentState)
        originalUnsatisfiedSubHs ← unsatisfiedSubHs

        loop
            unsatisfiedSubH ← unconstrainted subHierarchy from unsatisfiedSubHs
            <subNode, subSubHs> ← unsatisfiedSubH
            <subMatchedGoal, smt, sssg, sd, spr> ← subNode

            if ∃goal ∈ satisfiedGoals where isInverse(goal, subMatchedGoal) = TRUE
            then
                unsatisfiedSubHs ← unsatisfiedSubHs \ {unsatisfiedSubH}
                subHierarchies ← subHierarchies \ {unsatisfiedSubH}
                hierarchy ← <node, subHierarchies>
            endif
        loopwhile originalUnsatisfiedSubHs ≠ unsatisfiedSubHs
    endif

    if matchedGoal satisfied in currentState and noUnsatisfiedAreCleanup = TRUE
    then
        hierarchy ← <>

        if decomp not empty and unsatisfiedSubHs is empty then
            <rule, ruleDecompCount> ← decomp
            rule ← refineRule(rule, matchedTask, satisfiedSubGoals)
        endif

        return hierarchy
    endif

    if unsatisfiedSubGoals is empty then
        if ruleDecompCount > DECOMPOSITION_LIMIT then
            prevRules ← prevRules ∪ {rule}
```

```
        endif
        <goal, goalToStateMap, varConstraints, state> ← matchedGoal
        goalToStateMap ← findBestGoalToStateMap(goal, currentState, varConstraints)
        matchedGoal ← <goal, goalToStateMap, varConstraints, currentState>

        rulesTried ← {}
        loop
            hierarchy ← decompose(matchedGoal, currentState, knownRules,
                                  prevRules ∪ rulesTried)

            if hierarchy is empty then
                return FAIL
            endif

            <node, subHierarchies> ← hierarchy
            <mg, rule, mt, ssg, d, prevRules> ← node

            hierarchy ← updateRuleDecompCount(hierarchy, rule, ruleDecompCount)
            hierarchy ← updateHierarchy(hierarchy, currentState, knownRules)

            if hierarchy = FAIL then
                rulesTried ← rulesTried ∪ {rule}
            endif
        loopwhile hierarchy = FAIL
    endif

    return hierarchy
```

A.1.7

getUnconstrainedSatisfiedGoals(hierarchy, currentState)
    unconstrainedSatisfiedGoals ← {}
    <node, subHierarchies> ← hierarchy
    <matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules> ← node

    if matchedGoal satisfied in currentState **then**
        unconstrainedSatisfiedGoals ← unconstrainedSatisfiedGoals ∪ {matchedGoal}
    endif

    for each unconstrained subHierarchy in subHierarchies **do**
        unconstrainedSSGs ← **getUnconstrainedSatisfiedGoals**(subHierarchy, currentState)
        unconstrainedSatisfiedGoals ← unconstrainedSatisfiedGoals ∪ unconstrainedSSGs
    endfor

    return unconstrainedSatisfiedGoals

A.1.8

```
decompose(matchedGoal, currentState, knownRules, prevRules)
     goalStateDifference ← calcGoalStateDiff(matchedGoal, currentState)
    rulesToTry ← knownRules \ prevRules

    possibleDecomps ← {}
    for each rule in rulesToTry do
        <task, subGoals, constraints> ← rule
        taskToStateMap ← findBestTaskToStateMap(task, goalStateDifference)

        matchedTask ← <task, taskToStateMap, currentState>
        decomp ← <rule, 1>
        node ← <matchedGoal, matchedTask, {}, decomp, prevRules>

        subHierarchies ← {}
        for each subGoal in subGoals do
            subHierarchy ← initializeHierarchy(subGoal, constraints)
            subHierarchies ← subHierarchies ∪ subHierarchy
        endfor

        hierarchy ← <node, subHierarchies>
        possibleDecomps ← possibleDecomps ∪ {<hierarchy, taskToStateMap>}
    endfor

    if possibleDecomps is empty then
        return <>
    endif

    <hierarchy, taskToStateMap> ← get best from possibleDecomps
    return hierarchy
```

A.1.9

updateRuleDecompCount(hierarchy, prevRule, prevRuleDecompCount)
    <node, subHierarchies> ← hierarchy
    <matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules> ← node
    <rule, ruleDecompCount> ← decomp

    if prevRule = rule **then**
        ruleDecompCount ← ruleDecompCount + 1
    else
        ruleDecompCount ← 1
    endif

    decomp ← <rule, ruleDecompCount>
    node ← <matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules>
    hierarchy ← <node, subHierarchies>
    return hierarchy

A.1.10

refineRule(rule, matchedTask, satisfiedSubGoals)
    <originalTask, originalSubGoals, originalConstraints> ← rule
    matchedTask ← **refineTask**(matchedTask)
    <task, taskToStateMap, state1> ← matchedTask

    subGoals ← {}
    subGoalToStateMaps ← {}
    for each matchedSubGoal in satisfiedSubGoals **do**
        matchedSubGoal ← **refineGoal**(matchedSubGoal)
        <subGoal, subGoalToStateMap, state2> ← matchedSubGoal
        subGoals ← subGoals ∪ {subGoal}
        subGoalToStateMaps ← subGoalToStateMaps ∪ {subGoalToStateMap}
    endfor

    constraints ← **refineSubGoalOrdering**(originalConstraints, satsifiedSubGoals)
    constraints ← **refineRuleVariables**(constraints, taskToStateMap, subGoalToStateMaps)
    rule ← <task, subGoals, constraints>
    return rule

A.1.11

```
updateInterleavings(hierarchy, interleavingStack)
    loop
        originalStack ← interleavingStack
        interleaving ← peek from interleavingStack
        for subHierarchy in interleaving do
            if subHierarchy no longer present in hierarchy then
                interleaving ← interleaving \ {subHierarchy}
            endif
        endfor
        if interleaving is empty then
            interleavingStack ← interleavingStack \ {interleaving}
        endif
    loopwhile originalStack ≠ interleavingStack

    interleaving ← {}
    if interleavingStack is empty then
        interleaving ← interleaveHierarchy(hierarchy)
    else
        topInterleaving ← peek from interleavingStack
        unconstrainedHs ← get unconstrained elements from topInterleaving
        interleavingm ← interleave(unconstrainedHs)
    endif

    if interleaving ≠ {} then
        interleavingStack ← push interleaving on to interleavingStack
    endif

    return interleavingStack
```

A.1.12

```
interleaveHierarchy(hierarchy)
    <node, subHierarchies> ← hierarchy

    if subHierarchies is empty then
        return {}
    endif

    unconstrainedSubHs ← get unconstrained elements from subHierarchies
    if |unconstrainedSubHs| = 1 then
        interleaving ← interleaveHierarchy(unconstrainedSubHs)
        return interleaving
    else
        interleaving ← interleave(unconstrainedSubHs)
        if interleaving ≠ {} then
            return interleaving
        else
            for h in unconstrainedSubHs do
                interleaving ← interleaveHierarchy(h)
                if interleaving ≠ {} then
                    return interleaving
                endif
            endfor
        endif
    endif
    return {}
```

A.1.13

