Title
Hierarchical Methods for Optimal Long-Term Planning

Author
Wolfe, Jason Andrew

Publication Date
2011

Peer reviewed|Thesis/dissertation

Hierarchical Methods for Optimal Long-Term Planning

by

Jason Andrew Wolfe

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Stuart Russell, Chair
Professor Christos Papadimitriou
Professor Rhonda Righter

Fall 2011

Hierarchical Methods for Optimal Long-Term Planning

Abstract

Hierarchical Methods for Optimal Long-Term Planning

by

Jason Andrew Wolfe

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Stuart Russell, Chair

This thesis addresses the problem of generating goal-directed plans involving very many elementary actions. For example, to achieve a real-world goal such as earning a Ph.D., an intelligent agent may carry out millions of actions at the level of reading a word or striking a key. Given computational constraints, it seems that such long-term planning must incorporate reasoning with high-level actions (such as delivering a conference talk or typing a paragraph of a research paper) that abstract over the precise details of their implementations, despite the fact that these details must eventually be determined for the actions to be executed. This multi-level decision-making process is the subject of hierarchical planning.

To most effectively plan with high-level actions, one would like to be able to correctly identify whether a high-level plan works, without first considering its low-level implementations. The first contribution of this thesis is an "angelic" semantics for high-level actions that enables such inferences. This semantics also provides bounds on the costs of high-level plans, enabling the identification of provably high-quality (or even optimal) high-level solutions.

Effective hierarchical planning also requires algorithms to efficiently search through the space of high-level plans for high-quality solutions. We demonstrate how angelic bounds can be used to speed up search, and introduce a novel decomposed planning framework that leverages task-specific state abstraction to eliminate many redundant computations. These techniques are instantiated in the Decomposed, Angelic, State-abstracted, Hierarchical A* (DASH-A*) algorithm, which can find hierarchically optimal solutions exponentially faster than previous algorithms.

Contents

Acknowledgments

This dissertation would not have been possible without the help and support of advisors, colleagues, friends, and family.

First and foremost, I would like to thank my advisor Stuart Russell. His wisdom, encyclopedic knowledge of the field, and ample guidance have shaped many of the ideas in this dissertation, and helped me become a better writer, researcher, and thinker.

I would also like to thank Bhaskara Marthi, who originated this research project, contributed to and helped refine many of the ideas presented here, and has taught me a lot about research over the years.

I am very grateful to Professors Christos Papadimitriou and Rhonda Righter for serving on my dissertation committee, and offering helpful comments on this manuscript. Along with Professor Dan Klein, they also helped guide this research in its earlier stages by serving on my qualifying exam committee.

This space is unfortunately too short to thank all those who have shaped my education and research, including a great number of brilliant faculty and students throughout undergrad and grad school at Berkeley. Conversations with Malik Ghallab, Pieter Abbeel, and many of the great researchers at Willow Garage stand out especially. Working on side-projects with Aria Haghighi and Dan Klein was also a great pleasure.

My fellow RUGS members have been a never-ending source of great conversations, inspiration, and helpful suggestions on countless research ideas, papers, and talks over the years: Norm Aleks, Nimar Arora, Emma Brunskill, Kevin Canini, Shaunak Chatterjee, Daniel Duckworth, Nick Hay, Gregory Lawrence, Bhaskara Marthi, Brian Milch, David Moore, Rodrigo de Salvo Braz, Erik Sudderth, and Satish Kumar Thittamaranahalli.

I am also grateful for all of the reviewers, researchers, and others I've had the pleasure to interact with throughout graduate school. In particular, the ICAPS community has inspired and welcomed papers based on many of the ideas in this thesis.

Last and far from least, I would like to thank my friends and family. My parents, Rich and Sue, have always offered unconditional love and support for my endeavors, whatever they may be. I can't thank you enough! My girlfriend, Sarah Reed, has been a pillar of support and a perfect partner in crime over the past three years — I can't wait for you to join me as Dr. Reed! To all of my family and friends: I can't tell you how much your love, support, and understanding mean to me — you've helped make my time in graduate school the best of my life thus far.

Chapter 1

Introduction

A central aspect of intelligence is the ability to *plan* ahead, selecting actions to do based on their expected impact on the world, including their influence on potential future action choices. In fact, nearly all actions that humans actually carry out on a day-to-day basis — striking a key, articulating a word, and so on — are only useful insofar as they take us tiny steps closer to achieving our long-term goals. Since the early days of Artificial Intelligence, hierarchical structure in behavior has been recognized as perhaps the most important tool for coping with the complex environments and long time horizons encountered in such real-world decision-making. Humans, who execute on the order of one trillion primitive motor commands in a lifetime, appear to make heavy use of hierarchy in their decision making— we entertain and commit to (or discard) high-level actions such as "write a Ph.D. thesis" and "run for President" without preplanning the motor commands involved, even though these high-level actions must eventually be refined down to motor commands (through many intermediate levels of hierarchy) in order to take effect.

Consider a simulated robot tasked with cleaning up a room by delivering a set of objects (e.g., dirty cups, magazines) to target regions (e.g., dishwasher, coffee table), an example that will be used throughout this thesis. To accomplish this task, the robot must execute a coordinated sequence of many thousands of low-level "*primitive*" actions, corresponding to small movements of its base and arm, articulations of its gripper, and so on. Attempting to find such a *solution* by directly searching over the space of all possible such sequences seems like a hopeless, or at least misguided, pursuit. A more sensible approach might be to attack the problem *top-down*, first deliberating about possible orderings for the high-level tasks, and then *refining* promising candidate plans, progressively filling in details such as base placements, grasp positions, and ultimately low-level trajectories, until a primitive solution is discovered.

This reasoning process is captured by hierarchical planning, which encodes each abstract task (such as "put away magazine 1" or "navigate to location (5,5)") as a *high-level action*

(HLA), defined by a set of possible implementations in terms of other actions. The principal computational benefit of using high-level actions seems obvious: high-level plans are much shorter, and thus good high-level plans ought to be much easier to find. However, this benefit can be fully realized only if high-level solutions can be identified—that is, only if we can establish that a given high-level plan can be refined to a primitive plan that achieves the goal, *before* explicitly considering its implementations. Moreover, if we desire an *efficient* solution — e.g., one that minimizes the total time taken to clean the room — we also require a method to assess the *costs* of high-level plans. For example, if we could conclude that "put away the magazine, then put away the cup" is a high-level solution, and moreover, is guaranteed to be cheaper than "put away the cup, then put away the magazine" (perhaps the cup is currently on the coffee table), we can immediately discard the latter plan and focus all of our efforts on refining the former plan into a high-quality primitive solution.

The primary challenges in implementing such an approach are twofold.

First, for this approach to work, the planner needs to know what high-level actions *do*. For example, to conclude that "put away the magazine, then the cup" is a high-level solution, the planner must know that "put away the magazine" can lead to a state where the magazine is on the coffee table, and moreover, "put away the cup" can be executed from that state to accomplish its stated goal. While many previous hierarchical planners have used precondition-effect annotations for HLAs, no existing proposals have correctly captured the full transition dynamics of HLAs, and thus supported the identification of high-level solutions. The first aim of this thesis is to rectify this situation and thereby realize the full benefits of hierarchical structure in planning.

Second, given a hierarchy (and perhaps transition models for its HLAs), we require efficient algorithms for searching through the space of hierarchical plans for high-quality solutions. While previous research has proposed a wide variety of hierarchical planning algorithms, the focus has typically been on the precise forms of hierarchy representation and refinement operations. Given these choices, a simple backtracking depth-first search has typically been used to find *any* primitive solution compatible with the hierarchy (perhaps with a greedy bias towards lower costs). Comparatively little attention has been paid to the problem of finding high-quality or *optimal* hierarchical plans. Filling this gap is the second primary objective of this thesis.

The ultimate goal of this line of work is to enable autonomous agents that exhibit robust, high-quality behaviors in real-world environments. In the general case, this requires reasoning about uncertain knowledge, probabilistic uncertainty, the actions of other agents, and so on. This thesis takes one step in this direction, tackling the simplest possible sequential decision problems, while leaving more expressive features such as uncertainty for future work. The remainder of this chapter provides a high-level overview of these classical planning problems, hierarchies, and the motivations behind and contributions of the dissertation.

1.1 Classical Planning

The techniques discussed in this thesis are (thus far) geared towards *classical planning* problems, which are sequential, discrete, deterministic, and fully observable. Section 2.1 describes these problems in detail; in short, their objective is to find a (shortest) path in a graph with *states* for nodes and *actions* for edges, from a particular initial state to a goal state. For example, suppose that the state of our example environment is captured by a set of discrete variables corresponding to grid points for the objects, robot base, gripper, and so on. If, in addition, our robot knows the precise initial state of the environment and can exactly predict the outcomes of its actions, then its cleaning task is a classical planning problem (see Figure 1.1).

Since planning problems are graph search problems, they can be solved optimally with generic procedures such as Dijkstra's algorithm. The main feature distinguishing planning from ordinary graph search is *representation*: a planning problem *implicitly* specifies an graph that has one state for each allowable combination of state variable values, and is thus *exponentially* large. This is accomplished by specifying the transition functions of actions compactly, in terms of *preconditions* on state variables that must hold before their execution, and *effects* that generate new values for a subset of state variables after execution. For instance, the BaseR$(1, 2)$ action moves our robot's base right one square to $(2, 2)$, and can be executed when the base is mobile at $(1, 2)$ and $(2, 2)$ is free of obstacles. (The action must be parameterized by the current position, because its preconditions and effects are determined by the action type and parameters but cannot depend on the current state.) The exponential size of the state space renders simple algorithms such as Dijkstra's intractable for all but the smallest problems, while opening doors to more complex algorithms that can exploit the structure of planning problems to more quickly find (optimal) solutions.

1.2 Hierarchical Planning

Hierarchical decision-making has been studied since the early days of AI: in 1962, Herbert Simon wrote "Hierarchy ... is one of the central schemes that the architect of complexity uses." Planning is no exception, and a wide variety of approaches to hierarchical planning have been proposed over the past four decades. Section 2.3 describes these approaches and their key properties in some detail; for now, we briefly summarize hierarchical task network (HTN) planning, the approach broadly taken in this thesis.

In hierarchical task network planning, in addition to the "primitive" classical planning domain, we are given a hierarchy consisting of high-level actions (HLAs). Each HLA is defined by a set of allowed *refinements* into other actions, high-level or primitive. In our example

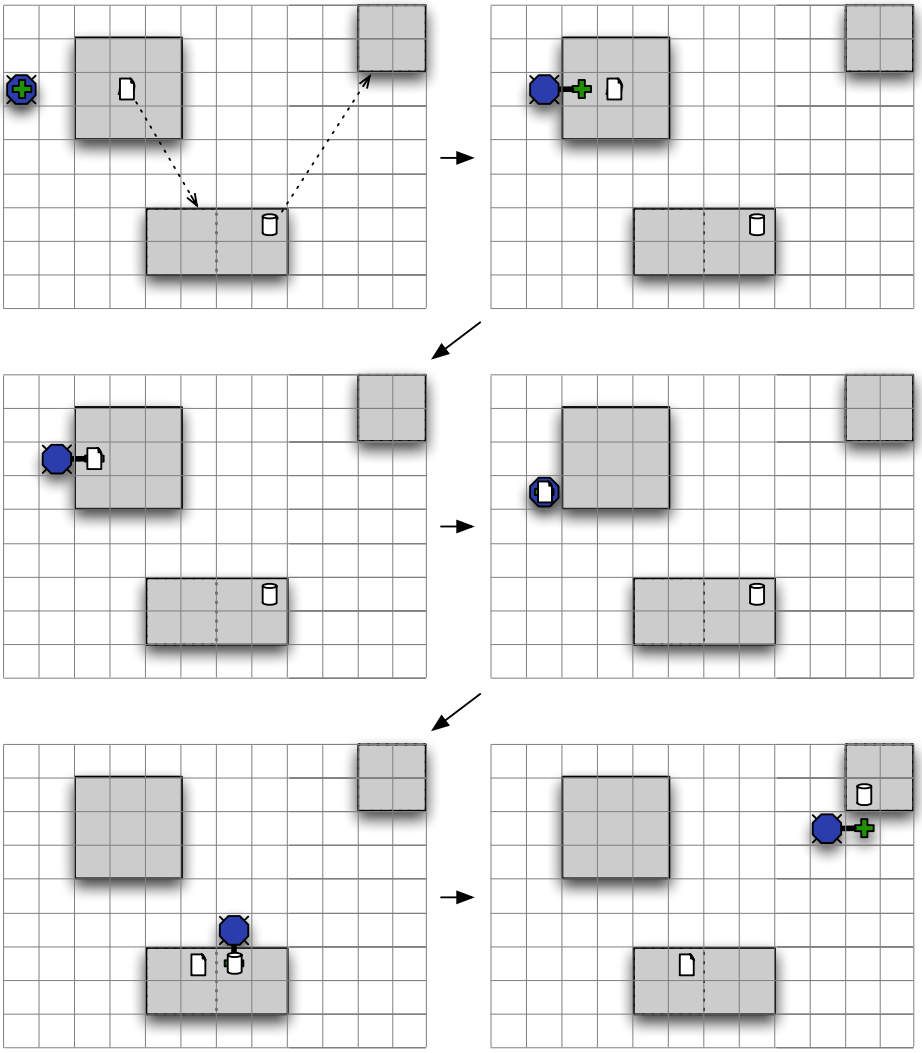

Figure 1.1: A "discrete manipulation" robotic planning problem, and a subset of states encountered along an optimal solution. A mobile robot base (blue octagon) can move to white squares of the grid, and its gripper (green cross) can extend to squares adjacent to the base that are not occupied by objects. The gripper can hold one object at a time, and can pick up and put down objects that are adjacent to it. In the initial state (top left), a magazine and cup must be moved from their initial positions to their goal regions (shown with dotted lines). To deliver the objects, the robot first unparks, moves one square to the right, parks, and extends the gripper to the right (top right). Then, it picks up the magazine (center left), unparks, and starts navigating to the lower table (center right). Upon arrival, it parks, extends the gripper down, places the magazine to the left and picks up the cup (bottom left). Finally, it navigates to the top right and puts down the cup, completing the optimal solution (bottom right).

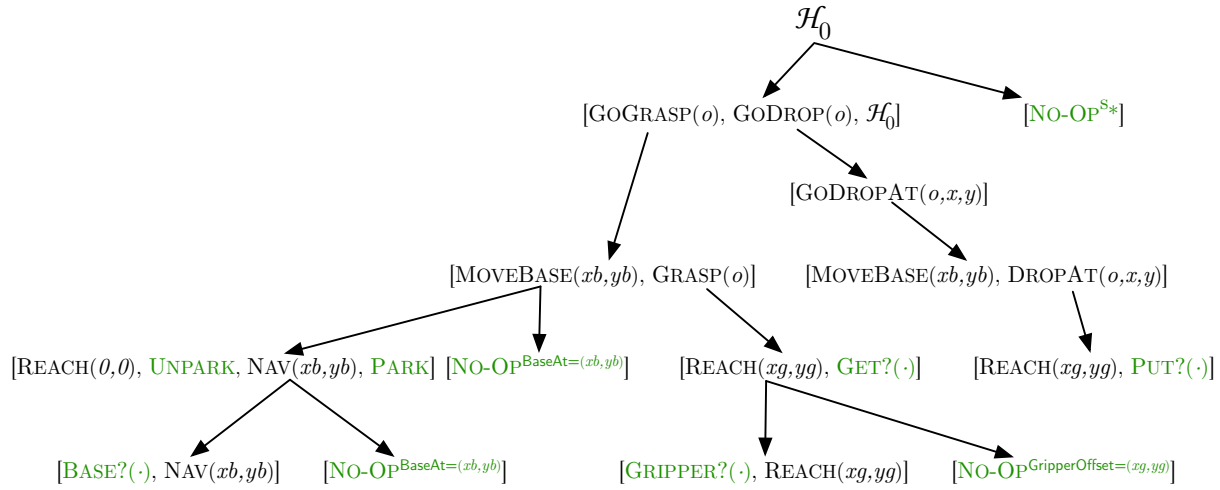

Figure 1.2: A schematic depiction of a hierarchy for the example domain. Primitive actions are in green, and HLAs are in black.

domain, we might consider an HLA GoDrop(o) that attempts to drop object o at a goal location, with refinements [GoDropAt(o, x, y)] ranging over potential goal positions (x, y) for o. Refinements can also be *recursive*; for instance, we might also include an HLA Nav(x, y) that attempts to navigate the base to (x, y), with refinements [BaseL/R/U/D($\cdot$), Nav(x, y)], or the empty sequence if the robot is already at (x, y). Figure 1.2 sketches a complete hierarchy for the example domain, including these and other HLAs. Given a hierarchy, planning begins with a plan consisting of a single "top-level" HLA called $\mathcal{H}_0$, and proceeds by repeatedly expanding HLAs out into their refinements until a fully primitive solution is generated.

In addition to ruling out primitive plans that do not conform to the hierarchy, the hope is that solutions can be separated from non-solutions at a high level (where plans are short), thus quickly pruning large fractions of the search space. In the ideal case, this could reduce the time complexity of planning from exponential (in the length of the final solution) to linear (Korf, 1987).

Unfortunately, previous attempts to identify high-level solutions have not been fully successful. The basic approach taken has been to assign precondition and effect annotations to the high-level actions, in essentially the same form as supplied for the primitive actions. This has a certain simplistic appeal; unfortunately, however, in most cases it is not possible to correctly capture all of the preconditions and effects of an HLA in this form (unless, e.g., the HLA admits only one refinement into a single primitive action sequence). For example, consider a simple HLA h with refinements $[a]$ and $[b]$, where a is a primitive action with precondition $u \wedge v$ and effect $e \wedge f$, and b is a primitive action with precondition $u \wedge w$ and effect $e \wedge g$ (see Figure 1.3). Then, u and e are clearly a precondition and effect of h

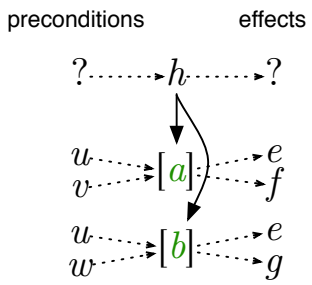

Figure 1.3: An example HLA h with two immediate refinements, $[a]$ and $[b]$. The preconditions and effects of a and b are given, but the corresponding annotations for h are unclear.

(respectively), but what of the remaining variables? It seems that the best we can do is to omit them from our description of h entirely, but this omission can deceive us in several ways. First, suppose that f is a goal of our problem; then, our description would seem to entail that h is not a solution (when in fact, we could always choose refinement $[a]$). Second, suppose that $e \wedge \neg f \wedge \neg g$ is our goal; then, we have the opposite problem: our description suggests that h is a solution, when in fact each of a and b causes one of the goal conditions to be violated.

A primary contribution of this dissertation is a solution to this problem, using a more expressive *angelic* language for HLA descriptions that can include *possible* preconditions and effects. Other contributions include a set of techniques for exploiting hierarchical structure in optimal planning, and several new families of hierarchically optimal search algorithms that exploit these techniques (along with angelic descriptions, in some cases) for faster planning. The main limitation of the techniques and algorithms described in this thesis compared to previous work in hierarchical planning is that we assume that the refinements of each HLA are *fully ordered* sequences of actions, and do not allow actions generated by different HLAs to be interleaved in a final solution. This assumption simplifies the analysis of HLA descriptions and other algorithmic techniques, while still allowing for natural hierarchies in many domains (such as our running example). However, as we discuss later, it can be somewhat restrictive in other domains, and we hope that future work will relax this restriction.

1.3 Outline of the Thesis

The dissertation begins with a detailed review of background material and previous work upon which our contributions are built. Chapter 2 is divided into three sections; the first and last formally introduce classical planning problems and hierarchies, in considerably more detail than the brief summaries above. The remaining section describes AND/OR graph search problems, generalizations of classical planning problems that most commonly arise in non-

deterministic or multi-agent planning settings, and reviews the broad classes of algorithms used for solving them. While the methods proposed in this thesis cannot yet be applied to *solve* AND/OR graph search problems, they will *use* techniques and algorithms developed for AND/OR graph search to more efficiently solve classical planning problems. Minor contributions of this section include a novel formalization of AND/OR search problems that can succinctly express both *top-down* and *bottom-up* search algorithms, which have primarily been considered separately in the literature, as well as a novel *hybrid* algorithm that combines some of the best features of both approaches.

Next, Chapters 3, 4, and 5 describe the primary contributions of the thesis.

Chapter 3 begins by proposing a simple method to *decompose* search for refinements of a high-level plan into separate subproblems, one for each action in the plan. For example, every optimal solution to put away a magazine via [GoGrasp(m), GoDrop(m)] consists of an optimal solution to grasp the magazine while ending in a particular intermediate state, followed by an optimal solution to put it away from this state. (Note that the reverse implication does not hold, because some intermediate states will result in a more efficient overall solution than others.) The advantage of decomposition is that subproblems may be repeated across the search space, and solution information from one instance can be *cached* and reused at later instances of the same subproblem. Furthermore, we show how to dramatically increase this caching by taking advantage of *state abstraction* in the form of HLA-specific relevance information. For example, the optimal solutions to GoGrasp(m) do not depend on the positions of objects other than the magazine (and any nearby objects), and can thus be shared between GoGrasp(m) subproblems from different world states that agree on all relevant variable values (but with, e.g., different cup positions). The chapter concludes with a sequence of increasingly efficient planning algorithms that can exploit these techniques to find hierarchically optimal solutions *exponentially* faster than previous brute-force algorithms.

Chapter 4 takes a step back from planning algorithms to consider the question of HLA descriptions introduced above. We introduce a novel *angelic semantics* for high-level actions, which provides descriptions of the effects of HLAs that are *true*—that is, they follow logically from the refinement hierarchy and the descriptions of the primitive actions. If achievement of the goal is entailed by the true descriptions of a sequence of HLAs, then that sequence must, by definition, be reducible to a primitive solution. Conversely, if the sequence provably fails to achieve the goal, it is not reducible to a primitive solution. When extended to include action costs, this semantics also allows for the identification of *provably optimal* high-level plans. The key insight behind the angelic approach is that *the planning agent itself* (not an adversary) will ultimately choose which refinement to make. Thus, instead of considering only preconditions and effects of an HLA h that hold under *all* refinements of h, we propose descriptions that generate a *set* of all states reachable by *any* refinement of h.

Unfortunately, such *exact* descriptions will typically be too large and computationally

expensive to be of practical utility. Thus, we propose principled *optimistic* and *pessimistic* approximations to the exact descriptions, which compactly describe *bounds* on the reachable sets and costs of high-level plans. For example, an optimistic description of h in our above example might have precondition u and effect $e \wedge \tilde{+}f \wedge \tilde{+}g$, meaning that h is definitely not applicable when u is false, and after executing h, e will definitely be made true, and f and g may *possibly* be made true as well. A pessimistic description could simply pick a refinement, stating that when $u \wedge v$ is true, h can definitely make $e \wedge f$ true (and possibly reach other states as well). We prove that optimistic and pessimistic descriptions of individual actions can be chained together to provide correct bounds on the outcomes of high-level sequences, and provide concrete representations and algorithms for computing such bounds efficiently. The chapter concludes by discussing potential approaches for computing angelic descriptions, and the relationships between angelic descriptions and descriptions of ordinary "primitive" actions (which are often intended to correctly capture even lower-level real-world processes).

Chapter 5 describes hierarchical planning algorithms that can take advantage of angelic descriptions and cost bounds. Our first algorithm, Angelic Hierarchical A* (AH-A*), applies the well-known A* algorithm to search through the space of high-level sequences generated by a hierarchy, using optimistic bounds as *admissible heuristics* that guarantee optimality of a discovered solution. AH-A* also includes a novel application of pessimistic descriptions for *pruning*, based on domination relationships among prefixes of generated plans. The remainder of the chapter builds upon this algorithm, incorporating the decomposition methods of Chapter 3 along with other, novel improvements, culminating in the DASH-A* (Decomposed, Angelic, State-abstracted, Hierarchical A*) algorithm. DASH-A* is a hierarchically optimal planning algorithm built upon a novel AND/OR graph search space, which can simultaneously reason about high-level actions and sequences at varying levels of action and state abstraction. Cost and reachability bounds provided by angelic HLA descriptions are used to quickly eliminate large swaths of provably suboptimal plans from the search space, leading to further exponential speedups over AH-A* or decomposed planning alone. Finally, we conclude with a brief discussion of *bounded suboptimal* variants of these algorithms, which can trade off solution quality for computation time in a principled manner, and may be a better fit for many practical applications.

Chapter 6 describes implementations of the algorithms introduced in the previous chapters, and compares their empirical performance with previous algorithms for (hierarchically) optimal planning, on a selection of discrete planning domains.

Chapter 7 discusses relationships to other planning research not covered in Chapter 2, as well as connections to work in other areas of AI and farther afield.

Finally, Chapter 8 concludes with a brief summary of the major contributions of the thesis, and a discussion of interesting directions for future work.

Some of the material in this thesis has been published in a series of conference papers and technical reports. The angelic approach was developed by Bhaskara Marthi, Stuart

Russell and myself, initially described at the International Conference on Automated Planning and Scheduling (ICAPS) in 2007 (Marthi et al., 2007a,b), and extended to include cost information the following year (Marthi et al., 2008, 2009). The first algorithm of Chapter 3 was presented at ICAPS in 2010, along with an implementation of our example mobile manipulation domain on a physical robot (Wolfe et al., 2010).

Chapter 2

Background

This thesis presents several new families of search algorithms that use hierarchies of high-level actions to *efficiently* find *high-quality* solutions to the simplest sequential decision problems. Section 2.1 describes these *classical planning* problems in detail, including relevant background material on problem representation, practical implementation details, and search strategies. Then, Section 2.2 discusses *AND/OR* search problems, which generalize classical planning problems by adding, e.g., nondeterminism. While the algorithms presented in this thesis can *not* (as yet) solve this more general class of problems, they will *use* the machinery presented to solve classical planning problems more effectively. Finally, Section 2.3 reviews historical approaches to hierarchical planning, and presents the definition of hierarchy used throughout the thesis.

The formalisms presented in this chapter sometimes deviate from their typical presentations. In particular, we attempt to simplify each formalism as much as possible, by presenting compilations that remove common additional features without compromising expressiveness. The purpose of these deviations is to reduce descriptive complexity, and thus present more comprehensible and easily analyzable results and search algorithms.

2.1 Classical Planning and Search

2.1.1 State-Space Search Problems

This section presents a standard representation-independent formalism for state-space search problems, which describes shortest-path problems in weighted, directed (multi-)graphs.

Definition 2.1. *A state-space search problem is given by a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0, s_*)$*:*

- $\mathcal{S}$ *is the* state space.

- $\mathcal{A}$ *is the* action space*, where* $\mathcal{A}_s \subseteq \mathcal{A}$ *is the subset of actions applicable in state* s*.*
- $\mathcal{T}(s,a)$ *is the* transition function*, specifying the state* $\mathcal{T}(s,a)$ *reached after doing action* $a \in \mathcal{A}_s$ *in state* $s \in \mathcal{S}$*.*
- $\mathcal{C}(s,a)$ *is the accompanying* cost function*, specifying the finite cost* $\mathcal{C}(s,a) \in [0,\infty)$ *of doing action* a *from state* s*.*
- s_0 *is the* initial state
- s_* *is the* goal state

In graph terminology, $\mathcal{S}$ is the set of nodes, each $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$ generates an edge from s to $\mathcal{T}(s,a)$ with cost $\mathcal{C}(s,a)$, and the task is to find a path from s_0 to s_* (where the sum of edge costs is minimized). The *successors* of a state s are those states reachable from s by applying a single action.

Remark. *It is more common to assume a* set *of goal states* $\mathcal{G}$*. We can always transform a problem with multiple goal states into our formalism by adding a new goal state* s_**, and a zero-cost action that leads from states in* $\mathcal{G}$ *to* s_**.*

Assumption 2.1. *We assume that* $\mathcal{S}$ *and* $\mathcal{A}$ *are finite. We also assume without loss of generality that* $\mathcal{C}(s,a) = 0$ *or* $\mathcal{C}(s,a) \geq 1$*, which can always be ensured by multiplying all costs by the reciprocal of the smallest non-zero action cost.*

To simplify notation, we extend $\mathcal{T}$ and $\mathcal{C}$ to the full domain of $\mathcal{A}$ by assuming the existence of a dummy sink state $s_\perp \neq s_*$ with $\mathcal{A}_{s_\perp} = \emptyset$. Then, $\mathcal{T}(s,a) = s_\perp$ and $\mathcal{C}(s,a) = \infty$ if $a \notin \mathcal{A}_s$ We also overload the transition and cost functions to work on sequences of actions in the obvious way, so if action sequence $\mathbf{a} = [a_1, ..., a_n]$, then we say that $\mathbf{a}$ has length n and define

$$\begin{aligned} \mathcal{T}(s,\mathbf{a}) &:= \mathcal{T}(\mathcal{T}(\mathcal{T}(s,a_1),...),a_n) \\ \mathcal{C}(s,\mathbf{a}) &:= \mathcal{C}(s,a_1) + \mathcal{C}(\mathcal{T}(s,a_1),a_2) + ... + \mathcal{C}(\mathcal{T}(s,\mathbf{a}_{1:n-1}),a_n), \end{aligned}$$

where $\mathbf{a}_{1:n-1}$ refers to the subsequence of $\mathbf{a}$ including actions with indices from 1 to $n-1$ (i.e., all but the last action). Under these simplifications, an action sequence $\mathbf{a}$ is applicable from state s iff $\mathcal{T}(s,\mathbf{a}) \neq s_\perp$, or, equivalently, iff $\mathcal{C}(s,\mathbf{a}) < \infty$.

Definition 2.2. *A* solution $\mathbf{a}$ *is a finite sequence of actions* $\mathbf{a} \in \mathcal{A}^*$ *where* $\mathcal{T}(s_0,\mathbf{a}) = s_*$*.*

Definition 2.3. *The* optimal cost $c^*(s)$ *to reach the goal* s_* *from a state* $s \in \mathcal{S}$ *is*

$$c^*(s) := \min_{\mathbf{a} \in \mathcal{A}^* \mid \mathcal{T}(s,\mathbf{a}) = s_*} \mathcal{C}(s,\mathbf{a}).$$

Definition 2.4. *An* optimal solution $\mathbf{a}^*$ *is a solution with minimal cost:*

$$\mathcal{T}(s_0, \mathbf{a}^*) = s_* \text{ and } \mathcal{C}(s_0, \mathbf{a}^*) = c^*(s_0).$$

Assumption 2.2. *To ensure that all optimal solutions have finite length, we assume (for now) that the search space is free of zero-cost cycles:*

$$\neg \left((\exists s \in \mathcal{S}, \mathbf{a} \in \mathcal{A}^+) \ \mathcal{T}(s, \mathbf{a}) = s \text{ and } \mathcal{C}(s, \mathbf{a}) = 0 \right).$$

With this assumption in place, we have the following results, which will be useful in proving the termination of search algorithms.

Lemma 2.3. *Given any cost $c \in [0, \infty)$, there are a finite number of action sequences $\mathbf{a} \in \mathcal{A}^*$ with cost $\mathcal{C}(s_0, \mathbf{a}) \leq c$.*

Proof. Since the state and action spaces are finite, there are finitely many plans with length at most $(c+1)|S|$. By the pigeonhole principle, every plan $\mathbf{a}$ with $|\mathbf{a}| > (c+1)|S|$ must visit some state at least $c+1$ times. By Assumption 2.1 and Assumption 2.2, $\mathcal{C}(s_0, \mathbf{a}) > c$. □

Theorem 2.4. *Every search problem either has no solutions, or admits a finite-length optimal solution.*

Proof. Suppose that there exists a solution with finite cost c. By Lemma 2.3, there exist finitely many plans with cost $\leq c$, each of finite length, and among these at least one must have minimal cost and thus be an optimal solution. □

Now, a *search algorithm* takes a search problem (and perhaps additional information) as input, and either outputs a solution, outputs failure (indicating that no solution exists), or fails to terminate.

Definition 2.5. *A search algorithm is* sound *if, when it returns a plan, that plan is always a solution.*

Definition 2.6. *A search algorithm is* complete *if, when a solution exists, it always eventually returns one. If no solution exists, it may return failure or fail to terminate.*

Definition 2.7. *A search algorithm is* optimal *if, when a solution exists, it always eventually returns an optimal solution. If no solution exists, it may return failure or fail to terminate.*

Of course, there is typically a trade-off between search time and solution quality, with optimality lying at one extreme. Other common points on this curve include *satisficing* algorithms, which attempt to find any solution (perhaps within a given fixed cost bound) as quickly as possible, and *bounded suboptimal* algorithms, which attempt to find a solution

whose cost is at most a given multiple of the optimal cost. This thesis will primarily be concerned with optimal search algorithms.[1] The reason is that while other criteria may sometimes be more useful in practice, understanding optimal search in a given setting is typically a prerequisite for developing algorithms that can make a principled tradeoff between solution quality and execution time.

2.1.2 Planning Problems: Representations and Examples

Classical planning studies very large search problems, in which the problem specification *implicitly* represents an exponentially larger state-space graph. Typically, the states are specified by assignments to a set of discrete variables. While most of the algorithms in this thesis are representation-independent (depending only on black-box functions such as $\mathcal{T}$), we present two common representations — STRIPS and SAS$^+$ — to support examples and later discussions of concrete implementations. Determining (bounded) plan existence in both representations is PSPACE-complete (Bylander, 1994; Jonsson and Bäckström, 1998). While more expressive formalisms such as ADL (Pednault, 1989) and PDDL (Ghallab et al., 1998) exist, many of their additional features can be efficiently compiled back into STRIPS (e.g., (Nebel, 2000)), and STRIPS and SAS$^+$ suffice for the problems considered here.

This section also describes our discrete robotic manipulation domain, and its representation in SAS$^+$, which will serve as a running example throughout the thesis.

2.1.2.1 STRIPS

STRIPS is a simple, widely used representation language for planning problems (Fikes and Nilsson, 1971). A STRIPS planning problem is presented in two parts: a *domain* that specifies a set of *predicates* and *action schemata* that capture the basic physics for a class of problems, and a *problem instance* that specifies a fixed set of objects, the set of propositions true in the initial state, and a set of goal propositions.

We present STRIPS by example, using a simple "nav-switch" domain (Marthi et al., 2008). In this domain, a single agent can navigate on a grid, moving one square up, down, left, or right at each step. Figure 2.1 shows a simple 3×3 example. The domain is made slightly more interesting by the addition of a single global "switch" that can face horizontally or vertically; move actions cost 2 if they go in the current direction of the switch and 4 otherwise. The switch can be flipped from horizontal to vertical and back with cost 1, but only from a subset of designated squares (e.g., $(2,1)$, $(3,2)$, and $(2,3)$ in Figure 2.1). The goal is always to reach a particular square (e.g., $(3,3)$) with minimum cost. For example, in

[1] More precisely, our focus is on hierarchically optimal search algorithms, to be defined in Section 2.3.

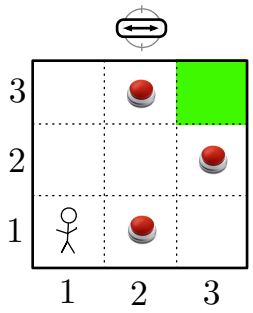

Figure 2.1: An instance of the nav-switch STRIPS planning domain.

the problem instance depicted in Figure 2.1, the only optimal solution is to go right, flip, up two, flip, and right again, incurring a total cost of 10.

This nav-switch domain can be encoded in (typed) STRIPS as follows.

- The problem state is captured by six predicates:
  - $\mathsf{AtX}(x)$ and $\mathsf{AtY}(y)$, where $x \in \mathsf{Xs}$, $y \in \mathsf{Ys}$, representing the position of the agent;
  - $\mathsf{LeftOf}(x_l, x_r)$ where $x_l, x_r \in \mathsf{Xs}$, representing that x_l is left of x_r;
  - $\mathsf{Below}(y_d, y_u)$ where $y_d, y_u \in \mathsf{Ys}$, representing that y_d is below y_u;
  - H, representing the orientation of the switch (true for horizontal); and
  - $\mathsf{CanFlip}(x, y)$ where $x \in \mathsf{Xs}$, $y \in \mathsf{Ys}$, representing that the switch can be flipped from position (x, y).
- The action schemata are $\textsc{UpH}(y_{from}, y_{to})$, $\textsc{DownH}(y_{from}, y_{to})$, $\textsc{LeftH}(x_{from}, x_{to})$, $\textsc{RightH}(x_{from}, x_{to})$, $\textsc{UpV}(y_{from}, y_{to})$, $\textsc{DownV}(y_{from}, y_{to})$, $\textsc{LeftV}(x_{from}, x_{to})$, and $\textsc{RightV}(x_{from}, x_{to})$, which move the agent one square in a direction when the switch is in the specified orientation, and $\textsc{FlipH}(x, y)$ and $\textsc{FlipV}(x, y)$, which flip the switch to the specified orientation. Each action schema consists of a *precondition* that must hold for the action to be applicable, an *effect* that is made to hold by executing it, and a cost. Preconditions and effects are conjunctions of *literals*, which are possibly-negated predicate applications. For example, the schema $\textsc{UpH}(y_{from}, y_{to})$ where $y_{from}, y_{to} \in \mathsf{Ys}$ has precondition $\mathsf{AtY}(y_{from}) \wedge \mathsf{Below}(y_{from}, y_{to}) \wedge \mathsf{H}$, effect $\neg\mathsf{AtY}(y_{from}) \wedge \mathsf{AtY}(y_{to})$, and cost 4. In other words, to move from position (x, y_{from}) to (x, y_{to}) with action $\textsc{UpH}(y_{from}, y_{to})$, the agent must be at the initial position, the destination must be one square above this position, and the switch must be horizontal. After doing this action, the agent is at the new position, and no longer at the initial position. Similarly, $\textsc{FlipH}(x, y)$ has precondition $\mathsf{AtX}(x) \wedge \mathsf{AtY}(y) \wedge \mathsf{CanFlip}(x, y) \wedge \neg\mathsf{H}$, effect H, and cost 1. The remaining eight action schemata are defined analogously.

Then, a problem instance for this domain specifies a set of objects for each type, an initial state, and a goal state. For the instance in Figure 2.1, these are as follows.

- Objects: $\mathsf{Xs} = \{\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3\}$, $\mathsf{Ys} = \{\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3\}$. Each predicate generates a set of ground Boolean *propositions* that can be true or false in a given state of the problem, one for each allowed combination of objects; thus, $|\mathcal{S}| = 2^p$ where p is the number of ground propositions. Similarly, each action schema generates a set of ground actions, which define the transition and cost functions $\mathcal{T}$ and $\mathcal{C}$.
- Initial state: $\mathsf{AtX}(\mathsf{x}_1) \wedge \mathsf{AtY}(\mathsf{y}_1) \wedge \mathsf{H} \wedge \mathsf{LeftOf}(\mathsf{x}_1, \mathsf{x}_2) \wedge \mathsf{LeftOf}(\mathsf{x}_2, \mathsf{x}_3) \wedge \mathsf{Below}(\mathsf{y}_1, \mathsf{y}_2) \wedge \mathsf{Below}(\mathsf{y}_2, \mathsf{y}_3) \wedge \mathsf{CanFlip}(\mathsf{x}_2, \mathsf{y}_1) \wedge \mathsf{CanFlip}(\mathsf{x}_2, \mathsf{y}_3) \wedge \mathsf{CanFlip}(\mathsf{x}_3, \mathsf{y}_2)$. All unspecified propositions are false in the initial state.
- Goal: $\mathsf{AtX}(\mathsf{x}_3) \wedge \mathsf{AtY}(\mathsf{y}_3)$, a conjunction of propositions that must be achieved.

The sole optimal solution to this problem is $[\textsc{RightH}(\mathsf{x}_1, \mathsf{x}_2), \textsc{FlipV}(\mathsf{x}_2, \mathsf{y}_1), \textsc{UpV}(\mathsf{y}_1, \mathsf{y}_2), \textsc{UpV}(\mathsf{y}_2, \mathsf{y}_3), \textsc{FlipH}(\mathsf{x}_2, \mathsf{y}_3), \textsc{RightH}(\mathsf{x}_2, \mathsf{x}_3)]$, with cost 10.[2]

In principle, this separation of domain and problem instance allows one to consider *lifted* solution algorithms that operate directly on the first-order predicates and action schemata. However, many competitive planning algorithms *ground* a STRIPS instance before planning, expanding each predicate and action schema out into its legal propositions and ground actions. In addition to enabling lifted solution algorithms, the domain/instance separation allows us to meaningfully consider *domain-configurable* planning algorithms that use additional domain-specific knowledge (e.g., a manually specified action hierarchy) to more effectively plan in the (potentially infinitely) many problem instances in that domain.

2.1.2.2 SAS$^+$

SAS$^+$ (Bäckström, 1992) is a generalization of ground STRIPS that allows for multi-valued state variables, which we will assume in parts of this thesis. A SAS$^+$ problem is represented by a tuple $(\mathcal{V}, \mathcal{D}, \mathcal{A}, s_0, s_*)$:

- $\mathcal{V}$ is the set of multi-valued *state variables*. Each state variable $v \in \mathcal{V}$ takes on values from a *domain* $\mathcal{D}_v$. Each state in the state space $\mathcal{S}$ assigns a value to each variable, so $\mathcal{S} = \bigotimes_{v \in \mathcal{V}} \mathcal{D}_v$. We write $s[v]$ for the value of variable v in state s.

[2]Note that, as described, nav-switch instances have *two* goal states (one per switch orientation). To conform with the above single-goal-state restriction, we can add a zero-cost action Goal with precondition that the agent is at the goal position and effect H, and add H as a goal condition.

- Each action $a \in \mathcal{A}$ is represented by precondition pre_a and effect post_a, which specify values for subsets of the state variables $\mathcal{V}_a^{pre}$ and $\mathcal{V}_a^{post}$, plus a cost c_a.[3] Action a is applicable in state s if $(\forall v \in \mathcal{V}_a^{pre})\ s[v] = \mathsf{pre}_a[v]$. After doing a, state $s' = \mathcal{T}(s, a)$ satisfies
$$s'[v] = \begin{cases} \mathsf{post}_a[v] & \text{if } v \in \mathcal{V}_a^{post} \\ s[v] & \text{otherwise.} \end{cases}$$
- The initial state s_0 is a complete assignment to state variables.
- The goal state s_* is also a complete assignment to state variables.[4]

SAS$^+$ often allows for more concise and natural representations than STRIPS. For instance, an SAS$^+$ version of the nav-switch domain would replace the three Boolean propositions $\mathsf{AtX}(x_i)$ with a single state variable AtX with domain $\{1, 2, 3\}$.

2.1.2.3 Running Example: Discrete Manipulation

As a running example, we use a discrete version of a robotic mobile manipulation domain (Wolfe et al., 2010). In this domain (see Figure 1.1), a mobile robot navigates through a grid environment to deliver each of a set of objects to its goal region. Objects are located on tables, which represent obstacles to the robot base; to manipulate them, when parked the base can extend a gripper up to a fixed distance, which can pick up or put down objects from any direction. Objects can only be picked up from non-goal locations, and put down in goal locations.[5] The robot's gripper must be located on the same square as the base when the base is mobile (unparked). Efficient solutions must optimize details ranging from the top-level task ordering, down to object and base placements, to paths for the base and gripper through the space.

Consider an $n \times n$ instance of this domain, where the base can traverse squares in set F, the gripper can reach up to r squares from the base, and the set of movable objects is O where each $o \in O$ has initial position o_i and set of goal positions o_G. We assume the set of all initial positions and goal regions are disjoint with F. We can encode this instance in SAS$^+$ with the state variables shown in Figure 2.2 and the actions in Figure 2.3.

[3] A distinction is traditionally made between preconditions and prevail conditions, which we omit because it is not relevant to this thesis.

[4] Traditionally, the goal is allowed to be a partial assignment. As mentioned in the previous sections, we can always accommodate this case by adding a zero-cost goal action with the original goal as precondition, which sets all remaining variables to (arbitrarily chosen) canonical values.

[5] These constraints are included because they reduce the reachable state space significantly, without affecting optimal solution quality in most cases.

args	variable	domain	init	represents
	Parked	$\{\mathsf{true}, \mathsf{false}\}$	true	is the base parked or mobile?
	BaseAt	F	$(1, 7)$	position of the robot base
	Holding	$O \cup \{none\}$	$none$	object currently held
	GripperOffset	$\{(x, y) : \lvert x\rvert + \lvert y\rvert \leq r\}$	$(0, 0)$	gripper offset from base
$o \in O$	ObjectAt(o)	$o_G \cup \{o_i, held\}$	o_i	position of object o
$o \in O$	CanMove(o)	$\{\mathsf{true}, \mathsf{false}\}$	true	can o be moved?
$x, y \in [1\ n]$	Free(x, y)	$\{\mathsf{true}, \mathsf{false}\}$	$\cdot$	is table position (x, y) free?

Figure 2.2: State variables for an SAS$^+$ encoding of the discrete manipulation domain

In the optimal solution shown in Figure 1.1, the robot begins by moving one square to the right with [UNPARK, BASER$(1, 7)$, PARK], and extending its gripper to pick up the magazine with [GRIPPERR$(2, 7, 0, 0)$, GETR$(2, 7, 1, 0, \mathsf{m})$]. After moving the gripper back onto the base, the robot unparks and navigates to the bottom table, which contains both the cup and the goal area for the magazine. Upon arrival at position $(7, 4)$, the robot extends the gripper down one square, drops the magazine to the left and picks up the cup to the right, and moves the gripper back home. Finally, the base unparks, navigates to $(10, 7)$, parks, extends the gripper up or right one square, and drops the cup on the table to complete the task.

2.1.3 State-Space Search Algorithms

This section briefly reviews several textbook state-space search algorithms (Russell and Norvig, 2009), as well as some practical techniques for making these algorithms efficient.

2.1.3.1 (Symbolic) Breadth-First Search

When every action has unit cost, the simplest optimal search algorithm is breadth-first search (BFS). This algorithm (see Algorithm 2.1) maintains a *fringe* of newly discovered states, and a *closed* set of states previously explored. In the ith iteration of the loop, the fringe contains the set of states reachable from s_0 with cost i, but not less. When the fringe contains s_*, an optimal solution of length i has been found, and this cost is returned. (If an actual solution is desired, a little extra bookkeeping is required to extract and return it.)

BFS traditionally operates on a single state at a time, where *fringe* is a queue and *closed* is a hash set. In this case, it finds a solution in time linear in the number of states

action	precondition	effect	cost
PARK	Parked = false	Parked = true	5
UNPARK	Parked = true, GripperOffset = $(0, 0)$	Parked = false	2
BASEL(x, y)	Parked = false, BaseAt = (x, y)	BaseAt = $(x-1, y)$	2
GRIPPERL(x, y, gx, gy)	Parked = true, BaseAt = (x, y), GripperOffset = (gx, gy), Free$(x+gx-1, y+gy)$ = true	GripperOffset = $(gx-1, gy)$	1
GETL(x, y, gx, gy, o)	Parked = true, BaseAt = (x, y), GripperOffset = (gx, gy), ObjectAt(o) = $(x+gx-1, y+gy)$, CanMove(o) = true, Holding = $none$	ObjectAt(o) = $held$, Holding = o, Free$(x+gx-1, y+gy)$ = true	1
PUTL(x, y, gx, gy, o)	Parked = true, BaseAt = (x, y), GripperOffset = (gx, gy), ObjectAt(o) = $held$, Free$(x+gx-1, y+gy)$ = true, Holding = o, $(x+gx-1, y+gy) \in o_G$	ObjectAt(o) = $(x+gx-1, y+gy)$, Holding = $none$, Free$(x+gx-1, y+gy)$ = false, CanMove(o) = false	1
... (BASE, GRIPPER, GET, PUT for directions U, R, D)			
GOAL	$(\forall o \in O)$ CanMove(o) = false	$\cdot$ = $goal$	0

Figure 2.3: Actions for an SAS$^+$ encoding of the discrete manipulation domain. Constraints imposed by the variable domains in Figure 2.2 are implicit.

Algorithm 2.1 (Symbolic) Breadth-First Search

```
function BFS()
    fringe ← {s_0}
    closed ← {}
    steps ← 0
    while fringe ≠ ∅ do
        if s_* ∈ fringe then return steps
        closed ← closed ∪ fringe
        fringe ← {s' : s ∈ fringe and a ∈ A_s and s' = T(s, a)} \ closed
        steps ← steps + 1
    return ∞
```

reachable from s_0 while incurring no more than the optimal solution cost. As written in Algorithm 2.1, another possibility exists: $\mathcal{T}$, *fringe* and *closed* can be represented *implicitly*, e.g., using logical formulae or binary decision diagrams (BDDs) (Bryant, 1992). In this case, each iteration of the loop can be carried out *symbolically*, without ever explicitly enumerating the states reachable at each depth (Cimatti et al., 1997), potentially leading to exponential speedups.[6] The algorithms described in Chapter 5 build upon this idea, in the context of hierarchical planning.

When actions can have variable costs, the natural generalization of BFS is Dijkstra's algorithm (Dijkstra, 1959). The next section reviews the A* algorithm, which extends Dijkstra's algorithm to allow for the use of heuristic information.

2.1.3.2 A* Search

A* (Hart et al., 1972) is an optimal search algorithm that can leverage *heuristic* cost bounds to increase search efficiency. The algorithm maintains a *fringe* of newly discovered states paired with the cost to reach them from s_0 along some particular path, as well as a *closed* hash table mapping previously explored states to the *optimal* cost to reach them from s_0. The fringe is represented as a priority queue (e.g., heap), sorted by $f(s) = c + h(s)$, where c is the cost incurred to reach s from s_0 and $h(s)$ is a heuristic lower bound on the remaining cost to reach s_* from s.

Definition 2.8. *A heuristic function $h(s)$ is* admissible *iff it never overestimates the cost to reach the goal: $(\forall s \in \mathcal{S})\ h(s) \leq c^*(s)$.*

Theorem 2.5. *(Hart et al., 1972) A* tree search (i.e., without a closed map) is optimal if h is admissible.*

[6] Other planning algorithms such as SATPLAN (Kautz and Selman, 1992) are based on similar ideas.

Algorithm 2.2 A*

```
function A*()
    fringe ← a priority queue on pairs (s, c) ordered by min c + h(s), initially containing {(s_0, 0)}
    closed ← an empty mapping from state to optimal solution cost
    while fringe not empty do
        (s, c) ← fringe.RemoveMin()
        if s = s_* then return c
        if closed[s] = undefined then
            closed[s] ← c
            for a ∈ A_s do fringe.Insert((T(s, a), c + C(s, a)))
        else assert c ≥ closed[s] /* otherwise, h is inconsistent */
    return −∞
```

A* tree search is optimal under an admissible heuristic because $f(s)$ is a lower bound on the cost of the cheapest solution that passes through s, and thus when s_* is first popped from the queue, every state on some optimal path to s_* must have already been expanded.

The optimality of A* graph search (as shown in Algorithm 2.2) depends on a stronger condition:

Definition 2.9. *A heuristic function $h(s)$ is* consistent *iff it obeys the following triangle inequality:*

$$(\forall s, a) \;\; h(s) \leq h(\mathcal{T}(s, a)) + \mathcal{C}(s, a).$$

Under a consistent heuristic, if s' is a successor of s, then $f(s') \geq f(s)$.

Theorem 2.6. *(Hart et al., 1972) A* graph search is optimal when h is consistent.*

Consistency ensures that when s is first popped from *fringe*, its associated cost c is optimal, and subsequent occurrences of s can be safely skipped.

Because A* examines only states with $f(s) \leq c^*(s_0)$, A* can avoid examining a significant portion of the state space when h is accurate. However, if there are many states on or near an optimal trajectory, A* still must examine the exponentially many states with $f(s) < c^*(s_0)$ even given an almost-perfect heuristic (Helmert and Röger, 2008).

Given a STRIPS planning problem, a variety of methods exist to automatically construct consistent heuristics for use with A* search, including pattern databases (Culberson and Schaeffer, 1996), planning graphs (Blum and Furst, 1995), causal graphs (Helmert, 2004), and landmarks (Richter and Westphal, 2010).

2.1.3.3 Suboptimal Search Algorithms

While optimal solutions are clearly desirable if we can find them, on sufficiently large problems A* can require too much time or memory to be practical. A variety of related search algorithms exist that can trade off optimality for computation time, while still providing bounds on the sub-optimality of the discovered solution.

The canonical such algorithm is weighted A* (Pohl, 1970), which is simply A* where the heuristic value $h(s)$ is multiplied by a weight $w \geq 1$. Weighted A* is guaranteed to find a solution whose cost is at most $w \times c^*(s_0)$, and often does so much faster than A*, because it greedily puts more weight on the estimated distance to the goal than the cost paid so far. Many variants upon this approach have been proposed, with explicit estimation search (Thayer and Ruml, 2010) being a recent promising contender.

One problem with weighted A* is that one must select the sub-optimality bound w in advance. Another family of *anytime* algorithms, including anytime repair A* (Likhachev et al., 2003), relax this restriction by repeatedly searching with decreasing values of w until computation time runs out, returning the best solution found thus far.

2.1.3.4 Practical Considerations

In addition to effective search algorithms and heuristics, efficient state-space planners require the ability to quickly compute the successors of a state. In particular, the most significant challenge is efficiently generating the set of actions $\mathcal{A}_s$ applicable in a given state s.

For instance, consider the nav-switch domain presented in Section 2.1.2.1. In principle, in an $n \times n$ world there are $8n^2$ possible move actions (e.g., $\textsc{LeftH}(\mathsf{x_1}, \mathsf{x_1})$) and $2n^2$ flip actions (e.g., $\textsc{FlipH}(\mathsf{x_1}, \mathsf{y_1})$). However, in any given state of the world, at most 5 such actions are applicable (moving one step, or flipping the switch). Efficiently generating this set of applicable actions directly, without explicitly testing every possible action for applicability, can reduce planning time by several orders of magnitude even in modestly-sized instances.

Data structures for this problem are called *successor generators*, and they are typically built up in two steps. First, static analysis is used to ground the action schemata, generating only ground actions that are potentially applicable in any state reachable from s_0. Then, these ground actions are organized into a data structure that supports efficient queries for the subset applicable in a particular reachable state. Helmert (2006) provides a detailed overview of these techniques; we briefly summarize them here, to serve as a basis for later discussion of related techniques in the hierarchical setting.

First, we must generate the feasible ground actions, by grounding each action schema. Consider the $\textsc{UpH}(y_{from}, y_{to})$ action schema. This schema allows for nonsensical actions such as $\textsc{UpH}(\mathsf{y_1}, \mathsf{y_1})$, which can never be applicable because $\mathsf{Below}(\mathsf{y_1}, \mathsf{y_1})$ is always false.

In particular, the Below predicate is *constant*: no legal action affects it, and so its values in the initial state must persist throughout planning. Such constant predicates can be automatically identified, and any ground action that does not match their instantiation in s_0 can be pruned (and then the constant predicates can be removed from the domain entirely). Because the number of possible ground actions can be many orders of magnitude larger than the consistent ones, rather than generate them all and then prune the inconsistent ones, it is often desirable to generate the consistent ground actions directly. This can be done by formulating a constraint satisfaction problem (CSP) in which the variables are the arguments to the action schema and the constant predicates in its precondition are the constraints. Each solution to the CSP generates a consistent ground action instantiation.

Second, we organize this set of consistent actions into a successor generator data structure that supports efficient lookups for the actions applicable in a particular reachable state. A simple method is to build a decision tree, where each internal node tests a single proposition p in state s. The actions are partitioned into three sets: those that require p, those that require $\neg p$, and those that don't care. Each leaf of the tree holds a set of ground actions with the same precondition. At query time, the applicable actions for a state are generated by traversing the tree, following the applicable and "don't care" branches from each node (when they exist), and unioning the sets of actions at the leaves reached.