```
interleave(coOrderedHierarchies)
    for <h1, h2> every two pair combination of hs from coOrderedHierarchies do
        for subH1 ← depthfirst traversal of unconstrained subHs of h1 do
            <node, subHs1> ← subH1
            <matchedGoal1, mt, ssg, d, pr> ← node
            for subH2 ← depth first traversal of unconstrained subHs of h2 do
                <node, subHs2> ← subH2
                <matchedGoal2, mt, ssg, d, pr> ← node
                if matchedGoal1 is the same as matchedGoal2 then
                    interleaving ← interleaveDecomps(subHs1, subHs2)
                endif
            endfor
        endfor
    endfor

    if interleaving = then
        return interleaving
    endif

    for every h3 in coOrderedHierarchies where h3 ≠ h1 and h3 ≠ h2 do
        for subH3 ← depth first traversal of unconstrained subHs of h3 do
            <node, subHs3> ← subH3
            <matchedGoal3, mt, ssg, d, pr> ← node
            if matchedGoal3 is the same as a goal in interleaving then
                interleaving ← interleaveDecomps(interleaving, subHs3)
            endif
        endfor
    endfor

    return interleaving
```

A.1.14

chooseAtomicAction(hierarchy)
<node, subHierarchies> ← hierarchy
<matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules> ← node

if subHierarchies is empty **then**
<task, taskToStateMap, state> ← matchedTask
<rule, decompCount> ← decomp
<task, action, constraints> ← rule
<actionName, actionArgs> ← action
actionArgs ← calcActionArgs(taskToStateMap, constraints, actionArgs)
action ← <actionName, actionArgs>
return <action, matchedTask>
endif

<action, matchedTask> ← **chooseActionFromCandidates**(subHierarchies)
return <action, matchedTask>

A.1.15

chooseActionFromCandidates(hierarchies)
candidates ← {}
for each unconstrained hierarchy in hierarchies **do**
<node, subHierarchy> ← hierarchy
percentSatisfied ← **calcPercentSatisfied**(subHierarchy)
candidates ← candidates ∪ {<subHierarchy, percentSatisfied>}
endfor

bestCandidates ← get elements with highest percentSatisfied from candidates
bestCandidate ← get element with shallowest sub-hierarchy from bestCandidates
<subHierarchy, percentSatisfied> ← bestCandidate
<action, matchedTask> ← **chooseAtomicAction**(subHierarchy)
return <action, matchedTask>

A.1.16

calcPercentSatisfied(hierarchy)

<node, subHierarchies> ← hierarchy

<matchedGoal, matchedTask, satisfiedSubGoals, decomp, prevRules> ← node

<rule, decompCount> ← decomp

<rule, subGoals, constraints> ← rule

return |satisfiedSubGoals| / |subGoals|

A.2 TADPOLE pseudo-code

A.2.1

```
learnLesson(knownRules, actionNames, firstState, secondState)
    partialParses ← {}
    prevState ← firstState
    currentState ← secondState

    while currentState is not END_OF_LESSON do
        stateDiff ← calcStateDiff(prevState, currentState)
        action ← determineActionExecuted(prevState, currentState)
        atomicHierarchy ← findBestAtomicHierarchy(stateDiff, knownRules, action)

        if partialParses is empty then
            partialParse ← <{atomicHierarchy}, atomicHierarchy, {}>
            partialParses ← partialParses ∪ {partialParse}
        else
            allNeighbouringParses ← {}
            for partialParse in partialParses do
                <topHs, newHs, partMatchedRules> ← partialParse
                topHs ← topHs ∪ {atomicHierarchy}
                partialParse ← <topHs, {atomicHierarchy}, partMatchedRules>

                parsesToExtend ← {partialParse}
                for parseToExtend in parsesToExtend do
                    parseToExtend ← extendParse(parseToExtend)
                    allNeighbouringParses ← allNeighbouringParses ∪ {partialParse}
                    neighbours ← getNeighbouringParses(parseToExtend)
                    parsesToExtend ← ∪ neighbours
                endfor
            endfor
        endif

        partialParses ← allNeighbouringParses
        prevState ← currentState
        currentState ← observeNextState()
    endwhile
```

```
bestParse ← get best element from partialParses

<topHs, newHs, partMatchedRules> ← bestParse
if |topHs| > 1 then
    bestParse ← completePartialParse(bestParse)
endif

learnedRules ← refineRules(bestParse, knownRules)
return learnedRules
```

A.2.2

```
findBestAtomicHierarchy(stateDiff, knownRules, actionExecuted)
    candidateRules ← {}
    for rule in knownRules do
        if rule is atomic then
            <task, action, constraints> ← rule
            if action = actionExecuted then
                candidateRules ← candidateRules ∪ {rule}
            endif
        endif
    endfor

    <newRule, taskToStateMap> ← constructAtomicRule(stateDiff, actionExecuted)
    <task, action, constraints> ← newRule
    matchedTask ← <task, taskToStateMap, stateDiff>
    bestNode ← <stateDiff, matchedTask, {}, newRule>

    for rule in candidateRules do
        <task, action, constraints> ← rule
        taskToStateMap ← matchTaskToStateDiff(task, stateDiff, {})
        if taskToStateMap is better than mapping in bestNode then
            matchedTask ← <task, taskToStateMap, stateDiff>
            bestNode ← <stateDiff, matchedTask, {}, rule>
        endif
    endfor

    hierarchy ← <bestNode, {}>
    return hierarchy
```

A.2.3

extendParse(partialParse)
    <topHs, newHs, partMatchedRules> ← partialParse
    for newH in newHs **do**
        partMatchedRules ← **extendPartialMatches**(partMatchedRules, newH)
        partMatchedRules ← partMatchedRules ∪ **matchToNew**(knownRules, newH)
    endfor

    partMatchedRules ← **matchToInterleaved**(partMatchedRules, topHs, newHs)
    partMatchedRules ← **matchToAlreadyAchieved**(partMatchedRules, topHs, newHs)

    partialParse ← <topHs, newHs, partMatchedRules>
    return partialParse

A.2.4

extendPartialMatches(partMatchedRules, newH)
    <newNode, subHs> ← newH
    <stateDiff, mt, {}, r> ← newNode
    extMatchedRules ← {}
    for matchedRule in partMatchedRules **do**
        <rule, mt, matchedSGs> ← matchedRule
        <task, subGoals, vc> ← rule

        subGoalsToSkip ← {}
        for matchedSG in matchedSGs **do**
            <goal, gtsm, h> ← matchedSG
            subGoalsToSkip ← subGoalsToSkip ∪ {goal}
        endfor
        subGoalsToMatch ← subGoals \ subGoalsToSkip

        for each subGoal in subGoalsToMatch **do**
            goalToStateMap ← **matchGoalToStateDiff**(subGoal, stateDiff,
                                                    matchedSubGoals, vc)
            if goalToStateMap not empty **then**
                matchedSG ← <subGoal, goalToStateMap, newH>
                matchedSubGoals ← matchedSubGoals ∪ {matchedSG}

```
                matchedRule ← <rule, mt, matchedSubGoals>
                extMatchedRules ← extMatchedRules ∪ {matchedRule}
            endif
        endfor
    endfor
    return partMatchedRules ∪ extMatchedRules
```

A.2.5

```
matchToNew(knownRules, newH)
    partMatchedRules ← {}
    for rule in knownRules do
        <task, subGoals, vc> ← rule
        <stateDiff, <>, mt, r> ← newNode
        for subGoal in subGoals do
            goalToStateMap ← matchGoalToStateDiff(subGoal, stateDiff, {}, vc)
            if goalToStateMap not empty then
                matchedSG ← <subGoal, goalToStateMap, newH>
                matchedSubGoals ← matchedSubGoals ∪ {matchedSG}
                matchedRule ← <rule, matchedSubGoals>
                partMatchedRules ← partMatchedRules ∪ {matchedRule}
            endif
        endfor
    return partMatchedRules
```

A.2.6

```
matchToInterleaved(partMatchedRules, topHs, newHs)
    extMatchedRules ← {}
    for matchedRule in partMatchedRules do
        <rule, mt, matchedSGs> ← matchedRule

        matchedHs ← {}
        for matchedSG in matchedSGs do
            <g, gtsm, hierarchy> ← matchedSG
            matchedHs ? matchedHs ∪ {hierarchy}
        endfor
```

```
        firstH ← get first element of matchedHs

        subGoalsToSkip ← {}
        for matchedSG in matchedSGs do
            <goal, gtsm, h> ← matchedSG
            subGoalsToSkip ← subGoalsToSkip ∪ {goal}
        endfor
        subGoalsToMatch ← subGoals \ subGoalsToSkip

        if matchedHs ∩ newHs ≠ {} then
            for each subGoal in subGoalsToMatch do
                goalToStateMappings ← get all mappings from matchedSGs
                allVarsBound ← allVarsBound(subGoal, vc, goalToStateMappings)
                if allVarsBound = TRUE then
                    for each topH in topHs do
                        if topH earlier than firstH then
                            <node, subHs> ← topH
                            <stateDiff, mt, mgs, r> ← node
                            gtsm ← matchGoalToStateDiff(subGoal, stateDiff, matchedSGs,
                                                        vc)
                            if gtsm not empty then
                                matchedSG ← <subGoal, gtsm, topH>
                                matchedSGs ← matchedSGs ∪ {matchedSG}
                                matchedRule ← <rule, matchedSubGoals>
                                extMatchedRules ← extMatchedRules ∪ {matchedRule}
                            endif
                        endif
                    endfor
                endif
            endfor
        endif
    endfor
    return partMatchedRules ∪ extMatchedRules
```

A.2.7

```
matchToAlreadyAchieved(partMatchedRules, topHs, newHs)
    extMatchedRules ← {}
    for matchedRule in partMatchedRules do
        <rule, matchedSGs> ← matchedRule

        matchedHs ← {}
        for matchedSG in matchedSGs do
            <g, gtsm, hierarchy> ← matchedSG
            matchedHs ← matchedHs ∪ {hierarchy}
        endfor

        firstH ← get first element of matchedHs
        <firstNode, subHs> ← firstH
        initialState ← get initial state of stateDiff

        subGoalsToSkip ← {}
        for matchedSG in matchedSGs do
            <goal, gtsm, h> ← matchedSG
            subGoalsToSkip ← subGoalsToSkip ∪ {goal}
        endfor
        subGoalsToMatch ← subGoals \ subGoalsToSkip

        if matchedHs ∩ newHs ≠ {} then
            for each subGoal in subGoalsToMatch do
                goalToStateMappings ← get mappings from matchedSGs
                allVarsBound ← allVarsBound(subGoal, vc, goalToStateMappings)
                if allVarsBound = TRUE then
                    gtsm ← matchGoalToState(subGoal, initialState, matchedSGs,
                                            vc)
                    if gtsm not empty then
                        matchedSG ← <subGoal, gtsm, initialState>
                        matchedSGs ← matchedSGs ∪ {matchedSG}
                        matchedRule ← <rule, matchedSubGoals>
                        extMatchedRules ← extMatchedRules ∪ {matchedRule}
                    endif
                endif
```

```
            endfor
        endif
    endfor
    return partMatchedRules ∪ extMatchedRules
```

A.2.8

```
getNeighbouringParses(partialParse)
     <topHs, newHs, partMatchedRules> ← partialParse
    if |topHs| > LIMIT then
        return {}
    endif

    neighbours ← getHoleFillingNeighbours(partialParse)

    completeMatchedRules ← {}
    for partMatchedRule in partMatchedRules do
        <rule, mt, matchedSGs> ← partMatchedRule
        <task, subGoals, vc> ← rule
        if |matchedSGs| = |subGoals| then
            taskMatching ← calcTaskMatching(task, matchedSGs)

            if taskMatching is not empty then
                partMatchedRule ← <rule, taskMatching, matchedSGs>
                completeMatchedRules ← completeMatchedRules ∪ {partMatchedRule}
            endif
        endif
    endfor

    ruleCombinations ← get all combinations of completeMatchedRules

    for rulesCombo in ruleCombinations do
        allMatchedTopHs ← {}
        newNodeMatched ← FALSE
        everyMatchedRulePartiallyIndependent ← TRUE
        everyMatchedRulePartiallyOverlapping ← TRUE
```

```
        for partMatchedRule in rulesCombo do
            <rule, <>, matchedSGs> ← partMatchedRule
            matchedTopHs ← {}
            for matchedSG in matchedSGs do
                <subGoal, goalToStateMap, topH> ← matchedSG
                <node, subHs> ← topH
                matchedTopHs ← matchedTopHs ∪ {topH}
                if node ∈ newNodes then
                    newNodeMatched ← TRUE
                endif
            endfor

            if matchedTopHs \ allMatchedTopHs = {} then
                everyMatchedRulePartiallyIndependent ← FALSE
            endif

            if matchedTopHs ≠ {}
              and matchedTopHs \ allMatchedTopHs = matchedTopHs then
                everyMatchedRulePartiallyOverlapping ← FALSE
            endif

            allMatchedTopHs ← allMatchedTopHs ∪ matchedTopHs
        endfor

        if allMatchedTopHs are contiguous
          and newNodeMatched = TRUE
          and everyMatchedRulePartiallyIndependent = TRUE
          and everyMatchedRulePartiallyOverlapping = TRUE then
            neighbour ← getNeighbour(partialParse, rulesCombo)
            neighbours ← neighbours ∪ {neighbour}
        endif
    endfor

    return neighbours
```

A.2.9

calcTaskMatching(task, matchedSGs)
    firstMatchedSG ← get first element of matchedSGs
    <goal, gtsm, topH> ← firstMatchedSG
    <node, subHs> ← topH
    <stateDiff, mt, mgs, r> ← node
    initialState ← get initial state of stateDiff

    lastMatchedSG ← get last element of matchedSGs
    <goal, gtsm, topH> ← firstMatchedSG
    <node, subHs> ← topH
    <stateDiff, mg, mt, r> ← node
    finalState ← get final state of stateDiff

    topStateDiff ← **calcStateDiff**(initialState, finalState)
    taskToStateMap ← **matchTaskToStateDiff**(task, topStateDiff, matchedSGs)

    if taskToStateMap is not empty **then**
        taskMatching ← <task, taskToStateMap, topStateDiff>
        return taskMatching
    else
        return <>
    endif

A.2.10

getNeighbour(partialParse, completedRules)
    <topHs, newNodes, partMatchedRules> ← partialParse
    matchedTasks ← {}
    for completedRule in completedRules **do**
        <rule, matchedTask, matchedSGs> ← completedRule
        matchedTasks ← matchedTasks ∪ matchedTask
    endfor

    if all matchedTasks the same **then**
        completedRules ← **combineRepeated**(completedRules)
    endif

```
newTopHs ← {}
for completedRule in completedRules do
    <rule, matchedTask, matchedSGs> ← completedRule
    matchedTopHs ← {}
    for matchedSG in matchedSGs do
        <subGoal, goalToStateMap, topH> ← matchedSG
        <node, subHs> ← topH
        <sd, matchedGoals, mt, r> ← node

        matchedGoals ← matchedGoals ∪ {matchedSG}

        node ← <sd, matchedGoals, mt, r>
        topH ← <node, subHs>
        topHs ← replace old topH with new topH
        matchedTopHs ← matchedTopHs ∪ {topH}
    endfor

    <task, tsm, stateDiff> ← matchedTask
    newTopNode ← <stateDiff, matchedTask, matchedSGs, rule>
    newTopH ← <newTopNode, matchedTopHs>
    newTopHs ← newTopHs ∪ {newTopH}
    topHs ← replace matchedTopHs with newTopH
endfor

for partMatchedRule in partMatchedRules do
    <rule, matchedTask, matchedSGs> ← partMatchedRule
    for matchedSG in matchedSGs do
        <subGoal, goalToStateMap, topH> ← matchedSG
        if topH ∉ topHs then
            partMatchedRules ← partMatchedRules \ {partMatchedRule}
        endif
    endfor
endfor

neighbour ← <topHs, newTopHs, partMatchedRules>
return neighbour
```

A.2.11

```
getHoleFillingNeighbours(partialParse)
    <topHs, newHs, partMatchedRules> ← partialParse
    neighbours ← {}

    for matchedRule in partMatchedRules do
        <rule, matchedSGs> ← matchedRule
        <task, subGoals, vc> ← rule

        unmatchedSGs ← subGoals
        matchedHs ← {}
        for matchedSG in matchedSGs do
            <g, gtsm, hierarchy> ← matchedSG
            matchedHs ← matchedHs ∪ {hierarchy}
            unmatchedSGs ← unmatchedSGs \ {g}
        endfor
        {unmatchedSG} ← unmatchedSGs