2.1.4 Partial-Order Planning

If we have access to a structured description of a planning domain (e.g., STRIPS descriptions), methods other than state-space search are possible. In partial-order planning, perhaps the best-studied alternative, plans are partially ordered networks of actions reaching backwards from the goal. Partial-order planning was pioneered in NOAH (Sacerdoti, 1975), systematized in TWEAK (Chapman, 1987) and SNLP (Mcallester and Rosenblitt, 1991), and studied and extended by many other researchers. While this thesis is only concerned with fully ordered planning algorithms, we briefly summarize the main ideas for comparison with previous hierarchical planning approaches, many of which were implemented within a partial-order planning framework.

The central intuition behind partial-order planning is *least commitment*. At each step, a partial-order planning algorithm attempts to make a smallest commitment that can make progress towards solving the problem. By deferring arbitrary choices until later, the planner will not have to backtrack over values for these choices if a fundamental inconsistency in its plan is later discovered.

Specifically, a partial-order planner searches through the space of partially ordered plans, which include actions (possibly with some arguments as-yet unspecified), causal links (e.g., action a effects a precondition of action b), and ordering constraints (e.g., a must come before b). In a solution, every precondition and goal condition must be satisfied by a causal link from

an action or the initial state, but actions concerned with different subsets of propositions need not be ordered. Partial-order planners are also often *lifted*, allowing as-yet uninstantiated variables as arguments to actions in their partial plans. At each planning step, a *conflict* in the current plan – either an unsatisfied precondition, or a threat to a causal link – is selected, and the algorithm branches over possible resolutions to this conflict. When an irresolvable conflict is detected, search must backtrack and try another branch.

2.2 AND/OR Graph Search

The previous section described classical planning problems, in which the search space is effectively an OR-graph: from each state (or partial-order plan), the agent gets to choose which action (or plan modification) to do next. Such problems are a subset of the larger class of *AND/OR* graph search problems, which include AND-nodes at which the agent must solve a *set* of subproblems to find a solution. AND/OR search problems arise in a wide variety of applications, including nondeterministic and probabilistic planning (where AND-nodes represent the observation of an uncertain outcome), natural language parsing (where AND-nodes represent sub-parses for adjacent sentence fragments), and hierarchical planning (where AND-nodes represent a sequence of abstract actions).

Unlike in classical search, where A* is the dominant optimal algorithm, in AND/OR search there exist a variety of non-dominated optimal search algorithms. Which algorithm is best in a particular situation can depend on a variety of factors, including whether the search space is a tree, directed acyclic graph (DAG), or general directed cyclic graph (DCG), whether the "top-down" or "bottom-up" branching factor is bigger, what heuristics are available, and so on.

This section defines a formalism for AND/OR graph search problems that generalizes the state-space formalism in the previous section, and reviews several relevant families of optimal AND/OR search algorithms.

2.2.1 AND/OR Search Problems

2.2.1.1 Formalism

This section presents a generic formalism for AND/OR search problems, combining ideas and notation from a variety of sources (Kumar et al., 1985; Kambhampati et al., 1995; Felzenszwalb and McAllester, 2007)

The basic entity in this formalism is a *subproblem*, which roughly corresponds to a state in the state-space framework. The problem-solving agent is given a single *root* subproblem $\mathcal{P}_0$ to solve. Each subproblem may be terminal (trivially solved), or it may admit a set

of decompositions (called *refinements*) into sets of other subproblems. *All* subproblems appearing in at least *one* refinement must be solved to solve the parent subproblem. Thus, solutions correspond to *graphs* in which $\mathcal{P}_0$ is the root, all leaves are terminal, and each nonterminal subproblem (OR-node) has a single child corresponding to a refinement (AND-node), each of whose child subproblems is the root of a solution graph. Figure 2.4 shows an example AND/OR graph and the tree it represents, and Figure 2.5 shows the two solutions for this graph.

Definition 2.10. *An AND/OR search problem is given by a tuple* $(\mathcal{P}_0, \mathcal{Z}, \mathcal{R})$*:*

- $\mathcal{P}_0$ *is the root subproblem.*
- $\mathcal{Z}(p)$ *returns an optimal solution (discussed shortly) for subproblem* p *if* p *is terminal, and otherwise returns* NT *(for nonterminal).*
- $\mathcal{R}(p)$ *returns the refinements of nonterminal subproblem* p*. Each refinement* $i \in \mathcal{R}(p)$ *can be written as a production* $\pi_i \to_{g_i} \lambda_i\ \rho_i$ *where* $\pi_i = p$*. The meaning is that a solution to subproblem* λ_i *with cost* c_λ *and a solution to subproblem* ρ_i *with cost* c_ρ *combine to yield a solution to the original subproblem* $\pi_i = p$ *with cost* $g_i(c_\lambda, c_\rho)$*.* g_i *is a* cost function*, often just addition, which we discuss further in Section 2.2.1.3.*

Remark. *This formalism assumes that every refinement consists of* exactly two *subproblems. Any AND/OR search with pairwise factorable cost functions can be compiled into this form, which leads to significantly simpler search algorithms in many cases.*[7]

Assumption 2.7. *We assume that each subproblem has a finite set of refinements, and that the set of all subproblems reachable by refinements of* $\mathcal{P}_0$ *is finite.*

For concreteness, Algorithm 2.3 shows perhaps the simplest AND/OR search algorithm, depth-first tree search (DFS). Of course, DFS is not optimal because it returns the first discovered solution regardless of cost, and not complete because it may, e.g., get stuck in a loop examining the first refinement while the second refinement represents a trivial solution.

Definition 2.11. *The* solution set *for a subproblem* p *or refinement* i*, written* $\langle\!\langle p \rangle\!\rangle$ *or* $\langle\!\langle i \rangle\!\rangle$*, is recursively defined by:*[8]

$$\langle\!\langle p \rangle\!\rangle := \begin{cases} \{\mathcal{Z}(p)\} & \text{if } \mathcal{Z}(p) \neq NT \\ \bigcup_{i \in \mathcal{R}(p)} \langle\!\langle i \rangle\!\rangle & \text{otherwise.} \end{cases}$$

$$\langle\!\langle i \rangle\!\rangle := \{\textsc{sol}(i, z_\lambda, z_\rho) : z_\lambda \in \langle\!\langle \lambda_i \rangle\!\rangle \text{ and } z_\rho \in \langle\!\langle \rho_i \rangle\!\rangle\},$$

[7] All of the search algorithms we describe can be adapted to directly handle general refinements containing zero or more subproblems, at the cost of some algorithmic complexity.

[8] With certain caveats regarding cyclic solutions, to be discussed in Section 2.2.1.3.

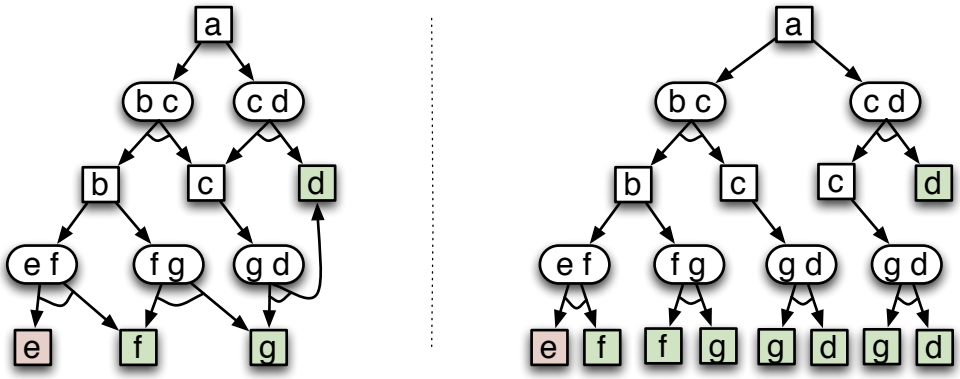

Figure 2.4: An acyclic example AND/OR graph (left), which implicitly represents a larger AND/OR tree (right). Boxes are OR-nodes, ovals are AND-nodes, green leaves are terminal (solved), and the red leaf has no refinements (a dead end).

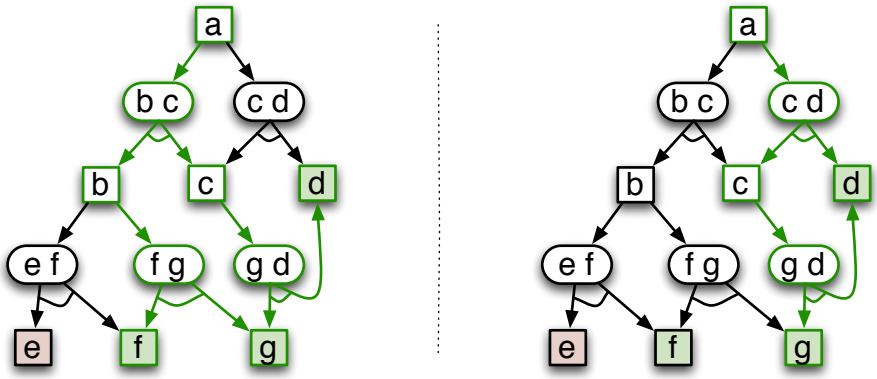

Figure 2.5: The two solutions (in green) for the AND/OR graph in Figure 2.4.

Algorithm 2.3 Depth-First Tree Search

```
function DFS(p)
    if Z(p) ≠ NT then return Z(p)
    for i ∈ R(p) do
        z_λ ← DFS(λ_i)
        if z_λ ≠ ⊥ then
            z_ρ ← DFS(ρ_i)
            if z_ρ ≠ ⊥ then
                return SOL(i, z_λ, z_ρ)
    return ⊥
```

where $\textsc{sol}(i, z_\lambda, z_\rho)$ *is a* solution constructor *that returns a node with label* i*, left child* z_λ*, and right child* z_ρ*.*

Each node in a solution graph is also labeled with a cost.

Definition 2.12. *The cost of a solution* z *is written* $c(z)$*. This cost is given for terminal solutions. For nonterminal solutions* $z = \textsc{sol}(i, z_\lambda, z_\rho)$*,* $c(z) := g_i(c(z_\lambda), c(z_\rho))$*. Solution costs are always finite.*

Remark. *For algorithmic convenience, we will sometimes assume a dummy solution* $\perp$ *with* $c(\perp) = \infty$*, representing the lack of any solutions.*

Finally, optimal costs and solutions are defined in the obvious ways.

Definition 2.13. *The* optimal cost *for subproblem or refinement* x *is* $c^*(x) := \min_{z \in \langle\langle x \rangle\rangle} c(z)$*.*

Definition 2.14. *The set of* optimal solutions *for* x *is* $\langle\langle x \rangle\rangle_* := \{z : z \in \langle\langle x \rangle\rangle \text{ and } c(z) = c^*(x)\}$*, just those solutions with minimal cost.*

Figure 2.6 shows optimal costs and solutions for an example AND/OR graph.

2.2.1.2 Examples

We briefly present some example AND/OR search problems. We defer discussion of our application to hierarchical planning to Chapters 3 and 5.

A first, trivial example shows that AND/OR search problems include state-space search problems as a special case. Let $\mathcal{P}_0 = s_*$, and for each state s and action $a \in \mathcal{A}_s$ define a refinement $\mathcal{T}(s, a) \to_+ s \;\; a_s$. Subproblem s_0 is terminal with cost 0, and action subproblems a_s are terminal with cost $\mathcal{C}(s, a)$.

Turn-based games can also be solved using AND/OR search (i.e., minimax). If after taking action a from state s, the opponent has two possible responses that lead to states s_1 and s_2, this generates a refinement $s \to_{\max} s_1 \;\; s_2$. Probabilistic planning is similar, except the cost function computes the expected cost (averaged over the probabilities of reaching states s_i) rather than the maximum.

2.2.1.3 Cost Functions

The previous examples included a variety of refinement cost functions $g_\cdot$, including addition, maximum, and expectation. This section reviews some key properties of cost functions,

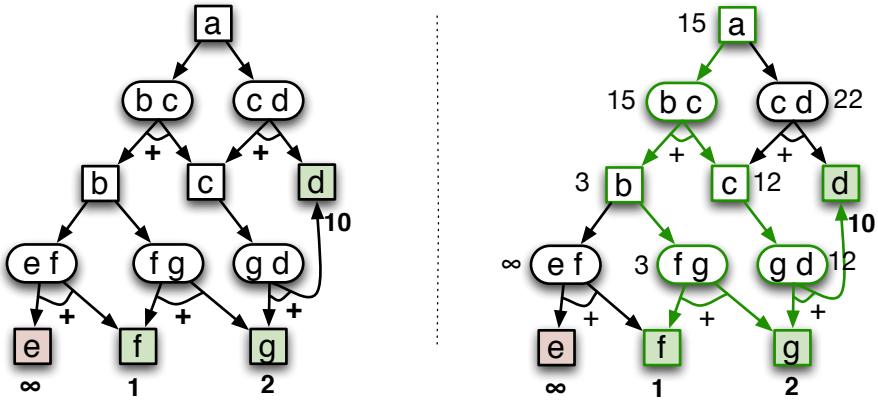

Figure 2.6: A concrete instantiation of the AND/OR graph of Figure 2.4 with terminal costs and additive cost functions (left), and the corresponding optimal costs and solution (right).

which have implications for the structure of the search space and types of search algorithms that can be used.

A first property of many cost functions is *monotonicity*.

Definition 2.15. *A cost function g_i is* monotonic *iff, for all costs $c_1, c_2, c_3 \in [0, \infty)$,*

$$c_1 \leq c_2 \;\Rightarrow\; (g(c_1, c_3) \leq g(c_2, c_3) \text{ and } g(c_3, c_1) \leq g(c_3, c_2))$$

A search problem is monotonic *iff all refinement cost functions are monotonic.*

This property is essential for ensuring *compositionality* of optimal solutions, holds for all of the above cost functions, and is necessary for all of the search algorithms we consider.

Theorem 2.8. *In a monotonic search problem, if any solution exists for nonterminal subproblem p, there always exists a refinement $i \in \mathcal{R}(p)$ s.t. there exist solutions for λ_i and ρ_i, and moreover, every combination of optimal solutions for λ_i and ρ_i yield an optimal solution for p:*

$$(\forall p : \langle\!\langle p \rangle\!\rangle \neq \emptyset)\ (\exists i \in \mathcal{R}(p))\ \ \langle\!\langle \lambda_i \rangle\!\rangle \neq \emptyset \text{ and } \langle\!\langle \rho_i \rangle\!\rangle \neq \emptyset \text{ and } (\forall z_\lambda \in \langle\!\langle \lambda_i \rangle\!\rangle_*, z_\rho \in \langle\!\langle \rho_i \rangle\!\rangle_*)\ \ \textsc{sol}(i, z_\lambda, z_\rho) \in \langle\!\langle p \rangle\!\rangle_*$$

Proof. Consider any optimal solution $z \in \langle\!\langle p \rangle\!\rangle_*$, and suppose $z = \textsc{sol}(i, z_\lambda, z_\rho)$. Let $z^*_\lambda \in \langle\!\langle \lambda_i \rangle\!\rangle_*$ and $z^*_\rho \in \langle\!\langle \rho_i \rangle\!\rangle_*$ be arbitrary optimal solutions for λ_i and ρ_i. By definition of optimality, $c(z^*_\lambda) \leq c(z_\lambda)$ and $c(z^*_\rho) \leq c(z_\rho)$. Thus, by monotonicity of g_i, $g_i(c(z^*_\lambda), c(z^*_\rho)) \leq g_i(c(z_\lambda), c(z_\rho))$, and $\textsc{sol}(i, z^*_\lambda, z^*_\rho)$ is optimal for p. □

Moreover, lower and upper bounds on optimal costs compose similarly:

Theorem 2.9. *In a monotonic search problem,*

$$(\forall p)\left(\left((\forall i \in \mathcal{R}(p))\ c^*(i) \in [u_i, v_i]\right) \Rightarrow c^*(p) \in \left[\min_{i\in\mathcal{R}(p)} u_i, \min_{i\in\mathcal{R}(p)} v_i\right]\right)$$

$$(\forall i)\left((c^*(\lambda_i) \in [u_\lambda, v_\lambda] \text{ and } c^*(\rho_i) \in [u_\rho, v_\rho]) \Rightarrow c^*(i) \in [g_i(u_\lambda, u_\rho), g_i(v_\lambda, v_\rho)]\right)$$

Proof. The lower bound in the first statement follows because each solution for p must be a solution for *some* refinement $i \in \mathcal{R}(p)$, and the upper bound follows because *every* solution to a refinement $i \in \mathcal{R}(p)$ is a solution to p. The second statement follows directly from monotonicity of g_i. □

In other words, if we know that every refinement of p costs at least c, then p must cost at least c. Similarly, if some refinement costs at most c, p must cost at most c. For refinements, monotonicity of g_i ensures that plugging in lower bounds yields a lower bound, and similarly for upper bounds.

Another key property is *superiority*.

Definition 2.16. *A cost function g_i is* superior *iff, for all costs $c_1, c_2 \in [0, \infty)$,*

$$g_i(c_1, c_2) \geq \max(c_1, c_2).$$

A search problem is superior *iff all refinement cost functions are superior.*

Superiority means that the cost of a solution is never less than the cost of its constituent sub-solutions. This property is necessary for *bottom-up* search algorithms, discussed shortly, and holds for all of the above cost functions *except* expectation (used in probabilistic planning). Superiority also has consequences for cyclic solutions, which we discuss next.

2.2.1.4 Cycles and Cyclic Solutions

AND/OR graphs can contain cycles (see Figure 2.7), whenever a subproblem p can include itself in a (refinement of a) refinement of itself. The mere existence of cycles has implications for the applicability and performance of various search algorithms, as in the pure OR-search setting. Not all cycles are created equal, however: some cyclic AND/OR graphs admit finite-cost *cyclic solutions*, and others do not.

Definition 2.17. *A* cyclic solution *is a subgraph of an AND/OR graph that*

1. *is a solution: contains $\mathcal{P}_0$, one child of each OR-node, and every child of each AND-node;*

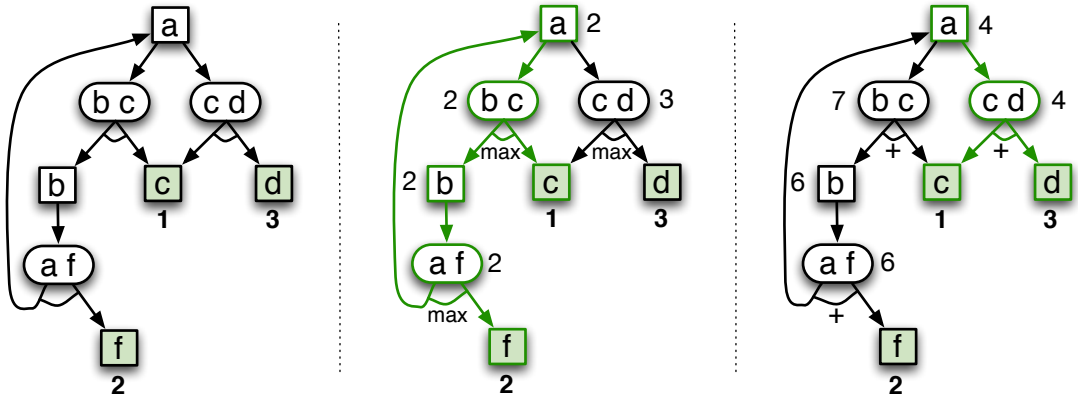

Figure 2.7: A cyclic AND/OR graph (left). With max as a cost function, the optimal solution is cyclic (center). With addition as a cost function, the optimal solution is acyclic, and the cyclic solution would be assigned infinite cost (right).

2. *is cyclic: contains at least one node that is an ancestor of itself; and*

3. *can be labeled with a consistent set of costs, so that each OR-node has cost equal to that of its child, each AND-node has cost equal to $g(\cdot, \cdot)$ of its children's costs, and $\mathcal{P}_0$ is labeled with finite cost.*

This thesis is only concerned with search problems that *do not* admit cyclic solutions. The existence of cyclic solutions is related to superiority (see Definition 2.16). In particular, *strictly* superior search problems cannot have cyclic solutions. Since addition is not strictly superior, we define an additional condition later that ensures that our search spaces are free of cyclic solutions.

2.2.1.5 Algorithm Overview

This thesis will have use for optimal AND/OR search algorithms, of which there are at least four broad classes.

- Algorithms that can find cyclic solutions include value iteration (Bellman, 1957), policy iteration (Howard, 1960), and real-time dynamic programming (Barto et al., 1995). These algorithms are typically overkill (i.e., slow compared to competing algorithms) for problems without cyclic solutions, so we do not consider them further.

- Bottom-up search algorithms start with the set of terminal subproblems, and work upwards, building up larger and larger optimal solutions until a solution to the root subproblem is reached. Bottom-up algorithms are especially effective at handling cycles

in the search space (but cannot find cyclic solutions). They tend to work well when the set of terminal subproblems is small, and the "upwards" branching factor is small (e.g., probabilistic context-free grammar parsing).

- Top-down search algorithms build and refine an explicit subgraph downwards from $\mathcal{P}_0$. At each step a nonterminal leaf is expanded, and updated cost *bounds* are propagated upwards towards the root. Cycles in the search space can be problematic for these algorithms, but they can be much more efficient than bottom-up algorithms when the "top-down" branching factor is smaller (e.g., chess checkmate problems).
- Finally, hybrid search algorithms combine elements of both top-down and bottom-up algorithms, capturing some of the desirable properties of both.

The next three sections describe the last three classes in more detail, including at least one relevant algorithm from each class.

2.2.2 Bottom-Up Search Algorithms

2.2.2.1 Framework

Bottom-up AND/OR search algorithms begin with the set of terminal subproblems, whose optimal solutions are known *a priori*. At each search step, they compose two known optimal solutions to build an optimal solution to a new, larger subproblem. This process continues until an optimal solution for the root subproblem $\mathcal{P}_0$ is generated.

In this sense, bottom-up algorithms are analogous to state-space search algorithms like Dijkstra's algorithm and A* search. In fact, the bottom-up algorithms we discuss are direct generalizations of these state-space algorithms for the AND/OR search setting.

While our AND/OR search space was defined top-down in terms of $\mathcal{P}_0$ and $\mathcal{R}$, bottom-up algorithms need access to the set of terminal subproblems and the inverse of $\mathcal{R}$.

Definition 2.18. *The set of* terminal subproblems $T := \{p : \mathcal{Z}(p) \neq NT\}$.

Definition 2.19. *The* predecessor function $\mathcal{P}(p, closed)$ *(the inverse of $\mathcal{R}$) takes a subproblem p and a mapping "closed" from subproblems to optimal solutions (including p), and returns a set of pairs (p_{new}, z_{new}) for each refinement $i \in \mathcal{R}(p_{new})$ where $p_{new} \notin closed$, $\lambda_i, \rho_i \in closed$, $p \in \{\lambda_i, \rho_i\}$, and $z_{new} = \text{SOL}(i, closed[\lambda_i], closed[\rho_i])$*

In other words, $\mathcal{P}(p, closed)$ returns the set of all new solutions that can be built from the optimal solution to p and other optimal solutions in *closed*.

Finally, like A*, bottom-up algorithms can use heuristic functions to guide their searches. However, the definition of a heuristic is somewhat more involved in this setting.

Definition 2.20. *Consider any solution for $\mathcal{P}_0$ that includes a solution to p, and remove the solution to p, leaving a "hole". Call the remaining partial solution a* solution context *for p. This context corresponds to a particular* context cost function*, which yields a cost for $\mathcal{P}_0$ when passed a cost for p. (With additive costs, this function just adds the costs of the remaining leaves.) Define the* optimal context cost *$g_p^*(\cdot)$ for a subproblem p to be the infimum over all such context cost functions.*

Remark. *In the additive case, the optimal context cost function is $g_p^*(x) = x + c^*(\mathcal{P}_0) - c^*(p)$*

Now, a bottom-up heuristic is just a function that estimates the optimal context cost. Admissibility and consistency are defined as in the single-agent setting.

Definition 2.21. *An* admissible *bottom-up heuristic is a set of functions (one per subproblem p) $\hat{g}_p(\cdot)$ that satisfy*

$$(\forall x)\ \ \hat{g}_p(x) \leq g_p^*(x).$$

Definition 2.22. *A* consistent *bottom-up heuristic is a set of functions (one per subproblem p) $\hat{g}_p(\cdot)$ that satisfy*

$$(\forall i \in \mathcal{R}(p))\ \ \hat{g}_p(g_i(c^*(\lambda_i), c^*(\rho_i))) \geq \max\left(\hat{g}_{\lambda_i}(c^*(\lambda_i)),\ \hat{g}_{\rho_i}(c^*(\rho_i))\right).$$

In fact, Definitions 2.8 and 2.9 for state-space search are special cases of these more general definitions, for additive cost functions under the mapping given in Section 2.2.1.2.

2.2.2.2 A* Lightest Derivation

A* lightest derivation (A*LD, (Klein and Manning, 2003; Felzenszwalb and McAllester, 2007)) is a straightforward generalization of A* to the AND/OR setting (see Algorithm 2.4).If the heuristic $\hat{g}$ is set to the identity, Knuth's lightest derivation (KLD, (Knuth, 1977)), a generalization of Dijkstra's algorithm, is recovered as a special case. Figure 2.8 shows the steps taken by KLD to optimally solve an example graph.

Theorem 2.10. *(Felzenszwalb and McAllester, 2007) When applied to a monotone, superior search problem with a consistent heuristic, A*LD always terminates with an optimal solution.*

Felzenszwalb and McAllester (2007) also developed a hierarchical extension of A*LD called HA*LD, which we discuss in Chapter 7.

2.2.3 Top-Down Search Algorithms

This section describes a framework for top-down search largely inspired by Kumar et al. (1985), as well as two algorithms within this framework: AO* and LDFS.

Algorithm 2.4 A* Lightest Derivation

function A*LD( )
  fringe ← a priority queue on pairs (p, z) ordered by min $\hat{g}_p(c(z))$, initially $\{(p, \mathcal{Z}(p)) : p \in T\}$
  closed ← an empty mapping from subproblems to their optimal solutions
  while *fringe* not empty **do**
    (p, z) ← *fringe*.RemoveMin()
    if $p = \mathcal{P}_0$ **then return** z
    if *closed*$[p]$ = undefined **then**
      closed$[p]$ ← z
      for $(p', z') \in \mathcal{P}(p, closed)$ **do**
        fringe.Insert((p', z'))
  return $\perp$

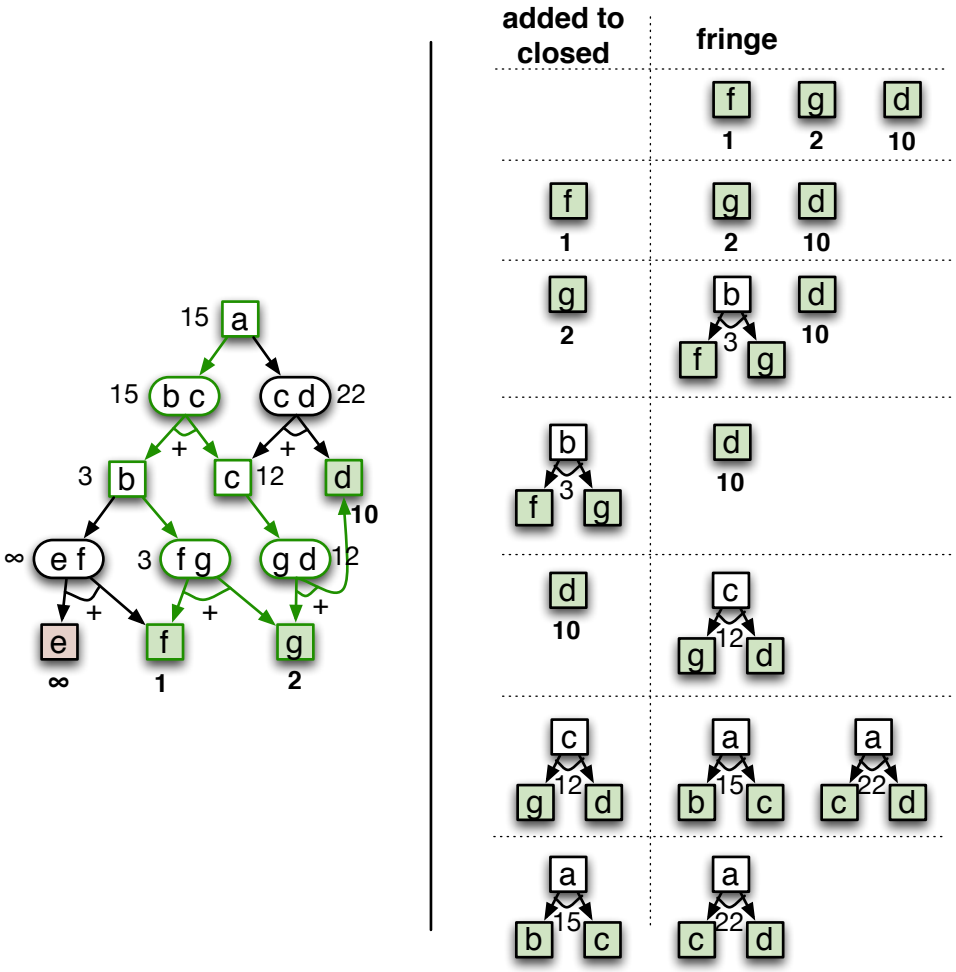

Figure 2.8: The AND/OR graph in Figure 2.6 (left), and steps to optimally solve it using Knuth's Lightest Derivation, i.e., A*LD with $\hat{g}(x) = x$ (right).

2.2.3.1 Framework

Top-down AND/OR search algorithms maintain an explicit graph, in which OR-nodes correspond to subproblems and AND-nodes correspond to refinements. Initially, this graph contains a single node corresponding to $\mathcal{P}_0$. At each search step, a single leaf subproblem node is selected to EXPAND, generating nodes corresponding to its refinements and their subproblems and adding them to the graph. Throughout search, each node maintains a *summary*, including a lower bound on its optimal cost and a flag indicating whether its optimal solution is yet known, computed from the summaries of its children in the graph (see Figure 2.9).

The pseudocode in Algorithm 2.5 and Algorithm 2.6 provides a modular scaffolding for top-down AND/OR search algorithms, upon which several of the search algorithms in this thesis will be built.

The first set of functions construct and operate on graph nodes. MAKEORNODE and MAKEANDNODE actually construct the nodes, each of which has a *name* (subproblem for OR, refinement for AND), lists of *parents* and *children*, and a current *summary*. Then, GETORNODE keeps a cache of constructed OR-nodes, so that multiple occurrences of a subproblem share the same OR-node, which has one parent for each constructed refinement referencing it.[9]

The last two functions in Algorithm 2.5 provide the interface used by search algorithms. MAKEROOTNODE creates a root OR-node with no parents, corresponding to the initial subproblem $\mathcal{P}_0$. Then, EXPAND expands a leaf node, constructing one child AND-node per refinement, each of which has two child OR-nodes corresponding to subproblems in the refinement (which are fetched from the cache, or created and cached if not yet present).

Next, Algorithm 2.6 describes pseudocode for operations on node *summaries*. First, function CURRENTSUMMARY(n, b) computes the current summary of a node n from the current summaries of its children, taking into account an existing lower cost bound b. A summary $\hat{z}$ encapsulates what is known about a subproblem p (or refinement) given the current graph:

- $\hat{z}.children$ is a the set of child nodes this summary depends on. $\hat{z}.children$ is empty for leaf nodes, a singleton containing the current-best child (with the lowest lower bound, breaking ties towards unrefinable nodes) for OR-nodes, and a list of both children at AND-nodes.

[9] Because we are not searching for cyclic solutions, we could skip this caching and perform a tree search instead. Algorithms for searching trees are typically simpler, but can be exponentially less efficient than the corresponding graph algorithms.

Algorithm 2.5 Top-Down AND/OR Graph Framework

```
function MakeOrNode(p)
    return a node n with
        n.name = p,
        n.parents = []
        n.children = leaf
        n.summary = h(p)

function MakeAndNode(parent, i)
    return a node n with
        n.name = i,
        n.parents = [parent]
        n.children = [GetOrNode(n, λ_i), GetOrNode(n, ρ_i)]
        n.summary = CurrentSummary(n, 0)

function GetOrNode(parent, p)
    if cache[p] = undefined then cache[p] ← MakeOrNode(p)
    cache[p].parents.Insert(parent)
    return cache[p]

function MakeRootNode()
    return MakeOrNode(P_0)

function Expand(n) /* n.children = leaf and n.summary.refinable? */
    n.children ← [MakeAnd(n, i) for i in R(n.name)]
```

Algorithm 2.6 Summaries

function CURRENTSUMMARY($n, bound$) `/*` $n.children \neq$ leaf `*/`
  if n is an OR-node **then**
    $best \leftarrow$ a dummy node with $best.summary.lb = \infty$ and $\neg best.summary.refinable?$
    for $c \in n.children$ **do**
      if $c.summary.lb < best.summary.lb$ or
        ($c.summary.lb = best.summary.lb$ and $\neg c.summary.refinable?$) **then**
          $best \leftarrow c$
    return a summary with $\hat{z}.children = [best]$,
      $\hat{z}.refinable? = best.summary.refinable?$, and
      $\hat{z}.lb = \max(best.summary.lb, bound)$
  else `/*` n `is an AND-node with 2 children */`
    $[\hat{z}_\lambda, \hat{z}_\rho] \leftarrow [n.children[0].summary, n.children[1].summary]$
    return a summary with $\hat{z}.children = n.children$,
      $\hat{z}.refinable? = \hat{z}_\lambda.refinable?$ or $\hat{z}_\rho.refinable?$, and
      $\hat{z}.lb = \max(g_{n.name}(\hat{z}_\lambda.lb, \hat{z}_\rho.lb), bound)$

function EXTRACTSOLUTION(n) `/*` n `is an OR-node with` $\neg n.summary.refinable?$ `*/`
  if $n.summary.lb = \infty$ **then return** $\perp$
  if n is a leaf **then return** $\mathcal{Z}(n.name)$
  $c \leftarrow$ the child in singleton $n.summary.children$
  return SOL($c.name$, EXTRACTSOLUTION($c.children[0]$), EXTRACTSOLUTION($c.children[1]$))

function UPDATESUMMARY(n)
  $old \leftarrow n.summary$
  $n.summary \leftarrow$ CURRENTSUMMARY($n, old.lb$)
  return $\neg$ ($old.lb = n.summary.lb$ and $old.refinable? = n.summary.refinable?$)

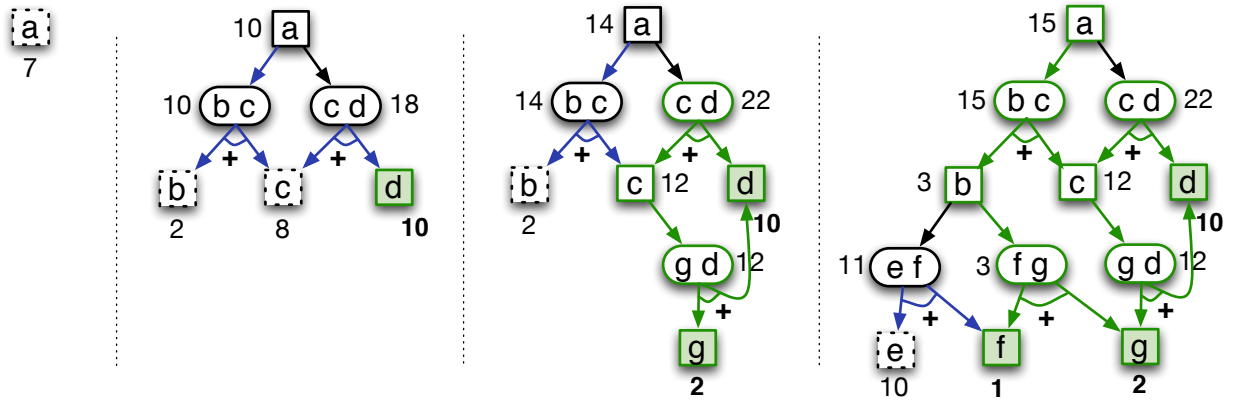

Figure 2.9: Steps to solve the AND/OR graph of Figure 2.6 using a top-down algorithm. Unexpanded leaf nodes (depicted with dashed borders) are assigned heuristic cost bounds. All other non-terminal nodes are assigned a summary based on the summaries of their descendants. Nodes with green borders are solved (unrefinable), and others are refinable. Blue lines indicate the current-best children of unsolved nodes, and green lines indicate optimal solutions for solved nodes. After expanding nodes a, c, and b (from left to right), a provably optimal solution is discovered. Node e remains unexpanded, because its heuristic cost bound guarantees that all solutions containing e are more expensive than the optimal solution at right.

- $\hat{z}.\mathit{refinable}?$ is false iff the graph contains a provably optimal solution for p (i.e., if a best child of an OR-node or all children of an AND-node are solved / not refinable), and otherwise true.[10]
- $\hat{z}.lb$ is a lower bound on solution cost, $\hat{z}.lb \leq c^*(p)$ computed from the current graph. If $\neg\hat{z}.\mathit{refinable}?$, then the lower bound is exact: $\hat{z}.lb = c^*(p)$. For an AND-node n, the lower bound is $g_{n.name}(x, y)$ where x and y are the lower bounds of the node's children. For an OR-node, it is the maximum of its current bound and the lower bound of its best child.[11]

When $\hat{z}.\mathit{refinable}? =$ false, ExtractSolution(n) can be used to recursively extract an optimal solution for $n.name$ by walking the graph, following $n'.\mathit{children}$ pointers at each OR-node n'. Finally, function UpdateSummary(n) simply replaces the summary of node n with its current summary, returning a flag indicating whether the summary's cost or refinable status was modified.

When a subproblem node is first created (as a leaf node, by MakeOrNode), its summary is initialized to $h(p)$ where h is a top-down heuristic function. If p is terminal ($\mathcal{Z}(p) \neq NT$),

[10] The interpretation of this flag will change slightly in Chapter 5.

[11] The maximum prevents information loss with inconsistent heuristics, to be discussed shortly.

then $h(p).\mathit{refinable}? = \mathsf{false}$. Otherwise, $h(p).\mathit{refinable}? = \mathit{true}$ with a heuristic lower cost bound $h(p).\mathit{lb}$. In both cases, $h(p).\mathit{children}$ is empty. As usual, we call $h(\cdot)$ *admissible* if it never overestimates the true cost:

Definition 2.23. *A top-down heuristic $h(\cdot)$ is* admissible *iff*

$$(\forall p)\ \ h(p).\mathit{lb} \leq c^*(p).$$

We could define a consistency property on $h(\cdot)$ as well. However, unlike in bottom-up search, we will see that consistency is not required for correctness of top-down search algorithms.

Definition 2.24. *A summary $\hat{z}$ of a node n is* correct *iff $\hat{z}.\mathit{lb} \leq c^*(n.\mathit{name})$ and $\neg\hat{z}.\mathit{refinable}?$ $\Rightarrow$ $\big((n.\mathit{summary}.\mathit{lb} = \infty$ and $\langle\!\langle n.\mathit{name}\rangle\!\rangle_* = \emptyset)$ or $\textsc{ExtractSolution}(n) \in \langle\!\langle n.\mathit{name}\rangle\!\rangle_*\big)$.*

Theorem 2.11. *Consider any graph initialized to* MakeRootNode(), *and modified by any sequence of* Expand *and* UpdateSummary *operations on its nodes, for any monotonic (but not necessarily superior) search problem and admissible heuristic $h(\cdot)$. Every summary in this graph is correct.*

Proof. First, note that by definition and admissibility of $h(\cdot)$, these properties hold for each leaf node constructed by MakeOrNode immediately after construction. Thus, they hold for the initial graph MakeRootNode().

Now, if these properties hold at every node in the graph, then they continue to hold at every node after calling Expand on any leaf node or UpdateSummary on any non-leaf node. Expand creates child connections that faithfully represent the decomposition structure of the original problem. Summaries of new nodes created by Expand and those updated by UpdateSummary must be valid by correctness of $h(\cdot)$, Theorem 2.8, Theorem 2.9, and the fact that the maximum of two valid lower bounds is also a valid lower bound. □

Corollary 2.12. *Any top-down search algorithm that begins with a graph initialized to* MakeRootNode(), *applies an arbitrary sequence of* Expand *and* UpdateSummary *operations, and then returns* ExtractSolution(root) *when $\neg\mathit{root}.\mathit{summary}.\mathit{refinable}?$ always returns optimal solutions (or fails to terminate).*

We conclude with a few more definitions, which will be of use in proving properties of top-down search algorithms.

Definition 2.25. *A node n is* consistent *iff it is a leaf node, or* UpdateSummary$(n) = \mathit{false}$.

In other words, a node is consistent iff its current summary correctly reflects the information available at itself and its immediate children in the graph. Next,

Definition 2.26. *The* active subgraph *for a node is the subgraph reachable from that node, retaining only edges from a node n to children in $n.summary.children$.*

Remark. *If $\neg n.summary.refinable?$, the active subgraph for that node corresponds to an optimal solution.*

Assumption 2.13. *A consistent, active subgraph for a node may contain cycles, even when a problem does not admit cyclic solutions. This can only occur when $h(\cdot)$ assigns a lower bound of 0 to a subproblem p with $c^*(p) > 0$. We henceforth assume that this is not the case.*

Given that forward and backward single-agent search are symmetric, the large differences between top-down and bottom-up AND/OR search may be surprising. As one might guess, the asymmetry arises due to AND-nodes. In particular, in a bottom-up search (or a forwards or backwards single-agent search), when a node is selected for expansion its optimal solution (in one direction) is known, and need not be revisited. In contrast, when a node is expanded in a top-down search we typically do not know its optimal context (or optimal solution).

2.2.3.2 AO*

AO* (Martelli and Montanari, 1973; Kumar et al., 1985) is a simple, optimal top-down search algorithm (see Algorithm 2.7). At each step, it selects an arbitrary refinable leaf in the active subgraph of the root to expand, and then updates the summaries of this node and its ancestors to restore consistency to the graph.[12] This process continues until the root is labeled unrefinable (see Figure 2.9).

By Corollary 2.12, we know that AO* is optimal when it terminates. Termination is guaranteed when the search space is free of cycles.

Lemma 2.14. *At the beginning of each iteration, every node in AO*'s current graph is consistent.*

Proof. The initial graph is consistent. Expanding a node n can render n itself, but no other nodes, inconsistent. UpdateAncestors restores consistency by propagating changes upwards in the graph, with each call to UpdateSummary(n') making n' consistent but possibly rendering the parents of n' inconsistent, until a set of nodes are reached whose summaries are unchanged and thus whose parents must remain consistent. □

Theorem 2.15. *In a search space without cycles, AO* is guaranteed to terminate with an optimal solution (or failure, if no solutions exist).*

[12]The version presented is technically an instance of B (Bagchi and Mahanti, 1983), because it gains efficiency by never propagating *decreases* in lower bounds that may arise due to inconsistencies in $h(\cdot)$.

Algorithm 2.7 AO*

```
function AO*()
    root ← MakeRootNode()
    while root.summary.refinable? do
        n ← ChooseLeaf(root)
        Expand(n)
        UpdateAncestors(n)
    return ExtractSolution(root)

function ChooseLeaf(n)
    while n.children ≠ leaf do
        n ← any c ∈ n.summary.children where c.summary.refinable?
    return n

function UpdateAncestors(n)
    if UpdateSummary(n) then
        for p ∈ n.parents do
            UpdateAncestors(p)
```

Proof. At the start of each iteration, every node is consistent, and $root.summary.refinable? = \mathsf{true}$. By the definition of consistency and lack of cycles, ChooseLeaf must find a refinable leaf node to Expand in finite time. By lack of cycles, UpdateAncestors must terminate. Because the set of all subproblems is finite and each subproblem can be Expanded at most once, the algorithm can only carry out a finite number of iterations, and must eventually terminate. □

If the search space contains cycles, however, UpdateAncestors and hence AO* can fail to terminate. Even when there are no cycles, a single iteration of the naive cost revision algorithm implemented by UpdateAncestors can take time exponential in the number of nodes in the graph. For example, Figure 2.10 shows a simple graph structure where each leaf expansion causes an exponential number of calls to UpdateAncestors, each of which increases the cost bound of a node by 1 (as if the tree corresponding to the graph was represented explicitly). In trees the worst case is a line graph, in which case each call to UpdateAncestors must update the entire graph, and thus the algorithm as a whole has cost $O(N^2)$ where N is the depth of the final solution (this worst case holds for any top-down search algorithm in the framework of Algorithm 2.5).

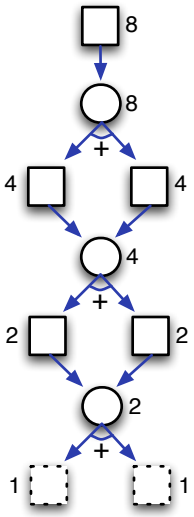

Figure 2.10: A worst-case graph for AO*'s cost revision operation (UpdateAncestors). Each layer of expansion doubles the cost bound at the root, but each call to UpdateSummary only increases a node's cost bound by 1. Thus, the total runtime for expanding a node at layer k is $O(2^k)$.

Algorithm 2.8 LDFS

```
function LDFS()
    root ← MakeRootNode()
    while root.summary.refinable? do LDFS-Pass(root)
    return ExtractSolution(root)

function LDFS-Pass(n)
    if n.children = leaf then
        Expand(n)
    else if n is consistent then
        LDFS-Pass(the first refinable node in n.summary.children)
    if ¬UpdateSummary(n) then LDFS-Pass(n)
```

2.2.3.3 LDFS

Learning depth-first search (LDFS, (Bonet and Geffner, 2005)), is another optimal top-down search algorithm. LDFS was originally described for explicit AND/OR graphs; Algorithm 2.8 shows a straightforward adaptation for our implicit setting.

Rather than searching all the way from the root to a leaf at each iteration, LDFS makes a series of passes over the graph. Each pass recursively explores the descendants of each node until it becomes solved, or its lower bound is increased. Because LDFS does not attempt to maintain consistency throughout the graph (just along a current path from the root), it can avoid updating summaries of nodes that are irrelevant to an optimal solution. Moreover, LDFS is optimal even in cyclic graphs, unlike AO* (which can fail to terminate).

Theorem 2.16. *(Bonet and Geffner, 2005) LDFS is guaranteed to terminate with an optimal solution when one exists. If not, it may fail to terminate.*

However, these advantages come at a price: LDFS is constrained to always explore the same (e.g., first) child at each AND-node until it becomes solved, and thus cannot balance its expansions between children of AND-nodes. Specifically, the best leaf to expand in an AND/OR graph is likely the leaf with *maximal uncertainty* among those in a current optimal subtree, because this will yield the maximum amount of information for a fixed amount of effort. If costs are divided roughly equally between the children of AND-nodes, this will often be the *shallowest* leaf eligible for expansion. In contrast, LDFS is constrained to expand a *deepest* leaf, which will often carry much less weight (see Figure 2.11).

Moreover, while LDFS is guaranteed to find optimal solutions even in the presence of cycles, in the worst case it may require $O(c^*(\mathcal{P}_0) * |\mathcal{P}|)$ calls to UpdateSummary to do so (where $|\mathcal{P}|$ is the total number of subproblems). For example, Figure 2.12 shows a graph where LDFS will perform poorly, requiring 99 summary updates of the root (with no further expansions) before optimality of the left branch is proven. This is preferable to the infinite loop that would ensue in AO*'s UpdateAncestors, but still far worse than the performance of a bottom-up algorithm such as KLD (whose runtime is independent of solution costs).

2.2.4 Hybrid Search Algorithms

As we have seen, bottom-up algorithms have a number of advantages compared to top-down algorithms, especially in search spaces with cycles. However, bottom-up algorithms are inappropriate for problems with a vast number of terminal subproblems, which will

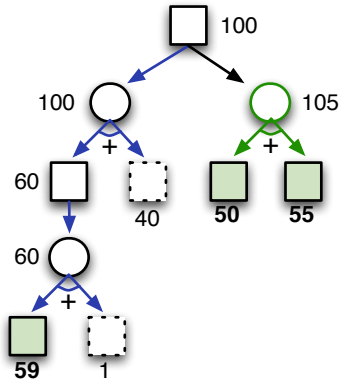

Figure 2.11: An AND/OR graph where LDFS's inability to balance refinements at AND-nodes may be inefficient. Expanding the leaf with heuristic bound 40 is likely to increase the cost bound of the left subtree (and thus prove optimality of the right subtree) much more quickly than expanding the leaf with bound 1 (e.g., assuming approximately relative heuristic error). However, because LDFS has already traversed the left branch, it cannot expand the leaf with bound 40 until the leaf with bound 1 has been solved (or proven to have cost at least 7, solving the problem).

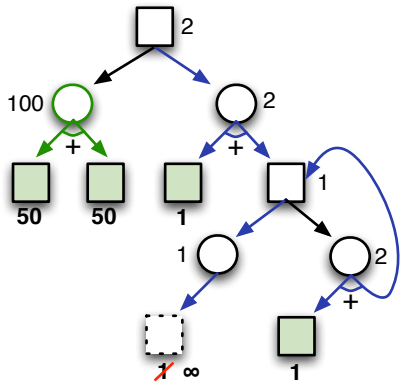

Figure 2.12: A cyclic AND/OR graph in which LDFS's cost revision strategy performs poorly. The dashed node has just been expanded, and found to be a dead-end (thus increasing its cost bound from 1 to ∞). After returning to the root to restore consistency along the path traversed (updating the root's bound to 3), LDFS requires 98 passes down the right branch, raising the cost bound of the root by 1 in each pass, to prove optimality of the left branch.

Algorithm 2.9 AO$*_{\text{KLD}}$ Cost Revision

```
function UpdateAncestors(n)
    activeBounds ← an empty map from active nodes to previous lower bounds
    fringe ← min-priority queue on n.summary.lb, breaking ties towards ¬n.summary.refinable?
    AddActiveAncestors(activeBounds, n)
    for (n, bound) ∈ activeBounds do
        n.summary ← CurrentSummary(n, bound)
        if n.summary.lb < ∞ then fringe.Insert(n)
    while fringe ≠ ∅ do
        n ← fringe.RemoveMin()
        activeBounds.Remove(n)
        for p ∈ n.parents do
            if p ∈ activeBounds then
                ẑ ← p.summary
                p.summary ← CurrentSummary(p, activeBounds[p])
                if ẑ ≠ p.summary then fringe.Insert(p) /* change priority if present */

function AddActiveAncestors(activeBounds, n)
    oldBound ← n.summary.lb
    if n ∉ activeBounds and UpdateSummary(n) then
        activeBounds[n] ← oldBound
        n.summary ← s⊥ /* s⊥.lb = ∞ */
        for p ∈ n.parents do
            AddActiveAncestors(activeBounds, p)
```

be our focus. Fortunately, it is easy to construct hybrid search algorithms that use an overall top-down search strategy like AO*, while gracefully handling cycles by replacing UpdateAncestors with a version of a bottom-up algorithm such as KLD.

Proposals for hybrid algorithms include $\text{CFC}_{\text{REV}*}$ (Jiménez and Torras, 2000), which is a very complex hybrid of relatives of AO* and KLD, and Loopy AO* (Hansen and Zilberstein, 2001), which combines AO* with Value or Policy Iteration and is designed for finding cyclic solutions. We present a very simple hybrid algorithm, a direct combination of AO* and KLD, which we call $\text{AO}*_{\text{KLD}}$ (see Algorithm 2.9). This algorithm replaces the recursive UpdateAncestors used by AO* with a version that uses dynamic programming to restore consistency in a constant number of passes through the set of ancestors that need updating.

This version of UpdateAncestors first calls AddActiveAncestors to identify the set of nodes whose summaries need to change and record their current lower cost bounds. If a node has not yet been added to the active set, and it is no longer consistent, it is added to the active set and its summary is set to the worst possible value (with a lower bound of ∞). This ensures that all stale information due to cycles is forgotten, and guarantees that all potentially affected parents will be made inconsistent (regardless of any potential heuristic inconsistencies).[13] Then, holding all current summaries outside the active set fixed, UpdateAncestors runs a variant of KLD to compute the correct summaries of nodes inside this set in a single pass. In this process, the recorded bounds in *activeBounds* are used to ensure that no information is forgotten.

The resulting algorithm explores the space in the same manner as AO*, while avoiding the pathologies of the former. In particular, it is guaranteed to terminate even in search spaces with cycles, and its worst-case runtime is roughly $O(N^2)$, the best asymptotically possible for any top-down search algorithm.

2.3 Hierarchical Planning

Hierarchical planning has a long history, beginning with state abstraction in GPS (Newell and Simon, 1972), macro-operators in STRIPS (Fikes et al., 1972), and hierarchical task networks in NOAH (Sacerdoti, 1975). This section introduces hierarchical planning via an overview of these historical approaches, then presents the precise hierarchical formalism upon which the remainder of the thesis is based.

For our purposes, the objective of hierarchical planning is to solve the the *classical* planning problems of Section 2.1 *faster* (the relevance of AND/OR search to this pursuit will become clear in later chapters). As in that section, we assume a search problem described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0, s_*)$, and our objective is to find an (optimal) solution $\mathbf{a} \in \mathcal{A}^*$, where

[13] It may be possible to improve on this step when the heuristic is consistent, or it can be guaranteed that a node does not depend on any stale information from affected cycles.

we now refer to actions in $\mathcal{A}$ as *primitive actions*.

In addition, hierarchical planning supplies us with a set of *high-level actions* (HLAs) that represent abstract, temporally extended behaviors. The basic idea is to use these HLAs to solve a problem at successively more detailed levels of abstraction, rather than searching directly for a primitive solution that reaches all the way from s_0 to s_*. At each planning step, a high-level plan is elaborated to take more details into account, by *expanding* some HLA into one of a number of possible *refinements* consisting of lower-level actions. Because a high-level plan necessarily omits some details, it may not actually admit an expansion into a primitive solution, and in such cases search must backtrack and try a different refinement at a higher level.

In a perfect world, we would avoid such backtracking by restricting our attention to *high-level solutions*, plans that are guaranteed to admit at least one expansion into a primitive solution. Moreover, we would be able to *decompose* the problem of finding a primitive expansion of this plan into separate subproblems, one per HLA in the plan. In this case, the complexity of search would be greatly reduced, to the sum of the complexities of the multiple searches, and not their product (Minsky, 1963). For instance, suppose that we are given a uniform hierarchy in which each HLA has r possible expansions into l (high-level) actions, and the primitive actions are found at depth d. In this best case, the total number of actions considered to find a solution of length l^d is just $O(r * l^d)$. In contrast, if the primitive branching factor is b, the cost of finding this solution without the hierarchy would be exponentially greater: $O(b^{l^d})$.

Of course, to achieve these exponential speedups, we require the ability to identify high-level solutions. Hierarchical planning systems with this ability are said to possess the *downward refinement property*, first defined by Bacchus and Yang (1991).

Definition 2.27. *A hierarchy possesses the* downward refinement property *if it can support a proof that a plan is a high-level solution (i.e., admits at least one expansion into a primitive solution).*

A planning algorithm can *commit* to a high-level solution, focusing solely on refinements of this solution, without sacrificing completeness. The converse of this property, first defined by Tenenberg (1988), is also of interest.

Definition 2.28. *A hierarchy possesses the* upward solution property *if it can support a proof that a plan is* not *a high-level solution (i.e., admits no expansions into primitive solutions).*

A planning algorithm can *prune* a high-level non-solution, ignoring this plan and all of its refinements, without sacrificing optimality.

Both of these properties can dramatically improve the efficiency of a planner, by ruling out unnecessary swaths of the search space. While some previous planners have possessed

the upward solution property, we will see that the downward refinement property has been much more difficult to achieve. Chapters 4 and 5 describe some of the first hierarchical planning algorithms that possess versions of both of these properties.

2.3.1 Historical Approaches

As mentioned above, a wide variety of approaches to hierarchical planning have been proposed over the past decades. These approaches all have in common the notion of high-level actions, but differ in the meaning of these HLAs, how abstract plans are constructed, and the overall search strategy for finding a primitive solution. We will divide the previous approaches into three broad classes.