        <fstMatched, midUnmatched, sndMatched> ← groupMatchedHs(topHs, matchedHs)

        if matchedHs ∩ newHs ≠ {}
         and |matchedSGs| + 1 = |subGoals|
         and |midUnmatched| > 1 then
            firstUnmatchedH ← get first element of midUnmatched
            <node, subHs> ← firstUnmatchedH
            <firstStateDiff, mt1, mgs, r1> ← node
            initialState ← get earlier state of firstStateDiff

            lastUnmatchedH ← get last element of midUnmatched
            <node, subHs> ← lastUnmatchedH
            <lastStateDiff, <>, mt2, r2> ← node
            finalState ← get later state of lastStateDiff

            midStateDiff ← calcStateDiff(initialState, finalState)

            goalToStateMap ← matchGoalToStateDiff(unmatchedSG, midStateDiff,
                                                  matchedSGs, vc)
```

```
            if goalToStateMap not empty then
                <newRule, matchedST, matchedSSGs> ← learnRule(midStateDiff,
                                                              midUnmatched)
                for each topH in midUnmatched do
                    <node, subHs> ← topH
                    <stateDiff, mt, mgs, r> ← node
                    correspondingMatchedSSG ← get corresponding sub-sub-goal
                                              from matchedSSGs
                    node ← <stateDiff, mt, {correspondingMatchedSSG}, r>
                    topH ← <node, subHs>
                    midUnmatched ← replace old topH with new topH
                endfor

                node ← <midStateDiff, matchedST, {}, newRule>
                newTopH ← <newTopNode, midUnmatched>
                topHs ← replace midUnmatched with newTopH

                matchedSG ← <umatchedSG, goalToStateMap, newTopH>
                matchedSGs ← matchedSGs ∪ {matchedSG}

                taskMatching ← calcTaskMatching(task, matchedSGs)
                if taskMatching is not empty then
                    completedRule ← <rule, taskMatching, matchedSubGoals>
                    neighbour ← getNeighbour(partialParse, completedRule)
                    neighbours ← neighbours ∪ {neighbour}
                endif
            endif
        endif
    endfor
    return neighbours
```

A.2.12

```
groupMatchedHs(topHs, matchedHs)
    firstMatched ← {}
    unmatchedMid ← {}
    secondMatched ← {}
    secondMatchedComplete ← FALSE
    otherMatched ← FALSE
    for topH in topHs do
        if topH ∈ matchedHs then
            if secondMatchedComplete = TRUE then
                otherMatched ← TRUE
            endif

            if unmatchedMid = {} then
                firstMatched ← firstMatched ∪ {topH}
            else
                secondMatched ← secondMatched ∪ {topH}
            endif
        else
            if secondMatched = {} then
            unmatchedMid ← unmatchedMid ∪ {topH}
            else
                secondMatchedNodesComplete ← TRUE
            endif
        endif
    endfor

    if otherMatched = TRUE then
        return <, {}, {}>
    else
        return <firstMatched, unmatchedMid, secondMatched>
    endif
```

A.2.13

completePartialParse(partialParse)
    <topHs, newHs, pmr> ← partialParse

    firstH ← get first element of topHs
    <node, subHs> ← firstH
    <firstStateDiff, mt, mgs, r> ← node
    initialState ← get earlier state of firstStateDiff

    lastH ← get last element of topHs
    <node, subHs> ← lastH
    <lastStateDiff, mt, mgs, r> ← node
    finalState ← get later state of lastStateDiff

    stateDiff ← **calcStateDiff**(initialState, finalState)

    completedRule ← **learnRule**(stateDiff, topHs)
    completedParse ← **getNeighbour**(partialParse, {completedRule})
    return completedParse

A.2.14

refineRules(hierarchy, knownRules)
    <node, subHs> ← hierarchy
    <stateDiff, matchedTask, mgs, rule> ← node
    <task, subGoals, constraints> ← rule

    matchedSubGoals ← {}
    for subH in subHs **do**
        <subNode, subSubHs> ← subH
        <sd, mt, matchedGoals, r> ← subNode
        matchedSubGoal ← get matched goal corresponding to rule
        matchedSubGoals ← matchedSubGoals ∪ {matchedSubGoal}
    endfor

    knownRules ← knownRules \ {rule}
    rule ← **refineRule**(rule, matchedTask, matchedSubGoals)

```
knownRules ← knownRules ∪ {rule}

for subH in subHs do
    knownRules ← refineRules(subH, knownRules)
endfor

return knownRules
```

Appendix B

B.1 Example state description in the Kitchen Domain

drawer1: MadeOf=[wood], Colour=[white], Open=[no], Vertical=[yes], Type=[container, drawer], Temperature=[cool]
drawer1 → link(135L): [Supports, Contains] → spoon2
drawer1 → link(131L): [Supports, Contains] → knife2
drawer1 → link(116L): [PartOf] → bench1
drawer1 → link(125L): [Supports, Contains] → fork1
drawer1 → link(133L): [Supports, Contains] → fork2
drawer1 → link(119L): [ConsistsOf, ControlledBy] → drawerHandle1
drawer1 → link(121L): [Supports, Contains] → knife1
drawer1 → link(127L): [Supports, Contains] → spoon1

spoon2: MadeOf=[metal], Colour=[steel], Type=[spoon, cutlery], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
spoon2 → link(134L): [On, In] → drawer1

teacherArm: Type=[arm]
teacherArm → link(101L): [PartOf] → teacher1

spoon1: MadeOf=[metal], Colour=[steel], Type=[cutlery, spoon], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
spoon1 → link(126L): [On, In] → drawer1

sugar1: Colour=[white], Type=[food, powder, sugar], Temperature=[cool], Taste=[sweet]
sugar1 → link(122L): [On, In] → tin1

hopper: Vertical=[yes], Type=[student, robot], Temperature=[cool]
hopper → link(78L): [On] → floor1
hopper → link(77L): [ConsistsOf] → robotArm

faucet1: MadeOf=[metal], Colour=[steel], Type=[faucet], Graspable=[yes], Temperature=[cool], Clean=[yes]
faucet1 → link(8L): [On, Controls] → tap1
faucet1 → link(9L): [PartOf] → sink1

clock1: MadeOf=[ceramics], Colour=[blue], Time=[12:00], Type=[clock], Temperature=[cool], Fragile=[yes]
clock1 → link(102L): [PartOf] → wall1

cooktop1: MadeOf=[metal], Colour=[black], Type=[cooktop], Temperature=[cool], Clean=[yes]
cooktop1 → link(24L): [ConsistsOf] → heatingElement1
cooktop1 → link(25L): [ConsistsOf] → dial1
cooktop1 → link(26L): [PartOf] → bench1

teaBox1: MadeOf=[cardboard], Colour=[yellow], Open=[no], Type=[container, box], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
teaBox1 → link(106L): [Supports, Contains] → teaBag4
teaBox1 → link(60L): [Supports, Contains] → teaBag1

teaBox1 → link(27L): [In, On] → cupboard2
teaBox1 → link(74L): [Supports, Contains] → teaBag3
teaBox1 → link(34L): [ConsistsOf, ControlledBy] → teaBoxHandle1
teaBox1 → link(62L): [Supports, Contains] → teaBag2

stool1: MadeOf=[wood], Colour=[brown], Type=[stool, furniture], Clean=[no], Temperature=[cool]
stool1 → link(140L): [On] → floor1
stool1 → link(142L): [NextTo] → table1

flour1: Colour=[white], Type=[flour, food, powder], Temperature=[cool], Taste=[flour]
flour1 → link(2L): [On, In] → tin2

tap1: MadeOf=[metal], Colour=[steel], Vertical=[yes], Running=[no], Type=[tap], Temperature=[cool], Clean=[yes]
tap1 → link(12L): [Supports, ControlledBy] → faucet1
tap1 → link(11L): [PartOf, Controls] → sink1

kettle1: MadeOf=[metal], Colour=[grey], Open=[yes], Running=[no], Type=[container, electricalKettle], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
kettle1 → link(13L): [On] → bench1
kettle1 → link(70L): [ConsistsOf, ControlledBy] → button2

cup2: MadeOf=[ceramics], Colour=[white], Open=[yes], Type=[cup, dish, container], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
cup2 → link(110L): [On, In] → cupboard2

ham1: Colour=[pink], Type=[ham, food], Graspable=[yes], Temperature=[cold], Taste=[ham], Movable=[yes]
ham1 → link(144L): [On, In] → refrigerator1

cup1: MadeOf=[ceramics], Colour=[white], Open=[yes], Type=[dish, cup, container], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
cup1 → link(48L): [On, In] → cupboard2

kitchen1: Colour=[white], Type=[location, kitchen], Temperature=[cool], Size=[medium]
kitchen1 → link(47L): [ConsistsOf] → floor1
kitchen1 → link(45L): [ConsistsOf] → wall1
kitchen1 → link(44L): [ConsistsOf, HasExit] → door1

button2: MadeOf=[plastic], Colour=[red], Type=[button], Clean=[yes], Temperature=[cool]
button2 → link(69L): [PartOf, Controls] → kettle1

fork1: MadeOf=[metal], Colour=[steel], Type=[fork, cutlery], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
fork1 → link(124L): [On, In] → drawer1

knife2: MadeOf=[metal], Colour=[steel], Type=[cutlery, knife], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
knife2 → link(130L): [On, In] → drawer1

knife1: MadeOf=[metal], Colour=[steel], Type=[cutlery, knife], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
knife1 → link(120L): [In, On] → drawer1

table1: MadeOf=[wood], Colour=[brown], Type=[surface, table, furniture], Temperature=[cool], Clean=[no]
table1 → link(129L): [Supports] → towel1
table1 → link(137L): [ConsistsOf] → tableLeg1
table1 → link(139L): [ConsistsOf] → tableLeg2

table1 → link(143L): [NextTo] → stool1
table1 → link(80L): [On] → floor1

pot2: MadeOf=[metal], Colour=[steel], Open=[yes], Type=[container, pot], Graspable=[yes], Temperature=[cool], Clean=[no], Movable=[yes]
pot2 → link(99L): [On, In] → cupboard1

pot1: MadeOf=[metal], Colour=[steel], Open=[yes], Type=[kettle, container], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
pot1 → link(98L): [On, In] → cupboard1

breadLoaf1: Colour=[bread], Type=[bread, food], Graspable=[yes], Temperature=[cold], Taste=[bread], Movable=[yes]
breadLoaf1 → link(148L): [On, In] → bag1
breadLoaf1 → link(150L): [In] → refrigerator1

fork2: MadeOf=[metal], Colour=[steel], Type=[fork, cutlery], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
fork2 → link(132L): [On, In] → drawer1

bag1: MadeOf=[plastic], Colour=[clear], Open=[no], Type=[container, bag], Graspable=[yes], Temperature=[cold], Movable=[yes]
bag1 → link(149L): [Supports, Contains] → breadLoaf1
bag1 → link(146L): [On, In] → refrigerator1

milk1: Colour=[white], Type=[liquid, food, milk], Temperature=[cold], Taste=[milk]
milk1 → link(31L): [In, On] → milkBottle1

wall1: Colour=[white], Vertical=[yes], Type=[wall], Temperature=[cool], Clean=[yes]
wall1 → link(87L): [PartOf] → kitchen1
wall1 → link(86L): [ConsistsOf] → clock1

wall1 → link(85L): [ConsistsOf] → window1

tableLeg2: MadeOf=[wood], Colour=[brown], Vertical=[yes], Type=[tableLeg], Temperature=[cool]
tableLeg2 → link(138L): [PartOf] → table1

tableLeg1: MadeOf=[wood], Colour=[brown], Vertical=[yes], Type=[tableLeg], Temperature=[cool]
tableLeg1 → link(136L): [PartOf] → table1

refrigerator1: MadeOf=[metal], Colour=[white], Type=[container, refrigerator], Temperature=[cool], Clean=[yes]
refrigerator1 → link(30L): [Supports, Contains] → milkBottle1
refrigerator1 → link(22L): [ConsistsOf, ControlledBy] → fridgeHandle1
refrigerator1 → link(153L): [Contains] → bottleCap1
refrigerator1 → link(147L): [Supports, Contains] → bag1
refrigerator1 → link(151L): [Contains] → breadLoaf1
refrigerator1 → link(66L): [Supports, Contains] → cheese1
refrigerator1 → link(4L): [On] → floor1
refrigerator1 → link(145L): [Supports, Contains] → ham1

trashCan1: MadeOf=[plastic], Colour=[white], Open=[yes], Type=[trashCan, container], Temperature=[cool], Clean=[no]
trashCan1 → link(76L): [On] → floor1

fryingPan1: MadeOf=[metal], Colour=[steel], Open=[yes], Type=[container, fryingPan, pot], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
fryingPan1 → link(97L): [On, In] → cupboard1

sink1: MadeOf=[metal], Wet=[yes], Colour=[steel], Open=[yes], Type=[container,

sink], Temperature=[cool], Clean=[yes]