First, *ABSTRIPS* HLAs are defined by *subgoals* that specify what must be achieved, but leave the question of how it should be achieved open to the planner. Second, *macros* are simple HLAs that admit a single primitive expansion. Finally, *hierarchical task networks* include aspects of both approaches, defining HLAs by a set of possible expansions into lower-level actions, typically accompanied by additional annotations that attempt to specify and/or constrain what the HLA achieves.

2.3.1.1 ABSTRIPS

HLAs based on *state abstraction* are defined by sets of *subgoals* to be achieved at a particular point in a high-level plan. This approach was pioneered in the GPS system (Newell and Simon, 1972), fully automated in the ABSTRIPS system (Sacerdoti, 1974), and subsequently formalized and analyzed by Knoblock et al. (1991) and others.

The basic idea in this approach is to assign each proposition in a planning problem a *criticality* level in $1, \ldots, L$. A sequence of primitive actions is said to be *correct* at level l if it is a solution to the simpler planning problem where all preconditions at level $< l$ are completely ignored (technically, a homomorphic abstraction of the original problem (Korf, 1980)). Then, planning begins with the empty plan, which is correct at level $L + 1$. At each search step, an attempt is made to extend a correct plan at level $l + 1$ to a correct plan at level l, by interspersing actions to satisfy the new preconditions at level l. Sometimes this will not be possible, and search will have to backtrack and try a different plan at a higher level. This process continues until a correct plan is found at level 1, which by definition solves the original planning problem.

Intuitively, the criticality levels should roughly reflect how difficult a given proposition is to achieve. Constant and key goal propositions should be assigned the maximum level L, and unimportant propositions that are easy to achieve in any situation should be assigned lower criticality. For instance, Sacerdoti (1974) supposed that when turning on a lamp, the fact that the object in question is actually a lamp is most critical, followed by that he is in

the same room as the lamp, then that the lamp is plugged in, and finally that he is next to it. ABSTRIPS included a technique for automatically inferring these levels from a planning domain, which was subsequently improved upon, most notably by Knoblock (1990).

The efficiency of this approach depends on two key properties: (1) how much easier it is to expand a plan at level $l+1$ to a plan at level l than to find a plan at level l directly, and (2) how much backtracking is required. If the abstract plan does not sufficiently constrain the planning process, or too much backtracking is needed, it will be faster to just perform ordinary state-space planning (i.e., just search for a plan at level 1 directly). Various methods have been proposed for constraining the expansion process and thus improving the first ratio (e.g., (Knoblock et al., 1991)); however, these techniques may increase the amount of backtracking, so there is no guarantee that they will improve performance.

With this fact in mind, Bacchus and Yang (1991) set out to understand when backtracking across levels could be avoided. The result of this investigation was a semantic definition of the downward refinement property (DRP, Definition 2.27), along with some checkable conditions that could be used to guarantee the DRP in a particular ABSTRIPS hierarchy. For example, the DRP holds if each action only has preconditions and effects in a single level, or if the state spaces for literals $\leq l$ are completely connected for any value of literals $> l$ (for all l).[14] Unfortunately, these conditions are quite restrictive, e.g., the first condition means that the overall problem consists of L completely separate, non-interacting planning problems.

Further study has not yet discovered general classes of ABSTRIPS hierarchies that satisfy the DRP (and in fact, this may be a fruitless pursuit (Giunchiglia and Walsh, 1993)). Nevertheless, as long as the DRP "almost" holds and backtracking is minimized, ABSTRIPS-style planning can be an effective hierarchical planning method, either alone or in combination with other methods.

2.3.1.2 Macros

A macro is an HLA with a single allowed expansion into a particular sequence of primitive actions. Macros first appeared in the STRIPS system (Fikes et al., 1972) as generalizations of portions of previous solutions that could be applied to solve new planning problems.

A significant advantage of macros is that they behave exactly like primitive actions. Given STRIPS descriptions for a macro-action sequence, it is trivial to construct an exact STRIPS description of the macro-operator, by just collecting the preconditions and effects not provided or neutralized by other actions in the sequence. Thus, it is easy to check if an abstract plan containing macro HLAs is a "high-level" solution.

Unfortunately, because macros are (by definition) inflexible, it is typically difficult or impossible to find a compact set of powerful macros that suffice to solve all of the problems in

[14] These properties are related to the independent and serializable subgoals studied by Korf (1987).

a given planning domain. Of course, completeness and optimality can always be retained by *augmenting* (rather than replacing) the primitive actions in a domain with macros (Russell and Norvig, 2009). While this technique has been used with some success in satisficing planning (Botea et al., 2005), it can only hurt an optimal planner since the set of states that must be examined can be no smaller than in the original instance.

Thus, macros have found greater utility as tools for solving a single problem instance within specialized solution procedures. For instance, Korf (1985) showed that policies for "serially decomposable" problems such as Rubik's cube can be represented as small macro tables that can be applied with no search at all. Closer to this work, Jonsson (2009) demonstrated a technique that caches solutions to subgoals within a particular planning problem as macros, enabling certain exponential-length solutions (such as in the Towers of Hanoi) to be discovered and represented in polynomial time.

2.3.1.3 Hierarchical Task Networks

In hierarchical task networks (HTNs), upon which this thesis is based, each HLA is defined by a set of allowed *refinements* that specify different methods for accomplishing a task. Each refinement consists of a *sequence* of (high-level) actions, or more generally, a partially ordered network of actions.

Unlike macros, HTNs allow for choices in how each HLA should be implemented. This greater flexibility makes it much easier to design general-purpose hierarchies that can be successfully applied across a wide range of problem instances in a domain.

Unlike ABSTRIPS, HTNs can encode *procedural* knowledge about *how* a task should be accomplished. This has a number of benefits for real applications, including efficiency, easy encoding of domain constraints, and easily interpretable, hierarchically structured solutions.

In addition to the procedural knowledge encoded by the structure of the hierarchy, each HLA in an HTN is typically annotated with additional knowledge to help guide search towards a solution. A wide variety of such annotations have been proposed, including information about when an HLA should be applied (i.e., preconditions), what it accomplishes (i.e., effects), preferences on its refinements, and so on.

The first HTN planner was NOAH (Sacerdoti, 1975), which has been succeeded by a plethora of other planners including NONLIN (Tate, 1977), SIPE (Wilkins, 1983), SIPE-2 (Wilkins, 1990), O-PLAN (Drummond and Currie, 1989), UMCP (Erol et al., 1994b), DPOCL (Young et al., 1994), AbNLP (Fox and Long, 1995), SHOP (Nau et al., 1999), and SHOP2 (Nau et al., 2001). All of these planners use HTNs to more efficiently find solutions to classical planning problems (or generalizations thereof). They differ primarily in the ways in which plans and refinements are represented, the HLA annotations and how they are used, and overall search strategies. Rather than discuss each individual planner in detail,

this section examines the similarities between approaches and the dimensions along which they differ.

First, similarities. In all cases, a planner is supplied with a primitive (e.g., STRIPS) planning problem, along with a (typically declarative) description of a hierarchy. The hierarchy is specified as a set of HLA schemata, each of which names a type of task that may be encountered in the domain. As in STRIPS action schemata, each HLA schema may take a set of arguments, which name objects in the domain to operate upon (e.g. $\textsc{MoveTo}(a, b)$). Each HLA is associated with a set of refinement schemata, each of which corresponds to a particular method of accomplishing the task, and may take arguments in addition to those provided to the HLA (e.g., $[\textsc{PickFrom}(a, c), \textsc{PlaceOn}(a, b)]$). The refinements may be specified as sequences of actions (high-level or primitive), but are typically partially ordered networks of actions, as in partial-order planning.

Much as in partial-order planning, HTN planners are typically satisficing, and based on depth-first search through the space of abstract, lifted partial-order plans (which may contain unbound variables). A notable exception is SHOP (Simple Hierarchical Ordered Planner, (Nau et al., 1999)), which conducts a depth-first search over fully ordered plans. Each search step may *refine* an HLA (expanding it out into its possible methods), or perform other operations such as the addition of an action, ordering constraint, causal link, or variable binding constraint. Such elaboration of a given plan continues until a primitive solution is found, or an inconsistency in the plan is detected and the search backtracks. Optimizing or even cost-sensitive search strategies are typically not considered: the closest we have seen are heuristics encoded in the hierarchy that attempt to explore refinements in order of expected cost, combined with a depth-first branch-and-bound search (Nau et al., 2003).

To reason about high-level plans and the correctness of HTN planning algorithms, a semantics for abstract plans is needed. Erol et al. (1994b) proposed such a semantics, associating the meaning of each a high-level plan with the set of all fully ordered, grounded, primitive plans that could be generated by repeated elaborations of the above sort.

In a subsequent paper, Erol et al. (1996) examined the complexity of HTN planning. Surprisingly, they proved that general HTN planning is *undecidable*, even in a finite state space. Planning can be made decidable by disallowing cyclic hierarchies (in which an HLA can be generated by one of its refinements), or requiring refinements to be fully ordered action sequences. Erol et al. (1996) also proved that when high-level plans are grounded and totally ordered, the setting for this thesis, planning is PSPACE-hard (in EXPTIME).

The first dimension of variation between approaches concerns the overall search strategy. While most planners use the same basic depth-first search strategy and set of plan elaborations (HLA refinement, causal link establishment, ordering constraint, binding constraint), they differ in their choices of which to apply when. In addition to these elaborations, some planners (e.g., (Young et al., 1994; Estlin et al., 1997; Kambhampati et al., 1998; Gerevini et al., 2008)) also allow actions to be directly added via means-end analysis (not just from

refinement of existing HLAs). Finally, many planners have also included "critics", more or less arbitrary procedures that can modify a plan after elaboration to take into account constraints, resource limitations, domain-specific guidance, and so on. (Erol et al., 1995).

Among these planners, SHOP and SHOP2 choose a notable strategy: they always choose to refine a *first* HLA in their current plan. As a consequence, when an HLA is refined the precise state of the world it will be executed from is known, and HLA specifications can include arbitrary procedures that generate, prune, and heuristically order their sets of refinements based on this state.

Another planner of note is UCPOP+PARSE (Barrett and Weld, 1994), which turns the usual approach upside-down. Rather than generating plans top-down, it combines ordinary state-space planning with a procedure that prunes primitive prefixes that are not consistent with an HTN-style hierarchy.

The second dimension of variation involves the additional HLA annotations, which are typically used to prune undesirable plans from the search space (with some partial exceptions described in Chapter 7). These annotations are necessary because as described thus far, HLAs are just implicit representations of sets of primitive plans. All of the information that allows us to conclude anything about *what* a plan achieves — whether it works or fails, its necessary ordering constraints, causal links, and so on — lives only at the bottom level in the descriptions of the primitive actions. HLA annotations purport to carry this information upwards in the hierarchy, and enable informed decisions about which high-level plans can be pruned, without first expanding them into ground primitive plans.

Aside from the SHOP family of planners just discussed, these annotations are typically *declarative* specifications, much like STRIPS descriptions. A wide variety of such annotation types have been proposed, including:

- *Supervised conditions*: abstract causal links between actions in a refinement
- *Unsupervised conditions*: ordinary preconditions to be achieved by actions external to an HLA
- *Usewhen conditions*: preconditions that must hold to do an HLA, but which the planner should not actually try to achieve
- *High-level effects*: effects that may be asserted as direct consequences of doing an HLA (regardless of the refinement chosen, potentially sacrificing soundness), or treated as constraints on the planning process (e.g., the refinement chosen must include a primitive that causes the effect)
- *External conditions* (Tsuneto et al., 1998): automatically derived preconditions that are required by *every* refinement of an HLA

Erol et al. (1994b) analyzed these proposals in some detail. They can be broadly divided into three classes: (1) those that capture constraints already imposed by the structure of the hierarchy (e.g., external conditions) and have no effect on the hierarchy semantics; (2) those that add additional constraints, and may disallow primitive solutions that would otherwise be allowed by the hierarchy (e.g., supervised, unsupervised, and usewhen conditions, and the second type of high-level effect); and (3) those that add additional solutions, and compromise soundness with respect to the primitive domain (e.g., the first type of high-level effect).

The first and third classes of conditions seem uncontroversially beneficial and detrimental, respectively. As for the second class, to the extent that the constraints imposed are seen as *part of*, rather than *descriptions of*, the structure of the hierarchy, they do not compromise the correctness of hierarchical search (and may be able to speed up search significantly). However, the way in which these constraints are used often lacks a well-defined semantics. For instance, the well-known Gift of the Magi problem (Russell and Norvig, 2009) illustrates that an apparently inconsistent high-level plan can sometimes be expanded to a consistent primitive solution by a combination of refinement and interleaving. On the other hand, proposals with a well-defined semantics have been so constraining that they rule out hierarchies that can effectively decompose a problem, and thus have not found much use in practice (Yang, 1990).

In summary, while many hierarchical planners use precondition–effect annotations for HLAs, their exact meaning is often unclear. After the collapse of the "hierarchical" track at the first International Planning Competition, McDermott (2000) wrote as follows:

> *The semantics of hierarchical planning have never been clarified . . . the hierarchical planning community is used to thinking of library plans as advice structures, which was a drastic departure from our assumption that the basic content of the plan library contained no "advice", only "physics". . . . The problem is that no one has ever figured out how to reconcile the semantics of hierarchical plans with the semantics of primitive actions.*

2.3.2 This Thesis: Simplified Hierarchical Task Networks

In this thesis, we adopt a simplified version of the hierarchical task network (HTN) formalism described in the previous section, assuming that plans are *fully ordered*, *grounded* sequences of (high-level) actions. We also make an number of minor, mostly cosmetic simplifications to ease theoretical analysis and algorithm design.

Without a doubt, these simplifications reduce the expressiveness of a hierarchy (as illustrated by the complexity results reported above). However, by doing so they enable a number of theoretical and practical tools for reasoning about and improving search through

hierarchical plan space.[15] These tools are sufficiently powerful that we are able to, unlike in previous works, focus on *hierarchically optimal* planning algorithms that are guaranteed to find a least-cost solution among those generated by the hierarchy. The remainder of this section formally defines our simplified HTN formalism, and presents an example hierarchy and simple search algorithm for concreteness. Then, the next three chapters introduce these tools and a variety of novel, hierarchically optimal search algorithms built upon them.

In particular, Chapter 3 describes two caching methods that capitalize on hierarchical structure, and presents hierarchically optimal search algorithms that exploit this caching for efficient search without the need for any HLA annotations. Then, Chapter 4 introduces and analyzes a novel "angelic semantics" that assigns *correct* precondition-effect annotations to HLAs, which can be used to prune, commit to, and bound the costs of high-level plans without first reducing them to primitive action sequences (and thus satisfies both the downward refinement and upward solution properties). Finally, Chapter 5 presents hierarchically optimal planning algorithms that use these annotations, including several that leverage the caching techniques of Chapter 3 by incorporating ideas from AND/OR graph search (Section 2.2).

2.3.2.1 Formalism

We assume a classical planning problem $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0, s_*)$, as described in Section 2.1.1. On top of this problem, we assume we are given an *action hierarchy*.

Definition 2.29. *An* action hierarchy *consists of:*

- $\hat{\mathcal{A}}$*, a set of* high-level actions *(HLAs),*
- $\mathcal{H}_0 \in \hat{\mathcal{A}}$*, a designated* top-level action*, and*
- $\mathcal{I}(h)$*, a function that specifies the set of* immediate refinements *of HLA* h*, each of which is a sequence of exactly two actions* $\mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^2$.

The reader no doubt notices similarities to the AND/OR search framework presented in Section 2.2.1.1; we proceed with some analogous definitions here, deferring a full exploration of the connections between the frameworks to Chapter 5.

Remark. *This formalism assumes that every refinement consists of* exactly two *actions. Any hierarchy can be converted to this form, by factoring longer refinements and padding shorter refinements with* NO-OP *primitives.*[16]

[15] As discussed in Chapter 8, we hope that these techniques will eventually be generalized to apply in a variety of more expressive settings.

[16] We assume that we can add NO-OP primitives to the primitive domain, since they do not change the set

Assumption 2.17. *We assume that the set of HLAs $\hat{\mathcal{A}}$ is finite, and each HLA has a finite set of refinements.*

In what follows, we let $\mathbf{a} + \!\!+ \, \mathbf{b}$ denote the concatenation of action sequences $\mathbf{a}$ and $\mathbf{b}$, and a_1 and $\mathbf{a}_{2:|\mathbf{a}|}$ denote the first and remaining actions (respectively) of sequence $\mathbf{a}$.

Definition 2.30. *A high-level plan $\mathbf{a}$ can be* refined *at position i iff $a_i \in \hat{\mathcal{A}}$, yielding refinements $\mathcal{I}_i(\mathbf{a})$ wherein a_i is replaced by one of its immediate refinements:*

$$\mathcal{I}_i(\mathbf{a}) := \{\mathbf{a}_{1:i-1} + \!\!+ \, \mathbf{b} + \!\!+ \, \mathbf{a}_{i+1:|\mathbf{a}|} : \mathbf{b} \in \mathcal{I}(a_i)\}.$$

Then, a *primitive refinement* of a high-level action or plan is a primitive action sequence reachable by repeated refinement operations.

Definition 2.31. *The* primitive refinements $\mathcal{I}^*(\mathbf{a})$ *of an action sequence $\mathbf{a}$ consist of all primitive action sequences in $\mathcal{A}^*$ reachable by repeated refinement operations:*

$$\mathcal{I}^*(\mathbf{a}) = \begin{cases} \{[a]\} & \text{if } \mathbf{a} = [a] \text{ and } a \in \mathcal{A} \\ \bigcup_{\mathbf{r} \in \mathcal{I}(a)} \mathcal{I}^*(\mathbf{r}) & \text{if } \mathbf{a} = [a] \text{ and } a \in \hat{\mathcal{A}} \\ \{\mathbf{r}_{a_1} + \!\!+ \, \mathbf{r}_{\mathbf{a}_{2:|\mathbf{a}|}} : \mathbf{r}_{a_1} \in \mathcal{I}^*([a_1]) \text{ and } \mathbf{r}_{\mathbf{a}_{2:|\mathbf{a}|}} \in \mathcal{I}^*(\mathbf{a}_{2:|\mathbf{a}|})\} & \text{if } |\mathbf{a}| > 1. \end{cases}$$

We will also occasionally be concerned with the set of all plans (primitive or not) reachable by repeated refinement operations:

Definition 2.32. *The* transitive refinements $\mathcal{I}^+(\mathbf{a})$ *of an action sequence $\mathbf{a}$ consist of all action sequences (primitive or not) reachable by one or more refinement operations from $\mathbf{a}$.*

Our objective will typically be to find a *hierarchically optimal* solution, a primitive solution that reaches s_* from s_0 with minimal cost among all primitive refinements of $\mathcal{H}_0$. To simplify our formal definition of hierarchical optimality, we first make another cosmetic simplification:

Assumption 2.18. *Every primitive refinement $\mathbf{a} \in \mathcal{I}^*(\mathcal{H}_0)$ applicable in s_0 is a solution:*

$$(\forall \mathbf{a} \in \mathcal{I}^*(\mathcal{H}_0))\ \mathcal{T}(s_0, \mathbf{a}) \in \{s_\perp, s_*\},$$

or, equivalently,

$$(\forall \mathbf{a} \in \mathcal{I}^*(\mathcal{H}_0))\ \mathcal{C}(s_0, \mathbf{a}) < \infty \Leftrightarrow \mathcal{T}(s_0, \mathbf{a}) = s_*,$$

of basic solutions. In light of this translation, we sometimes ignore the binary restriction when it leads to a simpler presentation. Note that NO-OP primitives add trivial zero-cost cycles to the state space; we thus replace Assumption 2.2 with related restrictions below.

This might seem extraordinarily restrictive, but in fact it is without loss of generality: given any hierarchy, we can simply add a new top-level action $\mathcal{H}'_0$ with a single refinement $[\mathcal{H}_0, \text{NO-OP}^{s_*}]$, where NO-OP^{s_*} is a zero-cost primitive action with precondition that the current state is the goal state, and no effects.[17] With this simplification, hierarchical optimality can be defined as follows.

Definition 2.33. *A hierarchically optimal solution* $\mathbf{a}^*$ *is a primitive refinement of* $\mathcal{H}_0$ *with minimal cost from* s_0*:*

$$\mathbf{a}^* \in \arg\min_{\mathbf{a} \in \mathcal{I}^*([\mathcal{H}_0])} \mathcal{C}(s_0, \mathbf{a})$$

Remark. *Because the hierarchy may constrain the set of allowed primitive action sequences, the cost of a hierarchically optimal solution must be greater than or equal to the cost of the cheapest primitive solution* $c^*(s_0)$*.*

We accept this potential optimality gap, in exchange for the significant efficiency gains that can be provided by the hierarchy.

For now, we do not consider any annotations on the HLAs in a hierarchy, including high-level preconditions or effects. However, we note that the former can be simulated in our framework, by prefixing the refinements of an HLA with a NO-OP^c primitive with precondition c as described above.

Finally, we sometimes impose additional constraints on the types of cycles that can appear in the hierarchy.

Definition 2.34. *A hierarchy is* recursive *iff there exists an HLA* h *and a transitive refinement of* h *that contains* h*:*

$$(\exists h \in \hat{\mathcal{A}}, \mathbf{a} \in \mathcal{I}^+([h]))\ h \in \mathbf{a}.$$

Definition 2.35. *A hierarchy is* forward cycle-free *(FCF) if there does not exist a state* s*, HLA* h*, and transitive refinement* $\mathbf{a}$ *of* h *that contains* h*, immediately preceded by a primitive prefix that cycles back to* s *from* s*:*

$$\neg(\exists s \in \mathcal{S}, h \in \hat{\mathcal{A}}, \mathbf{a} \in \mathcal{I}^+([h]), i \in [1, |\mathbf{a}|])\ \mathcal{T}(s, \mathbf{a}_{1:i-1}) = s \text{ and } h = \mathbf{a}_i.$$

As we will see in the next section, recursion is a powerful property that allows for natural encodings of goal-directed HLAs for tasks such as navigation. While such HLAs can introduce forward cycles, they are often not problematic for search algorithms; as we saw in Section 2.2, the real issue is with cycles that do not make progress or increase the cost of a candidate plan, as these can lead to cyclic solutions or finite-cost, infinite-length plans. The precise definition of a problematic cycle can differ based on the setting. For now, we consider hierarchical search problems that are *forward, zero-cost cycle-free* (FZCF).

[17] This transformation also simplifies hierarchical search algorithms, removing the need to explicitly consider a goal state/criterion.

Definition 2.36. *A hierarchy is* forward, zero-cost cycle-free *(FZCF) if there does not exist a state s, HLA h, and transitive refinement $\mathbf{a}$ of h that contains h, immediately preceded by a primitive prefix that cycles back to s from s with zero cost:*

$$\neg(\exists s \in \mathcal{S}, h \in \hat{\mathcal{A}}, \mathbf{a} \in \mathcal{I}^{+}([h]), i \in [1, |\mathbf{a}|])\ \mathcal{T}(s, \mathbf{a}_{1:i-1}) = s \text{ and } \mathcal{C}(s, \mathbf{a}_{1:i-1}) = 0 \text{ and } h = \mathbf{a}_i.$$

Remark. *Many natural hierarchies seem to be forward, zero-cost cycle-free. In particular, if the state space is free of zero-cost cycles, a hierarchy is forward zero-cost cycle-free as long as it does not contain* left-recursive *HLAs that can appear as the first action in their transitive refinements.*

Right-recursive HLAs do not violate this property, as they introduce at least one positive-cost primitive action. In fact, the next section describes two FZCF hierarchies that include right-recursive HLAs.

2.3.2.2 Example Hierarchies

This section presents three example hierarchies.

First, the following trivial hierarchy generates a standard forward search space (see Section 2.1), and will be useful when analyzing the properties of hierarchical search algorithms.

Definition 2.37. *The* "flat" hierarchy *consists of a single HLA $\mathcal{H}_0$, which has recursive refinements $[a, \mathcal{H}_0]$ for each primitive action $a \in \mathcal{A}$, plus one additional refinement* $[\text{NO-OP}^{s_*}]$.

Theorem 2.19. *Primitive refinements of $\mathcal{H}_0$ applicable in s_0 are in one-to-one correspondence with solutions to the original planning problem with the same costs.*

Proof. Trivial. □

Next, we discuss a hierarchy for the simple nav-switch domain of Section 2.1.2.1, which has three HLA types. First, a HLA $\text{NAV}(x_{from}, y_{from}, x_{to}, y_{to})$ navigates from (x_{from}, y_{from}) to (x_{to}, y_{to}) without flipping the switch. $\text{NAV}(x_{from}, y_{from}, x_{to}, y_{to})$ has nine refinements: $\left[\text{NO-OP}^{\mathsf{AtX}(x_{to}) \wedge \mathsf{AtY}(y_{to})}\right]$ for when the agent is at the destination (x_{to}, y_{to}), and eight other refinements of the form $[\text{LEFTH}(x_{from}, x_{from-1}), \text{NAV}(x_{from-1}, y_{from}, x_{to}, y_{to})]$ that make a single move followed by a recursive NAV. Second, the top-level HLA $\text{GO}(x_{from}, y_{from}, x_{to}, y_{to})$ goes to (x_{to}, y_{to}) from (x_{from}, y_{from}), perhaps flipping the switch at one or more locations along the way. $\text{GO}(x_{from}, y_{from}, x_{to}, y_{to})$ has two types of refinements: either a direct $[\text{NAV}(x_{from}, y_{from}, x_{to}, y_{to}), \text{GOAL}]$ without flipping, or sequence $[\text{NAV}(x_{from}, y_{from}, x_s, y_s)$, $\text{FLIPH/V}(x_s, y_s)$, $\text{GO}(x_s, y_s, x_{to}, y_{to})]$, which navigates to a location (x_s, y_s) from which the switch can be flipped, flips it, and then recursively goes to the destination.

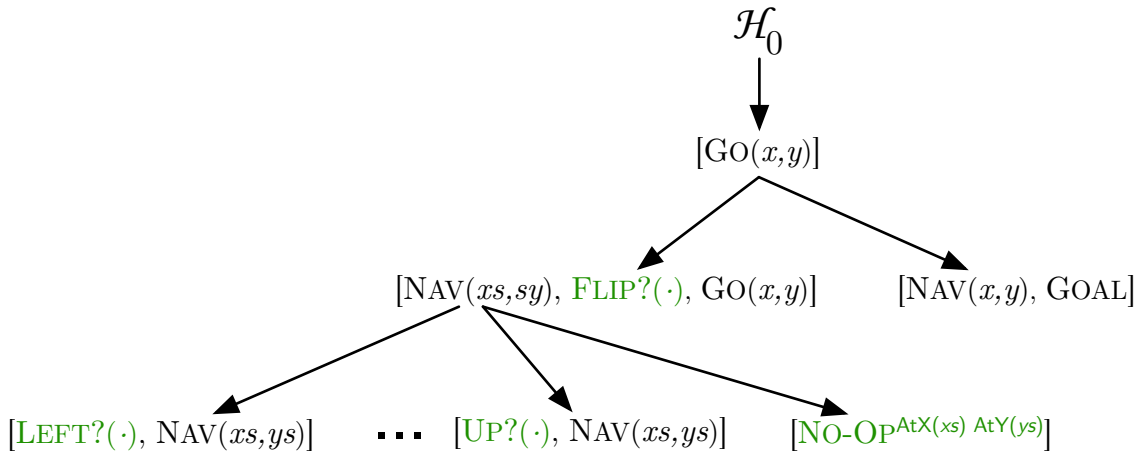

Figure 2.13: A hierarchy for the nav-switch domain (omitting arguments corresponding to the current position).

HLA	**Refinements**
$\mathcal{H}_0$	$[\text{GoGrasp}(o), \text{GoDrop}(o), \mathcal{H}_0]$ for $o : \mathsf{CanMove}(o)$ or $[\text{no-op}^{s*}]$
$\text{GoGrasp}(o)$	$[\text{MoveBase}(x_b, y_b), \text{Grasp}(o)]$ for (x_b, y_b) in reach of o
$\text{GoDrop}(o)$	$[\text{GoDropAt}(o, x, y)]$ for $(x, y) \in o_G$
$\text{GoDropAt}(o, x, y)$	$[\text{MoveBase}(x_b, y_b), \text{DropAt}(o, x, y)]$ for (x_b, y_b) in reach of (x, y)
$\text{Grasp}(o)$	$[\text{Reach}(dx, dy), \text{Get?}(\cdot)]$ where (dx, dy) next to o
$\text{DropAt}(o, x, y)$	$[\text{Reach}(dx, dy), \text{Put?}(\cdot)]$ where (dx, dy) next to (x, y)
$\text{Reach}(dx, dy)$	$[\text{Gripper?}(\cdot), \text{Reach}(dx, dy)]$ or $\left[\text{no-op}^{\mathsf{GripperOffset}=(dx,dy)}\right]$
$\text{MoveBase}(x, y)$	$[\text{Reach}(0, 0), \text{Unpark}, \text{Nav}(x, y), \text{Park}]$ or $\left[\text{no-op}^{\mathsf{BaseAt}=(x,y)}\right]$
$\text{Nav}(x, y)$	$[\text{Base?}(\cdot), \text{Nav}(x, y)]$ or $\left[\text{no-op}^{\mathsf{BaseAt}=(x,y)}\right]$

Figure 2.14: A hierarchy for the discrete manipulation domain. (See Figure 1.2 for a graphical depiction.)

Finally, we describe a realistic hierarchy for the discrete manipulation domain of Section 2.1.2.3, which will serve as a running example throughout the thesis. (See Figure 2.14, and the graphical depiction in Figure 1.2.) To solve the problem ($\mathcal{H}_0$), we GoGrasp(o) an object o, GoDrop(o) it at its goal, then recursively $\mathcal{H}_0$ (or stop, if all objects are delivered). GoGrasp(o) navigates to a base position within reach of o with MoveBase, then grasps it using Grasp(o), which refines to a Reach followed by a primitive Get?($\cdot$) (from some direction). Similarly, to drop an object o at its goal with GoDrop(o), we first select a goal position (x, y) to GoDropAt(o, x, y) it at. Then, GoDropAt(o, x, y) refines to a MoveBase to get the base in range followed by DropAt(o, x, y), which positions the gripper with Reach and then places the object with primitive Put?(). Finally, Reach and Nav recursively refine into sequences of primitive Gripper?($\cdot$) and Base?($\cdot$) actions that move the gripper or base by one square in a given direction. This hierarchy structures the solution space, but leaves all of the details mentioned above open for optimization.

Practical techniques for programmatically generating the set of refinements of an HLA (applicable from a given state) given the structure of a hierarchy are discussed in Section 4.3.

2.3.2.3 Simple Hierarchically Optimal Search

This section describes a very simple hierarchically optimal search algorithm, both for concreteness and as a starting point for the more sophisticated algorithms to come. In particular, it defines a *state space* for hierarchical search, to which a standard algorithm like uniform-cost search (i.e., Dijkstra's algorithm) can be applied to find a hierarchically optimal solution. This state-space representation is similar to approaches used by several previous planners (Barrett and Weld, 1994; Kuter et al., 2005).

The "hstates" in this state space are tuples $(s, \mathbf{a})$ consisting of a state $s \in \mathcal{S}$ from the original problem, and a sequence of remaining (possibly high-level) actions $\mathbf{a}$ to do from s. The initial hstate is $(s_0, [\mathcal{H}_0])$, and the goal hstate is $(s_*, [])$. Finally, the successor hstates from a reachable non-goal hstate $(s, \mathbf{a})$ (with non-empty $\mathbf{a}$) are:

- $(\mathcal{T}(s, a_1), \mathbf{a}_{2:|\mathbf{a}|})$ with cost $\mathcal{C}(s, a_1)$, if $a_1 \in \mathcal{A}$ is primitive.
- $(s, \mathbf{b} +\!\!+ \mathbf{a}_{2:|\mathbf{a}|})$ with cost 0 for each $\mathbf{b} \in \mathcal{I}(a_1)$, if $a_1 \in \hat{\mathcal{A}}$ is high-level

In other words, if the first action is primitive we apply it to the state portion of the hstate, and otherwise we consider all refinements of the remaining sequence at this first action. Figure 2.15 shows some example hstates encountered along an optimal solution to the discrete manipulation problem instance of Figure 1.1. Optimal state-space search algorithms applied to this formulation are hierarchically optimal, for FZCF hierarchies.

Figure 2.15: The first nine "hstates" encountered along an optimal solution for the discrete manipulation instance of Figure 1.1. Each hstate consists of a state (the relevant fragment of which is shown at left) paired with a sequence of actions remaining to do from this state. Each hstate is labeled with the total cost incurred to reach it, shown at left.

Lemma 2.20. *For any $s \in \mathcal{S}$ and $h \in \hat{\mathcal{A}}$, only a finite number of hstates are reachable from $(s, [h])$ with zero cost in a FZCF hierarchy.*

Proof. Let $N = |\mathcal{S}| \times |\hat{\mathcal{A}} \cup \mathcal{A}| + 2$. There are only finitely many successor chains of length N beginning with $(s, [h])$. By the pigeonhole principle, each such chain must include two hstates $(s', \mathbf{a}')$ and $(s'', \mathbf{a}'')$ where $s' = s''$ and $a'_1 = a''_1$. By Definition 2.36, these hstates are not reachable from one another with zero cost. □

Lemma 2.21. *For any $s \in \mathcal{S}$, $h \in \hat{\mathcal{A}}$, and finite cost $c \in [0, \infty)$, only a finite number of hstates are reachable from $(s, [h])$ with cost $\leq c$ in a FZCF hierarchy.*

Proof. Let Z be the finite set of zero-cost states reachable from $(s, [h])$ by Lemma 2.20. Each element of Z can have only finitely many successors, each of which is in Z or has cost ≥ 1. Thus, only finitely many states have cost < 1. Recursively applying this logic $c + 1$ times establishes that only finitely many hstates are reachable with cost $\leq c$. □

Theorem 2.22. *Hierarchical uniform-cost search (H-UCS), which is just uniform-cost search applied to this state space, is* hierarchically optimal *for FZCF hierarchies.*

Proof. This state space generates the primitive refinements of Definition 2.31 in a straightforward, left-to-right manner, with pruning for inapplicable prefixes. Because every prefix of an applicable primitive refinement must be applicable, and because every hstate reached while generating a primitive refinement has cost no greater than the final primitive refinement, if search terminates it must terminate with a hierarchically optimal solution. Finally, by Lemma 2.21, the algorithm must terminate as long as a finite-cost solution exists. □

We note that H-UCS may be faster or slower on this problem than UCS on the original state space (depending on how many plans the hierarchy rules out, versus how many different suffixes each state may appear with). While unlikely to be very useful in practice, H-UCS serves as a useful starting point for algorithms to come.

Chapter 3

Decomposed Hierarchical Planning

This chapter introduces a novel family of algorithms that search for hierarchically optimal solutions in the formalism of Section 2.3.2.1, without any additional annotations on the HLAs involved. These algorithms improve on the simple H-UCS algorithm of Section 2.3.2.3 by exploiting the structure of hierarchical plan space with two complementary techniques.

The first technique is *decomposition*, which divides the task of finding solutions for a high-level plan $\mathbf{a}$ into separate subproblems, one per action in the plan. More precisely, decomposition exploits the fact that every primitive refinement of $\mathbf{a} = [h_1, h_2]$ that optimally reaches state s'' from s is a concatenation of a primitive refinement of h_1 that optimally reaches some intermediate state s' from s with a primitive refinement of h_2 that optimally reaches s'' from s' (see Figure 3.1). The advantage is that, e.g., the optimal primitive refinements of h_1 from s can be computed and *cached* the first time they are needed, and then reused throughout the search space each time this subproblem reappears. However, decomposed caching alone is not too powerful, since optimal search must still explore all states reachable under the hierarchy (while incurring no more than the optimal cost).

The second technique, *state abstraction*, builds upon decomposition, greatly increasing the opportunities for caching and sharing of solutions. The basic observation is that typically, only a portion of the state is *relevant* for doing a given HLA. For example, when planning for a robot base movement $\textsc{Nav}(x, y)$, only the current position of the robot base is relevant; similarly, when planning for a gripper movement $\textsc{Reach}(x, y)$, only the robot base and gripper positions and the positions of nearby objects are relevant. Subproblems that differ only in irrelevant parts of the state share the same optimal primitive refinements, and thus state abstraction can be incorporated into a decomposed planning algorithm by simply ignoring irrelevant parts of the state when identifying repeated subproblems. Chapter 7 describes relationships to related definitions of state abstraction used in previous work in hierarchical reinforcement learning and planning.

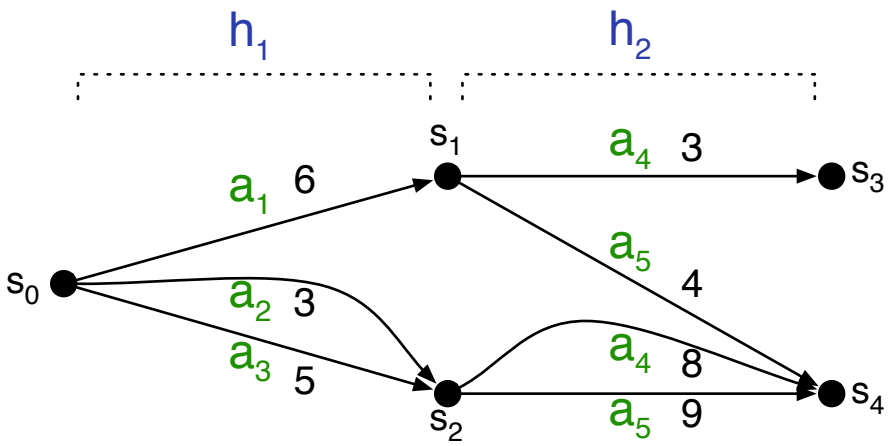

Figure 3.1: A schematic of a high-level plan $[h_1, h_2]$, where h_1 has primitive refinements $[a_1]$, $[a_2]$ and $[a_3]$, and h_2 has primitive refinements $[a_4]$ and $[a_5]$. Plan $[h_1, h_2]$ can reach states s_3 and s_4 from s_0, with optimal solutions $[a_1, a_4]$ and $[a_1, a_5]$ (respectively). Note that while $[a_2]$ is the cheapest primitive refinement from h_1 (and the optimal solution to reach state s_2 via h_1), a_2 does not appear in any optimal primitive refinement of $[h_1, h_2]$ because the refinements of h_2 are more expensive from intermediate state s_2 than from s_1.

These techniques are formalized in the next section. Then, the following section presents a simple exhaustive search algorithm that utilizes decomposition and state-abstracted caching. Finally, the last section presents a pair of cost-ordered algorithms (like H-UCS) that exploit decomposition and state abstraction without exploring the entire hierarchical search space, and thus may be more useful in practice.

3.1 Decomposition, Caching, and State Abstraction

3.1.1 Decomposition and Caching

The basic idea behind decomposition is to recursively divide the hierarchical search space into *subproblems*, each of which is concerned with the outcomes of a *single* (high-level) action, and then patch together the solutions to these subproblems. The advantage over algorithms that search directly through the space of high-level plans (like H-UCS) is that a given subproblem only needs to be solved a single time, and then the discovered solutions can be reused whenever this subproblem is encountered again. The resulting decomposed search space is analogous to the AND/OR graphs of Section 2.2, where an OR-node represents a single-HLA subproblem and an AND-node represents a sequential subproblem for the actions in a particular refinement of some HLA. However, decomposed hierarchical search cannot

be captured directly in the formalism of Section 2.2.1.1, because each subproblem may have *multiple* types of optimal solutions corresponding to different reachable states, and these types are not known *a priori* but must be discovered through search (see Figure 3.1).

Specifically, this chapter considers a simple class of *forward subproblems*, which are essentially equivalent to the "hstates" of Section 2.3.2.3.

Definition 3.1. *A* forward subproblem *p is a tuple $(s, \mathbf{a})$, which corresponds to doing (some primitive refinement of) sequence $\mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*$ from state $s \in \mathcal{S}$. For subproblem $p = (s, \mathbf{a})$, we define $p.state := s$ and $p.actions := \mathbf{a}$.*

As in Section 2.3.2.3, the objective is to solve the root subproblem $(s_0, [\mathcal{H}_0])$. Solutions to subproblem $(s, \mathbf{a})$ correspond to primitive refinements of $\mathbf{a}$ that are applicable from s.

Definition 3.2. *The* solutions *$z \in \langle\!\langle (s, \mathbf{a}) \rangle\!\rangle$ to forward subproblem $(s, \mathbf{a})$ consist of all primitive action sequences $z.seq \in \mathcal{I}^*(\mathbf{a})$ that are applicable from s, along with their costs $c(z) := \mathcal{C}(s, z.seq)$ and resulting states $z.outcome := \mathcal{T}(s, z.seq)$.*[1]

We also define a concatenation operation on solutions, and a set of trivial (empty) solutions:

Definition 3.3. *$\top_s$ is the empty solution for state s: $\langle\!\langle (s, []) \rangle\!\rangle = \{\top_s\}$. The concatenation operation $z_1 +\!\!+\ z_2$ produces a new solution by concatenating the actions of z_1 and z_2, adding their costs, and taking the outcome state of z_2. Concatenation is only defined when $z_1.outcome = p_2.state$, where $z_2 \in \langle\!\langle p_2 \rangle\!\rangle$.*

Finally, we define a *refinement* operator, which generalizes the successor function of Section 2.3.2.3 by adding bookkeeping for solutions.

Definition 3.4. *The* forward refinement *operator $\mathcal{R}(p, z)$ takes a nonterminal forward subproblem $p = (s, \mathbf{a})$ with $\mathbf{a} \neq []$ and a solution with $z.outcome = s$, and produces a list of pairs of next forward subproblems and corresponding solutions to reach their initial states, by applying the first action of $\mathbf{a}$ to s if primitive or refining it if high-level:*

$$\mathcal{R}((s, \mathbf{a}), z) = \begin{cases} \emptyset & \text{if } a_1 \in \mathcal{A} \text{ and } a_1 \notin \mathcal{A}_s \\ \{((\mathcal{T}(s, a_1), \mathbf{a}_{2:|\mathbf{a}|}),\ z +\!\!+ a_1)\} & \text{if } a_1 \in \mathcal{A}_s \\ \{((s, \mathbf{b} +\!\!+ \mathbf{a}_{2:|\mathbf{a}|}), z) : \mathbf{b} \in \mathcal{I}(a_1)\} & \text{if } a_1 \in \hat{\mathcal{A}}, \end{cases}$$

where $z +\!\!+ a$ extends solution z with primitive action a by appending a to $z.seq$, adding $\mathcal{C}(z.outcome, a)$ to $c(z)$, and updating $z.outcome$ to $\mathcal{T}(z.outcome, a)$.

[1] While the cost and outcome state are redundant given the action sequence, they are included in the solution so that they can be retrieved in constant time (independent of the length of the action sequence). In some problems (e.g., Towers of Hanoi), this can make an exponential difference in runtime.

For example, suppose that solution z has $z.outcome = s$ and $z.seq = [a_1, a_2]$, HLA h has immediate refinements $[a_3, h]$ and $[a_4, a_5]$, and a_3 leads to state s' from s whereas a_4 is inapplicable from s. Then, $\mathcal{R}((s, [h, a_6]), z) = \{((s, [a_3, h, a_6]), z), ((s, [a_4, a_5, a_6]), z))\}$, and these pairs have further refinements $\mathcal{R}((s, [a_3, h, a_6]), z) = \{((s', [h, a_6]), z')\}$ and $\mathcal{R}((s, [a_4, a_5, a_6]), z) = \{\}$, where $z'.outcome = s'$ and $z'.seq = [a_1, a_2, a_3]$.

This refinement operator can be used to generate solutions by repeated application, yielding the same search space used by H-UCS.

Theorem 3.1. *The solutions $\langle\langle p \rangle\rangle$ for forward subproblem $p = (s, \mathbf{a})$ can be computed by:*

$$\langle\langle p \rangle\rangle = \begin{cases} \{\top_s\} & \text{if } \mathbf{a} = [] \\ \{z_1 +\!\!+ z_2 : (p', z_1) \in \mathcal{R}(p, \top_s) \text{ and } z_2 \in \langle\langle p' \rangle\rangle\} & \text{otherwise.} \end{cases}$$

Proof. Same as the proof of Theorem 2.22. □

However, solutions can also be generated via decomposition, by exploiting the following equivalence.

Theorem 3.2. *If $\mathbf{a}$ is nonempty with first action a_1 and remaining actions $\mathbf{a}_{2:|\mathbf{a}|}$, then the solution set for forward subproblem $p = (s, \mathbf{a})$ satisfies:*

$$\langle\langle p \rangle\rangle = \{z_1 +\!\!+ z_2 : z_1 \in \langle\langle (s, [a_1]) \rangle\rangle \text{ and } z_2 \in \langle\langle (z_1.outcome, \mathbf{a}_{2:|\mathbf{a}|}) \rangle\rangle\}.$$

Proof. Follows directly from Definition 2.31. □

In other words, the solutions for an action sequence subproblem can be generated by first solving a subproblem for its first action, and then solving additional subproblems for its remaining actions, one for each reachable intermediate state (see Figure 3.2). Because solutions with different outcome states cannot be directly compared without taking the context of the subproblem into account (e.g., a cheap solution leading to a bad state might be worse than an expensive solution leading to a good state, see Figure 3.3), the definition of optimal solutions in this setting is more complex than in the formalisms of the previous chapter.

Definition 3.5. *The* optimal solutions $\langle\langle p \rangle\rangle_*$ *for a forward subproblem p consist of the minimum-cost solutions to reach each state:*

$$\langle\langle p \rangle\rangle_* := \{z : z \in \langle\langle p \rangle\rangle \text{ and } c(z) = c^*(p, z.outcome)\},$$

where $c^*(p, s)$ is the minimum cost of any solution for p that reaches state s:

$$c^*(p, s) := \min_{z \in \langle\langle p \rangle\rangle : z.outcome = s} c(z).$$

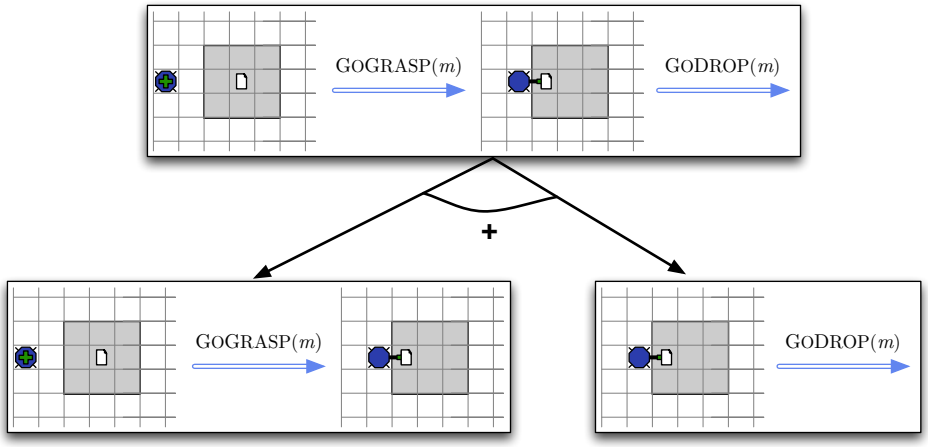

Figure 3.2: Given a specific intermediate state for action sequence [GoGrasp(m), GoDrop(m)], the task of finding an optimal solution decomposes into separate, independent subproblems: one to reach this intermediate state from s_0 using GoGrasp(m), and one to follow with GoDrop(m).

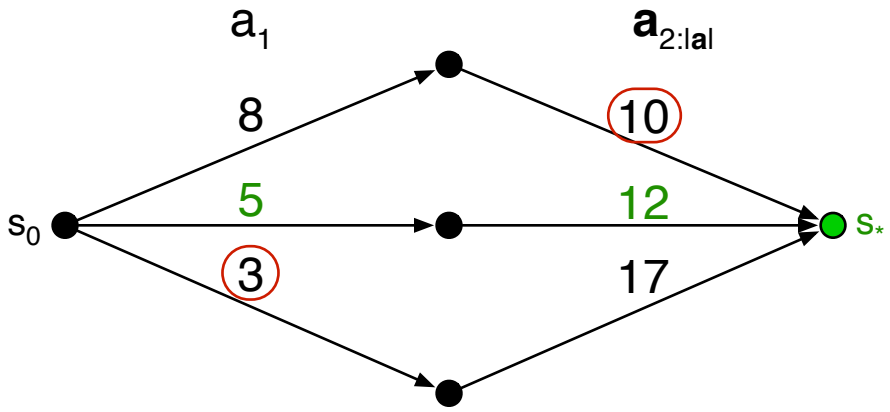

Figure 3.3: The bottom-left sequence is the cheapest refinement of a_1, and the top-right sequence is the cheapest refinement of $\mathbf{a}_{2:|\mathbf{a}|}$. However, the overall optimal solution (with total cost 17) passes through the center state, and does not include either of these refinements.

Hierarchically optimal search algorithms need only consider a *minimal* subset of $\langle\langle p \rangle\rangle_*$ containing a *single* optimal solution per reachable state (see Figure 3.4).

3.1.2 State Abstraction

Within this formalism, state abstraction amounts to the observation that if s_1 and s_2 only differ in ways *irrelevant* to action a, then forward subproblems $(s_1, [a])$ and $(s_2, [a])$ have the same sets of optimal solutions. In particular, suppose we are given a function $\textsc{EnterContext}(s, a)$ that captures this notion of relevance.

Definition 3.6. *Let* $s' = \textsc{EnterContext}(s, a)$ *capture the* context *of state* s *relevant for doing action* a*. The result is a new partial state* s' *defined on a subset of the variables of* s*, with the same values given by* s *(i.e., a projection of* s *onto the variables relevant to* a*). This function must satisfy the following condition: if* $s' = \textsc{EnterContext}(s_1, a) = \textsc{EnterContext}(s_2, a)$*, then* $(\forall \mathbf{b} \in \mathcal{I}^*(a))\ \ \mathcal{C}(s_1, \mathbf{b}) = \mathcal{C}(s_2, \mathbf{b})$ and *no action in* $\mathbf{b}$ *affects a variable not defined in* s'*.*[2]

For a primitive action a, the relevant variables are just those mentioned in its precondition or effect: $\mathcal{V}_a^{pre} \cup \mathcal{V}_a^{post}$. Then, the variables relevant to an HLA are the variables relevant to any primitive action appearing in one of its primitive refinements. This notion of relevance can be automatically computed, starting at the primitive actions and moving up the hierarchy towards $\mathcal{H}_0$ (to which all variables in s_0 are typically relevant).

Since a partial state generated by $\textsc{EnterContext}(s, a)$ is in effect a complete state for the subproblem of doing a, we will henceforth use the term "state" to refer to these partial states as well as complete states for the overall problem. We also assume that the transition and cost functions, refinement operator, and other definitions above have been extended in the obvious way to work on partial states.

Given a known (optimal) solution set for a subproblem $p = (s, a)$, it is trivial to construct an (optimal) solution set for another subproblem $p' = (s', a)$ with $\textsc{EnterContext}(s, a) = \textsc{EnterContext}(s', a)$.

Definition 3.7. *Define a* contextualization *operation* $\textsc{Contextualize}(s, s')$*, which produces a state with all variable values from* s'*, along with any remaining variable values from* s *that were not defined in* s'*. Furthermore, extend the solution concatenation operation so that* $(z +\!\!+ z').outcome = \textsc{Contextualize}(z.outcome, z'.outcome)$*.*

[2]This condition is sufficient for primitive actions as considered here that lack conditional effects, since when $\mathbf{b}$ is applicable, its effects on s_1 and s_2 are guaranteed to be the same. With conditional effects, we would also have to require that the final values for variables defined in s' are the same in both cases.

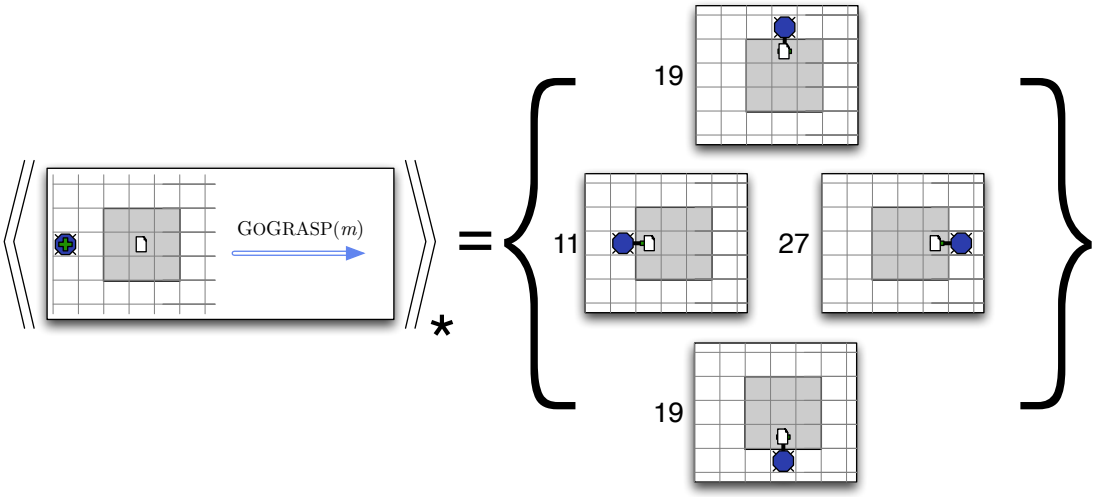

Figure 3.4: Reachable states and optimal costs for the forward subproblem of GoGrasping the magazine from the initial state of Figure 1.1. For instance, the sole optimal solution to reach the left state is [NO-OP$^{\mathsf{GripperOffset}=(0,0)}$, UNPARK, BASER$(1,7)$, NO-OP$^{\mathsf{BaseAt}=(2,7)}$, PARK, GRIPPERR$(2,7,0,0)$, NO-OP$^{\mathsf{GripperOffset}=(1,0)}$, GETR$(2,7,1,0,\mathsf{m})$].

Given a solution z_1 with $z_1.outcome = s$, a subproblem $p = (\text{ENTERCONTEXT}(s,a),a)$, and a solution $z_2 \in \langle\!\langle p \rangle\!\rangle$, this contextualization corresponds to combining the variables relevant to a from $z_2.outcome$ with the remaining (irrelevant) variables from $z_1.outcome$, which remain unchanged by any primitive refinement of a (see Figures 3.5 and 3.6).

Theorem 3.3. *Consider a subproblem $p = (s,a)$, and let Z be the optimal solutions for the abstracted version of this subproblem: $Z = \langle\!\langle(\text{ENTERCONTEXT}(s,a),a)\rangle\!\rangle_*$. Then,*

$$\langle\!\langle p \rangle\!\rangle_* = \{\top_s +\!\!+\ z : z \in Z\}.$$

Proof. Follows directly from Definitions 3.6 and 3.7. □

The effectiveness of state abstraction as defined here can depend heavily on the particular encoding of the problem state. For instance, the $\mathsf{Free}(x,y)$ state variables for our discrete manipulation example are essentially redundant given $\mathsf{ObjectAt}(o)$ for each object o. However, they can dramatically increase the potential for state abstraction. For example, GoGrasp(o) only needs to depend on the values of $\mathsf{Free}(x,y)$ for squares (x,y) neighboring $\mathsf{ObjectAt}(o)$, whereas without $\mathsf{Free}(\cdot)$ it would need to depend on $\mathsf{ObjectAt}(o')$ for each other object o'.