sink1 → link(5L): [ConsistsOf] → faucet1
sink1 → link(1L): [ConsistsOf] → drain1
sink1 → link(6L): [ConsistsOf, ControlledBy] → tap1
sink1 → link(7L): [NextTo] → bench1

milkBottle1: MadeOf=[plastic], Colour=[white], Open=[no], Type=[container, bottle], Graspable=[yes], Temperature=[cold], Movable=[yes]
milkBottle1 → link(23L): [In, On] → refrigerator1
milkBottle1 → link(32L): [Supports, Contains] → milk1
milkBottle1 → link(37L): [ConsistsOf, ControlledBy] → bottleCap1

cupboardHandle2: MadeOf=[plastic], Colour=[blue], Type=[handle], Graspable=[yes], Temperature=[cool], Clean=[yes]
cupboardHandle2 → link(54L): [PartOf, Controls] → cupboard2

cupboardHandle1: MadeOf=[plastic], Colour=[blue], Type=[handle], Graspable=[yes], Temperature=[cool], Clean=[yes]
cupboardHandle1 → link(67L): [PartOf, Controls] → cupboard1

door1: MadeOf=[wood], Colour=[brown], Open=[no], Vertical=[yes], Type=[door, exit], Temperature=[cool], Clean=[yes]
door1 → link(90L): [ConsistsOf] → keyhole1
door1 → link(91L): [ConsistsOf, ControlledBy] → doorHandle1
door1 → link(88L): [PartOf, LeadsTo] → kitchen1

teaBoxHandle1: MadeOf=[cardboard], Colour=[yellow], Type=[handle], Graspable=[yes], Clean=[yes], Temperature=[cool]
teaBoxHandle1 → link(33L): [PartOf, Controls] → teaBox1
teaBoxHandle1 → link(81L): [In] → cupboard2

plate2: MadeOf=[ceramics], Colour=[white], Type=[dish, plate], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
plate2 → link(50L): [In] → cupboard2
plate2 → link(114L): [On] → plate1

floor1: MadeOf=[wood], Colour=[brown], Type=[surface, floor], Temperature=[cool], Clean=[no]
floor1 → link(79L): [Supports] → trashCan1
floor1 → link(41L): [Supports] → teacher1
floor1 → link(43L): [PartOf] → kitchen1
floor1 → link(42L): [Supports] → hopper
floor1 → link(10L): [Supports] → refrigerator1
floor1 → link(40L): [Supports] → cupboard1
floor1 → link(141L): [Supports] → stool1
floor1 → link(39L): [Supports] → bench1
floor1 → link(38L): [Supports] → table1

dial1: MadeOf=[plastic], Colour=[black], Type=[dial], Graspable=[yes], Temperature=[cool], Clean=[yes]
dial1 → link(56L): [PartOf] → cooktop1
dial1 → link(55L): [Controls] → heatingElement1

teacher1: Vertical=[yes], Type=[robot, teacher], Temperature=[cool]
teacher1 → link(83L): [On] → floor1
teacher1 → link(82L): [ConsistsOf] → teacherArm

plate1: MadeOf=[ceramics], Colour=[white], Type=[dish, plate], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
plate1 → link(112L): [On, In] → cupboard2
plate1 → link(115L): [Supports] → plate2

doorHandle1: MadeOf=[wood], Colour=[brown], Type=[handle], Graspable=[yes], Temperature=[cool]
doorHandle1 → link(104L): [PartOf, Controls] → door1

cheese1: Colour=[yellow], Type=[food, cheese], Graspable=[yes], Temperature=[cold], Taste=[cheese], Movable=[yes]
cheese1 → link(64L): [On, In] → refrigerator1

button1: MadeOf=[plastic], Colour=[black], Type=[button], Temperature=[cool], Clean=[yes]
button1 → link(20L): [PartOf, Controls] → blender1

robotArm: Type=[arm]
robotArm → link(100L): [PartOf] → hopper

tin1: MadeOf=[metal], Colour=[white], Open=[yes], Type=[container, tin], Graspable=[yes], Temperature=[cool], Movable=[yes]
tin1 → link(35L): [On, In] → cupboard2
tin1 → link(123L): [Supports, Contains] → sugar1

tin2: MadeOf=[metal], Colour=[black], Open=[yes], Type=[container, tin], Graspable=[yes], Temperature=[cool], Movable=[yes]
tin2 → link(3L): [Supports, Contains] → flour1
tin2 → link(52L): [On, In] → cupboard2

drain1: MadeOf=[metal], Open=[yes], Type=[drain], Temperature=[cool]
drain1 → link(0L): [PartOf] → sink1

cupboard1: MadeOf=[wood], Colour=[white], Open=[no], Type=[cupboard, container], Temperature=[cool], Clean=[yes]
cupboard1 → link(68L): [ConsistsOf, ControlledBy] → cupboardHandle1

cupboard1 → link(72L): [Supports, Contains] → fryingPan1
cupboard1 → link(75L): [On] → floor1
cupboard1 → link(73L): [Supports, Contains] → pot1
cupboard1 → link(71L): [Supports, Contains] → pot2

cupboard2: MadeOf=[wood], Colour=[brown], Open=[no], Vertical=[yes], Type=[cupboard, container], Temperature=[cool], Clean=[yes]
cupboard2 → link(93L): [Contains] → teaBag1
cupboard2 → link(84L): [Contains] → teaBoxHandle1
cupboard2 → link(29L): [PartOf, On] → bench1
cupboard2 → link(28L): [Supports, Contains] → teaBox1
cupboard2 → link(107L): [Contains] → teaBag2
cupboard2 → link(53L): [Supports, Contains] → tin2
cupboard2 → link(95L): [Contains] → teaBag3
cupboard2 → link(46L): [Supports, Contains] → tin1
cupboard2 → link(111L): [Supports, Contains] → cup2
cupboard2 → link(65L): [ConsistsOf, ControlledBy] → cupboardHandle2
cupboard2 → link(51L): [Contains] → plate2
cupboard2 → link(49L): [Supports, Contains] → cup1
cupboard2 → link(113L): [Supports, Contains] → plate1

window1: MadeOf=[glass], Colour=[clear], Vertical=[yes], Type=[window], Temperature=[cool], Clean=[yes], Fragile=[yes]
window1 → link(103L): [PartOf] → wall1

towel1: MadeOf=[material], Colour=[red], Type=[towel], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
towel1 → link(128L): [On] → table1

bench1: MadeOf=[granite], Colour=[black], Type=[surface, bench], Temperature=[cool], Clean=[yes]

bench1 → link(18L): [NextTo] → sink1
bench1 → link(19L): [On] → floor1
bench1 → link(14L): [Supports] → kettle1
bench1 → link(117L): [ConsistsOf] → drawer1
bench1 → link(109L): [Supports] → blender1
bench1 → link(16L): [Supports, ConsistsOf] → cupboard2
bench1 → link(17L): [ConsistsOf] → cooktop1

blender1: MadeOf=[glass], Colour=[clear], Open=[yes], Type=[blender, container], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
blender1 → link(108L): [On] → bench1
blender1 → link(21L): [ConsistsOf, ControlledBy] → button1

drawerHandle1: MadeOf=[plastic], Colour=[blue], Type=[handle], Graspable=[yes], Clean=[yes], Temperature=[cool]
drawerHandle1 → link(118L): [PartOf, Controls] → drawer1

heatingElement1: MadeOf=[metal], TurnedOn=[no], Colour=[black], Type=[heatingElement], Temperature=[cool], Clean=[no]
heatingElement1 → link(57L): [PartOf] → cooktop1
heatingElement1 → link(58L): [ControlledBy] → dial1

fridgeHandle1: MadeOf=[plastic], Colour=[white], Type=[handle], Graspable=[yes], Temperature=[cool], Clean=[yes]
fridgeHandle1 → link(15L): [PartOf, Controls] → refrigerator1

teaBag4: MadeOf=[tea], Colour=[brown], Type=[teaBag], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
teaBag4 → link(89L): [On, In] → teaBox1

bottleCap1: MadeOf=[plastic], Colour=[blue], Type=[screwCap], Graspable=[yes],

Temperature=[cold], Movable=[yes]
bottleCap1 → link(152L): [In] → refrigerator1
bottleCap1 → link(36L): [PartOf, Controls] → milkBottle1

teaBag3: MadeOf=[tea], Colour=[brown], Type=[teaBag], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
teaBag3 → link(63L): [In, On] → teaBox1
teaBag3 → link(94L): [In] → cupboard2

teaBag2: MadeOf=[tea], Colour=[brown], Type=[teaBag], Graspable=[yes], Temperature=[cool], Clean=[yes], Movable=[yes]
teaBag2 → link(61L): [In, On] → teaBox1
teaBag2 → link(96L): [In] → cupboard2

teaBag1: MadeOf=[tea], Colour=[brown], Type=[teaBag], Graspable=[yes], Clean=[yes], Temperature=[cool], Movable=[yes]
teaBag1 → link(92L): [In] → cupboard2
teaBag1 → link(59L): [On, In] → teaBox1

keyhole1: Colour=[black], Vertical=[yes], Type=[keyhole], Temperature=[cool]
keyhole1 → link(105L): [PartOf] → door1

B.2 Example sequence of atomic actions for a lesson