More general versions of this idea are possible. For instance, one can restrict the context to include only precondition (but not effect) variables, at the cost of a somewhat more in-

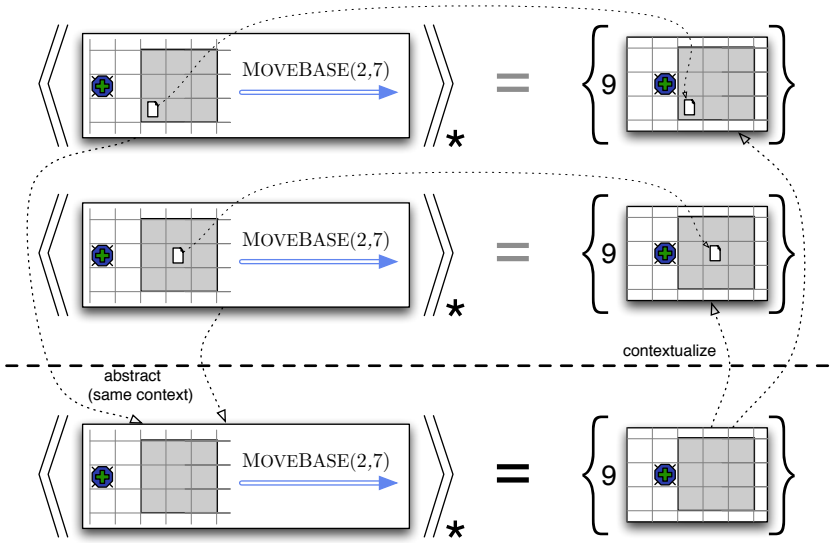

Figure 3.5: Solution sharing between MoveBase$(2, 7)$ subproblems for different states with the same context EnterContext$(\cdot, a)$, and thus the same optimal solutions. Top left: Two MoveBase$(2, 7)$ subproblems where the magazine is at different locations. Bottom left: a common abstraction of the subproblems, where state s is replaced with EnterContext$(s,$ MoveBase$(2, 7))$, which omits the irrelevant detail of the magazine position. Bottom right: the optimal solution set is directly computed for the abstracted subproblem. Top right: the solution sets for both original subproblems are generated by contextualizing this solution set, adding back the original magazine position.

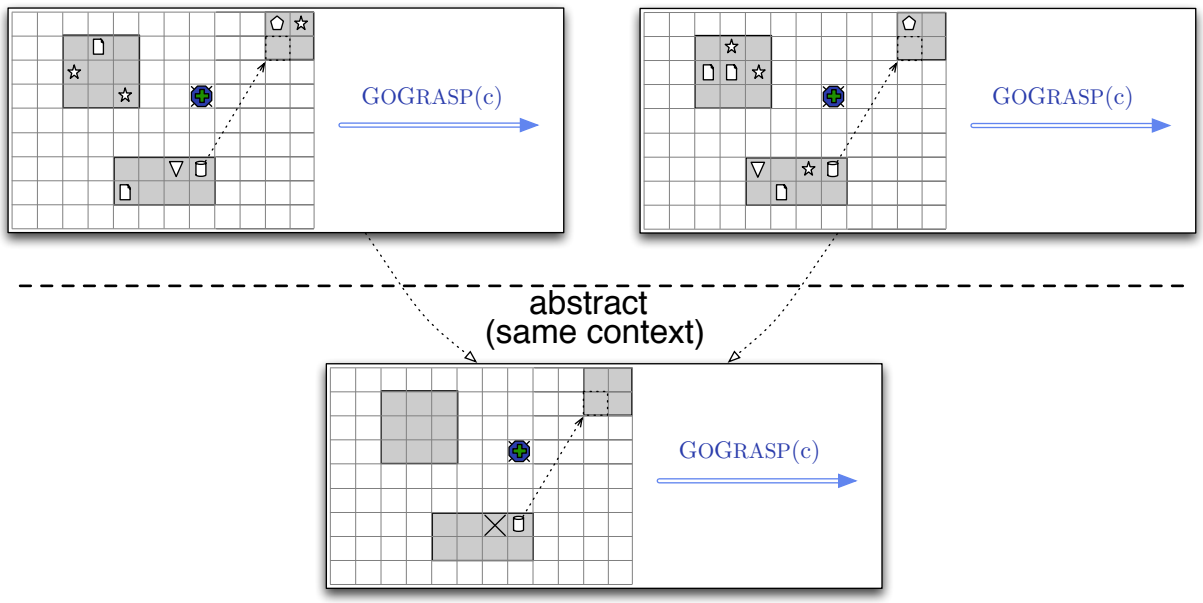

Figure 3.6: A more interesting example of state abstraction. Top: the planner encounters two subproblems of grasping the cup from quite different states. Bottom: these two states share the same state-abstracted context, which includes only the configuration of the robot, cup, and which squares neighboring the cup are occupied by some object (the identity of which is irrelevant to the grasp).

Algorithm 3.1 SAHTN

function SOLUTIONS-A((s, a))
  $p \leftarrow (\text{ENTERCONTEXT}(s, a), a)$
  if $cache[p] = \text{undefined}$ **then**
    $cache[p] \leftarrow \bigsqcup_{(p', z') \in \mathcal{R}(p, \top_{p.state})} \text{SOLUTIONS-P}(z', p'.actions)$
  return $cache[p]$

function SOLUTIONS-P($z, \mathbf{a}$)
  $sols \leftarrow \{z\}$
  for i from $1, ..., |\mathbf{a}|$ **do**
    $sols \leftarrow \bigsqcup_{z \in sols} \bigsqcup_{z' \in \text{SOLUTIONS-A}((z.outcome, a_i))} z +\!\!+ z'$
  return $sols$

function SAHTN()
  return the sole element of SOLUTIONS-A($(s_0, \mathcal{H}_0)$), or failure if empty

volved contextualization operation and more complex equality checks for states. Or, one can abstract away the identities of exchangeable objects, so that, e.g., the results of navigation planning can be shared between identical robots.

3.2 Exhaustive Decomposed Search

Algorithm 3.1 shows pseudocode for a simple, *exhaustive*, hierarchically optimal search algorithm called SAHTN (State-Abstracted HTN). This algorithm explores the space of all subproblems reachable by decomposition and refinement from $(s_0, \mathcal{H}_0)$ up to equivalence determined by ENTERCONTEXT$(\cdot, \cdot)$, by directly implementing Theorems 3.1 and 3.2.

SAHTN consists of two mutually recursive functions. SOLUTIONS-A computes the optimal solutions for a single-action subproblem $p = (s, a)$ and caches them under the pair $(\text{ENTERCONTEXT}(s, a), a)$, by applying one step of the refinement operation $\mathcal{R}$ and then unioning the results of SOLUTIONS-P on each refinement. The results are collected with a statewise-minimum $\bigsqcup$ operator, which retains a single minimum-cost solution for each reachable state, and discards the remaining solutions. Then, SOLUTIONS-P computes the solutions for an action *sequence* by applying the decomposition enabled by Theorem 3.2. Finally, the top-level SAHTN procedure simply returns the sole result of SOLUTIONS-A$((s_0, \mathcal{H}_0))$.[3]

Theorem 3.4. *In* forward cycle-free *hierarchies, SAHTN is* hierarchically optimal.

[3]Recall Definition 2.1 and Assumption 2.18, which guarantee (without loss of generality) that the search problem has at most a single goal state s_*, and no primitive refinement of $\mathcal{H}_0$ reaches a state other than s_* from s_0.

Proof. The optimality of SAHTN follows directly from Theorems 3.1, 3.2, and 3.3. □

Remark. *In hierarchies with forward cycles (Definition 2.35), SAHTN fails to terminate.*

It is easy to construct domains in which SAHTN is *exponentially faster* than H-UCS or primitive search algorithms like UCS. For example, in the Towers of Hanoi domain, SAHTN can find exponential-length optimal solutions in polynomial time as long as $z_1 +\!\!+ z_2$ concatenates sequences lazily (e.g., building a graph that compactly describes the final solution).

While it is possible to extend SAHTN to handle problems with cycles, we instead move directly to the cost-ordered algorithms of the next section. In addition to gracefully handling (non-zero-cost) cycles, these algorithms explore only the portion of the hierarchy necessary to find an optimal solution, and can thus be much more efficient than SAHTN.

3.3 Cost-Ordered Decomposed Search

This section presents two related algorithms for *cost-ordered* hierarchically optimal search with decomposition and state abstraction. In other words, these algorithms perform the same basic computations as SAHTN, but use a (set of) priority queues to examine a subproblem reachable with minimum cost from s_0 at each search step (like H-UCS). The primary advantage is that the first discovered solution must be hierarchically optimal, and thus the remainder of the search space need not be explored.

We first describe a *top-down* algorithm called DSH-LDFS (Decomposed, State-abstracted, Hierarchical LDFS), whose overall search strategy is similar to LDFS (Section 2.2.3.3). This algorithm maintains a graph of OR-nodes, each of which represents a uniform-cost search for optimal solutions to a single-HLA subproblem. Summary information and solutions are propagated upwards in the graph, where they create subsequent subproblems at higher levels when more actions remain (via Theorem 3.2).

DSH-LDFS is hierarchically optimal in FZCF hierarchies, but its performance can suffer due to the inherent inefficiencies of top-down search (especially in the presence of cycles). We present it primarily to help explain a second algorithm called DSH-UCS (Decomposed, State-abstracted H-UCS) that performs the same basic computations as DSH-LDFS more efficiently using a single global priority queue.

3.3.1 DSH-LDFS: Recursive Top-Down Search

Algorithm 3.2 shows pseudocode for DSH-LDFS, a recursive, top-down, cost-ordered, hierarchically optimal search algorithm. Search is carried out in a graph of OR-nodes, each

Algorithm 3.2 DSH-LDFS

```
function MakeOrNode()
    return a node n with
        n.queue = [], a priority queue on (z, a) or (z, child) tuples, ordered by
                    c(z) + child.queue.PeekMin() (or 0, breaking ties towards (z, a))
        n.closed = []
        n.solutions = MakeChannel()

function GetOrNode(queue, z, a)
    s ← EnterContext(z.outcome, a_1)
    if cache[(s, a_1)] = undefined then
        cache[(s, a_1)] ← MakeOrNode()
        for (p', z') ∈ R((s, a_1), ⊤s) do
            cache[(s, a_1)].queue.Insert((z', p'.actions))
    Subscribe(cache[(s, a_1)].solutions, λz' queue.Insert((z ++ z', a_{2:|a|})))
    return cache[(s, a_1)]

function DSH-LDFS-Pass(n, c)
    while n.queue.PeekMin() ≤ c do
        entry ← n.queue.RemoveMin()
        if entry = (z, a) then
            if n.closed[(z.outcome, a)] = undefined then
                n.closed.Insert((z.outcome, a))
                if a = [] then
                    Publish(n.solutions, z)
                    break
                else
                    n.queue.Insert((z, GetOrNode(n.queue, z, a)))
        else /* entry = (z, child) */
            DSH-LDFS-Pass(child, c − c(z))
            n.queue.Insert((z, child))

function DSH-LDFS()
    sols ← []
    root ← GetOrNode(sols, ⊤s_0, [H_0])
    while root.queue.PeekMin() < ∞ do
        if sols is not empty then return sols[0][0]
        DSH-LDFS-Pass(root, root.queue.PeekMin())
    return ⊥
```

of which represents a partially completed search for the optimal solutions of a particular single-action forward subproblem (s, a).

The search strategy of DSH-LDFS is a variant of learning depth-first search (see Section 2.2.3.3). Each iteration of the top-level DSH-LDFS loop initiates a DSH-LDFS-Pass that recursively explores the subgraph rooted at a node, until its lower bound is increased or a new optimal solution (for a previously unseen output state) is generated. Throughout this process, each OR-node n corresponding to forward subproblem $p = (s, a)$ maintains three pieces of state about its current search.

First, $n.queue$ is a priority queue on tuples t representing subsets of refinements of p, ordered by a lower bound on the cost of the next optimal solution for a that might be produced from t. The first element of each tuple is a solution z to reach $z.outcome$ from s by a primitive refinement of a prefix of an immediate refinement of a (or simply $[a]$ if a is primitive). Then, the second element can be either an action sequence $\mathbf{a}$ or an OR-node *child*. If the former, the tuple is *unexplored*, and sequence $\mathbf{a}$ specifies the actions remaining to do from $z.outcome$, e.g., $(z.seq +\!\!+ \mathbf{a}) \in \mathcal{I}^+(a)$. If the latter, the tuple is *in progress*, and *child* is an OR-node corresponding to an in-progress search for optimal solutions to a *next* subproblem from $z.outcome$. The queue is ordered by the lower cost bound of each tuple, which is $c(z)$ plus the current bound of *child* for in-progress tuples, breaking ties towards unexplored tuples (which helps surface solutions faster).

Next, $n.closed$ is a closed list representing the set of unexplored tuples that have already been processed (i.e., have generated a corresponding in-progress tuple). The first time an unexplored tuple $(z, \mathbf{a})$ is popped from the queue, z is guaranteed to be an optimal solution to reach $z.outcome$ from s by a primitive refinement of a prefix of an immediate refinement of a (and $\mathbf{a}$ is the remaining suffix of the immediate refinement). Because this solution is optimal, future instances of this state and remaining action sequence can be ignored.

Finally, $n.solutions$ is a communication *channel*, a data structure used throughout this thesis when a growing set of operations must be carried out on a growing set of data. A new datum d can be *published* to a channel c using Publish(c, d). Similarly, interested parties can *subscribe* to a channel c using Subscribe(c, f), where f is a *callback* function. The channel ensures that $f(d)$ is called once for each datum d and callback f, regardless of the order in which the publications and subscriptions are made. In this case, each datum published on $n.solutions$ is an *optimal solution* for a different outcome state of the subproblem represented by n, and each callback subscribed to $n.solutions$ represents a different parent context in which the subproblem corresponding to n has appeared.

The search logic of DSH-LDFS is captured by DSH-LDFS-Pass(n, c), which recursively explores OR-node n and its descendants until a new optimal solution for n is found and published, or the lower cost bound of n is increased above c. Each iteration of the main loop pops the best tuple off the queue and operates on it as follows.

- If the entry $(z, \mathbf{a})$ is unexplored and not on the closed list:
  - If it has no remaining actions ($\mathbf{a} = []$), z represents an optimal primitive refinement of a that reaches a new state $z.outcome$, and this optimal solution is published.
  - Otherwise $\mathbf{a} \neq []$, and the OR-node corresponding to doing the first action in $\mathbf{a}$ from $z.outcome$ is retrieved from the cache (or created if needed). A subscription to its solution channel is registered, which inserts an unexplored entry corresponding to doing the remaining actions in $\mathbf{a}$ from each solution state, and the child itself is added to the queue.
- Otherwise, the entry is in progress $(z, child)$, and search recursively descends into *child* with the cost $c(z)$ to reach *child* within this subproblem subtracted from the current bound c. When the recursive call completes, *child* is added back to the queue, either with greater cost than before, or having published a new solution (generating a new queue entry at n, and perhaps other nodes as well, via subscription callbacks).

Intuitively, DSH-LDFS is hierarchically optimal because it implicitly considers the same sets of plans as H-UCS, and just explores them in a more efficient manner by exposing and then exploiting the existence of repeated subproblems. In particular, if GetOrNode is modified to return a fresh OR-node on each call without caching the results, DSH-LDFS (very nearly) degrades to a baroque implementation of H-UCS.[4]

Theorem 3.5. *DSH-LDFS is* hierarchically optimal *in FZCF hierarchies.*

Proof. First, note that the cost bound $n.queue$.PeekMin() of an OR-node n can never decrease. Thus, when DSH-LDFS-Pass(n, c) is called, it must always be the case that $n.queue$.PeekMin() $\geq c$. Now, in an FCZF hierarchy, if DSH-LDFS-Pass(n, c) is on the call stack, a recursive call on the same node DSH-LDFS-Pass(n, c') must have $c' < c$. Thus, this recursive call must terminate immediately, and we need not worry about leaving nodes in an inconsistent state during recursive calls. Moreover, the total number of single-action forward subproblems $N \leq |\mathcal{S}||\hat{\mathcal{A}} \cup \mathcal{A}|$ is finite, so we need not worry about infinitely deep recursive chains.

Next, we prove the invariant that, before each iteration of the loop in DSH-LDFS-Pass, for every OR-node n representing a subproblem $p = (s, [a])$, and every primitive solution $z \in \langle\langle p \rangle\rangle$, the priority queue of n (plus published solutions, and a *child* entry that would be added back after an in-progress recursive call completes) contains *some* entry that admits a primitive solution $z' \in \langle\langle p \rangle\rangle$ where $z'.outcome = z.outcome$ and $c(z') \leq c(z)$, and whose current cost bound is $\leq c(z')$.

[4]Specifically, this modified DSH-LDFS would perform the same computations as H-UCS (up to tiebreaking), except that it would eliminate fewer repeated "hstates" from the search space.

This invariant is trivially satisfied by the initial node GetOrNode$(\cdot, s_0, [\mathcal{H}_0])$, which contains one queue entry for each immediate refinement of $\mathcal{H}_0$ from s_0. Now, we just need to show that each iteration of the loop preserves the invariant. There are several cases.

First, if an unexplored entry $(z, \mathbf{a})$ is discarded due to an item on the closed list corresponding to a previous entry $(z', \mathbf{a})$ with $z'.outcome = z.outcome$, because the queue is monotonic we must have $c(z') \leq c(z)$. Thus, for any primitive solution $z +\!\!+ z''$ where $z'' \in \langle\!\langle (z.outcome, \mathbf{a}) \rangle\!\rangle$, $z' +\!\!+ z''$ must reach the same state with no greater cost, and this entry can be safely discarded without compromising the invariant.

When the unexplored entry is not discarded and has nonempty sequence $\mathbf{a} \neq []$, it temporarily *delegates* its solutions to a child OR-node. The operations carried out by GetOrNode obey the refinement and decomposition equations proved correct above, and so long as the child obeys the invariant, it will be maintained at this node by the added child entry. When $\mathbf{a} = []$, the invariant ensures that the discovered solution z must be optimal for $z.outcome$. Then, z is published, creating new entries at all parent subproblems and returning control of the invariant for solutions passing through this state. In the final in-progress case, the contents of the queue at this level are not changed (other than re-prioritizing the current element), and so the invariant is preserved so long as it is preserved at lower levels.

Applying this invariant to the root node shows that DSH-LDFS is hierarchically optimal as long as it terminates. For termination, we note that for a given subproblem, there are at most $1+|\mathcal{S}||\mathcal{I}(a)|$ unique entries that can be added to the closed list (where $|\mathcal{I}(a)|$ represents the total number of actions in the immediate refinements of a). Thus, the **while** loop at each node can carry out at most a finite number of non-recursive iterations before terminating with a new solution or cost bound increased by at least 1. Because the recursion depth is at most N, each call must eventually terminate in this manner. Finally, because the number of subproblems, possible output states, and optimal solution cost are all finite, the algorithm must terminate with at most a finite number of calls to DSH-LDFS-Pass. □

DSH-LDFS can be much more efficient than SAHTN when there exist many state-abstracted subproblems that are not forward-reachable with less than the hierarchically optimal solution cost. Moreover, it is easy to see that the number of refinement operations carried out can be no greater than H-UCS. However, the performance of DSH-LDFS can still be dominated by our previous algorithms, because of the high overhead of propagating costs in a top-down search graph (see Section 2.2.3).

3.3.2 DSH-UCS: Flattened Search

This section presents DSH-UCS, which performs the same basic decomposed search as DSH-LDFS with lower overhead, using a single "flat" priority queue. The basic insight behind

Algorithm 3.3 DSH-UCS

```
function MakeSubproblem(mincost)
    return a node n with
            n.solutions = MakeChannel()
            n.mincost = mincost

function DecomposeSubproblem(queue, z, a, sp)
    s ← EnterContext(z.outcome, a_1)
    if cache[(s, a_1)] = undefined then
        cache[(s, a_1)] ← MakeSubproblem(c(z))
        for (p', z') ∈ R((s, a_1), ⊤_s) do
            queue.Insert((z', p'.actions, cache[(s, a_1)]))
    Subscribe(cache[(s, a_1)].solutions, λz' queue.Insert((z ++ z', a_{2:|a|}, sp)))

function DSH-UCS()
    root ← MakeSubproblem(0)
    fringe ← [(⊤s_0, [H_0], root)], a priority queue on (z, a, sp) tuples ordered by c(z)+sp.mincost,
                                                    breaking ties towards tuples with a = [].
    closed ← []
    while fringe ≠ ∅ do
        (z, a, sp) ← fringe.RemoveMin()
        if closed[(z, a, sp)] = undefined then
            closed.Insert((z, a, sp))
            if a = [] then
                if sp = root then return z
                else Publish(sp.solutions, z)
            else
                DecomposeSubproblem(fringe, z, a, sp)
    return ⊥
```

DSH-UCS is that every time DSH-LDFS actually performs expansions within an OR-node n (i.e., enters the **while** loop in DSH-LDFS-PASS(n, c)), the cost of the forward context encapsulated by the call stack (i.e., $root.queue$.PEEKMIN$() - c$) is always the same. In particular, it is always equal to the minimum cost to reach this subproblem from s_0 under the hierarchy. This means that all of the queues and closed lists in DSH-LDFS can be collapsed into a *single* fringe and closed list, as in a bottom-up search, where each entry now includes a pointer to its subproblem (c.f., OR-node), and the minimum cost to reach that subproblem is added to its total cost. In other words, the optimal context cost for each subproblem is represented explicitly, rather than as a recursive chain of prefix costs.

Algorithm 3.3 shows pseudocode for DSH-UCS, which involves only minor modifications (and simplifications) to the code in Algorithm 3.2. The top-level search encapsulated by DSH-UCS now resembles a standard Dijkstra-style search, where unexplored entries for all existing subproblems are mingled in the same priority queue, and so there is no longer any need for in-progress entries. Each unexplored entry consists of a tuple $(z, \mathbf{a}, sp)$, where sp corresponds to the OR-node at which this entry would have lived (now called a subproblem), and z and $\mathbf{a}$ capture a solution for a prefix of a refinement of sp and its corresponding suffix (as in DSH-LDFS). The priority of the entry is given by $c(z) + sp.mincost$, where $sp.mincost$ is the cost of the cheapest solution that leads to sp, equivalent to the context cost with which sp would always be explored in DSH-LDFS. The basic operations carried out in each loop iteration are identical to DSH-LDFS (with in-progress entries and recursion removed).

Lemma 3.6. *Given equivalent tiebreaking choices, after an equivalent number of* ***while*** *loop iterations (excluding the recursive case in DSH-LDFS), entries in the queue and closed list of DSH-UCS are in one-to-one correspondence with unexplored entries on the queues and closed lists of all OR-nodes in DSH-LDFS.*

Proof. The operations carried out in Algorithm 3.2 and Algorithm 3.3 are in direct correspondence. □

Theorem 3.7. *DSH-UCS is* hierarchically optimal *in FZCF hierarchies.*

Proof. Follows directly from Theorem 3.5 and Lemma 3.6. □

Because it performs the same basic computations with lower overhead, DSH-UCS should be preferred to DSH-LDFS for most direct applications. However, DSH-LDFS is still of interest for comparison (and perhaps combination) with the recursive top-down algorithms explored in Chapter 5.

Chapter 4

Angelic Semantics

This chapter returns to the issue of HLA annotations discussed in Section 2.3.1.3, introducing a novel "angelic semantics" that correctly captures the transition models of HLAs (in the formalism of Section 2.3.2.1). Unlike previous proposals, angelic descriptions of HLAs are *true*; that is, they follow logically from the refinement hierarchy and the descriptions of the primitive actions. Coupled with a sound, efficiently implementable approximation scheme, these descriptions can enable algorithms that realize the full benefits of hierarchical structure in planning.

Specifically, if achievement of the goal is entailed by the true descriptions of a sequence of HLAs, then that sequence must, by definition, be reducible to a primitive solution. Conversely, if the sequence provably fails to achieve the goal according to the descriptions, it is not reducible to a primitive solution. Thus, the downward refinement property and its converse are automatically satisfied. Finally, if the descriptions entail neither success nor failure of a given plan, then further refinement will resolve the uncertainty.

So far so good. But, what can truly be asserted about the preconditions or effects of a high-level action? Chapter 2 described two unambiguous types of such assertions. First, an HLA with a single primitive refinement (a macro) can easily be assigned a correct description; however, most interesting HLAs are not macros, and the tricky issues arise with multiple refinements. Second, an HLA can correctly be assigned a precondition (called an external condition) when some action in *every* primitive refinement requires it.[1]

This section moves beyond these restricted cases to describe full *transition models* for arbitrary HLAs, which correctly specify *all* of their preconditions and effects. This requires abandoning STRIPS-style conditions that are required to hold under *all* refinements, and

[1] One might also attempt to assign an HLA an effect that appears in all refinements. Indeed, in this case any successful execution of the HLA must result in a state where the effect holds. However, to guarantee that the effect is made to hold, the precondition of the HLA must also be strong enough to guarantee that at least one primitive refinement of the HLA will be applicable from a given state.

instead focusing on all possibilities that can occur under *any* refinement. This is a form of nondeterminism, but a key observation is that *the planning agent itself* (not an adversary) will ultimately choose which refinement to take, and thus which possibility is achieved.

A similar distinction between adversarial and non-adversarial refinements of abstract specifications occurs in the semantics of programming languages. For example, a multi-threaded program is correct only if it works under *all* legal thread sequencings, because the actual sequencing depends on unknown ("adversarial") properties of the machine and process scheduler. On the other hand, a nondeterministic algorithm succeeds iff *some* sequence of choices succeeds—the choices are assumed to be made with the aim of succeeding. In the programming languages literature, the term *angelic nondeterminism* is used for this case (Jagadeesan et al., 1992). Thus, we call our proposal an *angelic semantics* for HLAs.

This chapter begins with an overview of a simplified angelic approach that ignores solution costs and is only concerned with reachability, to demonstrate the basic concepts and how they fit together. This is followed by a detailed presentation of the full angelic approach, which incorporates action costs and thus can be used for hierarchically optimal search. Finally, the chapter concludes by discussing practical representations for angelic descriptions and hierarchies, sketching a potential approach for computing descriptions, and briefly exploring connections between the approach and the descriptions of (e.g, STRIPS) "primitives" that are often intended to encapsulate more complex, real-world behaviors.

Planning algorithms based on the techniques introduced here are the topic of Chapter 5.

4.1 Angelic Descriptions for Reachability

4.1.1 Reachable Sets and Exact Descriptions

We begin with the concept of a *reachable set*: the reachable set of HLA h from state s is the set of all states reachable with finite cost from s by *some* primitive refinement of h (see Figure 4.1). If (and only if) the goal state s_* is reachable from the start state s_0 by a given high-level plan, this plan is a *solution*, and can be expanded to a primitive refinement that solves the original problem. In fact, we have already seen an extension of this concept in the guise of solutions to forward subproblems: the reachable set of action a from state s is just $\{z.outcome : z \in \langle\!\langle (s, [a]) \rangle\!\rangle\}$. Figure 3.4 shows a concrete example of a reachable set in our discrete manipulation domain.

Then, the *exact description* of an HLA is simply its reachable set, specified as a function of the initial state s.

Definition 4.1. *The* exact description $\bar{E}_h$ *of an action h is a mapping from an initial state*

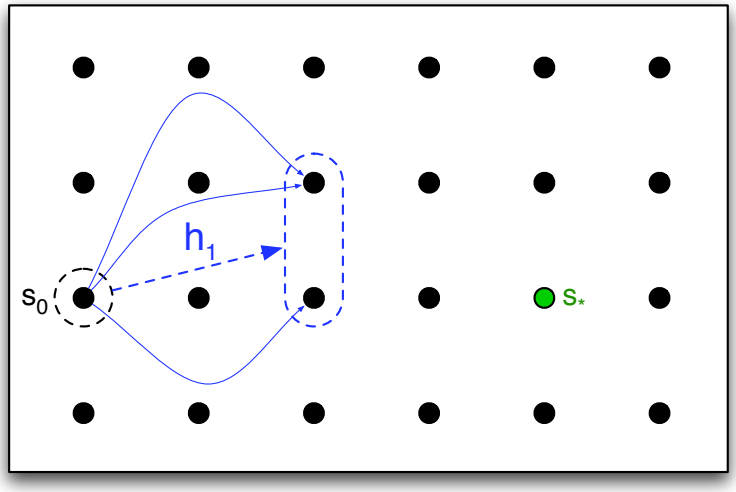

Figure 4.1: The exact reachable set of h_1 (depicted by a dashed oval) from the initial state is the set of all states reachable by primitive refinements of h_1 (depicted by solid lines).

to the set *of states reachable by doing some refinement of h from s:*

$$\bar{E}_h(s) := \{s' : s' = \mathcal{T}(s, \mathbf{a}) \text{ and } \mathbf{a} \in \mathcal{I}^*(h) \text{ and } s' \neq s_\perp\}.$$

Remark. *If a is primitive, then $\bar{E}_a(s) = \{\mathcal{T}(s, a)\} \setminus \{s_\perp\}$.*

Exact descriptions can be extended to functions of *sets* of states, where the output is the set of states reachable from *some* initial state via *some* refinement (see Figure 4.2).

Definition 4.2. *If S is a set of initial states, then*

$$\bar{E}_h(S) := \bigcup_{s \in S} \bar{E}_h(s).$$

With this extension, the exact description for a sequence of actions $\mathbf{a}$ is just the functional composition of the exact descriptions of its constituent actions.

Definition 4.3. *Given an action sequence $\mathbf{a} = [a_1, \ldots, a_n]$,*

$$\bar{E}_{\mathbf{a}} := \bar{E}_{a_n} \circ \ldots \circ \bar{E}_{a_1}.$$

This composition respects the semantics of the original problem. In fact, it is essentially a restatement of the decomposition results of Section 3.1.1.

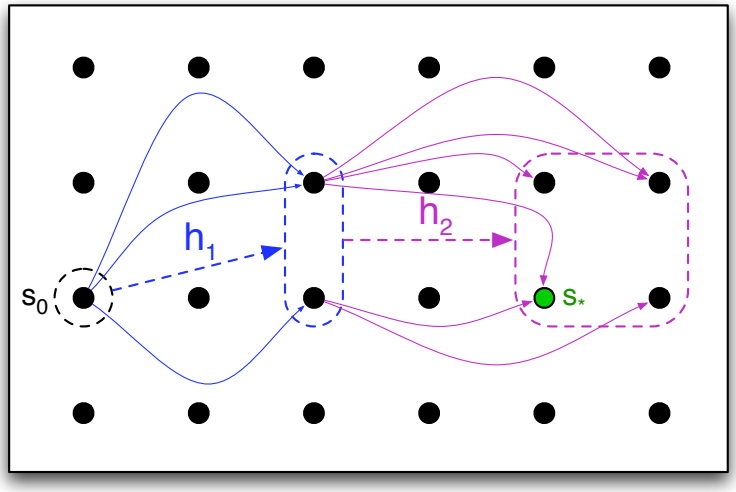

Figure 4.2: The exact reachable set for sequence $[h_1, h_2]$ is the union of the reachable sets of h_2 from each state reachable by h_1. Because this reachable set contains the goal state s_*, plan $[h_1, h_2]$ is a high-level solution.

Theorem 4.1. *When exact descriptions that obey Definition 4.1 are extended and composed via Definitions 4.2 and 4.3, the resulting sequence descriptions are correct:*

$$(\forall s \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ \bar{E}_{\mathbf{a}}(s) = \{s' : s' = \mathcal{T}(s, \mathbf{a}') \text{ and } \mathbf{a}' \in \mathcal{I}^*(\mathbf{a}) \text{ and } s' \neq s_\perp\}.$$

Thus, there exists a primitive refinement of a sequence that reaches a state if and only if that state is contained in the exact reachable set of the sequence.[2]

$$(\forall s, s' \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ s' \in \bar{E}_{\mathbf{a}}(s) \Leftrightarrow (\exists \mathbf{a}' \in \mathcal{I}^*(\mathbf{a}))\ s' = \mathcal{T}(s, \mathbf{a}').$$

Exact descriptions theoretically satisfy the downward refinement and upward solution properties, allowing us to correctly separate high-level solutions from non-solutions in all cases.

Corollary 4.2. *An action sequence* $\mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*$ *is a* solution *iff* $s_* \in \bar{E}_{\mathbf{a}}(s_0)$.

Of course, to reap the benefits of these properties in practice, we would require the ability to efficiently evaluate these exact descriptions. Unfortunately, this is too much to ask for in general; for instance, a *single* evaluation of $\bar{E}_{\mathcal{H}_0}$ for the flat hierarchy described in Section 2.3.2.2 must be PSPACE-complete in general, since it provides an answer to the STRIPS plan existence problem.

[2]We defer proofs of the theorems in this section to Section 4.2, which includes proofs of more general results that take action costs into account.

To sidestep this result, we introduce *optimistic descriptions*, which generate an upper bound (superset) of the exact reachable set, and *pessimistic descriptions*, which generate a lower bound (subset) of the exact reachable set. Compact, efficiently evaluable optimistic and pessimistic descriptions always exist (for some degree of accuracy). Moreover, optimistic descriptions support proofs that a sequence *cannot* reach the goal under *any* refinement, while pessimistic descriptions support proofs that a sequence *surely* reaches the goal under *some* refinement. The price paid for approximation is that in some cases neither inference will apply, and the refinements of a plan must be examined to resolve the uncertainty.

4.1.2 Optimistic and Pessimistic Descriptions

An optimistic description *never underestimates* the true reachable set of an HLA, and a pessimistic description *never overestimates* its true reachable set (see Figure 4.3).

Definition 4.4. *$\bar{O}_a$ and $\bar{P}_a$ are* optimistic *and* pessimistic *descriptions (respectively) of an action $a \in (\mathcal{A} \cup \hat{\mathcal{A}})$ iff*

$$(\forall s \in \mathcal{S})\ \bar{P}_a(s) \subseteq \bar{E}_a(s) \subseteq \bar{O}_a(s).$$

Both optimistic and pessimistic descriptions must be exact for primitive actions:

$$(\forall s \in \mathcal{S}, a \in \mathcal{A})\ \bar{P}_a(s) = \bar{E}_a(s) = \bar{O}_a(s).$$

Like exact descriptions, optimistic and pessimistic descriptions can be extended to sets of states and sequences of actions while still preserving their respective properties.

Theorem 4.3. *When extended to sets and sequences following Definitions 4.2 and 4.3, optimistic and pessimistic descriptions still provide valid bounds on reachable sets:*

$$(\forall s \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ \bar{P}_{\mathbf{a}}(s) \subseteq \bar{E}_{\mathbf{a}}(s) \subseteq \bar{O}_{\mathbf{a}}(s).$$

If a state is contained in the pessimistic reachable set of a sequence, there exists a primitive refinement of that sequence that reaches that state. Moreover, if there exists a primitive refinement that reaches a state, that state must be contained in the optimistic reachable set.

$$\left(\forall s, s' \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*\right)\ \ s' \in \bar{P}_{\mathbf{a}}(s) \Rightarrow \left((\exists \mathbf{a}' \in \mathcal{I}^*(\mathbf{a}))\ s' = \mathcal{T}(s, \mathbf{a}')\right) \Rightarrow s' \in \bar{O}_{\mathbf{a}}(s)$$

If follows directly that if the pessimistic reachable set of a sequence contains the goal, that plan is a solution; and if the optimistic reachable set *does not* contain the goal, that plan is *not* a solution (see Figure 4.4):

Corollary 4.4. *If $s_* \in \bar{P}_{\mathbf{a}}(s_0)$, then $\mathbf{a}$ is a solution. If $s_* \notin \bar{O}_{\mathbf{a}}(s_0)$, then $\mathbf{a}$ is not a solution.*

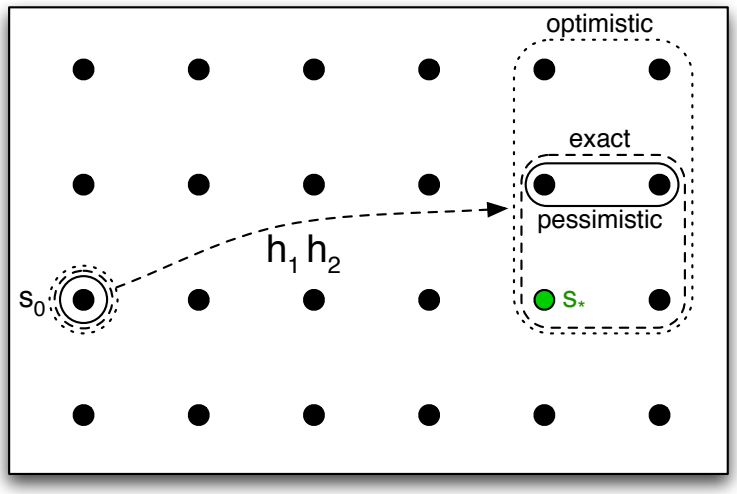

Figure 4.3: Possible optimistic (dotted rectangle) and pessimistic (solid oval) reachable sets for the example in Figure 4.2. Because the optimistic set contains the goal state but the pessimistic set does not, further refinement is required to determine that $[h_1, h_2]$ is a high-level solution.

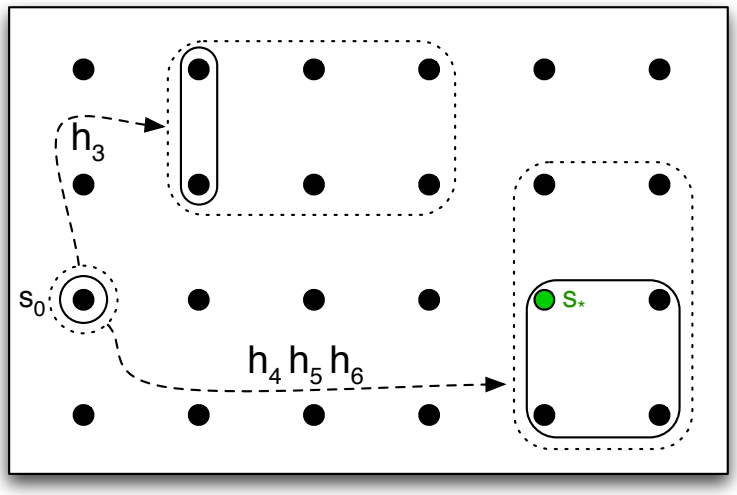

Figure 4.4: Optimistic and pessimistic reachable sets for two other plans. $[h_4, h_5, h_6]$ is provably a solution, because its pessimistic set contains the goal. $[h_3]$ is provably not a solution, because its optimistic reachable set does not contain the goal.

Planning algorithms can *prune* non-solutions without considering them further, and *commit* to solutions to the exclusion of all other plans, while remaining both sound and complete. Moreover, when the pessimistic reachable set of a plan contains a state, a planning algorithm can *decompose* the task of finding a successful primitive refinement of this plan that reaches this state into independent subproblems, one per action in the plan.

Theorem 4.5. *If* $\mathbf{a} = [a_1, \ldots, a_n]$ *and* $s_n \in \bar{P}_{\mathbf{a}}(s_0)$*, then:*

1. $(\exists s_1, ..., s_{n-1} \in \mathcal{S})(\forall 1 \leq i \leq n)\ s_i \in \bar{P}_{a_i}(s_{i-1})$
2. $(\forall 1 \leq i \leq n)\ (\exists \mathbf{b}_i \in \mathcal{I}^*(a_i))\ s_i = \mathcal{T}(s_{i-1}, \mathbf{b}_i)$
3. *Concatenating any such* $\mathbf{b}_i$ *yields a primitive refinement of* $\mathbf{a}$ *that reaches* s_n *from* s_0

In other words, there must exist a sequence of concrete intermediate states, starting with s_0 and ending with s_n, where each state can pessimistically reach the next via the next action in the plan. For each such subproblem $s_i \rightarrow^{a_i} s_{i+1}$, there must exist at least one solution, and any combination of independently discovered solutions of this sort can be concatenated to yield a successful primitive refinement for the entire sequence.

Concrete, practical representations for optimistic and pessimistic descriptions and reachable sets are discussed in Section 4.3.

4.1.3 Simple Angelic Search

Algorithm 4.1 shows pseudocode for a simple (non-optimal) angelic hierarchical planning algorithm called SAS (Simple Angelic Search), which uses optimistic and pessimistic descriptions to gain efficiency in the three ways just described. While (as discussed shortly) we do not expect SAS to perform well in practice, we present it to concretely illustrate the ways that angelic descriptions can be used in search. We defer discussion of practical (optimal) angelic planning algorithms to Chapter 5.

SAS begins with a top-level call to SAS$(s_0, \mathcal{H}_0, s_*)$, which attempts to find a primitive refinement of action $\mathcal{H}_0$ that reaches state s_* from s_0. This procedure contains the main loop, which operates on a queue of potential high-level plans, starting with the sequence $[\mathcal{H}_0]$. Each iteration pops a plan from the queue, immediately discards it if it fails to optimistically reach the goal, commits to and decomposes it if it pessimistically reaches the goal, and otherwise refines it (at some HLA) and places its refinements back on the queue.

Decomposition is handled by DecomposeSolution, which first computes the pessimistic reachable sets at each point in the plan. Then, stepping backwards from the goal,

Algorithm 4.1 Simple Angelic Search

function SAS(s_0, a, s_*)
  fringe $\leftarrow \{[a]\}$ `/* A queue */`
  while *fringe* $\neq \emptyset$ **do**
    $\mathbf{a} \leftarrow$ *fringe*.REMOVEFIRST()
    if $s_* \in \bar{O}_{\mathbf{a}}(s_0)$ **then**
      if $s_* \in \bar{P}_{\mathbf{a}}(s_0)$ and $|\mathbf{a}| \neq 1$ **then**
        return DECOMPOSESOLUTION($s_0, \mathbf{a}, s_*$)
      else
        for $\mathbf{a}' \in \mathcal{I}_i(\mathbf{a})$ **do** `/* At any HLA index` i `*/`
          fringe.INSERT($\mathbf{a}'$)
  return *failure*

function DECOMPOSESOLUTION($s_0, [a_1, ..., a_n], s_n$)
  sol $\leftarrow []$
  $S_0 \leftarrow \{s_0\}$
  for $i = 1, ..., n-1$ **do**
    $S_i \leftarrow \bar{P}_{a_i}(S_{i-1})$
  for $i = n, ..., 1$ **do**
    $s_{i-1} \leftarrow$ some $s \in S_{i-1}$ where $s_i \in \bar{P}_{a_i}(s_{i-1})$
    if a_i is primitive **then** *sol* $\leftarrow [a_i] +\!\!+$ *sol*
    else *sol* $\leftarrow$ SAS(s_{i-1}, a_i, s_i) $+\!\!+$ *sol*
  return *sol*

it chooses a sequence of concrete intermediate states (by Theorem 4.5), and solves each ensuing HLA subproblem by a recursive call to SAS.[3] Finally, it concatenates these solutions and returns the resulting primitive solution. This solution decomposition is related to the decomposed search strategy performed by algorithms in Chapter 3. The difference is that here search is still carried out over complete plans, and decomposition is applied only after a high-level solution is identified, rather than throughout search.

If the agent does not require a primitive solution, just a certificate of solvability, the algorithm can stop as soon as the first pessimistically succeeding plan is found. Or, in intermediate situations, an agent may interleave planning and execution, solving just the first problem in the decomposition before starting to execute actions in the real world.

SAS is sound and complete for non-recursive hierarchies, but may loop forever in recursive hierarchies, even given access to exact angelic descriptions. The issue is that while pessimistic descriptions guarantee that a primitive refinement of a plan exists, they do not provide any guarantee that a given plan is *good*, or makes progress towards the goal in a meaningful way. For instance, consider a navigation problem in an infinite grid world. *Every* plan is a high-level solution, and SAS can repeatedly commit to plans that actually take the agent farther and farther away from its goal. This issue arises because SAS does not have access to (estimates of) action costs, a shortcoming that is addressed in the next section.[4]

4.2 Angelic Descriptions with Costs

The previous section described the outcome of a high-level plan by its reachable set of states. However, these reachable sets say nothing about *costs* incurred along the way. This section describes a full version of the angelic semantics that includes cost information, enabling algorithms that can find *good* plans *quickly* by focusing on better-seeming plans first, and pruning provably suboptimal high-level plans without refining them further.

4.2.1 Valuations and Exact Descriptions

In this extended angelic semantics, we are concerned not only about which states are reachable by a high-level plan from some state, but also, for each reachable state, the *minimum cost* to reach it by some refinement.

[3]Finding the intermediate states efficiently requires the ability to *regress* sets through the pessimistic descriptions, which we discuss briefly in Section 4.3.

[4]While it is possible to avoid these issues and construct a sound, complete, terminating version of SAS (Marthi et al., 2007a), the efficiency of this algorithm would still suffer from the basic issues mentioned here.

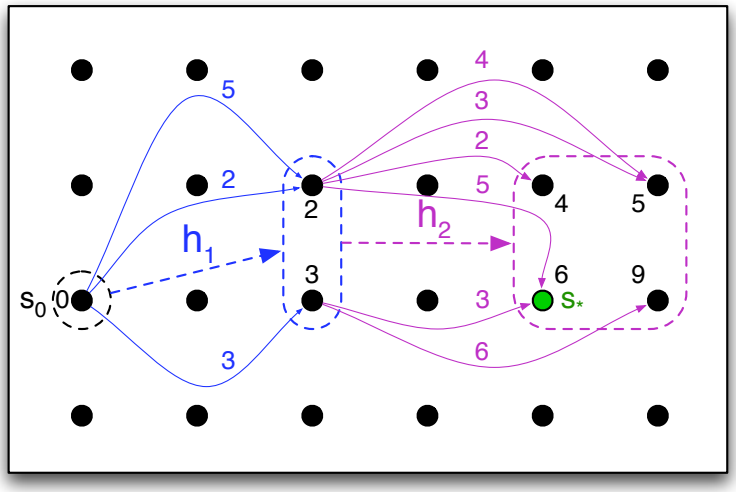

Figure 4.5: Exact valuations for the example in Figure 4.2. Each primitive refinement is labeled with its cost, and each state is labeled with the minimum cost to reach it from s_0. All states outside the dashed reachable sets are assigned infinite cost.

Definition 4.5. *Denote the minimum cost to reach s' from s by any primitive refinement of $\mathbf{a}$ by $c^*((s, \mathbf{a}, s'))$. This can be expressed directly, or in terms of forward solution sets:*

$$c^*((s, \mathbf{a}, s')) := \min_{\mathbf{a}' \in \mathcal{I}^*(\mathbf{a}) : \mathcal{T}(s, \mathbf{a}') = s'} \mathcal{C}(s, \mathbf{a}') = \min_{z \in \langle\!\langle (s, \mathbf{a}) \rangle\!\rangle : z.outcome = s'} c(z).$$

Full angelic descriptions return *valuations*, which extend reachable sets to include optimal cost information of this sort.

Definition 4.6. *A* valuation *is a function $v : \mathcal{S} \to [0, \infty]$. The* initial valuation v_0 *has $v_0(s_0) = 0$ and $v_0(s) = \infty$ for all $s \neq s_0$.*

Figure 3.4 shows an exact valuation for the GoGrasp$(\cdot)$ HLA in the discrete manipulation domain.

Definition 4.7. *The* exact description E_h *of h is a mapping from a state s to a valuation, which specifies the minimum cost to reach each state by some refinement of h from s:*

$$E_h(s) := v \quad \text{where} \quad (\forall s' \in \mathcal{S})\ v(s') = c^*((s, h, s')).$$

Remark. *If s' is not reachable by any refinement of h, $E_a(s)(s') = \infty$. Thus, this definition of exact descriptions generalizes the definition in the previous section:*

$$E_a(s)(s') < \infty \Leftrightarrow s' \in \bar{E}_a(s).$$

Exact descriptions can be extended to functions of valuations, by taking the minimum total cost to reach each output state over all states reachable in the input valuation (see Figure 4.5):

Definition 4.8. *If v is an initial valuation and E_h is an exact description, then*

$$E_h(v)(s') := \min_{s \in \mathcal{S}} \left(v(s) + E_h(s)(s')\right).$$

With this extension, the exact description for a sequence of actions $\mathbf{a}$ is again just the composition of the exact descriptions of its constituent actions.

Definition 4.9. *Given an action sequence $\mathbf{a} = [a_1, \ldots, a_n]$,*

$$E_{\mathbf{a}} := E_{a_n} \circ \ldots \circ E_{a_n}.$$

This composition process produces correct exact descriptions for sequences:

Theorem 4.6. *(Generalization of Theorem 4.1.) When exact descriptions that obey Definition 4.7 are extended and composed via Definitions 4.8 and 4.9, the resulting sequence descriptions are correct:*

$$(\forall s, s' \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ E_{\mathbf{a}}(s)(s') = c^*((s, \mathbf{a}, s'))$$

Proof. The proof is by induction. When $N = 1$, the theorem follows trivially from Definition 4.7. When $N > 1$,

$$\begin{aligned}
\min_{(s_1,\ldots,s_{N-1})} \sum_{i=1}^{N} E_{a_i}(s_{i-1})(s_i) &= \min_{(s_1,\ldots,s_{N-1})} \left(E_{a_N}(s_{N-1})(s_N) + \sum_{i=1}^{N-1} E_{a_i}(s_{i-1})(s_i) \right) \\
&= \min_{s_{N-1}} \left(E_{a_N}(s_{N-1})(s_N) + \min_{(s_1,..,s_{N-2})} \sum_{i=1}^{N-1} E_{a_i}(s_{i-1})(s_i) \right) \\
&= \min_{s_{N-1}} \left(E_{a_N}(s_{N-1})(s_N) + E_{a_{N-1}} \circ \ldots \circ E_{a_1}(v_0)(s_{N-1}) \right) \\
&= E_{a_N} \circ \ldots \circ E_{a_1}(v_0)(s_N)
\end{aligned}$$

□

These full exact descriptions allow us to correctly separate *hierarchically optimal* high-level solutions from non-optimal solutions (and non-solutions) in all cases:

Corollary 4.7. *An action sequence $\mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*$ is a* hierarchically optimal solution *iff $E_{\mathbf{a}}(s_0)(s_*) = E_{\mathcal{H}_0}(s_0)(s_*)$ (and $E_{\mathcal{H}_0}(s_0)(s_*) < \infty$, i.e., the problem is solvable).*

Of course, such full exact descriptions are even less likely to be practically applicable than the reachability versions of Section 4.1.1. Thus, for practical implementations of these concepts we again turn to optimistic and pessimistic descriptions, which will provide correct bounds on the exact valuations of high-level plans.

4.2.2 Optimistic and Pessimistic Descriptions

Full optimistic and pessimistic descriptions produce valuations that never overestimate or underestimate (respectively) the true cost to reach a state (see Figure 4.6). We first define a notion of *domination* for valuations, which generalizes the subset relation over reachable sets to incorporate costs.

Definition 4.10. *A valuation v_1 weakly* dominates *another valuation v_2, written $v_1 \preceq v_2$, iff it assigns no greater cost to each state:*

$$v_1 \preceq v_2 \Leftrightarrow (\forall s \in \mathcal{S})\ v_1(s) \leq v_2(s).$$

Moreover, v_1 strictly dominates *v_2, written $v_1 \prec v_2$, iff it assigns strictly lower cost to each reachable state:*

$$v_1 \prec v_2 \Leftrightarrow (\forall s \in \mathcal{S})\ v_2(s) = \infty \text{ or } v_1(s) < v_2(s).$$

We now formally define optimistic and pessimistic descriptions in terms of domination.

Definition 4.11. *An* optimistic description *of an HLA h is one whose output valuation always dominates the corresponding exact valuation, and a* pessimistic description *is one whose output valuation is always dominated by the exact valuation:*

$$(\forall s \in \mathcal{S})\ O_h(s) \preceq E_h(s) \preceq P_h(s).$$

For primitive actions $a \in \mathcal{A}$, we require that the descriptions are exact: $O_a = E_a = P_a$.

By extending these descriptions to input valuations and sequence of actions in the same manner as exact descriptions (see Definitions 4.8 and 4.9), we obtain valid bounds on the reachable states and costs of high-level sequences.

Theorem 4.8. *When optimistic and pessimistic descriptions that obey Definition 4.11 are extended and composed via Definitions 4.8 and 4.9, the resulting sequence descriptions are correct:*

$$(\forall v, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ O_{\mathbf{a}}(v) \preceq E_{\mathbf{a}}(v) \preceq P_{\mathbf{a}}(v).$$

Proof. When $|\mathbf{a}| = N = 1$, this follows trivially from Definition 4.11. Using induction for $N > 1$, for optimistic descriptions (the pessimistic case is symmetric):

$$\begin{aligned} O_{a_N} \circ \ldots \circ O_{a_1}(v_0)(s_N) &= \min_{s_{N-1}} O_{a_N}(s_{N-1})(s_N) + O_{a_{N-1}} \circ \ldots \circ O_{a_1}(v_0)(s_{N-1}) \\ &\leq \min_{s_{N-1}} E_{a_N}(s_{N-1})(s_N) + E_{a_{N-1}} \circ \ldots \circ E_{a_1}(v_0)(s_{N-1}) \\ &= E_{a_N} \circ \ldots \circ E_{a_1}(v_0)(s_N) \end{aligned}$$

□

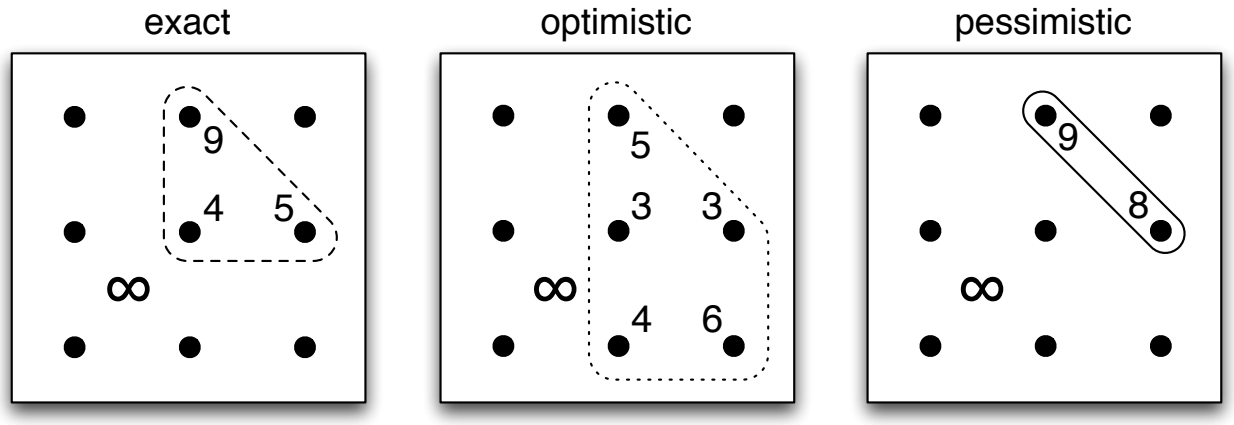

Figure 4.6: An exact valuation for a hypothetical plan (left), and potential optimistic (center) and pessimistic (right) approximations of this valuation.

Much of the remainder of this thesis will be concerned with the following consequences of this theorem.

Corollary 4.9. *If the optimistic valuation of a sequence* $\mathbf{a}$ *assigns cost* c *to a state* s'*, then no primitive refinement of* $\mathbf{a}$ *reaches* s' *with cost strictly less than* c*:*

$$(\forall s, s' \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ O_{\mathbf{a}}(s)(s') = c \ \Rightarrow\ \neg(\exists \mathbf{a}' \in \mathcal{I}^*(\mathbf{a}))\ s' = \mathcal{T}(s, \mathbf{a}') \text{ and } \mathcal{C}(s, \mathbf{a}') < c.$$

Similarly, if the pessimistic valuation of a sequence $\mathbf{a}$ *assigns cost* c *to a state* s'*, then at least one primitive refinement of* $\mathbf{a}$ *reaches* s' *with cost no greater than* c*:*

$$(\forall s, s' \in \mathcal{S}, \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*)\ P_{\mathbf{a}}(s)(s') = c \ \Rightarrow\ (\exists \mathbf{a}' \in \mathcal{I}^*(\mathbf{a}))\ s' = \mathcal{T}(s, \mathbf{a}') \text{ and } \mathcal{C}(s, \mathbf{a}') \leq c.$$

Proof. Follows directly from Theorem 4.8, Theorem 4.6, and Definition 4.5. □

As a special case, we note that if $O_{\mathbf{a}}(s_0)(s_*) > E_{\mathcal{H}_0}(s_0)(s_*)$, then no primitive refinement of $\mathbf{a}$ is a hierarchically optimal solution. During planning, such *provably suboptimal* plans can be immediately discarded, without sacrificing hierarchical optimality. Thus, optimistic descriptions are analogous to *admissible heuristics* in the state-space setting (see Section 2.1). Continuing the analogy, we can define a *consistency* property for HLA descriptions.