The following sequence of 69 atomic actions is an example of a lesson generated by the teacher. Note that TADPOLE does *not* see the atomic actions themselves but only their result. In this case it would see a sequence of 70 states in the form shown above.

MoveArm[cupboardHandle2], Grasp, PullArm, Release, LiftArm, MoveArm[cup1], Grasp, LiftArm, MoveArm[table1], Release, LiftArm, MoveArm[teaBoxHandle1], Grasp, PullArm, Release, LiftArm, MoveArm[teaBag1], Grasp, LiftArm, MoveArm[cup1], Release, LiftArm, MoveArm[teaBoxHandle1], Grasp, PushArm, Release, LiftArm, MoveArm[cupboardHandle2], Grasp, PushArm, Release, LiftArm, MoveArm[kettle1], Grasp, LiftArm, MoveArm[sink1], Release, LiftArm, MoveArm[faucet1], Grasp, TwistArm, TwistArm, Release, LiftArm, MoveArm[kettle1], Grasp, LiftArm, MoveArm[bench1], Release, Lift Arm, MoveArm[button2], PushArm, PushArm, LiftArm, MoveArm[kettle1], Grasp, LiftArm, MoveArm[cup1], TwistArm, LiftArm, MoveArm[bench1], Release, LiftArm, MoveArm[teaBag1], Grasp, LiftArm, MoveArm[trashCan1], Release, LiftArm

B.3 Example decomposition rule for making a cup of tea

MakeCupOfTea DECOMPOSITION (4):
==========
Graph:
183N: **VAR217** {}[(Type, liquid), (Type, water), (Temperature, hot), (Taste, tea), (Colour, clear)]
183N → link(0L): {}[On, In] → cup1

cup1: **VAR218** {MadeOf={ceramics=3}, Colour={white=3}, Open={yes=3}, Type={cup=3, dish=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cup1 → link(13L): {}[Supports, Contains] → 183N

trashCan1: **VAR221** {MadeOf={plastic=3}, Colour={white=3}, Open={yes=3}, Type={trashCan=3, container=3}, Clean={no=3}, Temperature={cool=3}}

trashCan1 → link(6L): {}[Supports] → teaBag1

teaBag1: **VAR220** {MadeOf={tea=3}, Colour={brown=3}, Type={teaBag=3}, Clean={yes=3}, Temperature={cool=3}}
teaBag1 → link(9L): {}[On] → trashCan1
SUB-GOALS:

Sub-Goal A

==========

Graph:

cupboard2: **VAR215** {MadeOf={wood=3}, Colour={brown=3}, Open={yes=2, no=2}, Vertical={yes=3}, Type={cupboard=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cupboard2 → link(1L): {}[Supports] → teaBag1
cupboard2 → link(4L): {}[Supports] → cup1

cup1: **VAR218** {MadeOf={ceramics=3}, Colour={white=3}, Open={yes=3}, Type={cup=3, dish=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cup1 → link(6L): {}[On] → cupboard2
cup1 → link(7L): {}[Supports] → teaBag1

teaBag1: **VAR220** {MadeOf={tea=3}, Colour={brown=3}, Type={teaBag=3}, Clean={yes=3}, Temperature={cool=3}}
teaBag1 → link(2L): {}[On] → cup1
teaBag1 → link(10L): {}[On] → cupboard2

Sub-Goal B

==========

Graph:

table1: **VAR216** {MadeOf={wood=3}, Colour={brown=3}, Type={surface=3, table=3, furniture=3}, Clean={no=3}, Temperature={cool=3}}
table1 → link(11L): {}[Supports] → cup1

cupboard2: **VAR215** {MadeOf={wood=3}, Colour={brown=3}, Open={yes=2, no=2}, Vertical={yes=3}, Type={cupboard=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cupboard2 → link(12L): {Supports=1} → teaBag1
cupboard2 → link(4L): {}[Supports] → cup1

cup1: **VAR218** {MadeOf={ceramics=3}, Colour={white=3}, Open={yes=3}, Type={cup=3, dish=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cup1 → link(5L): {}[On] → table1
cup1 → link(6L): {}[On] → cupboard2

teaBag1: **VAR220** {}
teaBag1 → link(13L): {On=1} → cupboard2
Sub-Goal C
==========
Graph:
183N: **VAR217** {}[(Type, liquid), (Type, water), (Temperature, hot), (Colour, clear)]
183N → link(1L): {}[On, In] → kettle1

kettle1: **VAR219** {MadeOf={metal=3}, Colour={grey=3}, Open={yes=3}, Type={container=3, electricalKettle=3}, Running={no=3}, Clean={yes=3}, Temperature={cool=3}}
kettle1 → link(0L): {}[Supports, Contains] → 183N
Sub-Goal D
==========
Graph:
table1: **VAR216** {}
table1 → link(7L): {Supports=3} → cup1

183N: **VAR217** {Colour={clear=3}, Type={water=3, liquid=3}, Tem-

perature={hot=3}}[(Taste, tea)]
183N → link(2L): {}[On, In] → kettle1
183N → link(1L): {}[On, In] → cup1

cup1: **VAR218** {MadeOf={ceramics=3}, Colour={white=3}, Open={yes=3}, Type={cup=3, dish=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cup1 → link(3L): {}[Supports, Contains] → 183N
cup1 → link(6L): {On=3} → table1
cup1 → link(4L): {Supports=3} → teaBag1

kettle1: **VAR219** {MadeOf={metal=3}, Colour={grey=3}, Open={yes=3}, Type={container=3, electricalKettle=3}, Running={no=3}, Clean={yes=3}, Temperature={cool=3}}
kettle1 → link(0L): {}[Supports, Contains] → 183N

teaBag1: **VAR220** {}
teaBag1 → link(5L): {On=3} → cup1
Sub-Goal E
==========
Graph:
table1: **VAR216** {}
table1 → link(7L): {Supports=3} → cup1

183N: **VAR217** {}
183N → link(5L): {In=3, On=3} → cup1

cup1: **VAR218** {MadeOf={ceramics=3}, Colour={white=3}, Open={yes=3}, Type={cup=3, dish=3, container=3}, Clean={yes=3}, Temperature={cool=3}}
cup1 → link(1L): {}[Supports] → teaBag1
cup1 → link(6L): {On=3} → table1
cup1 → link(4L): {Supports=3, Contains=3} → 183N

trashCan1: **VAR221** {MadeOf={plastic=3}, Colour={white=3}, Open={yes=3}, Type={trashCan=3, container=3}, Clean={no=3}, Temperature={cool=3}}
trashCan1 → link(2L): {}[Supports] → teaBag1

teaBag1: **VAR220** {MadeOf={tea=3}, Colour={brown=3}, Type={teaBag=3}, Clean={yes=3}, Temperature={cool=3}}
teaBag1 → link(0L): {}[On] → cup1
teaBag1 → link(3L): {}[On] → trashCan1

Partial Order: [[A, B], [C], [D], [E]]
Sub-goal dependencies:
Sub-goal A → []
Sub-goal B → []
Sub-goal C → [A, B]
Sub-goal D → [C]
Sub-goal E → [A, B, D]

Variables:
VAR220 → (teaBag1,0) (teaBag1,2) (teaBag1,3) (teaBag1,0) (teaBag1,T)
VAR221 → (trashCan1,T) (trashCan1,3)
VAR219 → (kettle1,2) (kettle1,1)
VAR218 → (cup1,0) (cup1,0) (cup1,T) (cup1,2) (cup1,3)
VAR217 → (183N,1) (183N,2) (183N,3) (183N,T)
VAR216 → (table1,0) (table1,3) (table1,2)
VAR215 → (cupboard2,0) (cupboard2,0)

Bibliography

[1] AGRE, P. E., AND CHAPMAN, D. Pengi: An implementation of a theory of activity. In *Sixth National Conference on Artificial Intelligence* (1987).

[2] ANDERSON, J. R. Act a simple theory of complex cognition. *American Psychologist* (April 1996).

[3] ANDERSON, J. R. Human symbol manipulation within an integrated cognitive architecture. *Cognitive Science 29* (2005), 313–341.

[4] ANDERSON, J. R., ALBERT, M. V., AND FINCHAM, J. M. Tracing problem solving in real time: fmri analysis of the subject-paced tower of hanoi. *Journal of Cognitive Neuroscience 17*, 8 (2005), 1261–1274.

[5] ANDERSON, J. R., BOTHELL, D., BYRNE, M. D., DOUGLASS, S., LEBIERE, C., AND QIN, Y. An integrated theory of the mind. *Psychological Review 111*, 4 (2004), 1036–1060.

[6] ANDREAE, D. B. *Representing, matching, and generalising structural descriptions of complex physical objects.* PhD thesis, Victoria University of Wellington, 1994.

[7] AYAN, N. F., KUTER, U., YAMAN, F., AND GOLDMAN, R. P. Hotride: Hierarchical ordered task replanning in dynamic environments, September 2007. ICAPS 2007 Workshop on Planning and Execution for Real-World Systems.

[8] BENSON, S., AND NILSSON, N. Reacting, planning and learning in an autonomous agent. *Machine Intelligence*, 14 (1995).

[9] BOEDDINGHAUS, J., RAGNI, M., KNAUFF, M., AND NEBEL, B. Simulating spatial reasoning using act-r. In *Proceedings of the Seventh International Conference on Cognitive Modeling* (2006), pp. 62–67.

[10] BYRNE, M. D. Using computational cognitive modeling to diagnose possible sources of aviation error. In *Proceedings of the 25th Annual Meeting of the Cognitive Science Society* (2003), pp. 204–209.

[11] CARBONELL, J. G., AND GIL, Y. Learning by experimentation: The operator refinement method. In *MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH, VOLUME III* (1996), Morgan Kaufmann, pp. 191–213.

[12] CHOI, D., KAUFMAN, M., LANGLEY, P., NEJATI, N., AND SHAPIRO, D. An architecture for persistent reactive behavior. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems* (2004), pp. 988–995.

[13] DYE, H. A. Diagrammatic reasoning: Route planning on maps with act-r. In *Proceedings of the 8th International Conference on Cognitive Modeling* (2007).

[14] EROL, K., HENDLER, J., AND NAU, D. S. UMCP: A sound and complete procedure for hierarchical task-network planning. In *International Conference on AI Planning Systems (AIPS)* (June 1994), pp. 249–254.

[15] EROL, K., NAU, D. S., AND SUBRAHMANIAN, V. S. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence 76* (July 1995), 75–88.

[16] FIKES, R. E., AND NILSSON, N. J. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[17] FLACK-YTTER, T., GREDEBCK, G., AND VON HOFSTEN, C. Infants predict other people's action goals. *Nature Neuroscience 9* (July 2006).

[18] FORBUS, K. D. *Qualitative physics: past, present, and future*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[19] FU, W.-T., AND ANDERSON, J. R. From recurrent choice to skill learning: A reinforcement-learning model. *Journal of Experimental Psychology: General 135*, 2 (2006), 184–206.

[20] HAY, A. W. Automatically extending the coverage of hierarchical task network planners. Master's thesis, The University of Auckland, 2008.

[21] HEBBAR, K., SMITH, S. J. J., MINIS, I., AND NAU, D. S. Plan-based evaluation of designs for microwave modules. In *ASME Design Technical Conference* (August 1996).

[22] ICHISE, R., SHAPIRO, D., AND LANGLEY, P. Learning hierarchical skills from observation. In *Proceedings of the 5th International Conference on Discovery Science* (2002), pp. 247–258.

[23] ILGHAMI, O., MUOZ-AVILA, H., NAU, D. S., AND AHA, D. W. Learning approximate preconditions for methods in hierarchical plans. In *International Conference on Machine Learning (ICML)* (August 2005).

[24] ILGHAMI, O., NAU, D. S., MUOZ-AVILA, H., AND AHA, D. W. Learning preconditions for planning from plan traces and htn structure. *Computational Intelligence*, 21(4) (November 2005), 388–413.

[25] KUTER, U., AND NAU, D. Controlled search over compact state representations, in nondeterministic planning domains and beyond. In *National Conference on Artificial Intelligence (AAAI)* (2006).

[26] KUTER, U., NAU, D., PISTORE, M., AND TRAVERSO, P. A hierarchical task-network planner based on symbolic model checking. In *International Conference on Automated Planning and Scheduling (ICAPS)* (June 2005), pp. 300–309.

[27] KUTER, U., NAU, D., REISNER, E., AND GOLDMAN, R. Conditionalization: Adapting forward-chaining planners to partially observable environments, September 2007. ICAPS 07 Workshop on Planning and Execution for Real-World Systems.

[28] LAIRD, J. E. Extending the soar cognitive architecture. In *In Artificial General Intelligence Conference* (2008).

[29] LAIRD, J. E., NEWELL, A., AND ROSENBLOOM, P. S. Soar: an architecture for general intelligence. *Artificial Intelligence 33*, 1 (1987), 1–64.

[30] LAIRD, J. E., WRAY III, R. E., MARINIER III, R. P., AND LANGLEY, P. Claims and challenges in evaluating human-level intelligent systems. In *Proceedings of the Second Conference on Artificial General Intelligence* (2009).

[31] LANGLEY, P., CHOI, D., AND ROGERS, S. Interleaving learning, problem solving, and execution in the icarus architecture, 2005. Technical Report, Computational Learning Laboratory, CSLI, Stanford University.

[32] LANGLEY, P., CHOI, D., AND SHAPIRO, D. A cognitive architecture for physical agents, 2004. Technical Report, Computational Learning Laboratory, CSLI, Stanford University.

[33] LANGLEY, P., AND ROGERS, S. Cumulative learning of hierarchical skills. In *Proceedings of the Third International Conference on Development and Learning* (2004).

[34] LEBIERE, C., WALLACH, D., AND TAATGEN, N. A. Implicit and explicit learning in act-r, 1998.

[35] MITCHELL, T. *Version Spaces: An Approach to Concept Learning*. PhD thesis, Stanford University, 1978.

[36] MUOZ-AVILA, H., AHA, D. W., BRESLOW, L., AND NAU, D. Hicap: an interactive case-based planning architecture and its application to noncombatant evacuation operations. In *AAAI/IAAI* (1999), pp. 870–875.

[37] NAU, D., AU, T.-C., ILGHAMI, O., KUTER, U., MUOZ-AVILA, H., MURDOCK, J. W., WU, D., AND YAMAN, F. Applications of shop and shop2, June 2004. University of Maryland Technical Report.

[38] NAU, D., AU, T.-C., ILGHAMI, O., KUTER, U., MURDOCK, J. W., WU, D., AND YAMAN, F. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20 (December 2003), 379–404.

[39] NAU, D., CAO, Y., LOTEM, A., AND MUOZ-AVILA, H. Shop: Simple hierarchical ordered planner. In *International Joint Conference on Artificial Intelligence (IJCAI)* (1999), pp. 968–873.

[40] NAU, D. S., CAO, Y., LOTEM, A., AND MUOZ-AVILA, H. Shop and m-shop: Planning with ordered task decomposition, June 2000. University of Maryland Technical Report.

[41] NAU, D. S., MUOZ-AVILA, H., CAO, Y., LOTEM, A., AND MITCHELL, S. Total-order planning with partially ordered subtasks. In *International Joint Conference on Artificial Intelligence (IJCAI)* (August 2001).

[42] NEJATI, N., LANGLEY, P., AND KONIK, T. Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning* (2006), pp. 665–662.

[43] NILSSON, N. Toward agent programs with circuit semantics, 1992. Technical Report STAN-CS-92-1412, Stanford University Computer Science Department.

[44] REDDY, C., AND TADEPALLI, P. Learning goal-decomposition rules using exercises. In *IN PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING* (1997), Morgan Kaufmann, pp. 278–286.

[45] SACERDOTI, E. D. A structure for plans and behavior. Tech. Rep. 109, AI Center, SRI International, Aug 1975.

[46] SHAPIRO, D., LANGLEY, P., AND SHACHTER, R. Using background knowledge to speed reinforcement learning in physical agents. In *Proceedings of the fifth international conference on Autonomous agents* (2001), pp. 254–261.

[47] SHEN, W.-M. *Autonomous Learning from the Environment*. Computer Science Press an imprint of W.H. Freeman and Company, 1994.

[48] SMITH, S. J. J., NAU, D. S., AND THROOP, T. A. A planning approach to declarer play in contract bridge. *Computational Intelligence 12* (1996), 106–130.

[49] WANG, X. Planning while learning operators. In *In Proceedings of the Third International Conference on AI Planning Systems* (1996), AAAI Press, pp. 229–236.

[50] WEST, R. L. Using computational cognitive modeling to diagnose possible sources of aviation error. In *Proceedings of the Twenty-first Conference of the Cognitive Science Society* (1999), pp. 296–301.

[51] WILKINS, D. E. Domain-independent planning: representation and plan generation. *Artificial Intelligence 22* (1984), 269–301.

[52] WU, D., PARSIA, B., SIRIN, E., HENDLER, J., AND NAU, D. Automating daml-s web services composition using shop2. In *International Semantic Web Conference (ISWC)* (November 2003).

[53] YANG, Q., WU, K., AND JIANG, Y. Learning action models from plan examples using weighted max-sat. *Artif. Intell. 171* (February 2007), 107–143.