Definition 4.12. *The overall optimistic and pessimistic transition functions are* consistent *iff refining a high-level plan never leads to weaker bounds:*

$$\begin{aligned}
&\big(\forall \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*, i, v, s\big) \quad (\forall \mathbf{a}' \in \mathcal{I}_i(\mathbf{a}))\ \ O_{\mathbf{a}}(v)(s) \leq O_{\mathbf{a}'}(v)(s), \\
&\big(\forall \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*, i, v, s\big) \quad (\exists \mathbf{a}' \in \mathcal{I}_i(\mathbf{a}))\ \ P_{\mathbf{a}'}(v)(s) \leq P_{\mathbf{a}}(v)(s).
\end{aligned}$$

In Chapter 5, we present a hierarchically optimal search algorithm based on A*, which can be more efficient when given consistent optimistic descriptions.

Similarly, if $P_{\mathbf{a}}(s_0)(s_*) = E_{\mathcal{H}_0}(s_0)(s_*)$ then some primitive refinement of $\mathbf{a}$ is hierarchically optimal, and a planning algorithm can correctly commit to this plan to the exclusion of all others. In addition, the agent can *decompose* the task of finding an optimal primitive refinement of $\mathbf{a}$ into independent subproblems, one per action in $\mathbf{a}$.

Theorem 4.10. *If* $\mathbf{a} = [a_1, \ldots, a_n]$ *and* $P_{\mathbf{a}}(s_0)(s_n) = c < \infty$*, then:*

1. $(\exists s_1, ..., s_{n-1} \in \mathcal{S})\ (\forall 1 \leq i \leq n)\ \ P_{a_i}(s_{i-1})(s_i) = c_i$ and $\sum c_i = c$,
2. $(\forall 1 \leq i \leq n)\ (\exists \mathbf{b}_i \in \mathcal{I}^*(a_i))\ \ \mathcal{T}(s_{i-1}, \mathbf{b}_i) = s_i$ and $\mathcal{C}(s_{i-1}, \mathbf{b}_i) \leq c_i$, *and*
3. *Let* $\mathbf{b} := \mathbf{b}_1 +\!\!+ \ ... \ +\!\!+ \ \mathbf{b}_n$. *Then* $\mathbf{b} \in \mathcal{I}^*(\mathbf{a})$, $\mathcal{T}(s_0, \mathbf{b}) = s_n$, *and* $\mathcal{C}(s_0, \mathbf{b}) \leq c$.

Proof. (1) follows directly from Definition 4.9. (2) follows from Corollary 4.9. (3) follows from Definition 2.31. □

In other words, when a plan $\mathbf{a}$ pessimistically reaches state s_n from s_0 with cost c, there must exist intermediate states s_i such that each action a_i in $\mathbf{a}$ pessimistically reaches s_i from s_{i-1}, and the pessimistic costs of these individual steps add up to c. These intermediate states effectively decompose the problem, generating independent subproblems for each a_i (where the objective is to find a primitive refinement of a_i that reaches s_i from s_{i-1} with no more than the pessimistic cost). Finally, concatenating any solutions to these subproblems generates a primitive refinement for $\mathbf{a}$ that reaches s_n from s_0 with cost no greater than c.

To use these results in practice we need to choose a compact representation for valuations, which can be progressed through the approximate descriptions of a sequence in turn. In the interest of maintaining compactness, we may sometimes choose to use an *approximate* progression algorithm that loses some information.

Definition 4.13. *An approximate progression algorithm corresponds to a relaxation of Definition 4.8, yielding (further) approximated descriptions* $\tilde{O}_h(v)$ *and* $\tilde{P}_h(v)$ *for each pair of original descriptions* O_h *and* P_h*. This algorithm is* correct *iff the errors only further weaken the descriptions:*

$$(\forall a \in (\mathcal{A} \cup \hat{\mathcal{A}}), v)\ \ \tilde{O}_h(v) \preceq O_h(v) \text{ and } P_h(v) \preceq \tilde{P}_h(v).$$

Correct approximate progression algorithms can safely be used in planning, without compromising the results of Theorems 4.8 and 4.10.

Theorem 4.11. *Theorems 4.8 and 4.10 hold under any correct approximate progression algorithm, replacing each* O_a *and* P_a *with their approximated counterparts* $\tilde{O}_a$ *and* $\tilde{P}_a$ *(and replacing* $\sum c_i = c$ *with* $\sum c_i \leq c$ *in part 1 of Theorem 4.10).*

Proof. Composing approximated optimistic descriptions always yields a sequence description that dominates the sequence description of the original (non-approximated) descriptions:

$$\begin{aligned} \tilde{O}_{a_2}(\tilde{O}_{a_1}(v)) &\preceq O_{a_2}(\tilde{O}_{a_1}(v)) \\ &\preceq O_{a_2}(O_{a_1}(v)), \end{aligned}$$

where the second step uses the fact that descriptions produced by Definition 4.8 are *monotonic* – that is, if $v_1 \preceq v_2$, then $O_a(v_1) \preceq O_a(v_2)$. The pessimistic result is analogous. □

Note that approximate progression may not preserve consistency. That is, even if a set of optimistic descriptions are consistent, their approximated counterparts may not be.

4.3 Representations and Inference

The previous section showed, via Theorems 4.8 and 4.10, that optimistic and pessimistic descriptions for high-level actions can provide correct bounds on the outcomes and costs of high-level plans. To utilize these results, a method for evaluating optimistic and pessimistic descriptions on sequences of actions is needed. We consider a simple such method based on *progression*, computing a valuation for a sequence of actions by beginning with the initial valuation v_0 and progressing through each action description in turn (i.e., $v_{i+1} = O_{a_i}(v_i)$), just as action sequences are typically evaluated in state-space planning.

To put this method into practice, compact representations for approximate angelic descriptions and valuations are required, along with efficient algorithms for progressing valuations through angelic descriptions. This section begins by describing a simple, generic interface that captures the details of a hierarchy and angelic descriptions needed for angelic hierarchical planning algorithms based on progression. We then describe two concrete possibilities for satisfying this interface. First, we describe a STRIPS-like language for declaratively specifying hierarchies and angelic descriptions, along with concrete algorithms that satisfy the interface by manipulating the declarative specifications. While this declarative language is sufficient to capture useful angelic descriptions for simple domains such as the nav-switch example of Section 2.1.2.1, further extensions are needed to effectively capture the behaviors of more complex HLAs such as those encountered in our discrete manipulation domain. We thus describe a second possibility, which is to write code that directly implements the interface for each HLA type, and thus escape the limitations of any particular declarative language.

4.3.1 Interface for Search Algorithms

This section describes a programmatic interface for HLAs that implements the hierarchy definitions of Section 2.3.2.1 and angelic methods of this chapter, which will be used by

the search algorithms in the next chapter. Concretely, this interface includes methods for progressing input valuations through the optimistic and pessimistic descriptions of an HLA to generate output valuations, generating its immediate refinements (restricted to those that may be applicable from a given reachable set), and computing its state-abstracted context for a particular input reachable set.

This interface is general enough to capture a wide variety of methods for implementing these operations on HLAs. However, it includes one simplifying assumption: that the representation for valuations is *simple*, consisting of an (arbitrary) reachable set representation, along with a *single* cost bound to reach states in this set.

Definition 4.14. *A* simple valuation $v := (S, c)$ *consists of a reachable set of states S together with a single cost bound c. Valuation v assigns cost c to all states in S, and ∞ to all other states.*

The restriction to simple valuations means that a single valuation cannot represent *correlations* between potential reachable states and costs. For example, an optimistic simple valuation describing the outcome of GoGrasp(m) from Figure 3.4 would have to assign the entire reachable set a lower cost bound of 11 (or less), losing information about the greater cost to reach states where the grasp was carried out from the top, bottom, or right side of the table.

Approximate progression with simple valuations is also typically not consistent. Thus, we will henceforth consider a simpler form of *reachability consistency* on the optimistic descriptions:

Definition 4.15. *The overall optimistic transition function is* reachability consistent *iff refining a high-level plan or using a smaller input set never leads to a larger reachable set:*

$$\begin{aligned} &\big(\forall \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*, S, i\big) \quad (\forall \mathbf{a}' \in \mathcal{I}_i(\mathbf{a})) \;\; \bar{O}_{\mathbf{a}}(S) \supseteq \bar{O}_{\mathbf{a}'}(S) \\ &\big(\forall \mathbf{a} \in (\mathcal{A} \cup \hat{\mathcal{A}})^*, S \supseteq S'\big) \qquad\qquad \bar{O}_{\mathbf{a}}(S) \supseteq \bar{O}_{\mathbf{a}}(S'). \end{aligned}$$

We restrict our attention to simple valuations primarily because they enable the (simpler, more effective) application of powerful search techniques, including the decomposed search strategy of Chapter 3 and a pruning technique based on pessimistic descriptions (described in then next chapter). For example, after doing GoGrasp(m) from a state where the robot's base begins in a different state, the reachable set will be the same, but the valuation will typically be different (because different costs will be assigned to each reachable state). By giving up on the ability to represent the differences in costs to reach each state, we thus dramatically increase the potential for state-abstracted caching.

However, it is likely that with some added algorithmic complexity, we could construct algorithms based on non-simple valuations that could still take full advantage of decomposition and pruning techniques. The end of the next section discusses potential ways to

move beyond simple valuations, while still allowing for compact representations and efficient inference algorithms.

We now introduce the HLA interface, which consists of four methods for each action a. Under the assumption of simple valuations, the inputs to optimistic and pessimistic descriptions can be reachable sets rather than full valuations, deferring responsibility for tracking the costs to reach these sets to the search algorithm.

- OptimisticOutcomeAndStatus(a, S) takes an action a and optimistic reachable set S as input, and returns a tuple $(S', c, \textit{refinable}?)$ representing the output optimistic set S' and optimistic step cost c, plus a binary flag *refinable*?. If *refinable*? = true, then a is a high-level action that can be refined from S using ImmediateRefinements(a, S). Otherwise, *refinable*? = false indicates that a cannot be refined from S. This is always the case for primitive actions $a \in \mathcal{A}$, but can also be used when $|S| > 1$ for high-level actions that cannot be productively refined without more information about the input state.[5]

- PessimisticOutcome(a, S) takes an action and pessimistic reachable set as input, and returns a tuple (S', c) representing the output pessimistic set and step cost.

- EnterContext(S, a) returns the subset of S relevant for doing a, allowing the exploitation of state abstraction (see Section 3.1.2).

- ImmediateRefinements(h, S) takes a high-level action h and an optimistic reachable set as input, and returns a set of immediate refinements of h, each of which is a sequence of actions. Immediate refinements with no applicable primitive refinements from S need not be returned in this set.

Primitive actions can be automatically retrofitted to this interface (since they are never refinable, ImmediateRefinements is omitted), using a version of the progression algorithm described in the next section.

4.3.2 Declarative Representations: NCSTRIPS

This section defines a fully declarative representation for hierarchies and angelic descriptions in STRIPS domains, and specifies algorithms that operate on these representations to satisfy the HLA interface declared in the previous section. These declarative specifications follow

[5]For example, it probably does not make sense to try to generate a detailed grasp plan from a reachable set where the position of the robot base is still uncertain. This is essentially a hint to the search algorithm about the order in which the hierarchical search space should be explored, and could no doubt be replaced by more principled techniques such as improved meta-level search control.

the tradition of domain-independent STRIPS planning, wherein the *knowledge* required for planning is completely separated from the *algorithms* for operating on this knowledge. Such declarative descriptions can be simple to specify, and free the hierarchy designer from worrying about details such as progression algorithms. Moreover, declarative specifications can be exploited for other kinds of inferences, including regression and automatic description verification. Drawbacks of this approach include the complexity of implementing efficient generic inference algorithms for a particular declarative language, along with any limitations that this particular language may place on the ease (or possibility) of expressing certain types of descriptions.

Specifically, this section first describes a method for representing reachable sets of states as disjunctive normal form (DNF) logical formulae. It then presents a STRIPS-style representation for high-level action schemata and their refinements, and sketches an algorithm for efficiently generating the immediate refinements of an HLA that may be applicable from a given set of states (represented in DNF). Finally, it concludes by introducing a representation for optimistic and pessimistic descriptions called NCSTRIPS (Nondeterministic, Conditional STRIPS), and defining a generic progression algorithm that can efficiently generate optimistic and pessimistic DNF reachable sets and cost bounds from these descriptions and a DNF specification of an input set.

4.3.2.1 DNF Reachable Sets

The algorithms discussed in this section consume and produce reachable sets represented as disjunctive normal form (DNF) formulae. A DNF formula is a disjunction of *clauses*, each of which is a conjunction of *literals* (possibly-negated propositions). The extension of the reachable set consists of those states whose propositions satisfy the formula. For example, in a simple STRIPS domain with propositions a, b, c, and d, the formula $(a \wedge \neg b \wedge c) \vee (\neg a \wedge \neg b \wedge c \wedge d)$ represents a set with three states: $a \wedge \neg b \wedge c \wedge \neg d$, $a \wedge \neg b \wedge c \wedge d$, and $\neg a \wedge \neg b \wedge c \wedge d$.

DNF is a convenient representation for reachable sets because each conjunctive clause represents a *partial state*, i.e., a set of states where some propositions have the same values in all states, and the remaining propositions may take on any combination of values. As we will see, this is convenient because it is easy to test whether a given partial state (possibly) satisfies a given precondition, by simply testing if propositions that are present have the required values. Moreover, satisfiability of a DNF formula can be tested in constant time (it is satisfiable iff it is not empty). However, DNF representations can also be exponentially larger than other potential representations such as binary decision diagrams (BDDs); we briefly discuss potential such alternative representations at the end of the section.

4.3.2.2 NCSTRIPS Hierarchies

We first describe how the structure of a hierarchy, consisting of parameterized HLA types and their allowed immediate refinements, can be specified using NCSTRIPS HLA schemata that generalize the STRIPS action schemata of Section 2.1.2.1. We present this representation via example, using examples from our nav-switch hierarchy (see Sections 2.1.2.1 and 2.3.2.2). Each HLA schema consists of the following components.

- A set of *typed parameters* for the HLA. For instance, $\text{Go}(x_1, y_1, x_2, y_2)$ has parameters x_1 and x_2 with type Xs and parameters y_1 and y_2 with type Ys.
- An optional *precondition*, which specifies constraints on the allowed bindings of the HLA parameters ($\text{Go}(x_1, y_1, x_2, y_2)$ has none).
- A set of *refinement schemata*, each of which consists of the following components.
  - A set of *typed parameters* introduced in the refinement.
  - An optional *precondition*, which specifies constraints on the refinement and HLA parameters under which this refinement is applicable.
  - An *expansion*, a sequence of actions (high-level or primitive) with arguments taken from the HLA or refinement parameters.

For example, $\text{Go}(x_1, y_1, x_2, y_2)$ has three refinement schemata:

1. The first schema corresponds to navigating directly, without flipping the switch. It has no parameters, no precondition, and the simple expansion $\text{Nav}(x_1, y_1, x_2, y_2)$.
2. The second schema corresponds to flipping at least one switch, when the current switch position is horizontal. It has parameters x' of type Xs and y' of type Ys, preconditions $\mathsf{SwitchAt}(x', y')$ and H, and expansion $[\text{Nav}(x_1, y_1, x', y'), \text{FlipV}(x', y'), \text{Go}(x', y', x_2, y_2)]$.
3. The third schema is similar to the second, but applies when the current switch position is vertical (replacing H by $\neg\mathsf{H}$ and FlipV by FlipH)

We now briefly sketch an algorithm for generating the immediate refinements of a particular instantiation of an NCSTRIPS action schema from a particular reachable set (represented in DNF). A first, naive approach would be to enumerate all possible instantiations of each refinement schema (ranging over all allowed values for the parameters), and then prune those instantiations in which the refinement preconditions are not satisfied by any clause in the DNF formula representing the input set. However, just like when grounding STRIPS action

schemata, this may be highly inefficient when many of the possible refinements include inconsistent variable bindings or are inapplicable from the input state set. Fortunately, the solution to that problem discussed in Section 2.1.3.4 can also be applied here: to generate the applicable refinements, we can construct a *constraint satisfaction problem* (CSP) for each refinement schema, where the variables are the parameters of the schema and the constraints are its preconditions with extension given by the DNF formula. The solutions to each such CSP can be efficiently enumerated, and each such solution generates a consistent immediate refinement that is applicable from the current input set.[6]

4.3.2.3 NCSTRIPS Angelic Descriptions

We now present the NCSTRIPS representation for optimistic and pessimistic descriptions, and describe an algorithm for efficiently progressing DNF reachable sets through these descriptions.

We first introduce the basic concepts by returning to the example of Section 1.2. In this example, the task was to construct a correct transition model for a simple HLA h with refinements $[a]$ and $[b]$, where a is a primitive action with precondition $u \wedge v$ and effect $e \wedge f$, and b is a primitive action with precondition $u \wedge w$ and effect $e \wedge g$ (see Figure 1.3).

Now, a first way to construct a correct transition model of h is to use a form of *conditional effects*. In particular, we can assign h an NCSTRIPS description consisting of two *clauses*, one corresponding to the STRIPS description of a and one for the STRIPS description of b.

$$\begin{aligned} u \wedge v &\rightarrow e \wedge f \\ u \wedge w &\rightarrow e \wedge g \end{aligned}$$

Then, given a particular initial state, e.g., where $u, v,$ and w are true and $e, f,$ and g are false, we can apply this description to generate the reachable set of h as follows. First, we represent the state as a conjunctive clause of literals: $u \wedge v \wedge w \wedge \neg e \wedge \neg f \wedge \neg g$. Then, for each conditional effect, we see if its precondition is satisfied by the clause (in this case, both are), and if so, generate an output clause in which the effect literals are made to hold. Finally, we disjoin the resulting clauses, reaching the DNF formula $(u \wedge v \wedge w \wedge e \wedge f \wedge \neg g) \wedge (u \wedge v \wedge w \wedge e \wedge \neg f \wedge g)$, which correctly describes the two concrete states reachable by h from this initial state. As we will see, the same basic logic can be applied to generate the reachable set from a set of input states represented in DNF, by progressing each clause independently and then disjoining the results.

While this conditional effect strategy can always represent the exact transition model of an HLA (by, in the worst case, including one clause for each primitive refinement), as

[6] However, unlike in Section 2.1.3.4, these CSPs must be solved at runtime, each time the immediate refinements of an HLA are generated. Thus, it may be desirable to do additional compilation beforehand, e.g., based on the values of constant predicates identified in a preprocessing phase.

described earlier such exact descriptions are typically far too large and/or expensive to be useful in practice. For instance, if an HLA g could potentially set each of 10 propositions to be true or false, 1024 clauses of the above sort would be needed to describe its exact transition model. We thus introduce a further notational extension, *possible effects*, which can much more compactly (often approximately) describe the transition model of an HLA.

In particular, in addition to allowing STRIPS-style effects of the form x or $\neg x$, which definitely cause x to be true or false (respectively), we also consider *possible effects* of the form $\tilde{+}x$, $\tilde{-}x$, and $\tilde{\pm}x$. A possible-add effect $\tilde{+}x$ of proposition x may cause x to be made true, or leave x with its former value. Similarly, a possible-delete effect $\tilde{-}x$ either sets x to false or has no effect on x. Finally, $\tilde{\pm}x$ represents a combination of a possible-add and possible-delete, and may leave x true or false regardless of its former value. These possible effects directly generate a reachable sets of states from a single description clause, and can thus lead to much more compact descriptions; for instance, the description of g above could be captured by a single clause with no precondition and effects $\tilde{\pm}x_1, ..., \tilde{\pm}x_{10}$.

Thus, when a clause has multiple possible effects, the semantics is that *any combination* of the possible effects may apply (and so the outcome is a DNF clause / partial state). This means that the *correlations* between variables cannot be captured, and so descriptions of this sort will typically be approximate. For instance, the pair of effects $x \wedge y$ and $\neg x \wedge \neg y$ cannot be represented exactly by a single clause with possible effects, because $\tilde{\pm}x \wedge \tilde{\pm}y$ represents the combinations $\neg x \wedge y$ and $x \wedge \neg y$ in addition to the desired pair.

Now, returning to our earlier example, we can express an optimistic description of h using a single clause with possible effects:

$$u \rightarrow e \wedge \tilde{+}f \wedge \tilde{+}g.$$

That is, when u is true, h definitely has effect e, and possibly-adds f and g as well. The reachable set generated by this description always includes all states actually reachable by h, but may include other states as well (for instance, where both f and g are set to true, or where f is set to true despite v being false in the input state). Similarly, a single-clause pessimistic description of h might simply pick a refinement, e.g.,

$$u \wedge v \rightarrow e \wedge f,$$

because NCSTRIPS cannot capture the correlations between both v and f and w and g in a single clause.

Formally, an NCSTRIPS representation of an optimistic or pessimistic description consists of a set of *clauses*, each of which has a precondition, effect, and cost expression. The precondition of each clause is a conjunction of literals, as in ordinary STRIPS. The effect is a conjunction of literals as in STRIPS, together with an additional set of possible-add, possible-delete, and $\tilde{\pm}$ effects that can be made to hold by doing the HLA when the precondition is met.

For example, an optimistic description of $\textsc{Go}(x_1, y_1, x_2, y_2)$ in the nav-switch domain can be expressed with a single clause:

$$\mathsf{AtX}(x_1) \wedge \mathsf{AtY}(y_1) \rightarrow \neg\mathsf{AtX}(x_1) \wedge \neg\mathsf{AtY}(y_1) \wedge \mathsf{AtX}(x_2) \wedge \mathsf{AtY}(y_2) \wedge \tilde{\pm}\mathsf{H}$$

with lower cost bound $2(|y_2 - y_1| + |x_2 - x_1|)$. That is, when the agent is at (x_1, y_1), this action can move the agent to (x_2, y_2), possibly flipping the switch along the way, with cost at least twice the Manhattan distance between the positions.

Similarly, a pessimistic description of $\textsc{Go}(x_1, y_1, x_2, y_2)$ might assume that the agent makes no further FLIPs, using two clauses:

$$\mathsf{AtX}(x_1) \wedge \mathsf{AtY}(y_1) \wedge \mathsf{H} \rightarrow \neg\mathsf{AtX}(x_1) \wedge \neg\mathsf{AtY}(y_1) \wedge \mathsf{AtX}(x_2) \wedge \mathsf{AtY}(y_2) \quad \text{and}$$

$$\mathsf{AtX}(x_1) \wedge \mathsf{AtY}(y_1) \wedge \neg\mathsf{H} \rightarrow \neg\mathsf{AtX}(x_1) \wedge \neg\mathsf{AtY}(y_1) \wedge \mathsf{AtX}(x_2) \wedge \mathsf{AtY}(y_2),$$

where the first clause has upper cost bound $4|y_2 - y_1| + 2|x_2 - x_1|$ and the second has bound $2|y_2 - y_1| + 4|x_2 - x_1|$

To describe the semantics of these descriptions, we explain how they generate an explicitly enumerated output valuation from a single input state. First, the set of NCSTRIPS clauses is filtered, retaining only clauses whose preconditions are met by the input state. Then, the effect description each remaining clause is expanded out into a set of ordinary STRIPS effects, one per subset of the possible effects. Each such effect is applied to the initial state, generating a reachable state with cost generated by the cost expression of the corresponding clause. Finally, any duplicate states are collapsed, retaining the minimum cost over all clauses that generated a state.

To generate the output valuation of an NCSTRIPS description from a *set* of input states represented by a DNF formula, we could first expand the formula out into an explicit set of states, apply this procedure to each state, and then disjoin the results (via Definition 4.8). However, this procedure would have runtime linear in the number of states allowed by the input formula, which can be *exponentially* larger than its representation. We thus describe an alternative procedure, which operates directly on the DNF representation of the input set and computes an output DNF reachable set in time *linear* in the input size.

This algorithm operates by progressing each pair consisting of a conjunctive clause of the input DNF formula and a clause of the description separately, and then disjoining the results. Each pair is progressed by:

1. conjoining description clause preconditions onto the DNF clause (and skipping this pair if a contradiction is created),
2. making all added (resp. deleted) literals true (resp. false), and finally
3. removing literals from the clause if false (resp. true) and possibly-added (resp. possibly-deleted).

Moreover, each such pair also produces a cost bound. When progressing optimistic (resp. pessimistic) valuations, we simply take the min (resp. max) of all these bounds to get the final cost bound, producing an output simple valuation. Regression through NCSTRIPS descriptions can be implemented similarly.

As a concrete example, consider progressing the reachable set

$$\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \mathsf{AtY}(0) \wedge \neg\mathsf{AtY}(1)$$

(where H is unknown) through the above descriptions of $\textsc{Go}(0, 0, 0, 1)$. The optimistic description has only a single clause, which yields an optimistic result of

$$\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \neg\mathsf{AtY}(0) \wedge \mathsf{AtY}(1)$$

with cost 2. The pessimistic description has two clauses; after conjoining the preconditions and applying the effects, these produce the output clauses and costs

$$\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \neg\mathsf{AtY}(0) \wedge \mathsf{AtY}(1) \wedge \mathsf{H}$$

with cost 4 and

$$\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \neg\mathsf{AtY}(0) \wedge \mathsf{AtY}(1) \wedge \neg\mathsf{H}$$

with cost 2. Combining these and projecting back to a simple valuation gives the final result

$$\begin{array}{l}(\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \neg\mathsf{AtY}(0) \wedge \mathsf{AtY}(1) \wedge \mathsf{H}) \quad \vee \\ (\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \neg\mathsf{AtY}(0) \wedge \mathsf{AtY}(1) \wedge \neg\mathsf{H})\end{array}$$

with cost 4, which could be further simplified to just

$$\mathsf{AtX}(0) \wedge \neg\mathsf{AtX}(1) \wedge \neg\mathsf{AtY}(0) \wedge \mathsf{AtY}(1).$$

The existence of this simple, efficient progression algorithm is a primary reason for considering this particular combination of representations (NCSTRIPS descriptions and DNF formulae). However, these representations have a number of disadvantages as well. First, since each (DNF clause, NCSTRIPS description clause) pair can generate a clause in the final DNF reachable set, the number of DNF clauses (and thus reachable set representation size) can grow exponentially with plan length. Fortunately, if a reachable set description grows too large, we can always choose to simplify its formula at runtime. For example, we can always drop clauses from pessimistic formulae, or drop variables from optimistic clauses (and then merge duplicate or subsumed clauses) without sacrificing correctness.

More importantly, NCSTRIPS descriptions and simple valuations with DNF reachable sets do not allow for compact representations of *correlations* between uncertain variable values, and between uncertain variable values and costs. While in some domains (such as

nav-switch) this is not problematic, in other domains it may prevent the effective use of angelic techniques. For instance, consider an HLA GoShopping corresponding to a trip to the grocery store where any number of items might be purchased, and where the cost of the trip is the dollar amount spent. An optimistic simple valuation of GoShopping would have to include the possibility that the agent bought every item in the store with a lower cost bound of 0, because it is also possible that the agent might come home empty-handed. To effectively handle such examples, more expressive representations for descriptions and valuations are needed that can take these correlations into account.

One simple extension would be to replace the single cost bound in simple valuations with an arithmetic expression, for example a weighted linear combination of indicator variables (e.g., GoShopping costs $2 \times \mathsf{BoughtMilk} + 3 \times \mathsf{BoughtCookies}$). Affine algebraic decision diagrams (AADDs, (Sanner and McAllester, 2005)), an extension of BDDs that include costs, present an even more promising alternative, enabling compact specifications of both reachable sets and costs with additive or multiplicative structure. This is a potentially fruitful area for future work, which may be key for enabling declarative specifications of angelic descriptions that are useful in many practical domains.

4.3.3 Procedural Specifications

Even with the extensions just described, however, it could be difficult to implement compact, accurate, efficiently computable declarative descriptions for HLAs in some domains. For example, after doing the GoDrop(o) HLA in our discrete manipulation domain, the base of the robot will always reside at some position within grasping distance of some destination location of o. Expressing this fact in NCSTRIPS would, among other things, require the ability to express universally quantification in NCSTRIPS description clauses (e.g., allowing clauses to introduce their own intermediate variables, such as potential destination locations of o). Taking this example a step farther, a declarative description of GoDrop(o) for a real robot could be extremely complex, and it is unlikely that a generic inference algorithm operating on this description could compete with, e.g., specialized inverse kinematics algorithms typically used to find feasible robot arm configurations in existing mobile manipulation systems.

These examples suggest that in some cases, it may be desirable to abandon intermediate declarative representations, and simply write code for each HLA type that directly implements the interface of Section 4.3.1. This section describes this procedural approach, which we have used to implement our discrete manipulation hierarchy (as well as a preliminary application to a physical robot, discussed in Chapter 7). The advantages of the approach include its simplicity, generality, and easy integration with existing specialized inference procedures (e.g., inverse kinematics for feasible grasps). On the other hand, procedural specifications may be more difficult to write, verify, and compose than the declarative

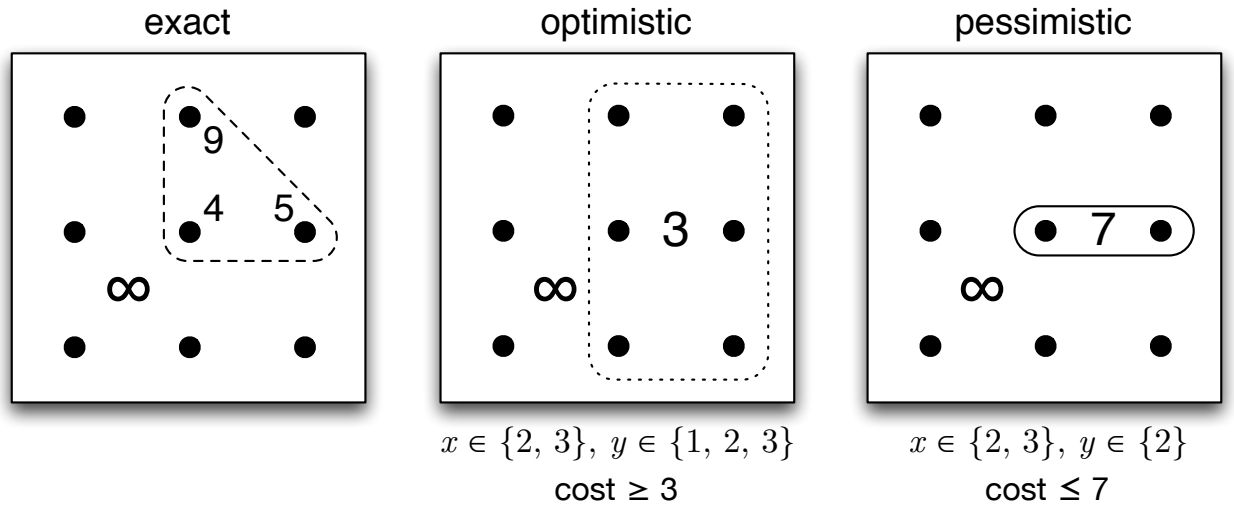

Figure 4.7: Possible *factored, simple* valuations for the example in Figure 4.6.

definitions discussed in the previous section.

To implement this approach, we must still choose a particular representation for reachable sets that the implemented procedures for each HLA type will operate on. In our implementation of the discrete manipulation domain, we use a *factored reachable set* representation that generalizes DNF clauses for the case of multi-valued state variables (i.e., SAS$^+$).

Definition 4.16. *A* factored state set S *is defined by, for each state variable* $v \in \mathcal{V}$*, a* set $S[v] \subseteq \mathcal{D}_v$ *of allowed domain values. The extension of this set is* $\bigotimes_{v \in \mathcal{V}} S[v]$.

This representation is quite restrictive, but has the significant advantages of simplicity and guaranteed compactness: each valuation is just a list of allowed values for each state variable, plus a single cost bound. Figure 4.7 shows example simple, factored valuations for the exact valuation depicted in Figure 4.6.

Given this representational choice, the procedural implementations for HLAs in the discrete manipulation domain are fairly straightforward.

As a simple example, our optimistic description of $\textsc{Nav}(x, y)$ reaches the destination position with cost equal to twice the minimum Manhattan distance from a possible current position (recall that each move step has cost 2). Its pessimistic description is vacuous (i.e., its pessimistic output set is empty), its immediate refinements are just those specified in Section 2.3.2.2, and its context retains only possible values for BaseAt and Parked.

A more interesting example is $\textsc{GoDrop}(o)$, which again is always refinable with refinements from Section 2.3.2.2 and a vacuous pessimistic description. Its context includes BaseAt, Parked, GripperOffset, Holding, and $\mathsf{Free}(x, y)$ for all positions (x, y) between the gripper and base, or within radius r of an allowed drop position for o. Finally, its optimistic

outcome always has the robot holding o, with its base and gripper offsets changed to the sets of all values that can result after a legal drop of o (see Figure 5.8). The corresponding optimistic cost is the minimum of two possible values.

1. If the robot's current base position is a possible destination position (i.e., the robot can possibly drop off the object without moving its base): the sum of the put cost, and the minimum Manhattan distance between a current gripper location and a location neighboring a dropoff location for o, and
2. The sum of
   - the minimum Manhattan distance between the current gripper offset and $(0, 0)$ (to move the gripper home before a base movement),
   - the cost to park and unpark the base,
   - the minimum over all destination base locations of the sum of:
     - twice the smallest Manhattan distance from a current possible base location, and
     - the Manhattan distance for the gripper from the destination base position to the nearest dropoff position for o, and
   - the cost of the put action itself (1).

In effect, we efficiently compute a lower bound on the cost to drop object o by computing simple functions on the input reachable set and the output reachable set, without ever having to enumerate the (potentially numerous) elements of either set. This description can be quite accurate for distant drop destinations where the cost is dominated by large navigation distances, or quite weak when many manipulations must be done in close quarters. However, by refining the action and selecting concrete base and drop locations, much of the slack in the descriptions is quickly removed.

Finally, our implementation of $\mathcal{H}_0$ optimistically reaches the goal state, with optimistic cost bound computed with a bipartite matching heuristic that estimates the minimum cost to put away all of the remaining objects.

4.4 Origins of Descriptions

This section elaborates on two aspects of the gap between high-level and primitive action descriptions, in light of the angelic approach.

4.4.1 Computing Descriptions

The bulk of this thesis assumes that HLA descriptions are given as part of the hierarchy, relying on the implementer to ensure their correctness. Algorithms for automatically computing HLA descriptions from the structure of a hierarchy could thus significantly ease the process of hierarchy design. Moreover, when combined with algorithms for automatic induction of hierarchical structure, they could enable fully domain-independent angelic hierarchical planning.

This section describes a simple algorithm for computing exact reachability descriptions (ignoring costs) in fully grounded domains, starting from the structure of the hierarchy and the descriptions of the primitive actions. This algorithm is not likely to be useful in practice, because it lacks the ability to produce approximate or lifted descriptions (i.e., it generates a separate exact description for each instantiation of each HLA, rather than a single description per HLA type that applies across all possible arguments to the HLA). Nevertheless, we include it as a potential first step towards reaching the above goals.

A first observation is that in a finite, fixed planning domain with a finite, FZCF hierarchy, it is theoretically easy to compute exact descriptions for all of the HLAs. For instance, one could simply apply DSH-UCS to find an optimal solution set for each HLA from each initial state (at great computational cost).

The remainder of this section describes a more efficient *bottom-up* method to compute these descriptions, which can make use of symbolic representations and exploit the compositional relationships between HLA descriptions in a domain. We assume that descriptions are represented *implicitly*, specifying the set of allowable transitions (s, s') using, e.g., logical formulae or BDDs (as mentioned in Section 2.1.3.1) over two copies of the propositions of the domain (one for s and one for s'). Then, the exact descriptions of the primitive actions are known, and the description of each HLA h must obey the equation

$$E_h = \bigcup_{\mathbf{a} \in \mathcal{I}(h)} E_{a_{|\mathbf{a}|}} \circ ... \circ E_{a_1}.$$

This defines a system of equations for the HLA descriptions. If the hierarchy is not recursive, this system can be evaluated directly to compute descriptions of all HLAs (starting at the lowest level, and proceeding upwards to $\mathcal{H}_0$).

In recursive hierarchies the equations are coupled, however, and this approach does not apply. Algorithm 4.2 shows an algorithm that works in general hierarchies, via an application of the Kleene fixed-point theorem. The algorithm begins by initializing the primitive descriptions to their exact values and the HLA descriptions to the empty set (e.g., false). It also creates a set "*updated*" of actions whose description information may be inconsistent with their parents, initially containing the set of primitives $\mathcal{A}$. Then, at each step the al-

Algorithm 4.2 Computing Exact Descriptions

```
function ComputeExactDescriptions()
    for a ∈ 𝒜 do
        E_a ← {(s, s') : s_⊥ ≠ s' = 𝒯(s, a)}
    for h ∈ Â do
        E_h ← ∅
    updated ← 𝒜
    while updated ≠ ∅ do
        a ← updated.RemoveFirst()
        for each HLA h and refinement a ∈ ℐ(h) where a ∈ a do
            new ← E_{a_|a|} ∘ ... ∘ E_{a_1}
            if E_h \ new ≠ ∅ then
                E_h ← E_h ∪ new
                updated.Insert(h)
    return E.
```

gorithm removes one action a from *updated*, computes the most current description of each refinement of each HLA h that includes a, adds any newly discovered transitions to the description of h, and finally adds h to *updated* if its description was changed. The algorithm is guaranteed to terminate with the correct descriptions since there are a finite number of actions, and each action's transition model can only be updated a finite number of times (once per possible transition, of which there are a finite number given the finite state space).

A variant of Algorithm 4.2 could also be used to compute optimistic and pessimistic descriptions, by allowing *simplification* steps that approximate a current description by a more compact upper or lower bound. Costs could also be incorporated, by using symbolic-numeric representations such as AADDs. This raises the difficult problem of how to automatically trade off description size with accuracy, however (a largely unsolved problem).

Moreover, to be useful in practice, we require *lifted* algorithms that can discover compact, approximate descriptions of the sort described in Section 4.3.2, which will be applicable across all problem instances in a given domain. While one could attempt to apply Algorithm 4.2 directly using a lifted transition function representation such as PDDL (an more expressive extension of STRIPS) or even first-order decision diagrams, there is no guarantee of termination in such cases. Effective algorithms for this case may have to move beyond the strictly deductive approach of Algorithm 4.2 to include *inductive* generalizations, which can be necessary for generating useful descriptions of cyclic HLAs such as our Manhattan-distance-based optimistic descriptions for Navigation HLAs.

4.4.2 Angelic Semantics, STRIPS, and the Real World

Somewhat farther afield, it is also of interest to compare angelic descriptions of HLAs — which correctly capture HLA semantics with respect to primitive, e.g., STRIPS, transition models — with the "primitive" STRIPS descriptions themselves, which are often intended to correctly capture real-world processes whose true transition models are given by the laws of physics. For instance, consider the "primitive" action PUTL(·) in our discrete manipulation domain. Our SAS$^+$ description of this action includes a precondition requiring that the robot's gripper grasps an object in a particular grid cell, and an effect stating that the object is now resting on the table in the cell to the left of the gripper. This description necessarily glosses over many details of the real world in order to make the problem description and solution processes practical (a form of *state abstraction*, as discussed in Section 2.3.1.1 or Section 3.1.2).[7] In particular, in a physical instantiation of this domain, both precondition and effect descriptions would correspond to an infinite set of states of the real world, including continuous object positions, joint angles, internal gear positions, and so on.

What, then, are the semantics of the "primitive" description in terms of these sets? Since we would like the plans discovered by our planning system to actually work in the real world, they must in some sense be analogous to pessimistic descriptions. Recall that pessimistic descriptions require that, for *every* state in the output set, there *exists* a state in the input set from which it can be reached (by some primitive refinement). This seems to be too strong a statement for our example, however; no doubt, there are unreachable world states that satisfy the output description, including those with infeasible joint angles, where the objects, gripper, or table are in collision, the internal gears are in impossible configurations with respect to the joint angles, and so on. Instead, it seems that the primitive STRIPS descriptions have a different interpretation, which we call *intersecting* semantics. Such descriptions state that, for *every* state in the *input* set, there *exists* a state in the output state that it can reach, essentially corresponding to ordinary (demonic) nondeterminism. In other words, in classical planning, primitive actions represent a line below which the world is not assumed to be fully controllable (i.e., the planning algorithm is not allowed to affect the low-level controllers used to implement the primitive actions), whereas everything above this line is fully controllable by the agent.

As one might expect, intersecting descriptions can be chained according to the same basic theorems as pessimistic descriptions, at least with respect to reachability. The primary difference is that since they are essentially a worst-case analysis, intersecting descriptions trade the ability to represent all solutions for the ease and compactness of representation of the existence of *any* solution. Thus, explicit *combinations* of angelic and intersecting descriptions seem like a particularly fruitful area for future work, offering the potential to reap the computational benefits of intersecting descriptions without sacrificing low-level completeness

[7] Simon (1962) discussed this issue at some length.

or solution quality. For example, by combining angelic uncertainty where applicable with adversarial (intersecting) nondeterminism, *pessimistic-intersecting* descriptions could accurately capture the important possibilities available to a planning agent, while easing the representational burden by ignoring the less important details. Marthi et al. (2007a) provide an overview of such pessimistic-intersecting descriptions, further elaboration of which we leave for future work.

In light of this discussion, we can aim to provide an answer to the question "why angelic hierarchical planning?" In short, angelic hierarchical planning allows us to leverage *incomplete* or *approximate* knowledge about high-level tasks. If we could express our discrete manipulation domain in STRIPS at a higher (e.g., blocks-world-like) level that omitted details like precise object positions, and perhaps even the robot itself, without sacrificing completeness or solution quality too much, we would almost certainly prefer that. This is simply too much to ask for in general, however: detailed reasoning about object and robot positions is sometimes needed to ensure feasibility (e.g., in cluttered environments), and can be crucial for finding high-quality solutions. On the other hand, many of the high-level tasks encountered in a real planning instance *are* easy: free-space navigation, or picking up an object on the edge of an uncluttered table, for example. Angelic hierarchical planning allows us to express the primitives of a problem at a low enough level that solution quality and feasibility can be guaranteed, while automatically identifying and leveraging knowledge about such easy subproblems that appear during planning (see Figure 4.8). This enables planning agents that reason at the highest level of abstraction possible for a given task *in a given context*, rather than having to settle on a single fixed abstraction level in advance for an entire problem.

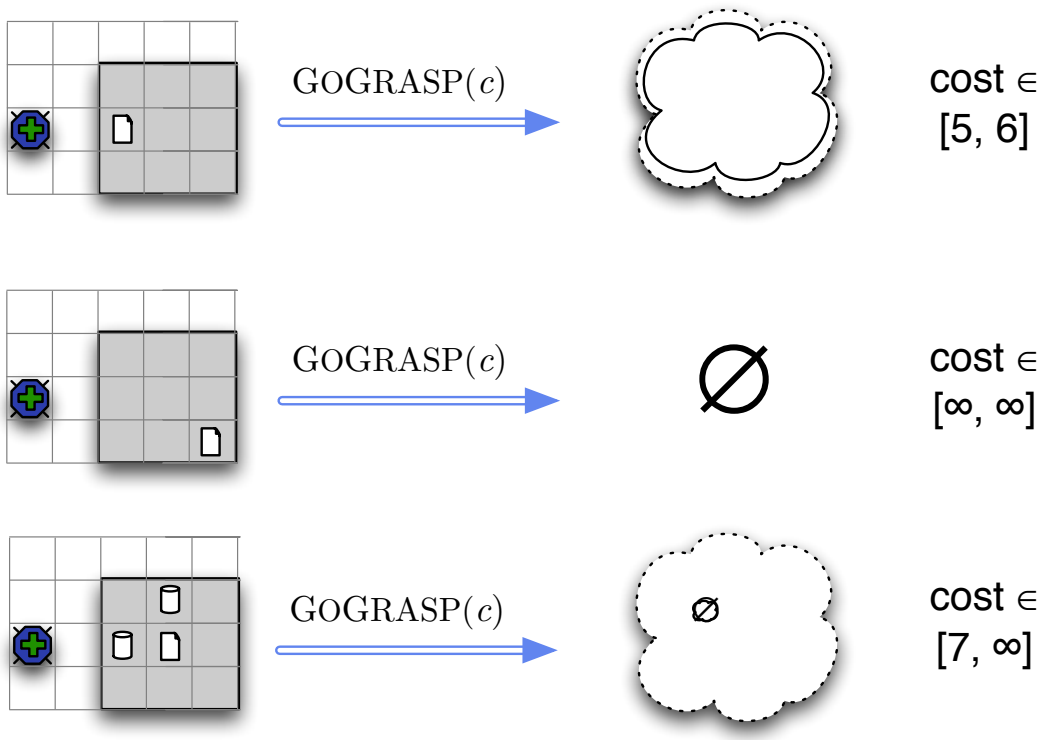

Figure 4.8: An illustration of how angelic reasoning can help focus computational effort on the hard problems encountered during search. The figure shows schematic depictions of potential optimistic and pessimistic reachable sets and cost bounds for the GoGrasp(c) HLA from three different states. Top: the object is on the edge of the table, and can be easily grasped. Simple angelic descriptions can likely capture this fact, yielding equal optimistic and pessimistic sets (corresponding to states after the grasp) with tight cost bounds (i.e., the grasp is definitely feasible with cost at most 6, and must cost at least 5). Center: the object is clearly out of reach, and cannot be grasped. Again, simple angelic descriptions can likely capture this fact, producing empty reachable sets and infinite cost bounds. In both of these "easy" cases, the agent need not think about the low-level details of the GoGrasp(c) HLA further (unless it is fairly certain that the top action will be part of its final plan, and the precise cost ends up mattering to determine its optimality). Bottom: The object is just within reach and there is a lot of clutter. In this case, the descriptions may provide weak information: the grasp may be feasible with cost at least 7, or it may be infeasible. Thus, further refinement of the GoGrasp(c) HLA is needed to accurately determine its feasibility and cost from this state.

Chapter 5

Angelic Hierarchical Planning

The previous chapter presented an "angelic" framework for generating bounds on the reachable sets and costs of high-level plans. This chapter introduces several classes of algorithms that can exploit these bounds to find hierarchically optimal solutions efficiently.

We first describe Angelic Hierarchical A* (AH-A*), an extension of H-UCS (see Section 2.3.2.3) that directly searches over potential high-level sequences and uses optimistic and pessimistic bounds to guide the search and prune provably dominated plans.

Then, we move on to algorithms that combine angelic descriptions with decomposition and state abstraction (see Chapter 3), which turn out to be naturally complementary. Starting with the simple setting in which each HLA can reach at most a single state, we show that the pairing of angelic bounds with decomposition generates AND/OR graphs that can be searched using the techniques of Section 2.2 for hierarchically optimal solutions. We generalize this algorithm to work on arbitrary hierarchies, introducing a number of novel extensions and modifications along the way. The resulting Decomposed, Angelic, State-abstracted Hierarchical A* (DASH-A*) algorithm simultaneously reasons about subproblems at various levels of state and action abstraction, and can find hierarchically optimal plans *exponentially faster* than previous algorithms (including the others in this thesis).

Finally, we briefly describe modifications of these hierarchically optimal algorithms for *bounded suboptimal* search. The resulting algorithms can find solutions faster at the expense of solution quality, and may be a better fit for some practical applications.

5.1 Angelic Hierarchical A*

5.1.1 Optimistic AH-A*

Section 2.3.2.3 presented a simple, hierarchically optimal algorithm called H-UCS, which performs a state-space-style search over "forward subproblems" $(s, \mathbf{a})$ consisting of a state s and a remaining high-level plan $\mathbf{a}$ to do from s. The queue of H-UCS was ordered by the total cost of the primitive actions executed thus far to reach state s, without regard for the remaining sequence $\mathbf{a}$. This section describes the Angelic Hierarchical A* (AH-A*) algorithm, which in its simplest "optimistic" incarnation converts H-UCS to an A* algorithm by charging for the remaining sequence $\mathbf{a}$ using its optimistic cost bound. More formally, the heuristic $h((s, \mathbf{a}))$ is given by $O_{\mathbf{a}}(s)(s_*)$, the optimistic bound on the cost of the cheapest primitive refinement of $\mathbf{a}$ that reaches the goal from s.

The following section describe how pessimistic bounds can be incorporated into AH-A*, but we first elaborate on several aspects of the "optimistic" version.

First, the bounds $O_{\mathbf{a}}(s)(s_*)$ can be computed using a simple algorithm based on progression (e.g., see Section 4.3.2.3). We start with a valuation v_0 with $v_0[s] = 0$ and $v_0[s'] = \infty$ for all other states s', compute $v_1 = O_{a_1}(v_0)$, ..., $v_n = O_{a_n}(v_{n-1})$, and then look up the final bound $v_n(s_*)$ (see Figure 5.1). The number of progression steps is linear in the length of $\mathbf{a}$, and in the worst case the intermediate valuations may become large (depending on the representations and simplifications employed). These costs can be mitigated somewhat by caching the valuations for steps, prefixes, and/or plan suffixes.[1]

Second, in AH-A* (unlike in H-UCS) it may be desirable to refine an action sequence at an HLA other than the first. Recall that in an A*-style algorithm, the basic goal is to increase the cost bounds of plans as quickly as possible (while maintaining correctness). Since refining an HLA may increase the bound for a plan by increasing the accuracy of its optimistic bounds (in addition to generating primitive prefixes as in H-UCS), one might suppose that the best strategy for AH-A* would be *balanced* expansion, i.e., always refining the highest-level action remaining. However, in general there will be a tension between refining the first HLA in order to generate applicable primitive prefixes and thus get better state-space pruning (from elimination of repeated subproblems), and refining in a balanced manner to more quickly increase the bounds on plans. We discuss this issue further after introducing the full version of AH-A* in the next section.

Finally, we prove the correctness of this "Optimistic AH-A*" algorithm. Because we are no longer restricted to "forward" algorithms that always refine the first HLA in a sequence,

[1] Prefix caching was the primary motivation behind abstract lookahead trees (Marthi et al., 2009), which also supported an improvement called "upward propagation" that is subsumed by our later discussion of decomposed algorithms.

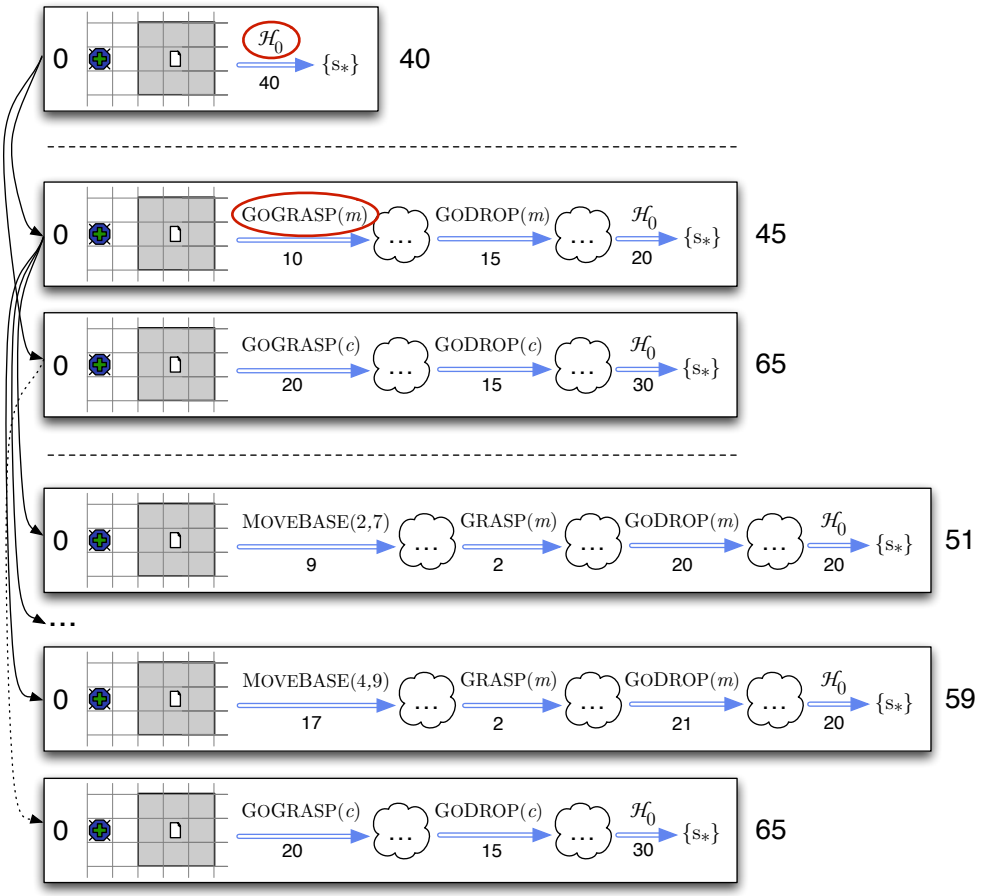

Figure 5.1: The fringe of Optimistic AH-A* for the first two search steps on the discrete manipulation instance of Figure 1.1. Clouds represent implicit optimistic reachable sets, and each action and plan is labeled with an optimistic cost bound. The algorithm begins with the plan $[\mathcal{H}_0]$ (top), and refines $\mathcal{H}_0$ to generate the two possible task orderings (middle). The optimistic cost bound of the first plan is smaller, and so the algorithm refines the first HLA in this plan, generating two new plans (bottom). Refinement of these plans continues until an optimal solution is found. In fact, the bottom plan will never be refined, because its optimistic cost bound of 65 is greater than the hierarchically optimal solution cost of 60.

we require a new cycle-free criteria for this setting:

Definition 5.1. *A hierarchy is* Optimistic Zero-cost Cycle-Free *(OZCF) iff, for each finite cost c, there are only a finite number of hierarchical plans with optimistic cost bound less than or equal to c:*

$$(\forall c) \;\; |\{\mathbf{a} : \mathbf{a} \in \mathcal{I}^+(\mathcal{H}_0) \text{ and } O_{\mathbf{a}}(s_0)(s_*) \leq c\}| < \infty$$

This class of OZCF hierarchies includes all FCZF hierarchies, so long as the optimistic descriptions do not assign zero bounds to HLAs with necessarily nonzero cost (from a given state), as well as other hierarchies that do not meet the FCZF restriction.

Theorem 5.1. *Optimistic AH-A* is* hierarchically optimal *in OZCF hierarchies with consistent optimistic descriptions.*

Proof. Optimistic AH-A* is simply A*, applied to hierarchical plan space. Corollary 4.9 guarantees that the bounds assigned by the optimistic descriptions are admissible, consistency of the optimistic descriptions entails consistency of the heuristic, and the OZCF restriction ensures that the algorithm must terminate so long as a finite-cost solution exists. □

Because of the approximation introduced by projecting into simple valuations, our optimistic descriptions are *not* actually consistent in the discrete manipulation domain. However, our descriptions exhibit a weaker property that we will call *weak consistency*, which is sufficient to ensure the correctness of Optimistic AH-A*.[2]

Definition 5.2. *Consider a sequence of refinements of* $(s_0, \mathcal{H}_0)$ *leading to subproblem* $p = (s, \mathbf{a})$*. Define the* derived heuristic *value for* $\mathbf{a}$ *from* s *as the maximum cost bound of any subproblem in this sequence, minus the cost of the primitive actions taken to reach* s*. If, for every subproblem* p*, the derived heuristic value is the same for* all *paths reaching* p *from* $\mathcal{P}_0$*, then the optimistic descriptions are* weakly consistent*.*

Corollary 5.2. *Optimistic AH-A* is hierarchically optimal in OZCF hierarchies with weakly consistent optimistic descriptions.*

Proof. Weak consistency ensures that when a subproblem $p = (s, \mathbf{a})$ is popped from the fringe, p must consist of a hierarchically optimal path to reach s among all possibilities that can be followed by $\mathbf{a}$. Suppose that p was popped from the fringe with cost c to reach s and derived heuristic value h. Furthermore, suppose that another potential subproblem p' was on the fringe, which could reach p with strictly lower cost $c' < c$ to s. By weak consistency,

[2]Specifically, our descriptions are weakly consistent when the first HLA in a sequence is always refined, which is the strategy used by Optimistic AH-A* in our experiments.

the total cost bound for p' (and all other subproblems on its path from $(s_0, \mathcal{H}_0)$ to p) must be $\leq c' + h < c + h$. Also by weak consistency, some ancestor of p on the popped path must have cost bound $= c + h$. This is a contradiction, because all nodes on the path through p' would have been expanded before this ancestor due to cost ordering of the fringe. □

5.1.2 AH-A* with Pessimistic Descriptions

As the theorems of the previous chapter might suggest, there are several ways to incorporate pessimistic descriptions into AH-A*. We first describe simple applications of pessimistic descriptions for tie-breaking, commitment, and solution decomposition, and explain why these cannot usually be of much help for hierarchically optimal search. Then, we describe a more powerful application of pessimistic descriptions that lies outside the usual A* framework, using domination to prune high-level plans with provably suboptimal *prefixes*.

5.1.2.1 Tie-breaking, Commitment, and Solution Decomposition

A first, simple idea is to use pessimistic bounds for *tie-breaking*. When choosing which HLA to refine in a given plan, a reasonable heuristic might be to refine the *most uncertain* HLA — that is, the HLA with the greatest gap between the cost bounds computed by its optimistic and pessimistic descriptions — in an attempt to raise the optimistic bound on the plan as much as possible. Moreover, breaking ties on the priority queue by pessimistic cost can help reduce the number of refinements considered for plans with optimistic bound equal to the hierarchically optimal solution cost, by (hopefully) ordering the optimal plans first.

Pessimistic descriptions could also be used for *commitment* and *solution decomposition*, via Corollary 4.9 and Theorem 4.10. Specifically, the first time a subproblem $(s, \mathbf{a})$ is popped from the queue with $O_{\mathbf{a}}(s)(s_*) = P_{\mathbf{a}}(s)(s_*)$, it must be hierarchically optimal, and we could commit to and decompose it as in Section 4.1.3.

However, these techniques cannot actually be of much help in this setting. For one, commitment only becomes possible when the optimistic and pessimistic descriptions for $\mathbf{a}$ become *exact*, which is unlikely to occur much before $\mathbf{a}$ is fully primitive (and the problem is solved already). Moreover, if the pessimistic descriptions are consistent and ties on the priority queue are broken by $P_{\mathbf{a}}(s)(s_*)$ and then recency, then commitment actually has *no effect* in which plans are refined by AH-A* (since no non-descendants can subsequently rise to the front of the priority queue under this tie-breaking rule). Similarly, solution decomposition can only help with the constant factors involved in computing bounds on valuations, but will not actually decrease the total number of refinement operations needed to reach a hierarchically optimal solution.

Thus, we henceforth assume the above tiebreaking rule, but do not consider commitment or decomposition in AH-A*. However, we note that such techniques can still be useful when

interleaving planning and execution, or searching for a *near*-optimal solution.

5.1.2.2 Aside: Pruning and Domination in State-Space Search

We briefly set aside AH-A* and angelic descriptions, returning to the simple state-space search setting of Section 2.1. In this setting, our objective is to find a solution with cost $c^* = c^*(s_0)$, by searching over *nodes* consisting of pairs (c, s) where s is a state and c is the cost incurred to reach s. In addition to the assumptions made in Section 2.1, we also assume access to a partial *domination* relation on nodes, and prove some general results about pruning and correctness under this relation (which we apply to AH-A* in what follows).

Definition 5.3. *Node* $n_1 = (c_1, s_1)$ weakly dominates *node* $n_2 = (c_2, s_2)$*, written* $n_1 \leq n_2$*, if* $c_1 + c^*(s_1) \leq c_2 + c^*(s_2)$

Definition 5.4. *Node* $n_1 = (c_1, s_1)$ strictly dominates *node* $n_2 = (c_2, s_2)$*, written* $n_1 < n_2$*, if* $c_1 + c^*(s_1) < c_2 + c^*(s_2)$

Remark. *Strict domination entails weak domination:* $n_1 < n_2 \Rightarrow n_1 \leq n_2$.

Suppose we get to observe only *portions* of these relations (if we had full access to them, optimal planning would be trivial). This section proves general results for how these observations can be used to *prune* nodes from a state-space search, without sacrificing optimality.

For strict domination, this is simple.

Theorem 5.3. *If* $n_1 < n_2$*, then* n_2 *can be safely* pruned.

Proof. Strict domination entails that no successor of n_2 can be optimal. □

Exploiting weak domination requires greater subtlety, however, because we must avoid *pruning cycles* that could eliminate all of the optimal nodes from the search space (see Figure 5.2). To enable correct pruning in this case, we first define a structure called the *delegation graph*.

Definition 5.5. *A* delegation graph *is a graph on the constructed nodes of a search problem. The* initial graph *contains the initial node* $(0, s_0)$ *and a sink node* $(\infty, s_\perp)$*. A* leaf node *is a node other than the sink with no outgoing edges. Three graph modification operations are available, which can be applied to any leaf node* n*:*

1. n *can be* expanded*, adding each successor node* n' *of* n *to the graph along with an edge from* n *to* n'*.*

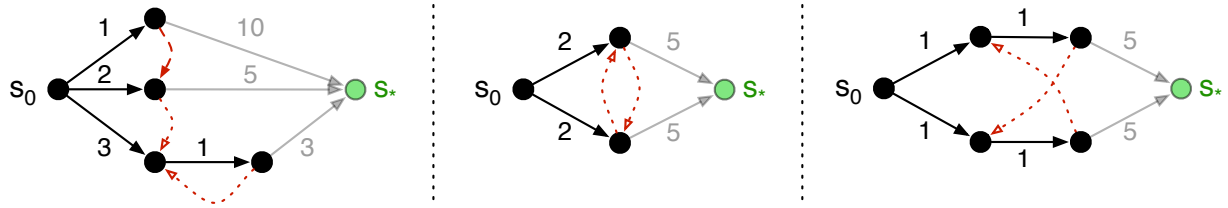

Figure 5.2: Examples of pruning and domination. Black circles and lines represent generated states and successor relations with step costs, respectively. Grey lines represent as-yet ungenerated paths to the goal state s_*. Dashed and dotted red, curved lines represent known strict and weak domination relationships (respectively). In the left graph, the top node can be safely pruned based on strict domination by the middle node, and the middle node can be safely pruned based on weak domination by the bottom-left node. The bottom-right leaf node *cannot* be pruned, however, because it is only weakly dominated by an ancestor. In the middle graph, *either one* of (but not both) nodes can be safely pruned based on mutual weak domination. Similarly, only one leaf node can be safely pruned from the right graph.

2. *n can be* strictly pruned *when $n' < n$ is observed for some other node n' in the graph, by adding an edge from n to the sink node.*

3. *n can be* weakly pruned *when $n' \leq n$ is observed for some other node n' in the graph, by adding an edge from n to n', provided that this does not create a cycle in the graph.*

Remark. *Note that this graph is on the* nodes*, not the* states*, of the search problem. Repeated state elimination can be accounted for by observing $(c, s) \leq (c', s)$ when $c \leq c'$ and $(c, s) < (c', s)$ when $c < c'$. We assume there are no zero-cost cycles in the state space.*

This graph allows us to safely carry out all strict pruning and some weak pruning, while preserving optimality (see Figure 5.3).

Theorem 5.4. *After any sequence of legal operations leading to delegation graph $\mathcal{G}$, at least one leaf node n exists that can be expanded to an optimal solution.*

Proof. Call a node $n = (c, s)$ of $\mathcal{G}$ *optimal* iff $c + c^*(s) = c^*(s_0)$. We prove the following invariant on the graph: for each optimal node n in $\mathcal{G}$, there exists a path in $\mathcal{G}$ from n to some optimal leaf node n'. If n is a leaf node this is trivially true ($n = n'$), so the invariant holds for the initial graph. Expansion preserves the invariant by definition. Strict pruning preserves it because strictly pruned nodes cannot be optimal. Weakly pruning a node n on another node n' preserves it because either n is not optimal, or both n and n' are optimal by definition of weak domination. In the latter case, the invariant ensures that there must exist a path from n' to some optimal leaf node n''. The added edge from n to n' creates a path

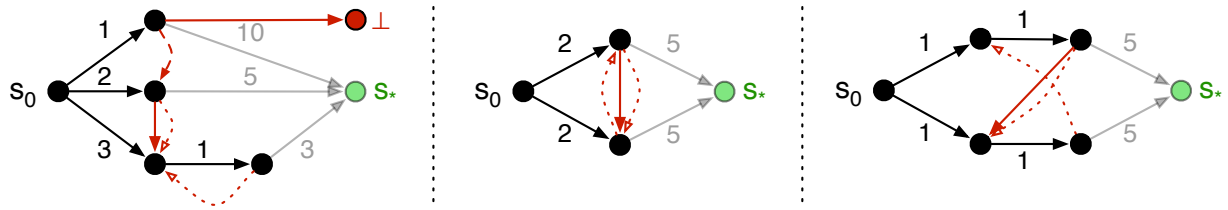

Figure 5.3: Safe pruning inferences for the examples in Figure 5.2 are correctly captured by the delegation graph. Solid red lines indicate pruning edges added to the delegation graph. The remaining unsafe weak pruning opportunities are prevented because they would create cycles in the delegation graph.

from n to n''. Moreover, because adding this edge does not create a cycle, $n \neq n''$ and n'' remains a leaf node, so the invariant is preserved. Finally, applying the invariant to $(0, s_0)$ establishes the theorem. □

5.1.2.3 AH-A* with Pruning

This pruning machinery can be applied to AH-A*, by observing that plan $\mathbf{a}$ (strictly) dominates plan $\mathbf{b}$ iff there exists a pair of prefixes for the plans where the pessimistic valuation of the prefix of $\mathbf{a}$ (strictly) dominates the optimistic valuation of the prefix of $\mathbf{b}$, and the remaining action suffixes of the plans are the same. Intuitively, this condition ensures that any state reached by a primitive refinement of $\mathbf{b}$ can also be reached by a (strictly) better primitive refinement of $\mathbf{a}$ (see Figure 5.4).

Theorem 5.5. *Consider any two action sequences* $\mathbf{a}$ *and* $\mathbf{b}$ *and indices* i *and* j *such that* $\mathbf{a}_{i+1:|\mathbf{a}|} = \mathbf{b}_{j+1:|\mathbf{b}|}$. *Then,* $\mathbf{a}$ strictly dominates $\mathbf{b}$ *from* s *if* $P_{\mathbf{a}_{1:i}}(s) \prec O_{\mathbf{b}_{1:j}}(s)$, *and* $\mathbf{a}$ weakly dominates $\mathbf{b}$ *from* s *if* $P_{\mathbf{a}_{1:i}}(s) \preceq O_{\mathbf{b}_{1:j}}(s)$.

Proof. Consider any primitive refinement $\mathbf{b}^* \in \mathcal{I}^*(\mathbf{b})$. This primitive refinement can be split into parts $\mathbf{b}_1^*$ and $\mathbf{b}_2^*$, where $\mathbf{b}_1^* \in \mathcal{I}^*(\mathbf{b}_{1:j})$ and $\mathbf{b}_2^* \in \mathcal{I}^*(\mathbf{b}_{j+1:|\mathbf{b}|})$. Because $\mathbf{a}_{i+1:|\mathbf{a}|} = \mathbf{b}_{j+1:|\mathbf{b}|}$, it must be the case that $\mathbf{b}_2^* \in \mathcal{I}^*(\mathbf{a}_{i+1:|\mathbf{a}|})$. Let $s' = \mathcal{T}(s, \mathbf{b}_1^*)$.

Now, suppose that $P_{\mathbf{a}_{1:i}}(s) \prec O_{\mathbf{b}_{1:j}}(s)$. Then, there must exist an $\mathbf{a}_1^* \in \mathcal{I}^*(\mathbf{a}_{1:i})$ s.t. $\mathcal{T}(s, \mathbf{a}_1^*) = s'$ and $\mathcal{C}(s, \mathbf{a}_1^*) \leq P_{\mathbf{a}_{1:i}}(s)(s') < O_{\mathbf{b}_{1:j}}(s)(s') \leq \mathcal{C}(s, \mathbf{b}_1^*)$. Thus, $\mathbf{a}^* := \mathbf{a}_1^* +\!\!+ \mathbf{b}_2^*$ is a primitive refinement of $\mathbf{a}$ that reaches the same state with strictly lower cost than $\mathbf{b}^*$, and so $\mathbf{a}$ strictly dominates $\mathbf{b}$. The same argument holds for weak domination, replacing $\prec$ by $\preceq$, $<$ by $\leq$, and "strictly lower" by "no greater". □

To put these definitions into practice, we require two further details: how such domination relationships should be identified, and how they should be used for pruning while avoiding

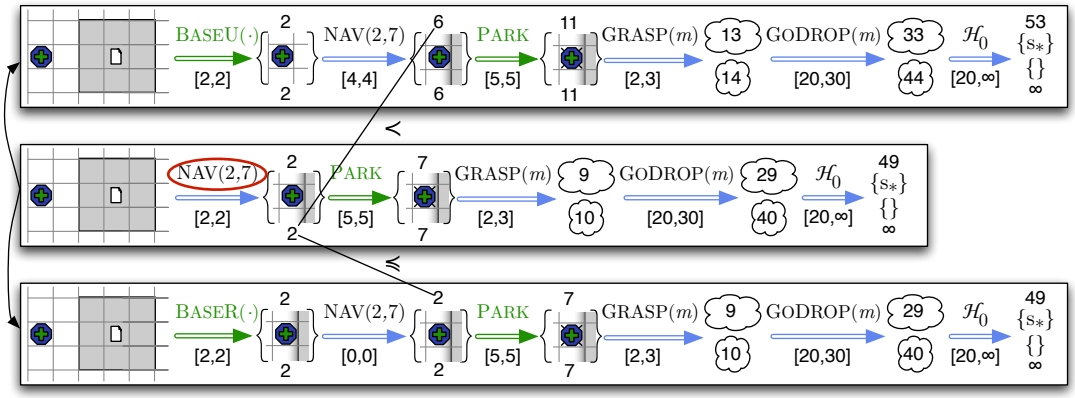

Figure 5.4: A concrete example of AH-A* pruning in the discrete manipulation domain. Refining the center plan generates the two plans at top and bottom, where the bottom plan represents the sole optimal refinement of NAV$(2, 7)$ in this context, and the top plan is a suboptimal refinement (i.e., a step in the wrong direction). Each HLA is labeled with an $[optimistic, pessimistic]$ cost interval, and the outcome of each prefix is labeled with optimistic (top) and pessimistic (bottom) simple valuations. The top plan can be strictly pruned, because its optimistic valuation after [BASEU$(\cdot)$, NAV$(2, 7)$] is strictly dominated by the pessimistic valuation after NAV$(2, 7)$ for the original plan. The bottom plan cannot be pruned, because while its optimistic valuation after NAV$(2, 7)$ is equal to (and thus weakly dominated by) the pessimistic valuation of the center plan, it is a child of that plan and so weakly pruning it would create a cycle in the delegation graph.

weak pruning cycles.

For the former problem, our general strategy will be to keep a database of pairs $(v, \mathbf{a}_2)$, one for each *prefix* $\mathbf{a}_1$ of each plan $\mathbf{a}$ constructed by AH-A*, where v is the pessimistic valuation reached by $\mathbf{a}_1$ and $\mathbf{a}_2$ is the remaining suffix (i.e., $v = P_{\mathbf{a}_1}(s_0)$ and $\mathbf{a}_1 +\!\!+ \mathbf{a}_2 \in \mathcal{I}^+(\mathcal{H}_0)$). Then, each time a plan $\mathbf{b}$ is selected for expansion, we can iterate through each prefix $\mathbf{b}_1$ and compute its *optimistic* valuation $v' = O_{\mathbf{b}_1}(s_0)$. If the database contains an entry $(v, \mathbf{b}_2)$ where $v \prec v'$ (resp. $v \preceq v'$), then this node is eligible for strict (resp. weak) pruning.

For this scheme to be practical, it should be possible to *efficiently* perform these database lookups, ideally in constant time. Since we are using simple valuations (which consist of a reachable set along with a single numeric cost bound), this would entail a way to efficiently find all entries in the database with the same plan suffix, and a *subset* of the reachable set of the query. Unfortunately, this is not possible in constant or near-constant time (Wolfe and Russell, 2007). Thus, we further simplify the problem by only looking for domination for *equal* reachable state sets, which can be implemented in constant time by hashing tuples $(S, \mathbf{a}_2)$ where S is a reachable set.

Strictly dominated action sequences identified by this procedure can be pruned immediately via Theorem 5.4. For weakly dominated sequences, we also need to avoid creating cycles in the delegation graph of Definition 5.5. One obvious way to accomplish this is to explicitly represent the delegation graph, and use efficient incremental cycle-checking methods to avoid creating cycles when weakly pruning. However, the best-known such methods have cost $O(n^{\frac{3}{2}})$ to build a graph with n nodes (Haeupler et al., 2008). Thus, we choose to forego some pruning opportunities for a simpler, easily checkable criterion: n can only be weakly pruned on n' when n' is still on the open list (i.e., a leaf node), which guarantees acyclicity and thus correctness.[3]

A final detail is that duplicate subproblem elimination (i.e., A* graph search) can not be applied together with this pruning without extra care. Elimination of duplicate subproblems is technically a special case of pruning, and may create cycles in the delegation graph if applied indiscriminately along with weak pruning. Since pruning should already (safely) capture most of the benefit gained by duplicate plan elimination, we simply turn off duplicate plan elimination when using weak pruning.

Pruning can be a powerful addition to AH-A*. In fact, pruning (including the special case of duplicate plan elimination) is the only mechanism combatting the exponential growth of potential plans at each level of AH-A*. For instance, consider a high-level plan consisting of N HLAs, each of which has two potential refinements. Refining this plan at each HLA generates 2^N descendants, one for each possible combination of refinements. Fortunately, many domination relationships will typically exist between these plans, cutting the set that

[3]More details on these ideas, including alternative strategies for correct pessimistic pruning, can be found in our previous work (Marthi et al., 2009).

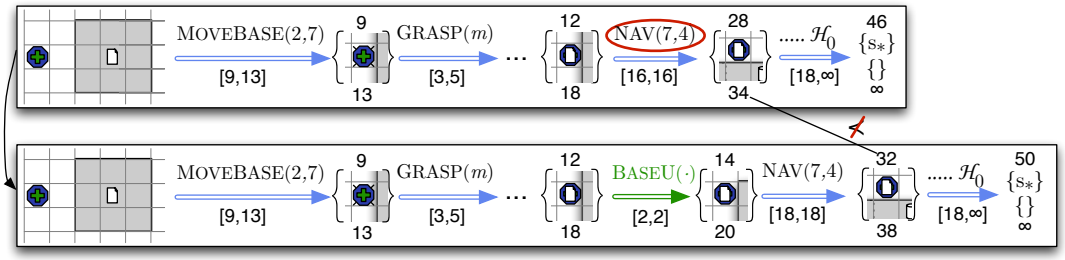

Figure 5.5: A situation similar to the top two plans of Figure 5.4, where one might expect strict pruning to apply. The circled navigation action in the top plan was refined, yielding the bottom plan (and presumably others). While the bottom plan introduces a suboptimal navigation step, and would be prunable if the actions before Nav were not present, it cannot be pruned in this case because the bounds on the earlier part of the plan are too loose.

must actually be considered down to a much more manageable size.

However, pruning on prefixes as discussed here still falls short in a number of ways. For one, as in ordinary A*, without a *perfect* heuristic (in this setting, both perfect optimistic and pessimistic descriptions) the algorithm can still be doomed to explore exponentially many plans with the optimal cost (e.g., if there are exponentially many optimal solutions). Moreover, because pruning is carried out on entire *prefixes*, the presence of an action early in the plan with loose descriptions can prevent any useful pruning on the remainder of the plan, even if the descriptions of the remaining actions are exact (see Figure 5.5). Thus, it typically remains best to refine the first HLA in a plan to maximize the opportunities for pruning, despite our desire described above to balance refinements evenly throughout a plan. These and other issues will be addressed by the fully decomposed angelic planning algorithms discussed in the next sections.

5.2 Singleton DASH-A*

This section shows how the angelic techniques of the previous section can be combined with decomposition and state abstraction, in restricted hierarchies where each HLA can optimistically reach at most one state (from each initial state).

The restriction to singleton reachable sets (which we remove in the next section) simplifies planning, because given the intermediate states $\{s_i\} := \bar{O}_{a_i}(s_{i-1})$ for a plan $\mathbf{a}$, planning for each action a_i can proceed completely independently of the other actions in $\mathbf{a}$. In particular, as we saw in Chapter 3 and Theorem 4.10, *any* concatenation of optimal primitive refinements for each a_i reaching s_i from s_{i-1} yields an optimal primitive refinement of $\mathbf{a}$.

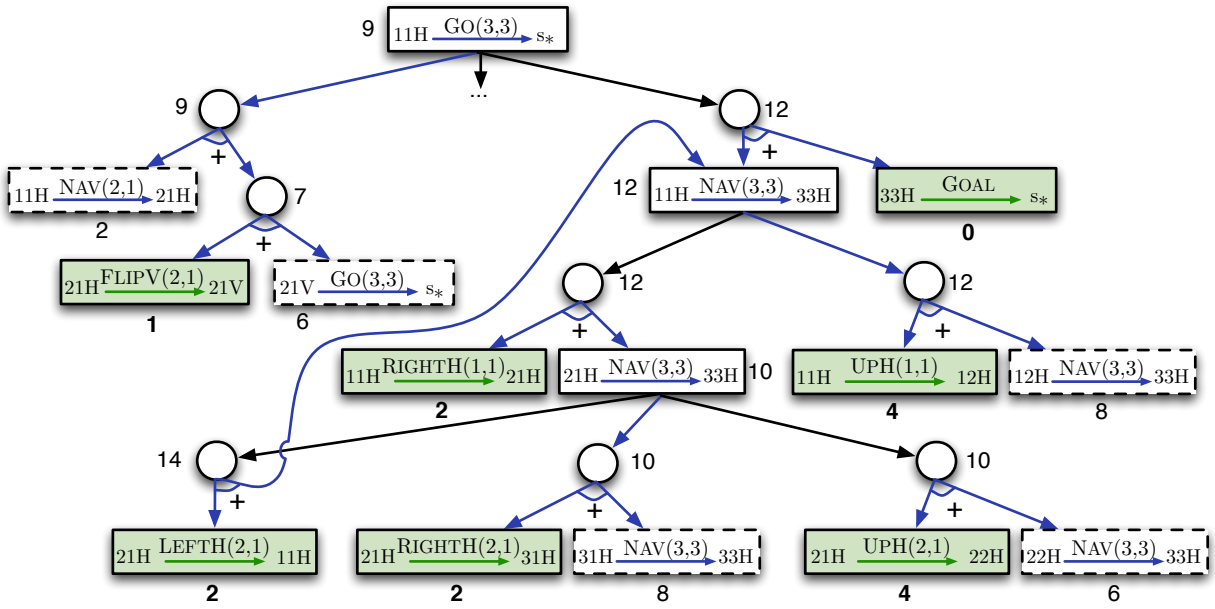

Figure 5.6: A portion of the AND/OR graph generated by singleton DASH-A* on the example nav-switch instance (see Section 2.1.2.1).

Moreover, search is even simpler than in Chapter 3, because the intermediate states for a sequence can be computed in advance using the optimistic descriptions (along with bounds on their optimal costs), rather than being discovered as each action is solved in turn. This leads to a search space that can be directly captured as an AND/OR graph in the formalism of Section 2.2.1.1. Each OR-node corresponds to selecting *some* immediate refinement for an HLA from some state, and each AND-node corresponds to finding optimal solutions for *all* action subproblems in a given immediate refinement. For example, Figure 5.6 shows a portion of the AND/OR graph corresponding to the nav-switch example presented in Section 2.1.2.1.

Formally, the AND/OR graph is defined in the formalism of Section 2.2.1.1 as follows. Subproblems are as described in Chapter 3: $p = (s, a)$ represents the task of doing the best refinement of a from s, and the initial subproblem $\mathcal{P}_0 = (s_0, \mathcal{H}_0)$. Iff a is primitive then p is terminal, and has optimal solution $\mathcal{Z}(p) = z$ with cost $c(z) = \mathcal{C}(s, a)$. Otherwise, a is high-level, $\mathcal{Z}(p) = NT$, and each immediate refinement $[b_1, b_2] \in \mathcal{I}(a)$ generates a refinement $(s, a) \rightarrow_+ (s, b_1)\ (s', b_2)$ where s' is the optimistic reachable state of b_1 from s. Finally, as in Chapter 3, state abstraction can be incorporated by simply caching OR-nodes under $(\textsc{EnterContext}(s, a), a)$ rather than (s, a) directly.

Given this graph definition, any search algorithm from Section 2.2 can be applied to search for a hierarchically optimal solution, yielding a family of algorithms we call *singleton DASH-A** (Decomposed, Angelic, State-abstracted Hierarchical A*). However, the bottom-

up algorithms are not particularly well-suited, both because the set of terminal subproblems is very large (one per (s, a) pair where $a \in \mathcal{A}_s$, modulo state abstraction), and because angelic descriptions naturally provide heuristics for top-down search. In particular, the top-down summary $\hat{z} = h(p)$ has lower bound $\hat{z}.lb = O_a(s)(s')$ (where s' is the sole reachable state). In acyclic domains any top-down algorithm can be used, but in cyclic domains an algorithm that can handle cycles (e.g., LDFS, or better, AO$*_{\text{KLD}}$) is required.

Theorem 5.6. *In OZCF hierarchies with singleton reachable sets, singleton DASH-A* with LDFS or* AO$*_{\text{KLD}}$ *is hierarchically optimal.*

Proof. This is a special case of Theorem 5.17, for the more general algorithm presented in the next section. □

Moreover, it is easy to see how this graph-based approach solves many of the problems inherent in the sequence-based approach of AH-A*. Consider a plan $[a_1, ..., a_n]$ where each HLA a_i has a single reachable state. If each a_i has k refinements, there are k^n possible plans generated by refining each HLA once, each of which could be considered separately by AH-A*. In contrast, DASH-A* represents this full space of plans *implicitly* in an AND/OR graph with just $O(kn)$ space and time, *without* the need for accurate pessimistic descriptions to enable pruning. We discuss potential uses for pessimistic descriptions in decomposed angelic search in the next section, after introducing the general DASH-A* algorithm.

Before moving to the more sophisticated techniques required for general (non-singleton) implicit angelic valuations in the next section, we briefly note that the simple approach of this section can easily be generalized to *explicit* valuations with more than one state in the reachable set, so long as reasoning about each reachable state is carried out *independently* of the others (see Figure 5.7). The basic idea is to use subproblems (s, a, s') that also specify a specific reachable state s', in addition to an action a and initial state s. Node (s, a, s') is an OR-node with one child for each refinement $[b_1, b_2]$ of a: (s, b_1, b_2, s'), which is itself an OR-node with one child for each state s'' reachable by b_1 from s: (s, b_1, s'', b_2, s'). Finally, this node is an AND-node with children (s, b_1, s'') and (s'', b_2, s'), since the sequence subproblem decomposes given concrete intermediate state s''.

Thus, the algorithm described in this section successfully augments the decomposed search strategy of Chapter 3 with the angelic heuristic bounds of Chapter 4, yielding potentially large reductions in the search space. The only price paid for these heuristic bounds, and the ability to balance refinements evenly across refinements (i.e., not just refining the first HLA in a sequence), is that a "flattened" algorithm like DSH-UCS is no longer possible and thus the top-down overhead discussed in Section 3.3.1 cannot be avoided in the same manner. The end of the next section discusses "hybrid" techniques that give up some efficiency in other places (e.g., caching) to help overcome this overhead when necessary.

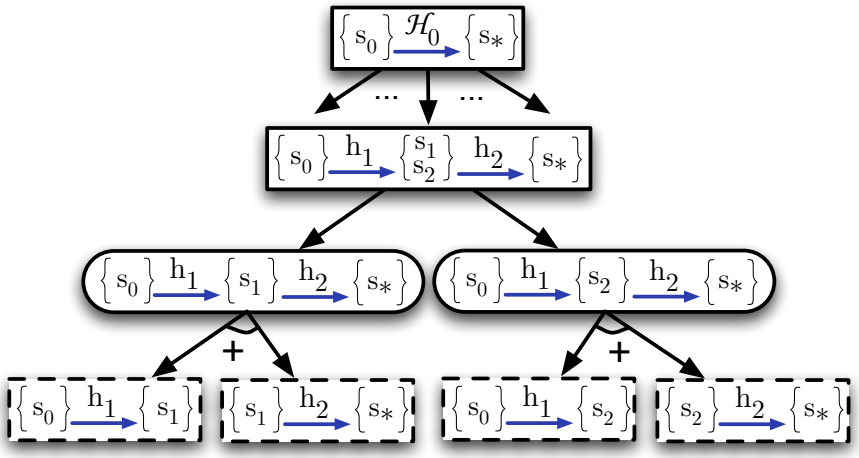

Figure 5.7: Singleton DASH-A* can easily be extended to solve problems with non-singleton reachable sets that are represented *explicitly*, by adding OR-nodes that choose a specific intermediate state for each pair subproblem.

5.3 DASH-A*

5.3.1 Introduction

While powerful, the approach taken in the previous section can fall short when HLAs have large reachable sets. For instance, consider the discrete manipulation instance of Figure 1.1. A human reasoner might approach this problem by first considering the two possible orderings for the object manipulations, corresponding to the plans $\mathbf{a}_1 :=$ [GoGrasp(m), GoDrop(m), GoGrasp(c), GoDrop(c)] and $\mathbf{a}_2 :=$ [GoGrasp(c), GoDrop(c), GoGrasp(m), GoDrop(m)]. Since the robot is close to m, and the destination of m is close to c, it might be obvious even at this high level that $\mathbf{a}_1$ is superior to $\mathbf{a}_2$, and the latter plan need not be refined further.

This sort of abstract reasoning is unavailable to a planner based on explicit reachable sets. In particular, the prefix GoGrasp(c), GoDrop(c) has a very large number of reachable states, consisting of all combinations of drop locations for c along with corresponding robot configurations (i.e., base and gripper positions). To conclude that $\mathbf{a}_2$ is expensive, the planner would have to reason about following with GoGrasp(m), GoDrop(m) from each such reachable state, independently of the others. In contrast, the algorithm presented in this section reasons with *implicit* angelic sets, and could quickly compute that this continuation is expensive from any such state regardless of its precise configuration (see Figure 5.8).

Our basic starting point is the singleton DASH-A* algorithm of the previous section.

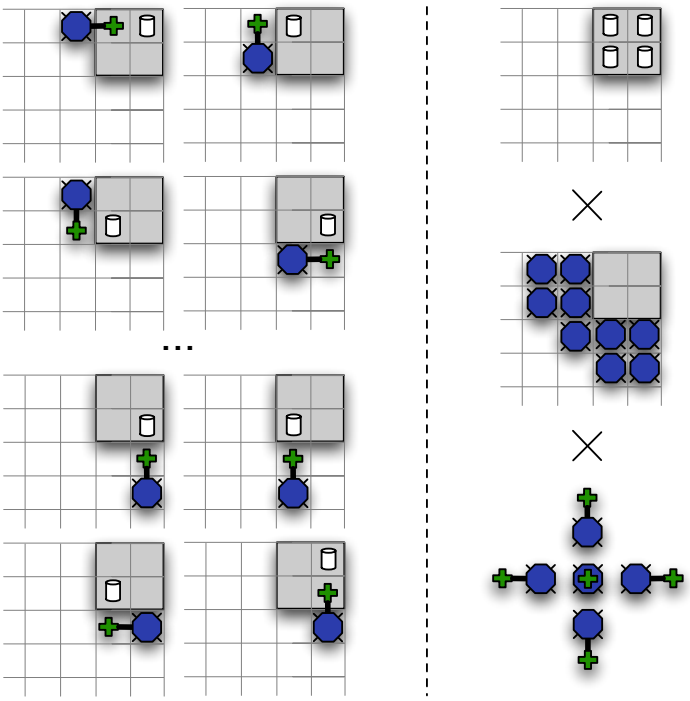

Figure 5.8: Left: an explicit listing of (a small subset of) the reachable states of GoDrop(c). Right: an implicit, factored representation of an optimistic approximation to this set.

However, we now consider subproblems of the form $(S, \mathbf{a})$, where S is a *reachable set* of states described implicitly, e.g. by a logical formula or factored state set (assuming simple valuations, as described in Section 4.3). As we will see, this extension requires substantial extensions to the algorithm in the previous section to ensure a systematic, efficient search.

A first observation is that, while a subproblem $p = (S, [a_1, a_2])$ can still be broken down into problems $p_1 = (S, a_1)$, $p_2 = (S', a_2)$ where $S' = \bar{O}_{a_1}(S)$, when $|S'| > 1$ these subproblems are no longer *independent* and thus the same concept of decomposition does not directly apply (recall the example of Figure 3.3). In particular, if c_1 and c_2 are optimal solution costs for p_1 and p_2, then $c_1 + c_2$ is only a *lower bound* on the optimal solution cost of p (because the output state corresponding to c_1 may not correspond to the input state for c_2). Nevertheless, if S' is large, refining p_1 and p_2 to increase this lower bound (while ignoring their correlations) may be an efficient way to prove that p is expensive and other plans should be considered instead.

For example, Figure 5.9 shows the first few steps of DASH-A* on (a slightly modified version of) our earlier example. While the initial bounds on the TIDY actions seem to indicate that the right plan is better, by considering the refinements of TIDY(m) from the implicit output set of TIDY(c), the algorithm is able to prove that the right plan is in fact much more expensive than it first seemed. Crucially, it is able to do so without explicitly enumerating the large number of concrete states contained in this implicit set.

Then, Figure 5.10 shows the next few steps of DASH-A* on this same instance, where the algorithm now refines both of the actions in the left plan. This increases the cost bound of this plan slightly, but it still remains much lower than the bound on the right plan (41 vs. 61). In fact, the algorithm will continue refining nodes in the left subtree until an optimal solution is found (with cost less than 61), without ever examining the right plan again.

To reach a fully primitive refinement of the left plan, however, DASH-A* will eventually have to *specialize* the work done on TIDY(c) from the abstract output set of TIDY(m), to find the best overall solution (consisting of an optimal primitive refinement of TIDY(m) that reaches some particular intermediate state s' in this set, followed by an optimal primitive refinement of TIDY(c) from concrete state s'). This propagation and specialization proceeds incrementally (i.e., for more and more refined versions of the intermediate set), at each point reusing as much of the work already done on TIDY(c) as possible. Figure 5.11 provides a rough sketch of how this propagation might work, for a simpler abstract example.

The full DASH-A* algorithm incorporates these ideas to efficiently carry out a systematic search across subproblems with varying levels of action and state abstraction (here referring both to implicit sets, and the state abstraction of Chapter 3). Perhaps surprisingly, the basic search strategy used by DASH-A* is the same as the simpler singleton variant in the

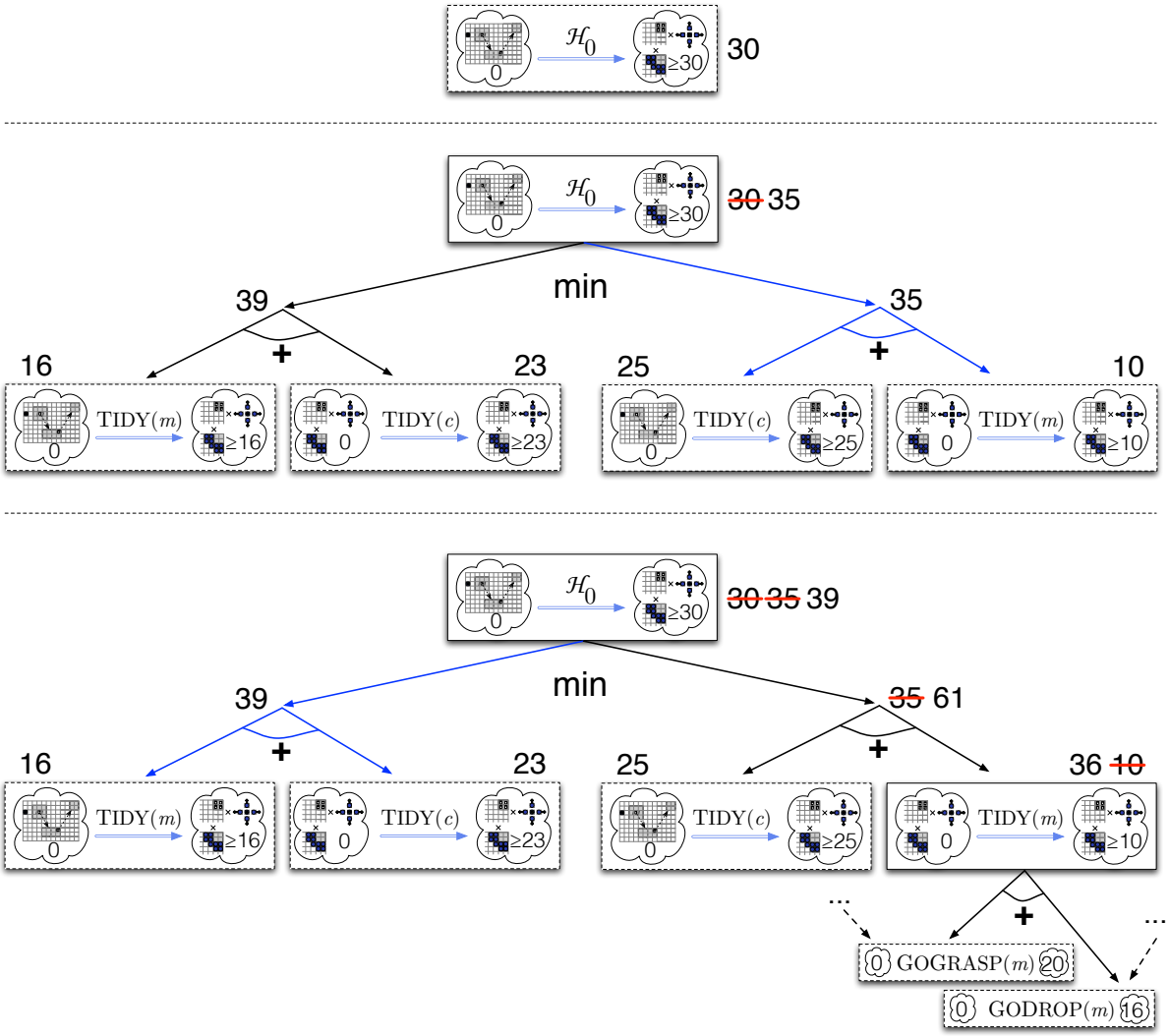

Figure 5.9: A rough sketch of the kinds of inferences DASH-A* is able to make. Top: initially, the agent's graph consists of just a root subproblem for $(s_0, [\mathcal{H}_0])$. Middle: after expanding the root, the agent has two plans: one for tidying the magazine and then the cup, and one for the opposite order (we modify the example hierarchy and cost bounds slightly for pedagogical purposes). Each plan is decomposed into two subproblems, one for doing the first action from the initial state, and one for doing the second action from the implicit optimistic reachable set of the first action. After propagating summary information from the new leaves to the root, the right plan looks cheaper than the left one (35 vs. 39). Bottom: upon expanding TIDY(m) in the right plan, the agent finds that this action is in fact much more expensive than its initial optimistic bound suggested, and the left plan now looks cheaper. (Continued in Figure 5.10.)

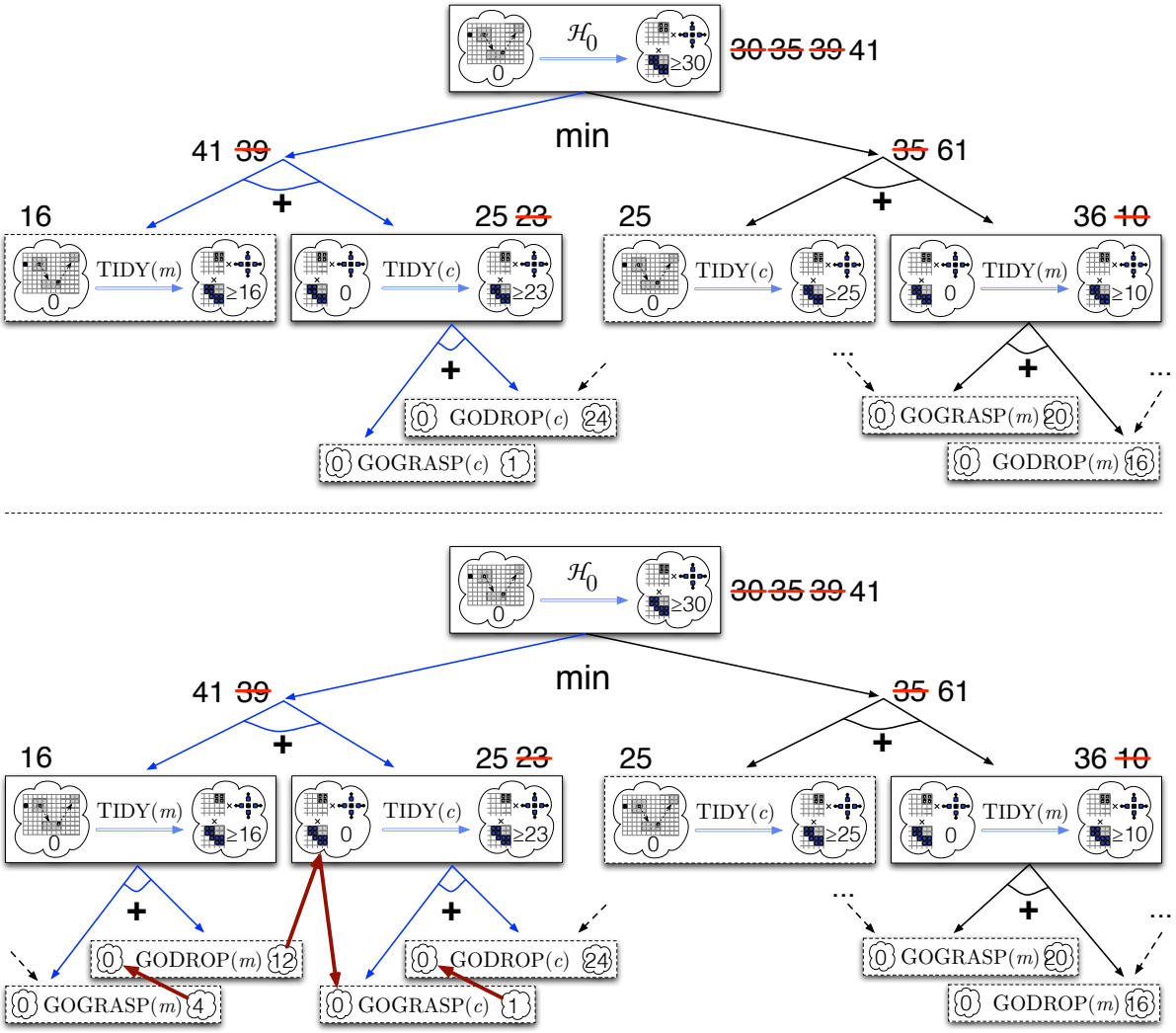

Figure 5.10: (A continuation of Figure 5.9.) Top: after expanding TIDY(c) in the left plan, its cost bounds increase slightly, but it still has much lower bounds than the right plan. Bottom: the algorithm continues by expanding TIDY(m) in the left plan. The left plan is in fact optimal with cost less than 61, so the agent will do no further expansions of the right plan. However, to reach a primitive solution, the algorithm will eventually have to propagate refined reachability information about the concrete output state of a particular way of doing TIDY(m) into TIDY(c), and from there down into its refinements (red arrows). This is the main complication that makes DASH-A* (significantly) more complex than the singleton/explicit variants considered in the previous section.

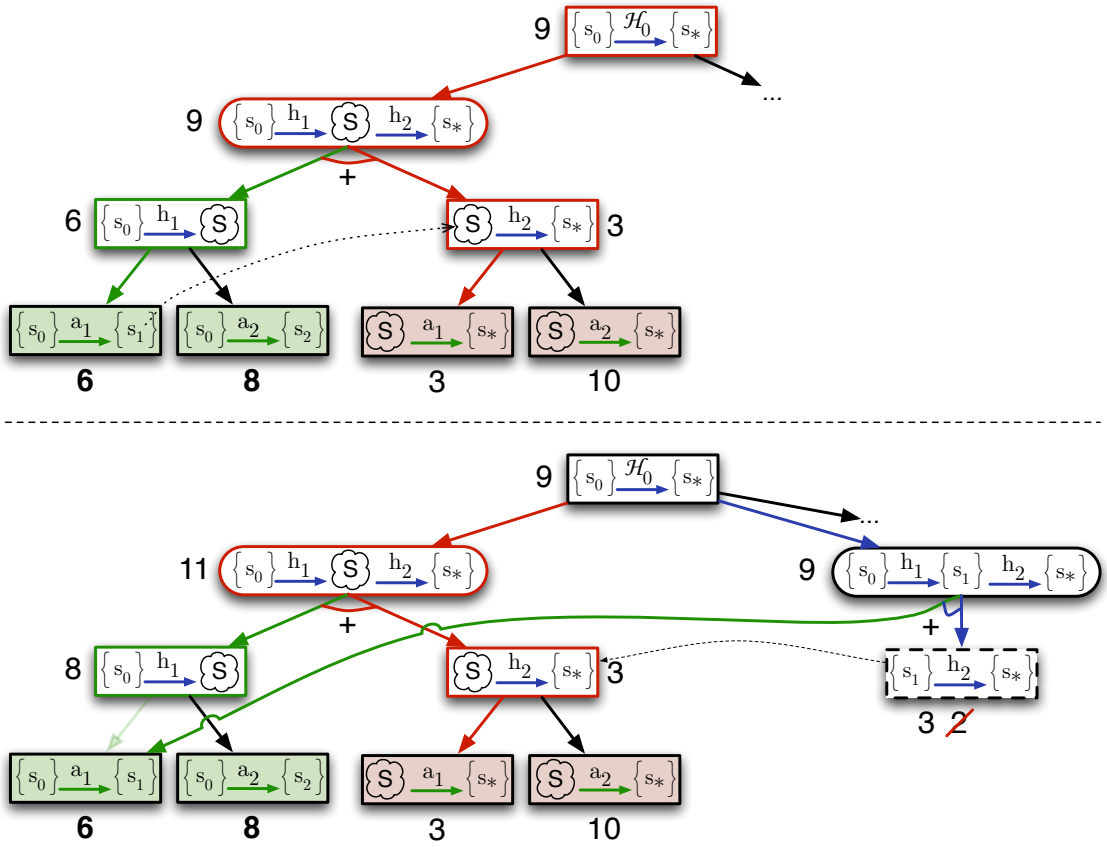

Figure 5.11: A rough sketch of how DASH-A* propagates refined reachability information from earlier subproblems into subsequent ones. Top: after several steps of graph expansion as in Singleton DASH-A*, the root becomes unrefinable. In particular, its left child has a smaller bound than its other children (not shown), but there are no refinable leaves left in its subtree (all of the leaves represent primitive actions). The node corresponding to action h_1 has been optimally solved by a_1 from the initial state, but the precise status and cost of the node corresponding to h_2 cannot be determined from implicit set S (the optimistic reachable set of h_1). Bottom: to make progress, refined information about the output set of h_1 (e.g., the specific output state $s_1 \in S$ of a_1) must be propagated into h_2. This is accomplished by constructing a new node for $[h_1, h_2]$ for intermediate state s_1, whose right child is a *specialization* of the previous h_2 that inherits its cost bound via *subsumption*.

previous section (i.e., DASH-A* can use standard AND/OR search algorithms such as AO*). Unlike in a typical AND/OR search, however, the AND/OR graph will be grown in a novel *non-local* way, as reachability information propagates forward through the search space.

Because the final DASH-A* algorithm is quite complex, we develop it in stages. We begin with a more detailed high-level overview of the algorithm and its data structures.

5.3.2 Overview

The previous section described the types of computations carried out by DASH-A* — expansion of nodes, propagation of summary and reachability information, and specialization of abstract plans — and the sorts of high-level inferences that can result from these computations. In order to make these high-level points clear, however, the examples simplified away many features of the algorithm that are crucial to its correct and efficient operation. This section provides an overview at a finer level of granularity, focusing on the core data structures and procedures used by the actual algorithm, as well as explaining the reasoning behind its major design decisions. In what follows, we assume that the reader is familiar with the AO* algorithm introduced in Section 2.2.3, including our definitions of nodes, summaries, and summary propagation.

The most significant simplification made in the previous section is that each subproblem was depicted as a single AND- or OR-node, as has been the case throughout this thesis. As we will see, in DASH-A* each subproblem will actually correspond to a connected *group* of several graph nodes, as well as other auxiliary data structures. The basic reason is that, to maintain a systematic search, the algorithm must keep track of what refinement information has been propagated into each parent context. Beginning with the abstract picture of the previous section, we introduce these components incrementally while explaining the issues that each component is meant to address.

Specifically, we pick up where the previous section left off, at the problem of propagating reachability information forward in the graph from refinements of earlier subproblems into subsequent subproblems with non-singleton input sets. As in Section 3.3.1, we will use a *channel* data structure to convey this reachability information, ensuring that each parent context receives all of the relevant information about each child. However, whereas in that section the channel was used to publish concrete solutions of a subproblem that reached a particular state, in DASH-A* a subproblem's channel conveys other *subproblems* corresponding to its (potentially non-primitive) *refinements*. Each published refinement p' of a subproblem p will reach a *subset* of the states reachable by p, allowing the algorithm to *incrementally* specialize subsequent subproblems (and hopefully, increase their cost bounds) before the fully primitive level is reached. For example, consider the bottom-left corner of Figure 5.10, corresponding to plan [GoGrasp(m), GoDrop(m)]. Upon refining GoGrasp(m), the agent generates more specific plans corresponding to particular base positions for the

grasp. By propagating reachability information about these plans into GoDrop(m) (which still leave other details such as the grasp position abstract), it can quickly prove that picking up the magazine from the top of the table is an expensive option that probably need not be considered further.

Concretely, our first step is to associate each subproblem node with a channel for publishing its refinements. We call this grouping of a node and a refinement channel a *subproblem description*.[4] Each description d corresponding to subproblem $(S, \mathbf{a})$ will include its node, input set $d.input = S$, output set $d.output = \bar{O}_{\mathbf{a}}(S)$, refinement channel $d.channel$, and other components to be introduced in what follows. We call subproblem descriptions for OR-nodes *atomic*, because they correspond to a single action ($|\mathbf{a}| = 1$). Descriptions for AND-nodes are called *pairs*, since they will always correspond to the sequential composition of two other subproblems (where the output set of the left child is equal to the input set of the right child).

Given this extension, when a leaf node corresponding to an atomic subproblem is refined, rather than adding its refinements (each a pair of two new atomic leaves) as children as depicted previously, the atomic subproblem can instead publish them on its refinement channel. Having passed responsibility for its refinements to its parent context(s) via these publications, the cost bound of its summary is increased to ∞.

Then, each pair subproblem p is a subscriber to the refinement channels of its left and right children l and r. When it receives a refinement r' published by its right child r, p publishes a new pair subproblem on its own channel with left child l and right child r'. Similarly, when it receives a refinement l' published by l, p publishes a new pair with left child l'. However, the output set of l' may be a *subset* of the output set of l (and thus the input set of r), and in this case the right child of its publication r' is a *specialization* of r for this refined input set.

The specialization of a subproblem description d for input set S proceeds as follows.

- If $S = d.input$, then the result is just d.
- Otherwise, if d is atomic corresponding to subproblem (S', a), the result is a new atomic subproblem for (S, a).
- Otherwise, d is a pair with left and right subproblem descriptions l and r. First, l is specialized for S, yielding l'. Then, r is specialized for $l'.output$, yielding r'. The result is a new pair subproblem for $[l', r']$.

The algorithm described thus far generates a forest of nodes, where each root node represents a (pairwise factored) transitive refinement of $(\{s_0\}, [\mathcal{H}_0])$. If we maintain a queue

[4]Note that the use of the word "description" here is distinct from in the previous chapter, where it was used to refer to angelic transition models.

of these root nodes, ordered by their lower cost bounds (i.e., at each step we choose the cheapest root, expand a leaf in its subgraph, and add all refinements published at this root back to the queue), we recover an implementation of Optimistic AH-A*.

Before proceeding to enhancements that improve upon this algorithm, we first present an alternative method for searching in this framework that will be more amenable to later improvements. In particular, rather than maintaining a separate queue over a forest of nodes, we would like to generate a single AND/OR graph that we can search directly with an existing algorithm like AO*.

A simple way to accomplish this is to add an additional OR-node to each subproblem description d, which we call its *outer* node $d.outer$. The outer node initially has the original node of d as a child, and it also subscribes to $d.channel$, adding each published refinement of d as a child as well. Thus, the summary of the outer node always captures all of the primitive refinements of a subproblem, initially through the original node, and later through its published refinements. With this extension, the forest is connected into a single AND/OR graph rooted at the outer node of the initial atomic subproblem $(\{s_0\}, [\mathcal{H}_0])$, and we can apply AO* to this graph to recover yet another implementation of Optimistic AH-A*.

So far, so good. But, of course, we have not yet actually gained anything for our troubles. For instance, consider a plan with n high-level actions, each of which has a single reachable state and k immediate refinements. As in the simpler version of Optimistic AH-A* presented earlier in the chapter, generating the set of plans corresponding to refining each HLA a single time still generates $O(n^k)$ nodes, rather than $O(nk)$ as we might hope.

The issue is that this algorithm publishes *every* refinement generated at a leaf node, and each such publication generates a further refinement publication at each parent, and so on until we reach nodes representing entire transitive refinements of the initial subproblem. However, recall that the purpose of publishing refinements is to propagate information about refined (smaller) reachable sets to other nodes in the graph. Thus, it makes sense to only publish those refinements d' of d where $d'.output \subset d.output$, and do something else with the remaining ones (with $d'.output = d.output$). In particular, when an atomic leaf d is refined, refinement subproblems d' with the same output set as d can be grouped as children of the original OR-node at d rather than being published (as in an ordinary AND/OR search). This *output grouping* prevents the exponential blowup just described, and is the first step towards an algorithm that can be exponentially faster than AH-A*.

The grouping just described does not fully solve this problem, however: the same issue can also arise at pair subproblems that are ancestors of the refined leaf. For example, when atomic subproblem GoGrasp(m) is expanded, each of its immediate refinements has a more specific output set (corresponding to particular base locations for the grasp), and is published. The parent pair subproblem for [GoGrasp(m), GoDrop(m)] receives each such refinement, and specializes GoDrop(m) for the refined output set of each grasp. However, these specializations of GoDrop(m) have the *same* output set as the unspecialized version

(because GoDrop also moves the base), and thus the new pair generated for each refinement also has the same output set as the original pair.

We would thus like to apply output grouping at pair subproblems as well. As described thus far, however, pairs have no appropriate OR-node at which the refinements can be stored. We thus augment our pair subproblem descriptions with new *inner* OR-nodes for this purpose. The inner node $d.inner$ for pair d initially has just its AND-node as a child, and supplants its AND-node as the attachment point for parent subproblems (and $d.outer$). For consistency, we also call the original OR-node of an atomic subproblem an inner node. Figures 5.13 and 5.14 show graphical depictions of these subproblem descriptions (including a few additional details to be described shortly).

After this change, the graph still intuitively captures the right structure, since each outer node still represents each refinement of a subproblem, either through the inner node (for generated refinements that do not provide new reachability information) or through published refinements directly attached to the outer node. However, this change breaks several other aspects of the algorithm, which we must repair before we can run AO* to enjoy its benefits.

A first problem is that calling UpdateAncestors on a refined leaf may no longer be sufficient to restore consistency to the summaries in the graph. For instance, consider a pair d whose left child l is an atomic subproblem for a primitive action, and whose right child r is an atomic subproblem for an HLA with a non-singleton output set S. Suppose that refining r generates two refinements: r_1, which is refinable with output set S and the same cost bound as r, and r_2, which is primitive and reaches a single state $s \in S$ with the same cost as r. Then, r_1 is added as a child of $r.inner$, whereas r_2 is published on $r.channel$, creating a new fully primitive, unrefinable pair for $[l, r_2]$ that is published at d and attached as a child of $d.outer$. When UpdateAncestors($r.inner$) is called to update summaries in the graph, it halts immediately because the summary of $r.inner$ has not changed (due to the attachment of r_1). This leaves the graph in an inconsistent state, because $d.outer$ (and perhaps its ancestors as well) should be marked unrefinable due to the attachment of r_2.

To address this issue, throughout each expansion operation we maintain a list of nodes *maybeUnrefinable* that have had new children attached, and at the end of the expansion operation we run a limited version of UpdateAncestors that only propagates changes in refinable status upwards in the graph, starting at each such node.

The other problem that arises from output grouping is related to specialization. Before, recall that (ignoring outer nodes) the graph was a forest of plans, each of which had pairs for interior nodes and unrefined, atomic subproblems at the leaves. (Plans including refined atomic subproblems had cost bound ∞, and were not part of the active graph for all intents and purposes.) Now, however, we can have live plans containing refined atomic nodes, which have their own subplans attached at inner nodes.

This is good, because it means that we can compactly represent combinations of refinements of different actions in a plan in a factored manner. However, with this change we must take extra care to ensure a systematic search. In particular, consider a pair $d = [h_1, h_2]$, and suppose that h_2 has been refined, yielding three refinement plans (pairs) r_1, r_2, and r_3. r_1 and r_2 had the same output set as h_2, and were attached as children of $h_2.inner$, but r_3 had a refined output set, and generated a publication at d corresponding to $[h_1, r_3]$. Now, suppose that we refine h_1, and one of its refinements r_0 has a smaller output set than h_1. In this case, we need to specialize h_2 for this refined output set. To ensure a systematic search, this specialization should capture the combination of r_0 with refinements r_1 and r_2, but not r_3 (otherwise, we would have two separate subproblems capturing plan $[r_0, r_3]$).

One potential path forward would be to create a new subproblem for just this subset of h_2's refinements from the output set of r_0. For instance, one could specialize the entire subgraph under h_2's inner node for this new input set. This could be wasteful, however, because h_2 might obviously be bad from this input set (e.g., perhaps it corresponds to grasping the magazine from the top of the table, so that the dropoff corresponding to h_2 becomes significantly more expensive). Moreover, it would significantly decrease the potential for caching (to be introduced momentarily), because it could result in many distinct copies of each subproblem description (for the same input set and action sequence) capturing different subsets of its refinements (those represented at its inner node at the time of specialization). These issues may be surmountable by a combination of laziness and a clever representation for such subset subproblems, but we have not yet found a way to tame the overwhelming bookkeeping complexity that seems to result.

While other solutions may be possible, we choose to sidestep this problem by modifying the way in which refinements are generated at pair subproblems. In particular, we only allow a given pair to generate refinements based on publications at *either* its left or right child, not both. We call a pair *left-expanding* if it responds to refinements at its left child, and *right-expanding* if it responds to refinements at its right child. This requires a further small change: a pair subproblem's AND-node must now connect to the *outer* node of the child whose publications it ignores (and the inner node of the other child, as before). This ensures that the ignored refinements are not lost, since the outer node always captures a bound on *all* refinements of a subproblem. In our implementation, we choose to make pairs right-expanding iff their left child has a singleton output set (and thus would not publish any refinements anyway), and otherwise left-expanding.

With these two changes (*maybeUnrefinable* updates and directional pair subproblems), we reach an state where we can again apply AO* for a systematic, hierarchically optimal search. Moreover, this algorithm avoids the exponential proliferation of plans in our earlier example, representing the set of all combinations of k refinements of n HLAs in $O(nk)$ space and time.

At this point, we have almost reached the full DASH-A* algorithm; just a few small

extensions remain. First, to yield a full graph-based algorithm, we can simply cache atomic subproblem descriptions so that at most a single description exists for each (S, a) subproblem encountered during search; then, state-abstracted caching requires just a bit more bookkeeping. However, in certain domains the cycles introduced by this caching can lead to infinite chains of publications and thus non-termination. We thus introduce another improvement called *hierarchical output grouping* to avoid this issue, and ensure termination and hierarchical optimality in all cases. Finally, when specializing the right child of a left-expanding pair at which some refinements have already been carried out, the algorithm described thus far is not able to reuse the information gathered by the previous refinements. Thus, we introduce a *subsumption* technique that rectifies this omission. We defer a full discussion of these techniques until Sections 5.3.6, 5.3.5, and 5.3.4 (respectively).

This subsection has introduced the key data structures and design decisions behind DASH-A*. The remainder of the section describes the algorithm in full detail. First, Section 5.3.3 introduces pseudocode for a first version of the algorithm that includes all of the techniques introduced here except for output grouping and the three extensions just mentioned.[5] Then, subsequent sections add the remaining techniques to yield the final DASH-A* algorithm, prove it correct, and present a family of examples in which DASH-A* can find hierarchically optimal solutions exponentially faster than other algorithms (including both DSH-UCS and AH-A*).

Figure 5.12 provides a summary view of the data structures used by DASH-A*, which may serve as a useful reference throughout the remainder of the chapter.

5.3.3 Initial Algorithm

This section describes pseudocode for the Simple Decomposed AH-A* algorithm, a simplified version of DASH-A* that has the same basic structure but omits a variety of improvements that make it efficient.

As mentioned above, the actual searching in Simple Decomposed AH-A* will be carried out by a standard AND/OR search algorithm such as AO$*_{\text{KLD}}$, and all of the complexities of the setting are encapsulated in the manner in which the AND/OR graph itself is constructed and extended over time. As such, we start by defining the top-level MakeRootNode and Expand operations that define the interface to this search algorithm. Algorithm 5.1 shows pseudocode for these operations, in terms of other operations and data structures to be defined in what follows.

To expand a leaf node n, which will correspond to an *inner* node of some atomic subprob-

[5] While not necessary in this simpler setting, the algorithm includes inner nodes and directional pair subproblems to serve as a foundation for the later improvements.

section	data structure	field	explanation
5.3.3	*maybeUnrefinable*	n/a	a global list of nodes that may have been rendered unrefinable via new child connections, within a single EXPAND operation
5.3.3	node n	$n.parents$	parents of n (which depend on n's summary)
		$n.children$	children of n (upon which n's summary depends)
		$n.summary$	a current summary of n
		$n.spdesc$	the subproblem description containing n
5.3.3	subproblem	$d.input$	the input reachable set of this subproblem
	description d	$d.output$	the output reachable set of this subproblem
		$d.outer$	the outer OR-node of this subproblem, which represents its full set of refinements
		$d.inner$	the inner OR-node of this subproblem, which represents only refinements not yet published at $d.channel$
		$d.channel$	a channel, upon which refinements of this subproblem are published (to communicate refined reachability information to subsequent subproblems)
5.3.5		$d.refmap$	a mapping from refined output sets for d to union subproblems, used for hierarchical output grouping
5.3.3	atomic subproblem	$d.hsp$	the subproblem $d.hsp = (d.input, [a])$ represented
5.3.4	description d	$d.irefs$	descriptions of immediate refinements (if expanded)
		$d.subsumers$	a list of subsuming subproblem descriptions
5.3.3	pair subproblem	$d.left$	the left subproblem description of this pair
	description d	$d.right$	the right subproblem description of this pair
		$d.and$	the AND-node of this pair
5.3.5	union subproblem	$d.irefs$	the constituent descriptions of this union
	description d	$d.specmap$	a mapping from refined input sets to existing specializations of this union
5.3.6	*cache*	n/a	a global cache of atomic subproblems descriptions, used to ensure that all instances of a given subproblem share the same description

Figure 5.12: A quick reference to the data structures used by the full version of DASH-A*. For each data structure and field (when applicable), we specify the section in which it will be introduced, and provide a brief explanation of its meaning.

Algorithm 5.1 DASH-A*: Top-level Operations

```
function EXPAND(leaf)
    d ← leaf.spdesc            /* the subproblem description for this leaf */
    maybeUnrefinable ← ∅       /* a global variable */
    for [a_l, a_r] ∈ IMMEDIATEREFINEMENTS(d.hsp.actions[0], d.input) do
        l ← GETATOMICSUBPROBLEMDESC((d.input, a_l))
        r ← GETATOMICSUBPROBLEMDESC((l.output, a_r))
        DOPUBLISH(d, MAKEPAIRSUBPROBLEMDESC(l, r))
    for n ∈ maybeUnrefinable do MARKUNREFINABLE(n)

function MAKEROOTNODE()
    return GETATOMICSUBPROBLEMDESC(({s_0}, [H_0])).outer
```

lem $p = n.spdesc.hsp$, each refinement of p generates a pair subproblem containing atomic subproblems for each action in the refinement, and these pairs are published at subproblem description $n.spdesc$. The root node of the search is the *outer* node of the description of subproblem $(\{s_0\}, [\mathcal{H}_0])$.

Before elaborating on these operations, we first make concrete the more familiar entities of *nodes* and *summaries* in this setting. Algorithm 5.2 show pseudocode for operations on nodes, which simply record the graph structure (in terms of child and parent lists), a current summary, and a containing subproblem description. MAKENODE(OR, d) creates an OR-node, and MAKENODE(AND, d) creates an AND-node; this type is used by CURRENTSUMMARY$(\cdot)$ (see Algorithm 2.5) to compute the summary of a node from its children. The graph structure is maintained by CONNECT(p, c), which makes p a parent of c. In addition, CONNECT(p, c) adds p to global list *maybeUnrefinable*, in case p becomes unrefinable (e.g., solved) by virtue of the newly connected child. Just before each EXPAND operation, each node correctly represents a *subset* of primitive refinements of its corresponding subproblem, which may change over time as the graph evolves.

Summaries are just as defined in Section 2.2.3.1, consisting of a *lower cost bound* $\hat{z}.lb$, refinable flag $\hat{z}.refinable?$, and active child set $\hat{z}.children$. The interpretation of the flag $\hat{z}.refinable?$ for summary $\hat{z}$ of node n is slightly different than in Chapter 2, however. While it is still the case that $\hat{z}.refinable? = \mathsf{true}$ iff there exists a refinable leaf node descendant in the active subgraph of n, $\hat{z}.refinable? = \mathsf{false}$ does not necessarily imply that an optimal solution is yet known for n. In particular, an optimal solution is known for n iff $\hat{z}.refinable? = \mathsf{false}$ and the input set of n is a singleton set; otherwise, no progress can be made on n (or its descendants) via expansion without more specific information about its input set, but an optimal solution may not yet be known (see Section 4.3.3).

Finally, in addition to the summary propagation operations carried out by the search algorithm itself (e.g., UPDATEANCESTORS(n) for AO$*_{\text{KLD}}$), we include an additional func-

Algorithm 5.2 DASH-A*: Nodes

```
function MakeNode(type, d)
    return a node of type type (AND/OR) with
        n.parents = []
        n.children = []
        n.summary = undefined
        n.spdesc = d

function Connect(parent, child)
    child.parents.Insert(parent)
    parent.children.Insert(child)
    maybeUnrefinable.Insert(parent)

function MarkUnrefinable(n)
    if n.summary.refinable? then
        ẑ ← CurrentSummary(n, n.summary.lb)
        if ¬ẑ.refinable? and ẑ.lb = n.summary.lb then
            n.summary ← ẑ
            for p ∈ n.parents do
                MarkUnrefinable(p)
```

tion MarkUnrefinable(n) that is called to propagate changes to summaries that occur at nodes other than the one expanded. This function propagates changes of summaries from refinable to unrefinable upwards in the graph from node n, without modifying any cost bounds. The potential loci for such changes are batched into a global list *maybeUnrefinable*, and MarkUnrefinable is called on each node in this list just before Expand returns.[6]

Next, Algorithm 5.3 shows pseudocode for constructing subproblems descriptions, as used by MakeRootNode and Expand. There are two basic types of subproblem descriptions, each of which corresponds to the subproblem $(S, \mathbf{a})$ of doing some primitive refinement of $\mathbf{a}$ from some element of reachable set S. First, an *atomic subproblem* (S, a) corresponds to doing a *single* action a. Second, a *pair subproblem* $(S, \mathbf{a} +\!\!+ \mathbf{a}')$ consists of the sequential composition of child subproblems $(S, \mathbf{a})$ and $(S', \mathbf{a}')$, where the input set of the latter is always the optimistic reachable set of the former: $S' = \bar{O}_{\mathbf{a}}(S)$. Each child may be an atomic subproblem or another pair (thus representing longer sequences).

MakeSubproblemDesc(i, o) constructs the basic structure shared by all subproblem descriptions. This includes the input reachable set $d.input = S$, the output reachable set

[6]The changes are batched rather than being applied directly after each Connect operation to ensure that the graph is in a consistent state (with all structural changes made, and all nodes assigned an initial summary) when MarkUnrefinable is called.

Algorithm 5.3 DASH-A*: Subproblem Construction

```
function MakeSubproblemDesc(i, o)
    d ← a subproblem description with
            d.input = i
            d.output = o
            d.outer = MakeNode(OR, d)
            d.inner = MakeNode(OR, d)
            d.channel = MakeChannel()
    Connect(d.outer, d.inner)
    Subscribe(d.channel, λx Connect(d.outer, x))
    return d

function MakeAtomicSubproblemDesc((S, a))
    d ← MakeSubproblemDesc(S, Ō_S(a))
    d.hsp ← (S, a)
    d.outer.summary ← d.inner.summary ← h((S, a))
    return d

function MakePairSubproblemDesc(l, r)
    d ← MakeSubproblemDesc(l.input, r.output)
    d.left ← l
    d.right ← r
    d.and ← MakeNode(AND, d)
    Connect(d.inner, d.and)
    if l.output is a singleton then   /* right-expanding */
        Connect(d.and, l.outer)
        Connect(d.and, r.inner)
        Subscribe(r.channel, λx DoPublish(d, MakePairSubproblemDesc(l, x)))
    else                              /* left-expanding */
        Connect(d.and, l.inner)
        Connect(d.and, r.outer)
        Subscribe(l.channel, λx DoPublish(d, MakePairSubproblemDesc(x,
                                                  Specialize(r, x.output))))
    d.inner.summary ← d.and.summary ← CurrentSummary(d.and, 0)
    d.outer.summary ← CurrentSummary(d.outer, 0)
    return d
```

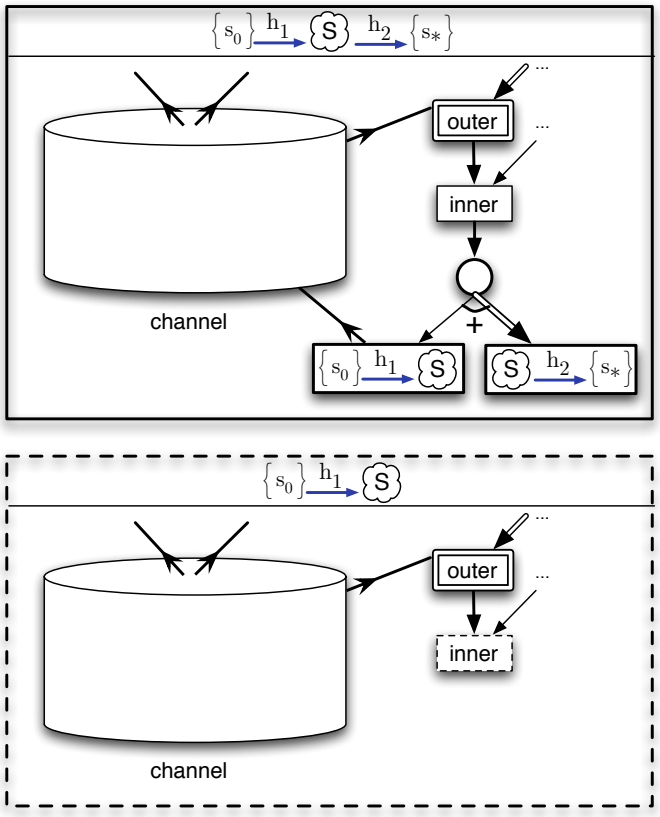

Figure 5.13: A "detail view" of two of the subproblems depicted as nodes in Figure 5.11. Each subproblem contains outer and inner OR-nodes, and a refinement channel. Top: a "left-expanding" pair subproblem. The circle is an AND-node, which is connected to the inner node of the left child subproblem (with a light arrow) and the outer node of the right child subproblem (with a double arrow). Lines with interior arrows represent subscriptions; for example, each publication by the left child subproblem will result in a publication at this node's channel, which will also be added as child of its outer node. Bottom: an atomic leaf (unexpanded) subproblem.

$d.output = \bar{O}_{\mathbf{a}}(S)$, two OR-nodes $d.inner$ and $d.outer$, and a channel $d.channel$ (see Figure 5.13). The *outer node* $d.outer$ represents the full set of all primitive refinements of $\mathbf{a}$ from S, and its summary $d.outer.summary$ always provides a valid lower bound on the minimum cost of any such primitive refinement $c^*((S, \mathbf{a}))$ (over all reachable states). The *refinement channel* $d.channel$ is used to publish information about refinements of d, corresponding to subproblems $(S, \mathbf{a}')$ where $\mathbf{a}' \in \mathcal{I}^+(\mathbf{a})$. While for now all refinements are directly published on this channel, later it will be used exclusively for subproblem descriptions d' with reachable sets $d'.output \subset d.output$ that are *strict* subsets of the reachable set of d. As in DSH-LDFS (see Section 3.3.1), channels are the conduit for propagating refined information about reachability forward to subsequent subproblems. Finally, the *inner node* $d.inner$ represents all primitive refinements of a subproblem that have not (yet) been published on $d.channel$.[7]

Thus, the children $d.outer.children$ of the outer node always consist of the inner node $d.inner$, plus the outer nodes of all descriptions that have been published on $d.channel$. The last lines of MakeSubproblemDesc(i, o) maintain this invariant, directly connecting $d.outer$ and $d.inner$, and then adding a subscription to $d.channel$ that connects $d.outer$ to $d'.outer$ for each published d'.

On top of this initialization, MakeAtomicSubproblemDesc$((S, a))$ adds a few more operations. First, it stores its corresponding subproblem (S, a) into $d.hsp$. Second, it initializes the summaries of its inner and outer nodes to $h((S, a))$, which returns a heuristic leaf summary with cost and status computed from OptimisticOutcomeAndStatus(a, S) (see Section 4.3.3).

MakePairSubproblemDesc(l, r) is a bit more complex, since it must also construct an AND-node and manage the transport of refined reachability information from its left subproblem l to its right subproblem r. It begins by storing l, r, and the AND-node in $d.left, d.right$, and $d.and$, and makes $d.and$ the sole child of $d.inner$. Then, it connects the AND-node to l and r in one of two ways, and finally initializes each node's current summary.

Each pair subproblem is either left-expanding or right-expanding (but not both), which determines how it is connected to l and r, and how it publishes refinements. If right-expanding, each refinement r' published by r generates a refinement corresponding to subproblem MakePairSubproblemDesc(l, r'), and refinements published by l are ignored. In this case, a systematic search is ensured by connecting $d.and$ to the *outer* node of l and the *inner* node of r. If left-expanding, each refinement l' published by l generates a refinement MakePairSubproblemDesc(l', r'), where r' is a *specialization* of r for the refined reachable set of l', defined shortly (see Figure 5.14). In this case, $d.and$ is connected to the *inner* node of l and the *outer* node of r (and refinements published by r are ignored). In

[7]For the time being, $d.inner$ will always have zero or one children (for atomic or pair subproblems, respectively) and is thus superfluous. Enhancements presented in subsequent sections will generate other children for $d.inner$.

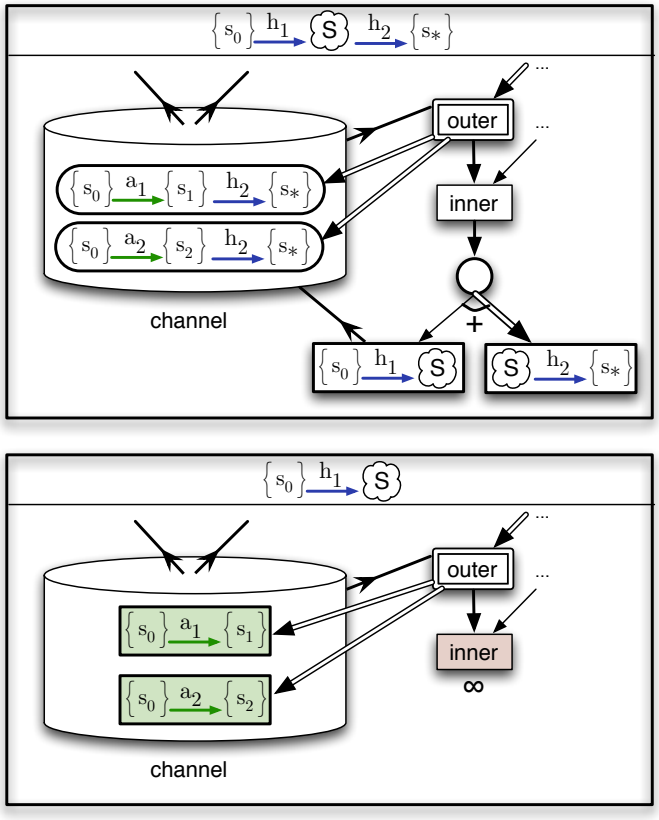

Figure 5.14: The state of the subproblems in Figure 5.13 after the leaf subproblem for h_1 has been expanded. Bottom: the two refinements of h_1 have been published in its channel, and their outer nodes are now children of its outer node. Top: these publications triggered additional publications at its parent pair subproblem, which are pairs consisting of a publication from h_1 and a specialization of the h_2 subproblem for the corresponding output set.

our implementation, all pairs are left-expanding unless l has a singleton output set, in which case there is no more information to be gained about the output set of l.[8]

Finally, Algorithm 5.4 introduces the three missing subproblem operations. For now, functions DoPublish and GetAtomicSubproblemDesc simply defer to Publish and MakeAtomicSubproblemDesc (respectively); improvements in subsequent sections will add additional logic. The interesting operation is Specialize(d, S'), which *specializes* description d representing subproblem $(S, \mathbf{a})$ to a new input set $S' \subseteq S$, returning a description corresponding to subproblem $(S', \mathbf{a})$. If $S' = S$, then no specialization is required and d is returned directly. Otherwise, if d is atomic, a fresh atomic subproblem for $(S', \mathbf{a})$ is returned. Finally, if d is a pair of subproblems l and r, first l is specialized for S' to l', then r is specialized for $l'.output$ to r', and l' and r' are wrapped in a new pair description.

Thus, Specialize(d, S') copies the graph of d down to the first level of atomic subproblems (which are created anew), sharing plan *suffixes* with d whose reachable sets have not changed as a result of specialization. Subsequent sections introduce additional improvements that can more heavily reuse the work embodied in d in this specialized context.

Before proceeding to discuss enhancements of this algorithm, we first prove its correctness.

Definition 5.6. *The subgraph rooted at a node n* represents *a primitive refinement* $\mathbf{b}$ *iff:*

- *n is an inner node of an unexpanded atomic subproblem (S, a), and $\mathbf{b} \in \mathcal{I}^*(a)$, or*
- *n is an OR-node, and some child in $n.children$ represents $\mathbf{b}$, or*
- *n is an AND-node, and there exist refinements $\mathbf{b}_1$ and $\mathbf{b}_2$ represented by the left and right children of n such that $\mathbf{b} = \mathbf{b}_1 + \mathbf{b}_2$.*

Lemma 5.7. *After any number of* Expand *operations, for each description d corresponding to subproblem $(S, \mathbf{a})$, each primitive refinement of $\mathbf{a}$ applicable from S is represented* exactly once *by the subgraph under $d.outer$.*

Proof. This is trivially true for the initial root subproblem, which represents all of its primitive refinements directly. When a subproblem is expanded, responsibility for its primitive refinements is transferred from $d.inner$ to the outer nodes of its newly constructed refinement pairs. These pairs initially capture the primitive refinements of each immediate refinement correctly, and so d and ancestors of $d.outer$ remain correct. The only difficulty involves pairs that depend directly on $d.inner$ (i.e., not through $d.outer$). Here, correctness is ensured since every parent of an inner node of d also subscribes to d, and re-publishes new pairs that capture the "missing" refinements. □

[8] More sophisticated policies are possible, including those where explicit decisions are made about when to publish refinements rather than doing so automatically.

Algorithm 5.4 DASH-A*: Subproblem Operations

```
function DoPublish(d, c)
    Publish(d.channel, c)

function Specialize(d, S)
    if d.input = S then return d
    if d is atomic then
        return GetAtomicSubproblemDesc((S, d.hsp.actions))
    else /*  d is a pair  */
        l' ← Specialize(d.left, S)
        r' ← Specialize(d.right, l'.output)
        return MakePairSubproblemDesc(l', r')

function GetAtomicSubproblemDesc(p)
    return MakeAtomicSubproblemDesc(p)
```

Lemma 5.8. *After each* Expand(n) *and* UpdateAncestors(n) *operation when applying* AO$*_{\text{KLD}}$ *to Simple Decomposed AH-A*, the summaries of all nodes are* consistent *(see Section 2.2) and provide correct bounds on the represented primitive refinements.*

Proof. All potential inconsistencies introduced by Expand(n) begin at node n. First, the lower bound on n itself increases to ∞. Second, subproblems in publication chains starting with the immediate refinements of $n.spdesc$ can be "caught" by outer nodes throughout the graph. Ignoring the first effect for now, the connection of these new subproblems to outer OR-nodes cannot change their cost bounds (assuming that they were correct before the Expand). However, if the new subproblems are unrefinable, the *status* of these OR-nodes and their ancestors can change, and in this case MarkUnrefinable restores consistency. Then, the only remaining inconsistency can be at n, and this is repaired by the call to UpdateAncestors(n). □

Lemma 5.9. *Consider any subgraph rooted at root.outer, where one child is selected at each non-leaf OR-node, both children are selected at each AND-node, and all leaf nodes have finite lower cost bound. Let $p_1, ..., p_n$ be the sequence of leaf subproblems corresponding to this subgraph, and $a_1, ..., a_n$ be the corresponding actions. Then for $i \in [1, n]$, $p_i.input \supseteq \bar{O}_{[a_1,...,a_{i-1}]}(\{s_0\})$. Moreover, if $a_1, ..., a_k$ are all primitive, then for $i \in [1, k+1]$, $p_i.input = \{\mathcal{T}(s_0, [a_1, ..., a_{i-1}])\}$.*

Proof. We first note the following properties. (1) For any pair subproblem d, we have $d.input = d.left.input$, $d.left.output = d.right.input$, and $d.output = d.right.output$. (2) For any OR-node corresponding to subproblem d, and any child of this OR-node corresponding to subproblem d', $d.input = d'.input$.

Next, we define the notion of an *effective prefix* for a subproblem p in the active subgraph. The effective prefix is the sequence $p'_1, ..., p'_l$ of atomic (not necessarily leaf) subproblems in this subgraph that generates the input set of p. In other words, $p'_1.input = \{s_0\}$, $p'_j.output = p'_{j+1}.input$, and $p'_l.output = p.input$, or equivalently, $p.input = \bar{O}_{[a'_1,...,a'_l]}\{s_0\}$ where a'_j is the action corresponding to p'_j.

Given a subproblem p, the effective prefix can be extracted as follows. First, walk upwards in the subgraph from p. Starting with the empty sequence, for each edge traversed corresponding to a right child of a pair subproblem, prepend the left child to the sequence. This yields a sequence of subproblems that satisfy the above conditions on input and output sets, but which may include pairs as well as atomic subproblems. Then, repeatedly expand each pair subproblem in this sequence into its left and right children, until no pair subproblems remain. (1) and (2) above guarantee that the sequence generated by this process is an effective prefix.

Now, consider any leaf subproblem p_i, and extract its effective prefix $p'_1, ..., p'_l$. Note that the actions in the leaf sequence $p_1, ..., p_{i-1}$ must represent a refinement of the sequence $p'_1, ..., p'_l$. Then, reachability consistency of the optimistic descriptions establishes the first part of the lemma.

To establish the other part, we first prove an auxiliary result by structural induction: for any pair subproblem d with singleton input set $d.input$, if all leaves in the left subgraph of d correspond to finite-cost primitive actions, then $d.left$ has a singleton output set and d is right-expanding. Moreover, if all leaves in the right subgraph of d are also finite-cost primitives, then $d.right$ and d itself have singleton output sets.

Suppose that d was left-expanding. Then its AND-node would be connected to the inner node of $d.left$, which must have a singleton input set and non-singleton output set. By the inductive hypothesis, $d.left$ cannot be a pair subproblem. However, $d.left$ also cannot be an expanded atomic subproblem because its inner node would have infinite cost bound, nor an unexpanded atomic subproblem because primitive actions have singleton outputs on singleton inputs. Thus, d is right-expanding, and $d.left$ has a singleton output set. Applying the same line of reasoning to $d.right$ establishes the full result.

Finally, suppose that leaves $p_1, ..., p_{i-1}$ are primitive. Then applying the auxiliary result to the pairs encountered when generating the effective prefix of p_i establishes that $|p_i.input| = 1$. Correctness of the optimistic descriptions together with the first part of the lemma guarantee that the single reachable state must be $\mathcal{T}(s_0, [a_1, ..., a_{i-1}])\}$. □

Lemma 5.10. *Consider any subproblem description d with singleton input and output sets ($|d.input| = |d.output| = 1$). $d.outer$ is only assigned an unrefinable summary $\hat{z}$ ($\hat{z}.refinable? = \mathsf{false}$) if $\hat{z}.lb = \infty$ and d is unsolvable, or $\hat{z}.lb < \infty$ and $\hat{z}$ is an optimal solution for d.*

Proof. Suppose that $d.outer$ is assigned an unrefinable summary $\hat{z}$ with finite cost. Consider

the sequence of leaf subproblems $p_1, ..., p_N$ corresponding to the active subgraph of $d.outer$, and let $p_1, ..., p_k$ be a maximal primitive prefix of this sequence. If $k = N$ then this sequence corresponds to an optimal solution. Otherwise, Lemma 5.9 ensures that $|p_{k+1}.input| = 1$. Then, by definition of OptimisticOutcomeAndStatus, p_{k+1} must be refinable. This is a contradiction, since $\hat{z}$ is refinable if any p_i is refinable. □

Theorem 5.11. *Simple Decomposed AH-A* with* AO$*_{\text{KLD}}$ *is* hierarchically optimal *in OZCF hierarchies.*

Proof. Together, Lemmata 5.7, 5.8, and 5.9 ensure that Simple Decomposed AH-A* can only return hierarchically optimal solutions, because every primitive refinement is represented in the structure of the graph, the leaves have valid cost bounds, and the summaries of the interior nodes are consistent.

Next, the algorithm cannot get "stuck", i.e., the root is unrefinable iff an optimal solution is known, or the problem has been proved unsolvable. This follows directly from Lemma 5.10, since the root always has singleton input and output sets $\{s_0\}$ and $\{s_*\}$.

Finally, the algorithm must terminate. First, we note that the graphs produced by Simple Decomposed AH-A* are always acyclic. Thus, calls to UpdateAncestors and MarkUnrefinable must terminate. Moreover, because publication chains can only travel *upward* (towards the root), each call to Expand must terminate as well. Finally, the OZCF restriction ensures that after a finite number of calls to Expand, the lower cost bound of the root must reach the optimal solution cost, and so the algorithm terminates. □

5.3.4 Subsumption

The first improvement to Simple Decomposed AH-A* takes advantage of *subsumption* relationships between subproblems.

Definition 5.7. *Subproblem* $(S, \mathbf{a})$ *is* subsumed *by subproblem* $(S', \mathbf{a})$ *if* $S \subseteq S'$.

Subsumption of this form has previously been exploited for search in partially observable domains with demonic nondeterminism (Genesereth and Nourbakhsh, 1993; Wolfe and Russell, 2007), where a solution for a set of states S can be reused on any *subset* of S. The consequences in the angelic setting are analogous.

Theorem 5.12. *Every primitive refinement of sequence* $\mathbf{a}$ *applicable from some state in set* S *is applicable from some state in* $S' \supseteq S$. *Thus, for any action sequence* $\mathbf{a}$, $c^*((S, \mathbf{a})) \geq c^*((S', \mathbf{a}))$

Proof. Trivial. □

Subsumption allows information gathered while refining a subproblem $(S', \mathbf{a})$ to guide search at all subproblems $(S, \mathbf{a})$ on subsets $S \subseteq S'$. While explicitly looking for such relationships could be prohibitively expensive (as discussed in Section 5.1.2.3), we naturally discover them during search: by definition, SPECIALIZE(d, S) is always subsumed by d. Thus, when specializing an atomic subproblem d, the constructed specialization can borrow information about its immediate refinements and cost bounds from d. In the best case, d may essentially already encode the optimal solution, with just a few remaining details that need to be fleshed out given the new input information in S. For instance, suppose we refine GOGRASP(c) in detail from the abstract output set of GODROP(m), discovering that one particular grasp position seems much cheaper than the others. If we later refine GODROP(m) and specialize GOGRASP(c) for its more specific outputs, subsumption enables the use of bounds discovered at the high level to avoid considering expensive (or infeasible) refinements of GOGRASP(c) from each such specific input set.

Algorithm 5.5 shows changes to Simple Decomposed AH-A* to incorporate these intuitions. First, two new fields are added to each atomic subproblem description d: $d.irefs$ records the set of immediate refinement subproblems of d (or *unexp* if d has not yet been EXPANDed), and $d.subsumers$ records the list of subproblems known to subsume d. Next, the case for atomic subproblems in SPECIALIZE(d, S) is modified in two ways: d is added as a subsumer of the specialized subproblem d', and the lower bounds of d' are updated to the *maximum* of their current values and the outer bound of d (which may encode knowledge propagated upwards from refinements taken at d). Finally, EXPAND(n) is extended to record immediate refinements in $d.irefs$ and take advantage of subsuming subproblems. If any expanded subproblem d' subsuming d is known, rather than generating the immediate refinements of d from scratch, the immediate refinements of d' are specialized instead. In addition to saving effort generating the refinements, this ensures that the subsumption and bound reuse are applied *recursively* throughout the existing subtree of d.

5.3.5 Same-Output Grouping

As described thus far, after expanding a subproblem, each published refinement can generate a cascade of refinement publications at its parent pair, and so on up the graph. Often, this is unnecessary. In particular, the purpose of publishing refinements is to communicate information about refined (strictly smaller) output sets to subsequent subproblems in the graph. Thus, if a refinement d' of d has $d'.output = d.output$, we need not publish it. Instead, we can store $d'.inner$ internally as a child of $d.inner$, and re-publish the refinements of d' directly at d. This "local" output grouping significantly compresses the subproblem graph when refinements of an HLA tend to have the same output set, which is often the case given

Algorithm 5.5 DASH-A*: Subsumption

```
function MakeAtomicSubproblemDesc((S, a))
    ... /* previous code from Algorithm 5.3 */
    d.irefs ← unexp
    d.subsumers ← []
    return d

function Specialize(d, S)
    if d.input = S then return d
    if d is atomic then
        d' ← GetAtomicSubproblemDesc((S, d.hsp.actions))
        d'.subsumers.Insert(d)
        d'.outer.summary.lb ← max(d'.outer.summary.lb, d.outer.summary.lb)
        d'.inner.summary.lb ← max(d'.inner.summary.lb, d.outer.summary.lb)
        return d'
    else ... /* code from Algorithm 5.4 for pairs */

function Expand(leaf)
    d ← leaf.spdesc
    maybeUnrefinable ← ∅
    d.irefs ← []
    if exists d' ∈ d.subsumers with d'.irefs ≠ unexp then
        for i ∈ d'.irefs do
            d.irefs.Insert(Specialize(i, d.input))
    else
        for [a_l, a_r] ∈ ImmediateRefinements(d.hsp.actions[0], d.input) do
            l ← GetAtomicSubproblemDesc((d.input, a_l))
            r ← GetAtomicSubproblemDesc((l.output, a_r))
            d.irefs.Insert(MakePairSubproblemDesc(l, r))
    for i ∈ d.irefs do DoPublish(d, i)
    for n ∈ maybeUnrefinable do MarkUnrefinable(n)
```

Algorithm 5.6 DASH-A*: Local Output Grouping

```
function DoPublish(d, c)
    if d.output ≠ c.output then
        Publish(d.channel, c)
    else
        Connect(d.inner, c.inner)
        Subscribe(c.channel, λ.x DoPublish(d, x))
```

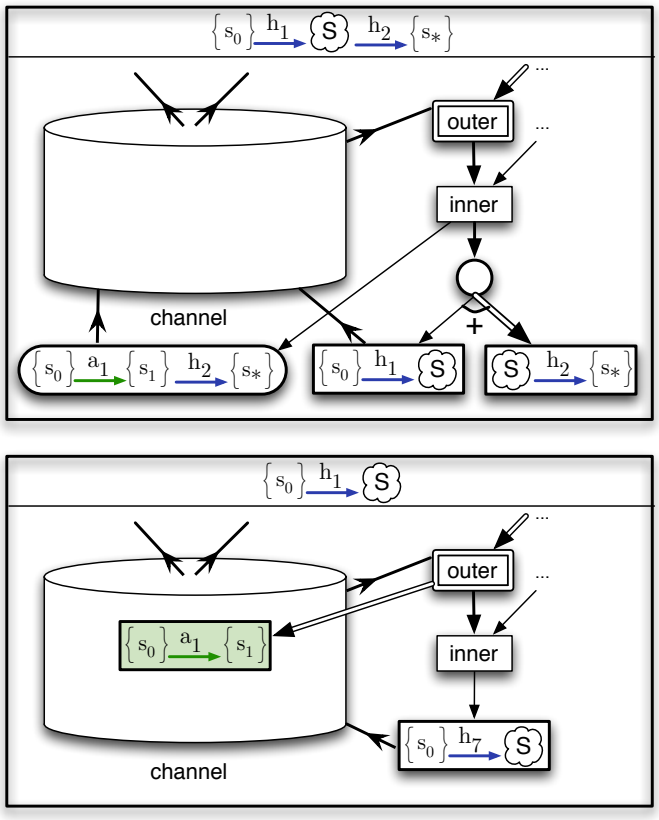

Figure 5.15: An example of local output grouping. Bottom: subproblem h_1 has two refinements, a_1 and h_7. Because h_7 has the same output set from s_0 as its parent h_1, its subproblem is captured as an inner child rather than being published. The subproblem corresponding to a_1 is published as usual, since its output set $\{s_1\}$ is a strict subset of S. Top: the publication of a_1 creates a new subproblem at the parent pair for sequence $[h_1, h_2]$, which is captured as an inner child (stopping the publication chain) because it has the same output set ($\{s_*\}$) as its parent.

the *funneling* nature of many useful HLAs (e.g., the output set of $\mathcal{H}_0$ is always $\emptyset$ or $\{s_*\}$).

Algorithm 5.6 shows a modified DoPublish(d, c) that directly implements this change. While this simple local approach can avoid many unnecessary refinement publications and thus lead to significant compression of the graph (see Figure 5.15), it has several drawbacks. First, it may still unnecessarily publish multiple refined subproblems with the same output set. Second, and more importantly, it is not sufficient to avoid issues that can arise in *cyclic* graphs, which we introduce in the next section. Figure 5.16 illustrates an example where two subproblems subscribe to each-other's refinement channels, leading to an infinite regress of solution publications with the same output set under local output grouping.

One approach to forestall the infinite regress would be to use an incremental or lazy publication strategy. In addition to adding more complexity to the overall search strategy, however, this would also be less than ideal because information about the cycle is effectively lost (and is just lazily unrolled in the publication stream). Thus, a more sophisticated method for output grouping is needed that is able to produce refinements that preserve cycles in the original graph.

Algorithm 5.7 shows pseudocode for such a method. The basic idea is to, in addition to grouping refinements of d with the same reachable set $d.output$ under $d.inner$, also *hierarchically* group refinements of d with the same reachable *subset* of $d.output$ into a single publication at d. In other words, (with some minor caveats) only a single refinement should be published at d for each reachable subset $\subset d.output$, which groups together all refinements c passed to DoPublish(d, c) with that set. This leads to additional compression of the graph, and moreover, guarantees termination even in the presence of cycles.

To implement this idea, we define a new subproblem type: a *union subproblem*, which corresponds to a set of subproblems with the same input and output sets. This subproblem stores a constituent set of subproblems $d.irefs$, which may grow over time, and a mapping $d.specmap$ of existing specializations of d as a function of input set.

Then, each subproblem d is augmented with a mapping $d.refmap$, which records the union subproblems published under d indexed under their output sets. In DoPublish(d, c), rather than directly publishing child c when it cannot be locally grouped into d, we add it to an union subproblem with its output set, creating and publishing one if necessary.

Finally, we must extend Specialize(d, S) to handle union subproblems d. This requires some care, since a union subproblem may grow to include itself recursively (e.g., as one side of a pair in $d.irefs$), and thus directly specializing its constituents could lead to an infinite loop. It turns out that this can be done easily and efficiently, however, by exploiting an analogy between atomic subproblems and union subproblems (both represent collections of immediate refinements of a sort). In particular, we simply make a new *leaf* union subproblem d' that looks like an atomic subproblem backed by d via subsumption. Then, our existing

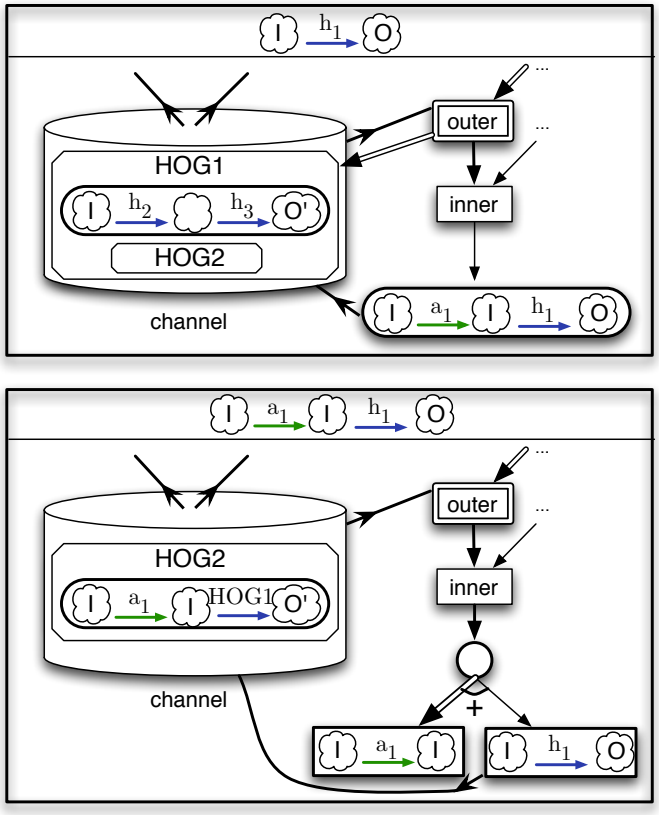

Figure 5.16: An examples of hierarchical output grouping, in a case where local (or no) output grouping would result in an infinite publication chain. The top box corresponds to a subproblem (I, h_1), and the second to a refinement of this subproblem $(I, [a_1, h_2])$, where the optimistic reachable set of a_1 from I is also I. h_1 also has a second refinement $[h_2, h_3]$, which leads to optimistic set O' from I. Without hierarchical output grouping, the publication of this refinement at the parent subproblem would generate a refinement publication at the child, which would be republished at the parent, and so on ad infinitum (corresponding to all plans of the form $\left[a_1^+, h_2, h_3\right]$). With hierarchical output grouping, the publication is wrapped in a union subproblem labeled HOG1. This resulting publication at the child is wrapped in another union HOG2. Finally, the infinite regress is avoided when HOG2 is added as a constituent of HOG1 at the parent, rather than being republished as a separate subproblem.

Algorithm 5.7 DASH-A*: Hierarchical Output Grouping

```
function MakeSubproblemDesc(i, o)
    ... /* previous code from Algorithm 5.3 */
    d.refmap ←an empty mapping from output set to union subproblem
    return d

function MakeUnionSubproblemDesc(i, o)
    d′ ← MakeSubproblemDesc(i, o)
    d′.irefs ← []
    d.specmap ←an empty mapping from input set to union subproblem specialization
    return d

function DoPublish(d, c)
    if d.output = c.output then
        u ← d
    else
        u ← d.refmap[c.output]
        if u = undefined or u.specmap ≠ [] or c.inner.summary.lb < u.inner.summary.lb
                                           or c.outer.summary.lb < u.outer.summary.lb then
            u ← d.refmap[c.output] ← MakeUnionSubproblemDesc(c.input, c.output)
            u.outer.summary ← c.outer.summary
            u.inner.summary ← c.inner.summary
            Publish(d.channel, u)
        u.irefs.Insert(c)
    Connect(u.inner, c.inner)
    Subscribe(c.channel, λx DoPublish(u, x))

function Specialize(d, S)
    if d.input = S then return d
    if d is union then
        if d.specmap[S] ≠ undefined then return d.specmap[S]
        d′ ← d.specmap[S] ← MakeUnionSubproblemDesc(S, d.output)
        d′.subsumers ← [d]
        s ← a refinable leaf summary with no children and bound d.outer.summary.lb
        d′.outer.summary ← d′.inner.summary ← s
        return d′
    else ... /* code from Algorithm 5.5 for atomics and Algorithm 5.4 for pairs */
```

machinery automatically populates d' with the specializations of the constituents of d if and when d' is subsequently selected for expansion. In addition to simplifying the code, deferring the hard work in this way automatically results in proper handling of cycles.

The caveat mentioned above is that because union subproblems are *open* (admitting the addition of new constituent subproblems), they do not have a single fixed semantics in terms of the primitive refinements they represent. This can cause problems in several ways. First, adding a new constituent to union d can actually *decrease* the cost of the best optimal solution represented by d, whereas our search machinery assumes that lower bounds should only *increase*. Second, if a new subproblem is added to d after it has been specialized, its specializations would have to be added to the specialization of d, and so on. To avoid both of these complexities, we simply create a fresh union subproblem in DoPublish when the old one has already been specialized or when adding this child would decrease its lower bound. Because these events can only occur a bounded number of times (there are only a limited number of subsets to specialize to, or costs to decrease to) and should be relatively infrequent in practice, this seems like a reasonable tradeoff of algorithmic efficiency for simplicity.

We conclude by noting that while this hierarchical output grouping machinery is somewhat heavyweight, it need not be applied at every subproblem. In particular, it suffices to apply it only at atomic and union subproblems, and only those that may be involved in certain kinds of cycles (although it may prove helpful elsewhere).

5.3.6 Caching and State Abstraction

With these improvements in place, we can introduce caching and state abstraction to yield the final DASH-A* algorithm.

Algorithm 5.8 shows the trivial change required for caching: we simply cache atomic subproblem descriptions, so that at most one description d exists for each atomic subproblem (S, a) encountered during search. This can introduce cycles in both the AND/OR graph and the "subscription graph." However, the OZCF restriction ensures that Assumption 2.13 is satisfied, and thus the AND/OR graph cycles are not problematic (for an appropriate search algorithm, such as LDFS or AO$*_{\text{KLD}}$). Similarly, the hierarchical output grouping of the previous section ensures that cycles in the subscription graph cannot lead to infinite loops (or gross inefficiencies).

Finally, Algorithm 5.9 shows the changes for *state-abstracted* caching. The basic change is just as in Section 3.1.2: atomic subproblems are cached under the state-abstracted pair $(\text{EnterContext}(s, a), a)$ rather than (s, a) directly. However, in this setting additional changes are required to properly *contextualize* reachable sets of a state-abstracted subproblem in different contexts. (In Section 3.1.2 we were able to hide these details in the

Algorithm 5.8 DASH-A*: Caching

```
function GetAtomicSubproblemDesc(p)
    if cache[p] = undefined then
        cache[p] ← MakeAtomicSubproblemDesc(p)
    return cache[p]
```

Algorithm 5.9 DASH-A*: State-Abstracted Caching

```
function GetAtomicSubproblemDesc((S, a))
    S ← EnterContext(S, a)
    if cache[(S, a)] = undefined then
        cache[(S, a)] ← MakeAtomicSubproblemDesc((S, a))
    return cache[(S, a)]

function MakePairSubproblemDesc(S, l, r)
    d ← MakeSubproblemDesc(S, Contextualize(Contextualize(S, l.output), r.output))
    ... /* previous code from Algorithm 5.3 */
    if l.output is a singleton (in context) then
        ... /* previous code from Algorithm 5.3 */
        Subscribe(r.channel, λx DoPublish(d, MakePairSubproblemDesc(S, l, x)))
    else
        ... /* previous code from Algorithm 5.3 */
        Subscribe(l.channel, λx DoPublish(d, MakePairSubproblemDesc(S, x,
                                            Specialize(r, Contextualize(S, x.output)))))
    ... /* previous code from Algorithm 5.3 */
    return d

function Specialize(d, S)
    S ← EnterContextOf(S, d.input)
    if S = d.input then return d
    ... /* code from Algorithm 5.5 for atomic and Algorithm 5.7 for union cases */
    if d is pair then
        l′ ← Specialize(d.left, S)
        r′ ← Specialize(d.right, Contextualize(S, l′.output))
        return MakePairSubproblemDesc(S, l′, r′)

function Expand(leaf)
    ... /* previous code from Algorithm 5.5 */
    for [a_l, a_r] ∈ ImmediateRefinements(d.hsp.actions[0], d.input) do
        l ← GetAtomicSubproblemDesc((d.input, a_l))
        r ← GetAtomicSubproblemDesc((Contextualize(d.input, l.output), a_r))
        d.irefs.Insert(MakePairSubproblemDesc(d.input, l, r))
    ... /* previous code from Algorithm 5.5 */
```

solution concatenation operator.) First, MakePairSubproblem(S, l, r) is extended to take an additional argument S representing the *true* input set of the subproblem (since $l.input$ may represent a further abstracted version of S). Then, in every place the output set of a child d of a pair was used previously, it is now lifted into the context of S using Contextualize$(S, d.output)$. Contextualize(S, S') returns a set S'' that includes all of the variable values from S', as well as values from S for any additional variables not contained in S'. Similarly, when entering Specialize(d, S), we abstract S using EnterContextOf$(S, d.input)$, which retains the same variables of S that are present in $d.input$.

5.3.7 Correctness of DASH-A*

By augmenting Simple Decomposed AH-A* with subsumption, output grouping, decomposition, and state-abstracted caching, we arrive at the full DASH-A* algorithm. These extensions can dramatically compress the AND/OR graph, and thus lead to much faster planning, while preserving hierarchical optimality. This section proves correctness of this final algorithm, which includes the latest version of each function defined in the previous sections:

- From Algorithm 5.1: Expand, MakeRootNode
- From Algorithm 5.2: MakeNode, Connect, MarkUnrefinable
- From Algorithm 5.5: MakeAtomicSubproblemDesc
- From Algorithm 5.7: MakeSubproblemDesc, MakeUnionSubproblemDesc, DoPublish
- From Algorithm 5.9: Specialize, Expand, GetAtomicSubproblemDesc, MakePairSubproblemDesc

Lemma 5.13. *With hierarchical output grouping, all chains of the form x is a publication of ... is a publication of z are finite.*

Proof. Hierarchical output grouping ensures that the output set of each refinement publication of x is a strict subset of the output set of x. Thus, the length of any chain starting with x is at most $|x.output|$, which is finite. □

Lemma 5.14. *Each call to* Specialize *must terminate.*

Proof. In the worst case, Specialize(d, S) copies the entire "pair subgraph" below d, where atomic and union subproblems are leaves and pairs have children $d.left$ and $d.right$. While

the underlying AND/OR graph may have cycles, this pair subgraph cannot. This is easy to see, because the left and right subproblems of a pair d are passed as arguments to its constructor, and thus cannot contain the as-yet-unconstructed d.

This is not the entire story, however, because when specializing a pair subproblem, the resulting pair may immediately generate publications (before the call to SPECIALIZE returns). This can happen when the left or right child passes the equivalent-input test (and thus is not specialized, and may already have publications), or is a leaf corresponding to a cached atomic or union subproblem, or is a newly-specialized pair recursively containing one of these cases. We do not have to worry about these publications generating additional calls, since the newly created pair subproblem cannot yet have any subscribers. However, each published refinement itself is a newly created pair, and may generate its own publications, and so on. By Lemma 5.13, these chains must be finite, and thus SPECIALIZE terminates. □

Lemma 5.15. *Each call to* PUBLISH *must terminate.*

Proof. The issue is that a publication might be caught by a subscriber, leading to another publication, and so off to infinity. As in Lemma 5.13, however, hierarchical output grouping prevents this from occurring.

Consider a hypothetical infinite recursive sequence of calls to PUBLISH. First, note that under hierarchical output grouping each subproblem can be published at most once. Thus, this sequence must embody the publication of an infinite number of new subproblems. Moreover, because pair subgraphs cannot have cycles, it must include an infinite number of subproblems that are published by atomic nodes, or by unions published by atomic nodes, and so on — call these *atomic transitive refinements*.

Now, publications cannot create new non-leaf atomic subproblems, and thus the total set of all atomic subproblems that can publish refinements is fixed throughout a given call to DOPUBLISH. Moreover, each atomic transitive refinement d can publish at most a finite number of *types* of nodes (one per $\subseteq d.output$), and the publication chain from each such chain must be finite (by *Lemma* 5.13). Thus, the above infinite sequence must include some subproblem *type* of some parent node infinitely many times.

This *almost* establishes a contradiction. The last remaining subtlety is that while each atomic or union subproblem d can store at most one union refinement of each type in $d.refmap$, each such refinement type may be re-generated due the "caveats" described in Section 5.3.5 (specifically, specialization of the existing union, or the addition of a lower-cost publication), and in principle this could allow for a recursive sequence containing an infinite number of union subproblem tokens of the same type. To rule out this possibility, we argue that a given union cannot be specialized *while on the publication call stack*. Then, the remaining case (cost decrease) can happen at most a finite number of times, and PUBLISH must terminate.

Consider a new union subproblem u, just created and PUBLISHed within DOPUBLISH. For this union to be specialized before the call to PUBLISH returns, it would have to be embedded (somewhere within) in the right half of a left-expanding pair subproblem. Now, new pair subproblems can be generated by publications in only two ways. First, if caught by a right-expanding pair, the generated pair will always also be *right-expanding*. Second, if caught by a left-expanding pair, the publication is always embedded on the *left* of the generated pair. In this case, specialization of the right side of the pair may ensue. However, the right side of a left-expanding subproblem is always an atomic subproblem, and so this can not generate new pairs or specialize u. □

Finally, Lemma 5.9 needs to be updated to account for the enhancements to DASH-A*.

Lemma 5.16. *(Extension of Lemma 5.9). Consider any subgraph rooted at root.outer, where one child is selected at each non-leaf OR-node, both children are selected at each AND-node, and all leaf nodes have finite lower cost bound. Let $p_1, ..., p_n$ be the sequence of leaf subproblems corresponding to this subgraph, and $a_1, ..., a_n$ be the corresponding actions. Then for $i \in [1, n]$, $p_i.input \supseteq \text{ENTERCONTEXT}(\bar{O}_{[a_1,...,a_{i-1}]}(\{s_0\}), a_i)$. Moreover, if p_1, ... p_k is a prefix of this sequence containing only primitive actions, then for $i \in [1, k+1]$, $p_i.input = \{\text{ENTERCONTEXT}(\mathcal{T}(s_0, [a_1, ..., a_{i-1}]), a_i)\}$.*

Proof. The basic proof strategy of Lemma 5.9 still applies, but must be updated to account for state abstraction and output grouping.

An effective prefix can be extracted in the same way, except that union subproblems must be expanded into a *set* of their constituents, yielding a factored prefix. Then, any prefix of leaves in the subgraph is guaranteed to be a refinement of *some* sequence consistent with this factored prefix.

Due to state abstraction, however, the output set of a subproblem in the effective prefix is not necessarily equal to the input of the next subproblem. Instead, we prove by induction that for any effective prefix $p'_1, ..., p'_l$ corresponding to actions $a'_1, ..., a'_l$, $p'_l.input = \text{ENTERCONTEXT}(\bar{O}_{[a'_1,...,a'_{l-1}]}(\{s_0\}), a'_l)$. By the definition of state abstraction, if $p'_{l-1}.input = \text{ENTERCONTEXT}(\bar{O}_{[a'_1,...,a'_{l-2}]}(\{s_0\}), a'_{l-1})$ then it must also be the case that $p'_{l-1}.output = \text{ENTERCONTEXT}(\bar{O}_{[a'_1,...,a'_{l-1}]}(\{s_0\}), a'_{l-1})$.

However, to get from $p'_{l-1}.output$ to $p'_l.input$ we must consider more context, because some variables irrelevant to p'_{l-1} may be relevant to p'_l. For each variable v relevant to p'_l, let $j < l$ be the last action in the effective prefix for which v was relevant (or 0 if v was not relevant to any previous actions). Then, consider the path from p'_j to p'_l in the active subgraph, which travels up from p'_j to some least common ancestor pair p_{lca}, and then back down to p'_l. At each hop along this path where the parent is not a pair, the output set must remain identical. At pairs, the value of v is preserved by the calls to CONTEXTUALIZE, and

the assumption that no intermediary actions can affect v. Thus, p'_l receives the correct value for each relevant variable, or $p'_l.input = \textsc{EnterContext}(\bar{O}_{[a'_1,\ldots,a'_{l-1}]}(\{s_0\}), a'_l)$. Applying this result to the effective prefix of p_i yields the first part of the lemma.

For the second part, we must update the proof of the auxiliary result. First, the context-relative definition of singleton does not change the result because of the nesting property of contexts, and because if a singleton output is lifted into a singleton context, the result must also be a singleton. Second, due to output grouping, now the left child of a left-expanding pair can be the inner node of a union or atomic subproblem. However, in either case the children of this inner node must be pairs with an identical output set, which again violates the inductive hypothesis. Given this updated result, the second part of the lemma follows as before.

□

Theorem 5.17. *DASH-A* with* $\text{AO}*_{\text{KLD}}$ *is* hierarchically optimal *in OZCF hierarchies.*

Proof. Lemmata 5.7 and 5.10 are preserved by all of the above improvements (when the definition of *representation* is extended in the obvious way for union subproblems). Together with Lemma 5.16, this ensures the full DASH-A* can only return hierarchically optimal solutions. Lemma 5.10 (substituting Lemma 5.16 for Lemma 5.9 in the proof) guarantees that DASH-A* cannot get stuck, and so it only remains to prove termination.

As mentioned above, the OCZF restriction ensures that both UpdateAncestors and MarkUnrefinable must terminate. Lemmata 5.14 and 5.15 establish that each Expand must terminate, despite the presence of cycles. Finally, because the total number of unique atomic subproblems constructed by GetAtomicSubproblemDesc is finite (at most $|(\mathcal{A} \cup \hat{\mathcal{A}})|2^{|\mathcal{S}|}$), only a finite number of Expand operations can be carried out before termination.
□

5.3.8 Analysis of DASH-A*

This section first demonstrates a class of problems for which DASH-A* is *exponentially faster* than previous algorithms, including DSH-UCS, AH-A*, and Simple Decomposed AH-A*. It then concludes with a discussion of future directions for improving DASH-A*.

Consider a planning problem consisting of n subproblems $p_1, \ldots, p_n$ in sequence. For example, perhaps we are building a large software system, where each p_i corresponds to writing a module with a fixed API, which p_{i+1} builds upon to perform its function. Each subproblem p_i may have exponentially many goal states (e.g., source code), but the variables relevant to later subproblems (e.g., the API) are the same in every goal state. Now, consider a hierarchy where $\mathcal{H}_0$ expands to $[\textsc{Solve}(p_1), \ldots, \textsc{Solve}(p_n)]$. In the optimistic outcomes $S_{i+1} = \bar{O}_{\textsc{Solve}(p_i)}(S_i)$, the conditions established by p_j for $j \leq i$ are known, but exponentially

many combinations of irrelevant variables from the previous subproblems are also allowed (represented implicitly). To solve this problem, DASH-A* can choose to first solve subproblem p_n from optimistic set S_n, finding the optimal plan to reach s_* (the goal of the combined problem). Then, it can choose to solve p_{n-1} next. Crucially, while refinements in p_{n-1} will generate specializations of the output set S_n, the variables *relevant* to p_n remain the same, and so DASH-A* automatically shares the same atomic subproblem corresponding to p_n between all refinements in p_{n-1}. It continues working backwards until an optimal solution to the entire problem is found, in time *polynomial* in n. In contrast, previous angelic algorithms (e.g., AH-A*) will be forced to consider the exponential number of combinations of solutions to the various subproblems, and those that just exploit state abstraction without angelic descriptions (e.g., DSH-UCS) will be forced to evaluate the exponentially many solutions to each individual subproblem.

In reality, things usually do not decompose so cleanly. Suppose that we tweak our example slightly, so that one of the state variables v_i connecting p_i with p_{i+1} has different values in different goal states of p_i. In this case, DASH-A* again begins with p_n, and may get most of the way to a concrete solution – until it gets stuck, needing to know the value of v_{n-1}. At this point it moves backwards to p_{n-1}, eventually producing sets with known values (and costs) for each value of v_{n-1}, and p_n will be *specialized* on these inputs. However, the initial effort spent on the general version of p_n is not wasted; subsumption bounds ensure that, rather than solving p_n from scratch for each such value, we quickly "patch up" our original partial solution, in the best case taking time *logarithmic* in the optimal solution length of p_n. This interleaving of refinements to earlier subproblems and specialization of later subproblems continues until the entire problem is solved. In the worst case, each subproblem p_i is solved once for each possible value of the connecting variable v_{i-1}, which is the best that we could hope for.

Thus, by combining features of previous algorithms such as AH-A* and DSH-UCS, DASH-A* can be exponentially more efficient than *either* on a given problem. In the worst case, the number of description evaluations performed by DASH-A* can be no greater than AH-A*, and typically only a constant worse than DSH-UCS (due to HLA evaluations, in addition to the primitive evaluations done by both algorithms).

The version of DASH-A* presented is the simplest angelic algorithm we could devise that could effectively exploit state abstraction and decomposition while efficiently handling cycles and reusing work spent refining subproblems with abstract input sets. A number of further improvements are possible, the details of which we leave for future work.

A first observation is that the algorithm presented above only takes *opportunistic* advantage of subsumption. Considerably greater sophistication is possible. For instance, one could attempt to use subset-lookup data structures to efficiently discover subsumption relationships not created by specialization. One could also *actively* propagate bounds along subsumption links, e.g., as a part of UpdateAncestors, to ensure that each node is assigned the best-

possible bound derivable from information in the current graph (the algorithm above only does this at the time the link is first discovered).

Another set of improvements involve *laziness*, and the granularity of search operations. With some complications, it is possible to construct a version of DASH-A* wherein the optimistic description evaluation for each atomic subproblem is an explicit *search step* (alongside EXPAND), rather than an automatic part of creating the subproblem. Along the same lines, each publication receipt could be made a search step as well, avoiding the potentially many operations performed by DASH-A* handling publications at nodes that are not currently thought to be part of an optimal solution.

Next, versions of the algorithm that can handle more expressive types of angelic descriptions could lead to faster convergence and fewer overall subproblem expansions.[9] For one, non-*simple* valuations (where the costs assigned to states in the reachable set can vary) could much more accurately capture reachability information in some cases. Another open problem is how pessimistic descriptions can be used for *pruning* in this framework, as we saw for AH-A*. The next section describes one way to extend DASH-A* to use pessimistic descriptions for *bounded suboptimal* search, but more powerful applications of pessimistic descriptions are likely possible.

Finally, the overhead of top-down search may be quite high in some cases, for example with very deep and narrow hierarchies (e.g., for navigation subproblems). In such cases, one might consider *hybrid* algorithms that use alternative search strategies at different nodes in the tree; for example, one could embed a version of DSH-UCS to more efficiently handle navigation subproblems.

5.4 Suboptimal Search

While our focus thus far has been on hierarchically optimal algorithms, in practice it may be desirable to trade off a bit of solution quality for (often, a lot of) computation time. Given an optimal search algorithm, it is quite easy to construct *bounded suboptimal* variants that do just this, searching for a solution whose cost is at most a given multiple of the true hierarchically optimal cost. This section outlines preliminary work in this area, including a recipe for constructing bounded suboptimal versions of the above algorithms, and a theoretical analysis demonstrating that bounded suboptimal search may be especially effective for angelic hierarchical planning.

Weighted A* (see Section 2.1.3.3) is a simple variant of A* in which the heuristic value is multiplied by a weight $w \geq 1$, and the cost of the discovered solution is guaranteed to be at most w times the optimal solution cost. A simple way to make a bounded suboptimal variant

[9] The opposite tack may also be of interest, i.e., projecting to a particularly constrained normal form to increase the potential for sharing.

of AH-A* or DASH-A* is to just directly apply this idea. The only stumbling block is that there is no longer a single "heuristic"; instead, there are optimistic cost bounds computed by HLAs at various levels, which are added together (along with the primitive action costs) to produce the overall bound for the plan.

A first attempt to apply weighted A* would be to leave primitive costs as they are, and multiply all HLA cost bounds by w. While correct, this would be ineffective. For one, it may take many refinements of $\mathcal{H}_0$ before primitive actions are generated, and during this time the weighting has *no effect* on search (because all plan costs are simply scaled by w). Moreover, one intuition behind weighted A* is that the weight w helps correct for the *bias* in the heuristic, which tends to underestimate costs in order to preserve admissibility. In an angelic search, however, we expect each HLA descriptions to have a *different* bias – in particular, higher-level HLAs may tend to have greater bias, because they operate at a greater level of abstraction. We can exploit this intuition by using a *different* weight $w_a > 1$ for each HLA, so that there is a smoother continuum from primitives ($w_a = 1$) to the top-level $\mathcal{H}_0$. It is easy to see that the resulting algorithm has a sub-optimality bound equal to the *greatest* weight w_a, and may be significantly more effective than ordinary weighted A*.

Even more promising is the idea of using *pessimistic descriptions* to aid in bounded suboptimal search. These bounds can serve at least two distinct purposes in this setting. First, if action a has optimistic cost bound o and pessimistic bound p, it can be assigned heuristic cost $\min(p, o \cdot w_a)$, or perhaps even $\min(p, o \cdot w_a, \frac{p+o}{2})$, while preserving the weighted A* guarantee. This ensures that we do not overestimate costs too much when we have pessimistic information indicating that the optimistic bound is fairly accurate.[10] Second, if the current optimistic bound on the best plan is o, we can use pessimistic bounds to immediately *commit* to any plan with pessimistic bound $p < o * w$. Unlike in optimal search where bounds must become exact for commitment, here we may be able to commit to a candidate plan at a high level given very few refinements.

AH-A* and DASH-A* can very easily be extended to compute pessimistic bounds as well as optimistic ones, by simply replacing optimistic input and output sets with *pairs* (S_O, S_P) of optimistic and pessimistic reachable sets.[11] The corresponding bounds (and perhaps weighted combinations as well) can then be summed along a plan in AH-A*, or incorporated into summaries in DASH-A*.

We conclude with a rough theoretical analysis indicating that bounded suboptimal search and angelic hierarchical planning may prove to be an especially effective combination in

[10] Along these lines, an even more promising approach would be to use Explicit Estimation Search (EES) (Thayer and Ruml, 2010), which can explicitly make use of non-admissible bounds, and also takes the amount of work left to refine a plan into a solution into account.

[11] An even better solution in DASH-A* would be to use separate AND/OR graphs for optimistic and pessimistic bounds, along with a third graph for their combinations. This would allow sharing of information about a given optimistic set O_s among all (S_o, S_p) pairs in which it is encountered.

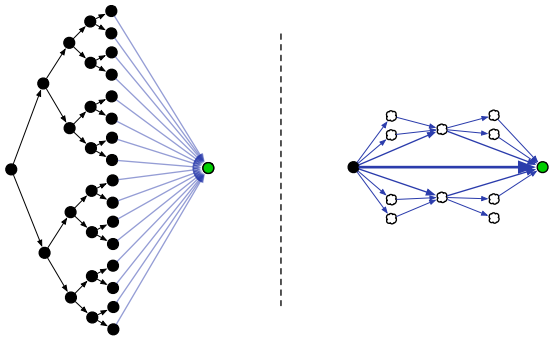

Figure 5.17: A graphical depiction of the potential power of bounded suboptimal hierarchical search. Suppose that resolving half the uncertainty in our heuristic bounds requires expanding half of the length (left) or hierarchical depth (right) of an optimal solution. The total number of node expansions required can be much lower in the hierarchical case.

practice. Consider a very simple primitive planning problem with two unit-cost primitive actions, in which *any* sequence of 1024 primitives is an optimal solution (with cost 1024). Suppose that there are no repeated states (i.e., the state is just a bit string recording which actions have been done thus far). Finding a provably optimal solution in this planning problem with anything less than a perfect heuristic has cost $> 2^{1024}$.

Now, suppose that we are willing to settle for a bounded suboptimal solution with $w = \frac{4}{3}$, and moreover, an oracle has already supplied us with a solution **a** with cost 1024 (not so great a feat, in this case). To prove that **a** is acceptable given our sub-optimality bound, we must prove that the optimal solution has cost ≥ 768.

Now, our heuristic matters (a lot): with no heuristic, we would have to run A* for $> 2^{768}$ steps, whereas a perfect heuristic could prove the bound with no search whatsoever. In the more interesting middle ground, suppose that $h(s) = \frac{c^*(s)}{2}$, i.e., the heuristic underestimates the true cost by a factor of 2. In this case, we can prove the bound with only 2^{512} steps of A* — an impressive reduction from 2^{768}, but still far from tractable (see the left side of Figure 5.17).

Suppose that, instead, we used AH-A* to prove the bound. Again keeping things simple, suppose that each HLA h at level i has two refinements, each containing two HLAs at level $i+1$ (and the primitives lie at level 10). Moreover, suppose that the accuracy of the optimistic descriptions increases *linearly* with depth, so that an HLA at level i underestimates the true cost by a factor of $0.5 + 0.05i$. Thus, the error at the top-level matches our state-space heuristic, and at the bottom level the primitive descriptions are exact. Now, suppose that we run AH-A* with this hierarchy, always refining the highest-level action remaining until the bound is established. It is easy to see that we hit the bound when all plans are at level 5, at which there are 2^{31} possible plans, which is within the realm of tractability (see the

right side of Figure 5.17).

Thus, angelic bounded suboptimal algorithms have the potential to be astronomically faster than non-hierarchical versions. By tweaking the numbers slightly, it is easy to generate even more impressive results; for instance, achieving a sub-optimality bound of 1.4 requires considering about 2^{440} primitive plans, versus 13 hierarchical plans. Of course, these precise results depend on the highly idealized problem structure and heuristic accuracies considered, which may not be representative of many real-world problems. Nevertheless, the ability of angelic bounded suboptimal algorithms to *divide* uncertainty across a plan and reduce it *uniformly* via balanced refinement is unique, and may prove to be a very effective tool for real-world decision-making.

Chapter 6

Experimental Results

This chapter provides empirical evidence for the claims made in this thesis. The empirical performance of (hierarchically) optimal algorithms introduced in previous chapters is compared on several planning domains: the nav-switch domain introduced in Section 2.1.2.1, the discrete manipulation domain introduced in Section 2.1.2.3, and a third "bit-stash" domain based on the example discussed in Section 5.3.8. After summarizing the implemented algorithms and experimental set-up, we provide empirical results and discussion for each planning domain in turn.

6.1 Implemented Algorithms

All of the search algorithms discussed in this thesis have been implemented in the Clojure programming language, along with the aforementioned planning domains. We first briefly review the algorithms tested, including a brief discussion of relevant implementation details and variants. The algorithms are divided into four broad classes.

- "Flat" search algorithms, which operate directly on a planning domain without using a hierarchy (see Section 2.1):
  - **UCS**: Uniform-Cost Search, i.e., A* with heuristic $h(s) = 0$ (see Section 2.1.3.2).
  - **A***: A* search with a domain-specific, consistent heuristic function (see Section 2.1.3.2).
- Simple hierarchical search algorithms, which use a planning domain and a hierarchy (see Section 2.3.2) but no angelic descriptions:
  - **DFBB**: Depth-First Branch-and-Bound, the algorithm used for optimal search by previous hierarchical planning systems such as SHOP2. This is a depth-first

graph search over the space described in Section 2.3.2.3, with pruning of plans whose cost is greater than that of the current best-known solution. We initialize this bound with the most accurate estimate available, and randomize the order of successors.

  - **H-UCS**: Hierarchical Uniform-Cost Search, a simple OR-search over "hstates" consisting of a state and remaining plan to do from this state, with repeated hstate elimination (see Section 2.3.2.3).
  - **SAHTN**: State-Abstracted Hierarchical Task Network planner, which exhaustively enumerates all hierarchically optimal solutions, using decomposition and state-abstracted caching to gain efficiency (see Section 3.2). The implemented version includes a "Dijkstra" modification that enables its application to cyclic hierarchies (Wolfe et al., 2010).
  - **DSH-LDFS**: Decomposed, State-abstracted, Hierarchical Learning Depth-First Search, which searches over the decomposed graph explored by SAHTN in cost order, and thus avoids examining parts of the search space that cannot be reached in less than the hierarchically optimal solution cost (see Section 3.3.1).
  - **DSH-UCS**: Decomposed, State-abstracted, Hierarchical Uniform-Cost Search, which performs the same basic operations as DSH-LDFS in a "flattened" manner (using a single priority queue) to reduce overhead.

- Variants of Angelic Hierarchical A* (see Section 5.1), which use angelic descriptions (see Chapter 4) but not decomposition or state abstraction:
  - **AH-A*-opt**: Optimistic Angelic Hierarchical A*, which extends H-UCS to incorporate heuristic cost bounds provided by optimistic descriptions (see Section 5.1.1). The implemented algorithm eliminates duplicate "hstates," caches the optimistic description evaluations for plan suffixes, and always refines the first HLA in a candidate plan.
  - **AH-A***: Full Angelic Hierarchical A*, which includes pruning using upper cost bounds provided by pessimistic descriptions (see Section 5.1.2). Specifically, strict pruning is computed over the set of all generated plans, and weak pruning (including duplicate plan elimination) is computed only over generated but unexpanded plans to guarantee the acyclic pruning requirement. Optimistic and pessimistic descriptions on suffixes are cached, and the first HLA in a plan is always refined.
  - **AH-A*-strict**: A variant of the previous algorithm that only carries out strict pruning (and also eliminates duplicate plans).

- Variants of DASH-A* (see Section 5.3), which use angelic descriptions in a decomposed hierarchical search:

- **DASH-A***: Decomposed, Angelic, State-abstracted, Hierarchical A*, which combines the decomposed, state-abstracted AND/OR graph used by DSH-LDFS with angelic descriptions and additional techniques (see Section 5.3). The implemented version of the algorithm uses optimistic, but not pessimistic angelic descriptions. This version uses default settings: state-abstracted caching, local output grouping (see Section 5.3.5), expands the rightmost refinable HLA in a plan, and searches using AO$*_{\text{KLD}}$ (see Section 2.2.4). All variants of DASH-A* discussed below use these same settings except where otherwise noted.
- **DAcH-A***: A variant of DASH-A* with subproblem caching, but no state abstraction (see Section 5.3.6).
- **DAxH-A***: A variant of DASH-A* with no subproblem caching whatsoever (i.e., without the improvements of Section 5.3.6).
- **DASH-A*-ldfs**: A variant of DASH-A* with the AO$*_{\text{KLD}}$ search strategy replaced by LDFS (see Section 2.2.3.3).
- **DASH-A*-hog**: A variant of DASH-A* with hierarchical output grouping (see Section 5.3.5).
- **DASH-A*-first**: A variant of DASH-A* that always expands the *first* HLA in a plan.
- **DASH-A*-sh**: A variant of DASH-A* that always expands a *shallowest* HLA in a plan. Each HLA type is manually assigned a depth, with $\mathcal{H}_0$ being shallowest, and lowest-level HLAs (such as NAV and REACH) being deepest. We also consider combinations of this refinement strategy with the other variants discussed above.

Next, we describe implementation details shared between the above algorithms.

- States are represented as persistent (immutable) maps from state variable names to variable values. Primitive actions are tuples consisting of a name, precondition map, effect map, and cost. For the flat search algorithms (UCS and A*), applicable primitive actions are generated using a successor generator data structure (see Section 2.1.3.4).
- Closed lists and caches are (mutable) hash tables, keyed on states (or state sets) and action sequence names (where applicable).
- Factored state sets are represented like ordinary states, except that each state variable name maps to a *set* of possible values. High-level actions and their refinements and angelic descriptions are represented with hand-coded procedures that operate on these state sets (see Section 4.3.3).

Finally, the configuration used for running experiments is as follows. For each domain, a subset of algorithms and problem configurations of increasing size was selected. Then, five

random instances of each problem size were generated, and each algorithm was tested once on each instance (except in the bit-stash domain, in which each algorithm was run three times on the single instance for each configuration). Each test was conducted on a single Intel Xeon 3.00 GHz CPU with 1.5 GB of RAM running Ubuntu Linux, inside a fresh 64-bit Java 6 Server VM that was warmed up for 10 seconds to ensure all algorithms were fully (JIT) compiled before measurements were taken. Algorithms were run until they returned a (hierarchically) optimal solution, or they exceeded 1 hour of run-time or 512 MB of heap space. A number of statistics were collected on each run, including CPU run-time and a count of the total number of primitive and angelic descriptions evaluated. For each algorithm and problem configuration, we report medians of these statistics over the instances tested.

6.2 Nav-Switch

The first results we report are on instances of the nav-switch domain (see Section 2.1.2.1). Each problem instance includes 20 randomly generated switch locations on an $n \times n$ grid, where n is increased from 5 to 500. For (flat) A* search, the heuristic assumes that only low-cost navigation actions will be taken (i.e., $h(s) = 2(|s_x - g_x| + |s_y - g_y|)$). For the hierarchical algorithms, the optimality-preserving hierarchy described in Section 2.3.2.2 is used. Angelic hierarchical algorithms are given exact optimistic and pessimistic descriptions for Nav, an optimistic bound for Go equal to the A* heuristic, and a pessimistic bound for Go corresponding to the exact cost of directly navigating to the goal without further Flips.

While the nav-switch domain is fairly uninteresting from a combinatorial planning perspective (the state space contains only $2n^2$ states), it can still provide considerable insight into the performance of hierarchical planning algorithms.

First, while the state space is small, the number of potential hierarchical plans is astronomically large: with 20 switch locations, there are on the order of 20! possible acyclic plans at the Nav level. In particular, along the way to the goal the agent can NAVigate to and Flip at any sequence of zero to all 20 switch locations. Hierarchical planning algorithms that cannot exploit the redundancy in these plans can quickly get lost in this space, and thus perform much worse than a simple state-space search algorithm like A*.

On the other hand, there is also something to be gained from hierarchy in this domain. Flat algorithms such as uniform-cost search or A* (with anything but a perfect heuristic) will typically have to explore at least half of the state space to find an optimal solution, and thus have $O(n^2)$ runtime. In contrast, angelic search algorithms are given exact cost bounds for Nav; if they can quickly zero in on an optimal high-level plan (consisting of Nav HLAs and primitive actions), they can in principle refine this to an optimal primitive plan in linear time, leading to an overall $O(n)$ runtime.

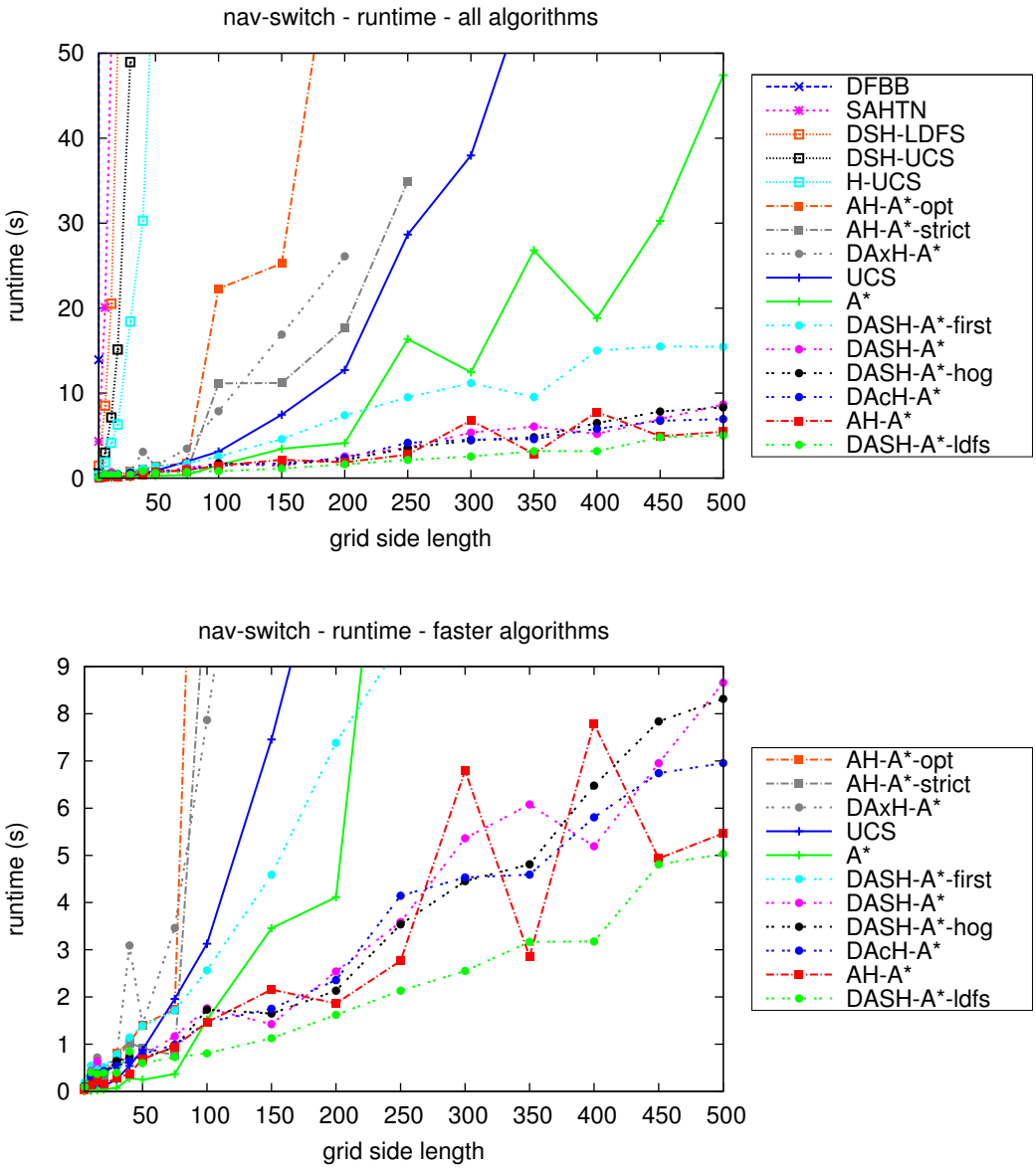

Figure 6.1: Median run-times for each implemented search algorithm, over five $n \times n$ nav-switch instances with 20 randomly generated switch locations. Algorithms are sorted in order of decreasing run-time. Top: results for all implemented search algorithms. Bottom: detail view including only the faster algorithms.

Figure 6.1 shows median runtimes for the algorithms discussed in the previous section on these nav-switch instances. As expected, baseline non-hierarchical algorithms UCS and A* exhibit runtime roughly quadratic in n, with A* being roughly twice as fast (depending on the accuracy of the heuristic in a particular selection of random instances).

At the far left, the non-angelic hierarchical algorithms DFBB, SAHTN, DSH-LDFS, DSH-UCS, and H-UCS all perform very poorly for essentially the reason discussed above. Specifically, the hierarchical search space contains roughly $20 \times 2n^2$ "hstates" of the form $(s, [\text{NAV}, \text{FLIP}, \text{GO}])$, which is much larger than the reachable state space. For SAHTN the situation is even worse; it exhaustively solves for all 21^2 possible NAVigation plans between key locations, nearly all of which will not appear in any high-quality plans. DFBB performs the worst of the bunch, since it may explore each "hstate" many times along different paths from the root.

Next, AH-A*-opt and AH-A*-strict variants of Angelic Hierarchical A* perform roughly a constant factor worse than flat A*. The heuristic bounds provided by optimistic descriptions effectively rule out almost all of the possible high-level plans, avoiding the pathologies exhibited by the non-angelic hierarchical algorithms. However, without weak pruning they are doomed to explore all $O(n^2)$ optimal solutions for each NAVigation action, and may do so for several promising high-level plans. In contrast, full AH-A* with weak pruning exhibits roughly linear run-time, since weak pruning ensures that each navigation action is solved with only $O(n)$ evaluations. Its performance can vary greatly, however, depending on the number of promising high-level plans that it must explore.

Finally, most of the DASH-A* variants exhibit the best performance, with fairly consistent, roughly linear runtime in problem size. Interestingly, they manage to do so without pessimistic descriptions or pruning. The reason is that they refine the final HLA in each plan first, generating sequences containing only NAV and primitive actions before refining any NAVigation actions. Because the optimistic descriptions of NAV are exact, an optimal high-level solution is discovered at this level, and only NAVigation actions in this solution are ever refined. Moreover, each NAVigation action can be extended to an optimal primitive solution with only linearly many HLA expansions, so long as ties are broken consistently at OR-nodes.

Among the DASH-A* variants, only DAxH-A* and DASH-A*-first seem to perform super-linearly. The reason is that the former cannot share identical NAVigation subproblems that arise in different plans, and the latter is not able to zero-in on an optimal high-level solution before beginning navigation planning. Variants DASH-A*, DAcH-A*, and DASH-A*-hog perform almost identically, as would be expected: state abstraction and hierarchical output grouping do not have any effect in this domain. Finally, DASH-A*-ldfs performs the best, due to lower overhead updating the summaries of ancestors than AO$*_{\text{KLD}}$.

6.3 Discrete Manipulation

Next, we report results on instances of the discrete manipulation domain (see Section 2.1.2.3). Each problem instance is a 20×20 grid with a 2×3 table in each corner and a 3×3 table in the center, and a gripper radius of 1. Difficulty is controlled by increasing the number of object tasks that must be completed by the robot: each object begins at a random location on one of the tables, and must be delivered to one of 4 randomly chosen locations on a different table. For each possible pickup and putdown location, two possible base positions are sampled. Because each object can be put down at multiple locations, the reachable state space is exponential in the number of objects.

On this domain, we consider a subset of the algorithms tested in the nav-switch domain. In particular, we did not implement a general heuristic or pessimistic descriptions in this domain, so we omit A* and the AH-A* algorithms with pruning. Hierarchical algorithms use the hierarchy described in Section 2.3.2.2, and the angelic descriptions described in Section 4.3.3

In addition to the features captured by the nav-switch instances, discrete manipulation instances include an exponential state space, deep hierarchies with many choice points, large reachable sets, and ample opportunities for exploiting state abstraction. Moreover, as discussed in the next chapter, the domain is an accurate representation of the structure of real mobile manipulation problems, and previous work has applied a version of this hierarchy for mobile manipulation planning on a physical robot (Wolfe et al., 2010).

Figure 6.2 shows run-times and counts of (primitive and optimistic) description evaluations for the algorithms tested, on discrete manipulation instances of increasing difficulty.

UCS and H-UCS scale poorly, due to the exponentially large state space (and even larger "hstate" space). AH-A* performs better, because its optimistic descriptions effectively rule out many courses of action at the high-level, but without pruning it too quickly falls short.

Interestingly, despite its exhaustive nature, SAHTN scales significantly better; it seems that state abstraction is especially powerful in this domain. Manipulations involving only a single object on a given table do not depend on the positions of other objects, and exploiting this observation can dramatically cut down on the combinatorial nature of the domain. As expected, DSH-LDFS and DSH-UCS improve on this further, by avoiding expanding HLAs corresponding to tasks that cannot be reached with less than the hierarchically optimal solution cost.

Finally, the DASH-A* variants perform the best over the range of problem sizes tested; while their runtime is still exponential in the number of objects, it is an order of magnitude faster than SAHTN and many orders of magnitude faster than UCS or H-UCS. However, it is not clear whether this trend would continue for larger problem sizes, as the runtime of

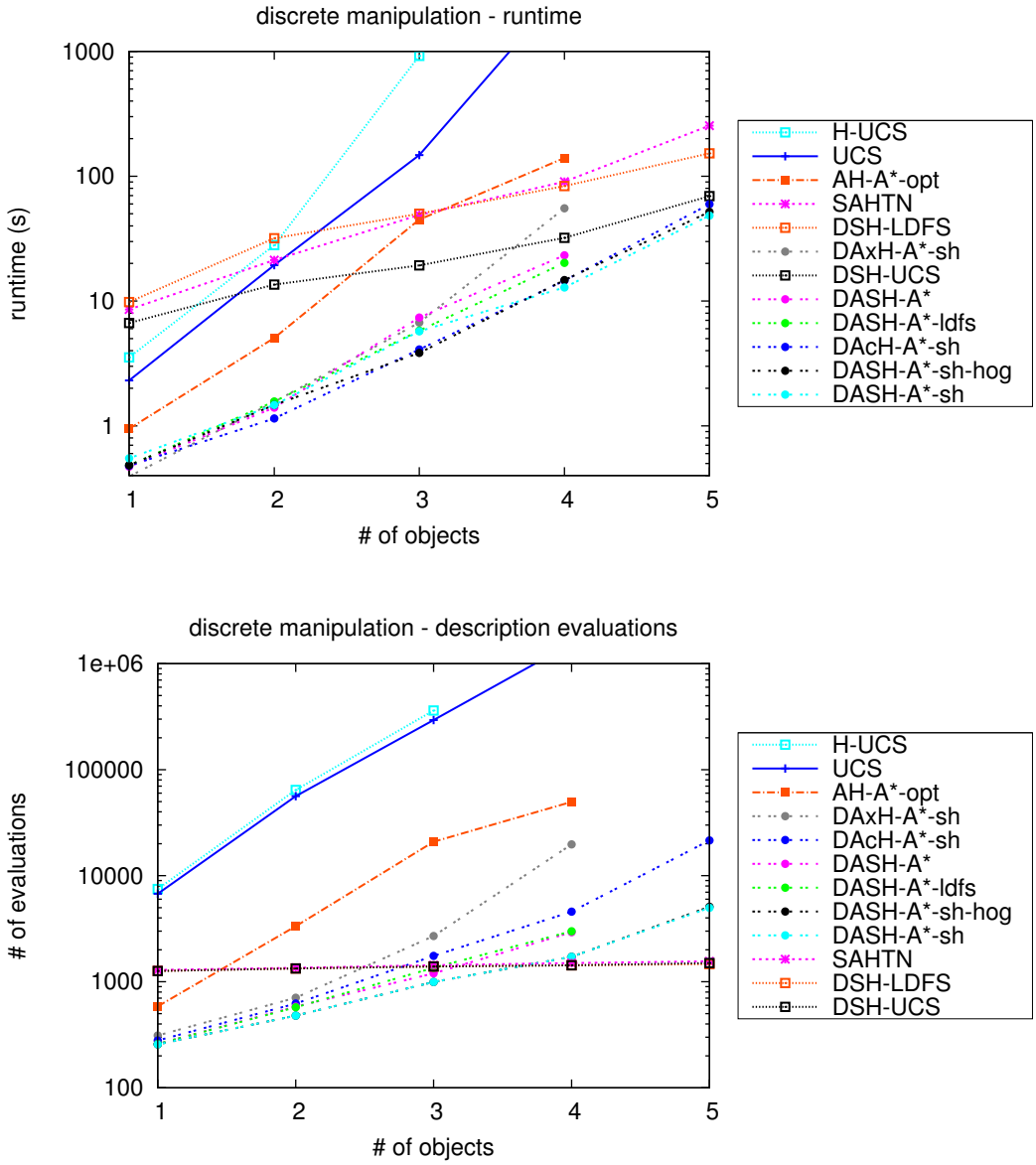

Figure 6.2: Median run-times (top) and number of description evaluations (bottom) for search algorithms, over five discrete manipulation instances with randomly generated object positions and goals. Algorithms are sorted in order of decreasing run-time or evaluations.

DSH-UCS is also growing slower over the tested range of problem sizes than the DASH-A* algorithms. This is perhaps surprising, since the heuristics of DASH-A* should result in it examining fewer primitive subproblems than non-heuristic algorithm DSH-UCS (under the same refinement strategy). There are several complementary explanations for this scaling behavior. First, DASH-A* incurs significant overhead due to its optimistic description evaluations on state sets (which have not been optimized, and could likely be sped up substantially), top-down search strategy, and additional data structures and bookkeeping, all of which can grow with the complexity of the problem instance and size of the resulting AND/OR graph. Second, while in a problem with few objects the optimal task ordering may be obvious at the high level, as the problems grow larger the number of near-optimal task orderings grows quickly. Moreover, the number of state-abstracted contexts each subproblem appears in also increases, and every such context must prove heuristically expensive for DASH-A* to avoid expanding it.

All variants of DASH-A* perform similarly, except for DAxH-A*, which begins to fall behind on larger instances due to the lack of caching. The best variant is DASH-A*-sh, which always expands the *shallowest* HLA in a candidate plan, effectively focusing on the regions of greatest uncertainty first. The other options are examined in combination with this feature, except for LDFS (which is incompatible with the shallowest choice, which does not satisfy the consistency property that LDFS requires). Again, hierarchical output grouping has little effect, because multiple refinements with the same output subset do not arise in this domain. Perhaps most interestingly, despite the competitive performance of DSH-UCS in this domain, ablating state abstraction has little effect on the performance of DASH-A*. It seems that the angelic heuristics already effectively rule out most of the repeated subproblems that would otherwise arise, and so state abstraction has little more to add.

The bottom chart plots median description evaluations for the same experiments. The results are qualitatively similar, with a few interesting deviations. Most noticeably, the description counts for the non-angelic decomposed algorithms are roughly constant across problem size. The reason is that each primitive action is evaluated only once in each state-abstracted context, and the number of such possibilities is dominated by the large number of potential base navigation actions. The angelic algorithms manage to avoid many of these primitive evaluations, but they also evaluate more and more optimistic descriptions with increasing problem size. Their relative ordering is roughly the same as for run-time, with differences somewhat more pronounced. It seems that expanding the shallowest HLA does have a significant effect on the number of description evaluations, but much of this difference is offset by the greater overhead incurred by a balanced refinement strategy.

6.4 Bit-Stash

Finally, we report results on a synthetic "bit-stash" domain roughly corresponding to the example in Section 5.3.8, which was constructed to illustrate that the combination of angelic descriptions and state abstraction in DASH-A* can exponentially outperform either feature alone. This domain consists of a sequence of N identical subproblems. The task in each subproblem is to "stash" k ones in a bit-stream of length 2^m. Each subproblem solution consists of 2^m primitive actions, each of which sets the i^{th} bit in turn. Setting the i^{th} bit to 0 has cost 1, and setting it to 1 has cost $1 + i$; thus, the optimal solution is always to stash the one bits in the first k positions.

The domain captures the features described in Section 5.3.8. First, each subproblem has a very large number of solutions, which is problematic for algorithms such as DSH-UCS that lack high-level heuristics. Second, there are an exponential number of combinations of solutions to different subproblems, which is problematic for algorithms such as AH-A* that lack state abstraction.

We consider a simple hierarchy for this domain. The top-level action $\mathcal{H}_0$ is right-recursive, generating a $\textsc{SetK}(s, 0, 2^m, k)$ HLA followed by a recursive $\mathcal{H}_0$ (or Goal if all subproblems are solved). $\textsc{SetK}(s, l, u, k)$ generates a primitive action sequence that sets bits l through u in subproblem s, including exactly k one bits. Its refinements $[\textsc{SetK}(s, l, l + \frac{u-l}{2}, x), \textsc{SetK}(s, l + \frac{u-l}{2}, u, k - x)]$ partition the sequence, ranging over $x \in [0, k]$ (or corresponding primitives, if $u - l = 1$). Its optimistic description possibly sets each bit in its region to 0 or 1, with cost bound $u - l + w(l + (l + 1) + \ldots (l + k - 1))$, where $w \leq 1$ is a discount factor modeling heuristic inaccuracy.

We compare UCS, DSH-UCS, AH-A*-opt, and DASH-A* on instances of this domain of increasing size, with $k = 3$ and $w = 0.9$. We consider three scenarios: increasing the subproblem size for a fixed number of subproblems, increasing the number of subproblems of a fixed size, and increasing both subproblem size and count simultaneously.

Figure 6.3 shows median runtimes for these four algorithms as we increase *either* subproblem size or count.

As we increase subproblem size for a single subproblem, all algorithms except DASH-A* quickly fail to scale. As expected, the best competitor is AH-A*, whose heuristics can rule out many of the suboptimal plans at the high-level. However, AH-A* is still at a disadvantage to DASH-A*, because the larger number of potential plans are represented much more compactly in the latter algorithm's decomposed data structures.

As we increase the number of subproblems for fixed $m = 3$, we also see the expected result: UCS and AH-A* quickly fall victim to the exponential number of combinations of solutions to different subproblems. In contrast, state abstraction in DSH-UCS and DASH-A*

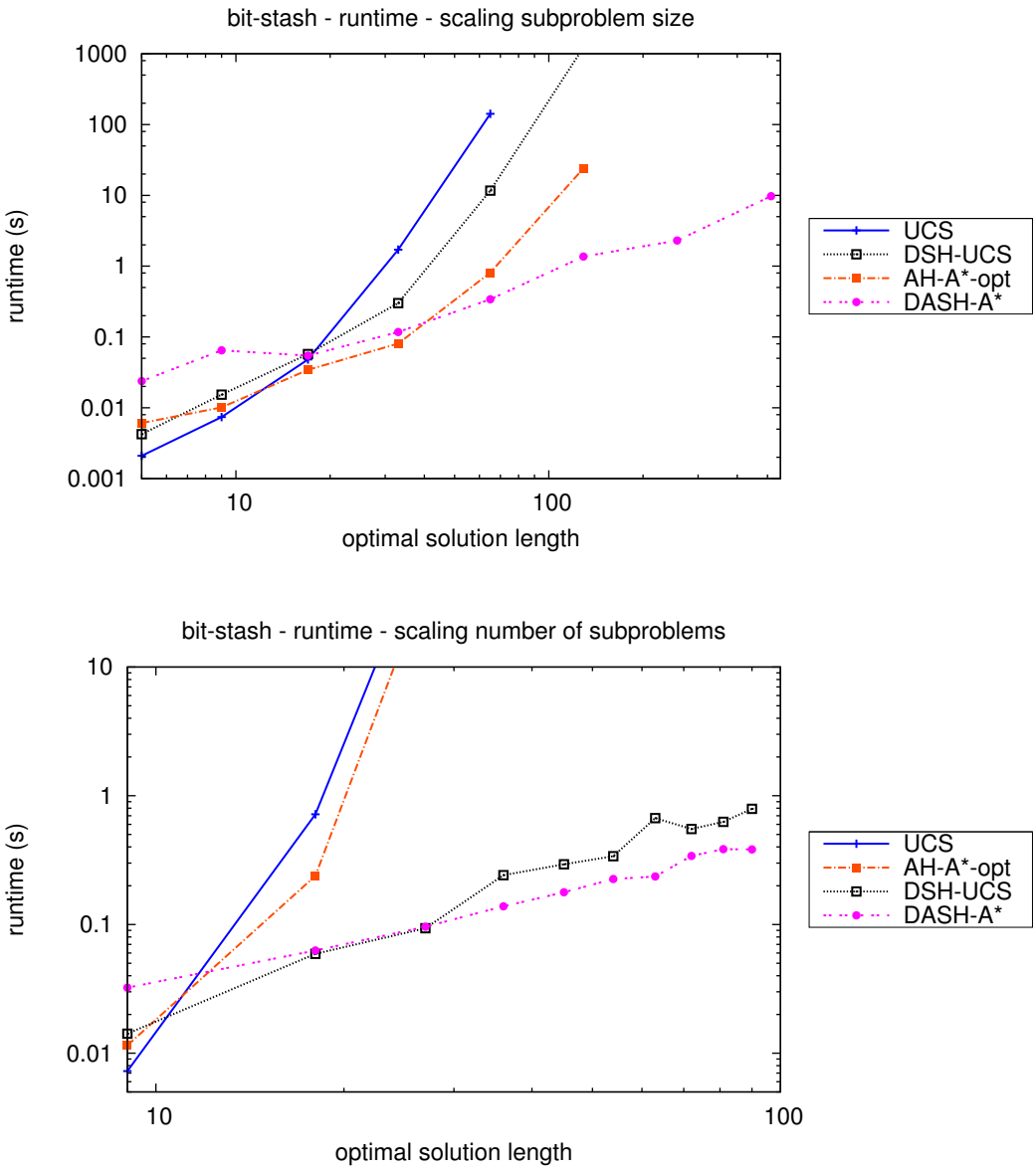

Figure 6.3: Median run-times for three runs each of four algorithms, over bit-stash instances of increasing subproblem size (top) or count (bottom).

prevents them from considering these combinations, allowing them to scale gracefully with subproblem count.

Finally, Figure 6.4 shows results for simultaneously scaling N and m. As suggested by the previous results, neither DSH-UCS or AH-A* can cope with both types of complexity, and all algorithms except DASH-A* scale very poorly. The bottom chart depicts the same results on a linear scale, where it is clear that the other algorithms are scaling exponentially with problem size, whereas DASH-A* seems to scale roughly quadratically (as would be expected, as discussed in Section 2.2).

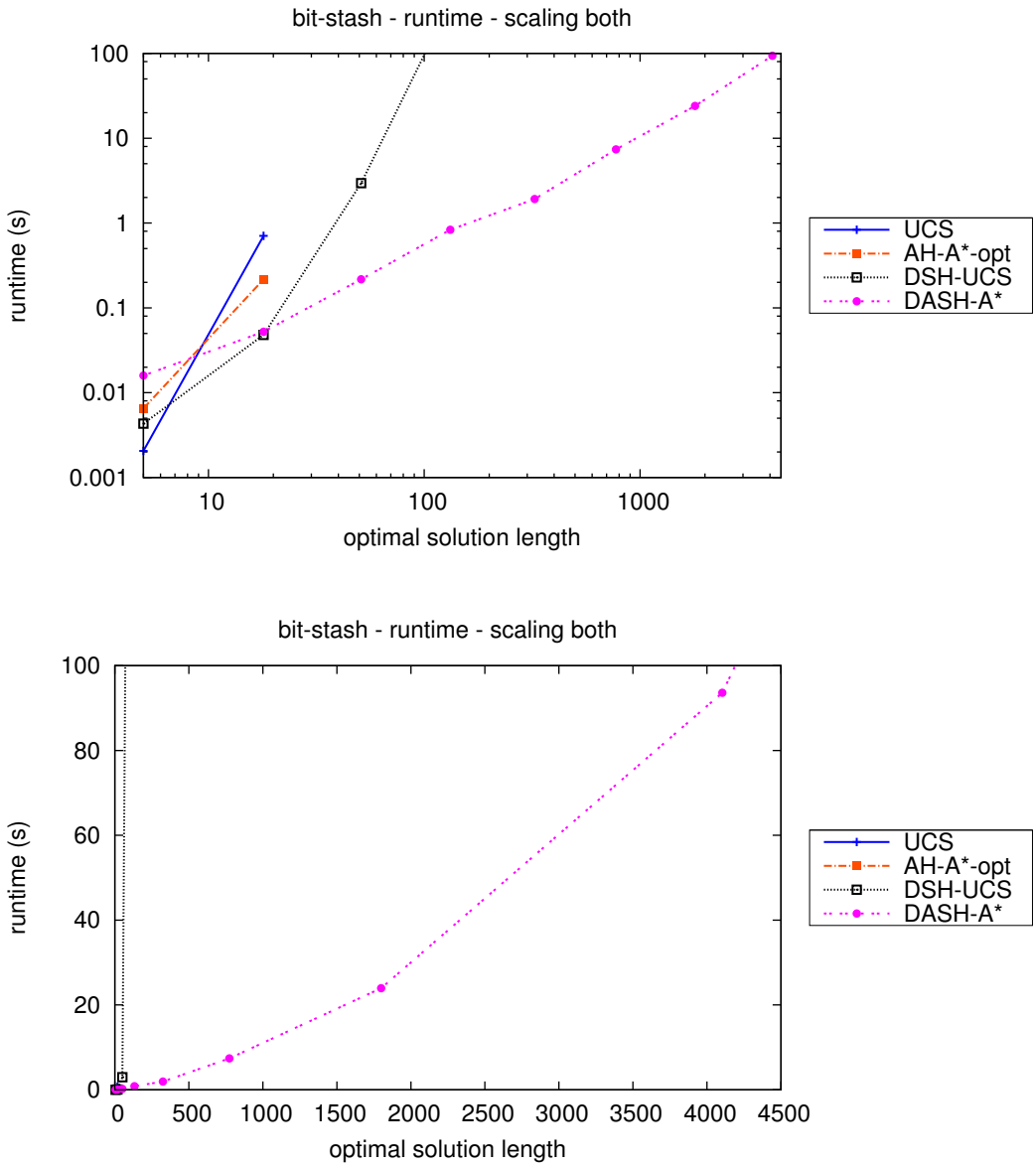

Figure 6.4: Median run-times for three runs each of four algorithms, over bit-stash instances of increasing subproblem size *and* count. Results are depicted both in log-log scale (top) and linear scale (bottom).

Chapter 7

Related Work

This chapter draws connections between the research reported above and related work not discussed in Chapter 2. Each section attempts to cover a different area of work, although some topics cut across these boundaries and are discussed in multiple sections.

7.1 Planning

We begin by discussing related work in three areas of planning: explanation-based learning, nondeterministic planning, and hierarchical task network planning.

7.1.1 Explanation-Based Learning

Explanation-Based Learning (EBL, (Mitchell et al., 1986)) is a powerful form of state abstraction, wherein deductive inferences (e.g., about discovered dead ends in the search space) are generalized, stored, and reused throughout the search space. Specifically, given an example of a target concept (e.g., a dead end), the method first constructs an explanation (e.g., proof) of why the example satisfies it, and then generalizes it to its weakest preconditions, generating a control rule that may be applicable in many future situations.

Perhaps most notably, cognitive architectures PRODIGY (Minton, 1988) and SOAR (Laird et al., 1987) have successfully used variants of EBL to learn control knowledge at various levels of abstraction. Work since has applied EBL to planning (Kambhampati, 1996) and reinforcement learning (Dietterich and Flann, 1997; Tadepalli and Dietterich, 1997). None of these systems do optimal planning, or use explicit hierarchies of the sort we consider.

A primary issue with EBL is that a very large number of rules can be gathered throughout search, and matching which are applicable in a given state can thus become very expensive,

to the point of actually slowing down search. This is known as the utility problem (Minton, 1988), and various solutions have been proposed to help eliminate less useful rules and thus reduce the overhead associated with the approach. While our use of state abstraction can be seen as a very limited form of EBL, it comes with a significant advantage: because all learned knowledge is hashed under a fixed key $(\textsc{EnterContext}(s, a), a)$, matching occurs in constant time and the utility problem is largely avoided.

7.1.2 Nondeterministic Planning

Like our angelic planners, algorithms for (partially observable) nondeterministic planning are also concerned with nondeterminism and reachable sets of states. Such planning algorithms are also typically based on AND/OR graph search, and exploit techniques such as symbolic reachable sets (Bertoli et al., 2001) and subsumption (Wolfe and Russell, 2007). However, unlike in the angelic setting, the nondeterminism is adversarial, and the actions do not admit refinements. Thus, the use of AND/OR graphs is actually quite different: the growth direction represents time rather than abstraction level, and AND-nodes represent adversarial choice rather than sequencing. As a consequence, most of the issues discussed in Chapter 5 do not arise in partially observable planning.

Somewhat closer to this research, a few recent works have explored the application of hierarchical task networks to nondeterministic planning problems (Kuter and Nau, 2004; Kuter et al., 2005). However, the search strategies have not considered angelic nondeterminism, instead resembling nondeterminized versions of the H-UCS algorithm of Chapter 2.

7.1.3 HTN Planning Semantics

Section 2.3.1.3 and Chapter 4 summarized previous work on hierarchical task network (HTN) planning, and its relation to our angelic approach. This section reviews a number of alternative approaches for capturing the semantics of high-level actions in HTN planning and related settings.

Erol et al. (1994a) published the first formal semantics of HTN planning, defining the meaning of a high-level plan in terms of its set of primitive refinements. This was a significant result for establishing the soundness and completeness of HTN planning algorithms, but did not provide an operational semantics that could be applied to improve planning efficiency.

Biundo and Schattenberg (2001) describe a system that combines HTN planning, state abstraction, and a form of pessimistic descriptions, but they do not connect them directly to solution algorithms. In addition, they begin by assuming the descriptions rather than viewing them as logical consequences of the primitive actions and refinement hierarchy.

McIlraith and Fadel (2002) describe a method for synthesizing descriptions of high-level actions that are specified using the Golog language. Their method produces successor state axioms that can, in certain cases, be converted to effect axioms of the sort that we consider. The descriptions will, however, be exact and therefore possibly grow very large for complex actions. More recently, Goldman (2009) described a semantics for HTN methods based on Golog, which can capture both angelic and adversarial nondeterminism.

Several groups of researchers have considered applications of *adversarial* semantics to HLAs. Clement and Durfee (1999) consider coordinating multiple agents, each with its own hierarchical plan; in this case an adversarial semantics is justified as each agent's plan must work *regardless* of the refinements chosen by the other agents. Kaelbling and Lozano-Perez (2011) consider a single hierarchical planning agent that interleaves planning and execution, justifying adversarial semantics as a computational shortcut (e.g., see Section 4.4.2).

Work by Doan and Haddawy (1995) is perhaps most closely related to the approach taken in this thesis. Their decision-theoretic refinement planning system (DRIPS) uses action abstraction along with automatically constructed analogues of our optimistic descriptions to find optimal *probabilistic* plans. However, they do not consider analogues of pessimistic descriptions, or sophisticated search algorithms of the sort considered here.

7.2 Hierarchies of State Abstractions

A variety of works have considered hierarchies based on *state abstraction*, wherein each level of the hierarchy is defined by a *subset* or *partitioning* (i.e., homomorphism) of the states at the next-lower level. Unlike ABSTRIPS and related algorithms in planning (see Section 2.3.1.1) which are based on very similar ideas, the focus here is on finding *optimal* or *hierarchically optimal* solutions. One can view state abstraction hierarchies as a special case of the angelic HTN-style action hierarchies considered in this thesis, wherein each HLA generates an arbitrary sequence of lower-level actions that terminates at a particular abstract state.

Botea et al. (2004) introduced hierarchical pathfinding A* (HPA*) to speed up search for high-quality paths in two-dimensional maps (e.g., for autonomous agents in computer games). Each level of the hierarchy is defined by a *subset* of states corresponding to entrances and exits to regions of states at the next-lower level. HPA* performs a hierarchically optimal search within this hierarchy, using exact descriptions at each level pre-computed by exhaustive search. Efficiency is gained by amortizing the cost of this pre-computation across the solution of many problem instances within the same map.

The observation that an optimal solution to an abstracted problem results in an admissible heuristic for the original problem (Lawler and Wood, 1966) has a long history in AI (Gaschnig, 1979; Pearl, 1984; Culberson and Schaeffer, 1996). Recently, researchers have pro-

posed a variety of (primitive-)optimal search algorithms that apply this idea hierarchically to speed up A* search without the need for a domain-specific heuristic.

Raphael (2001) introduced coarse-to-fine dynamic programming (CFDP), which applies this idea "top-down", starting with a maximally abstract space and progressively expanding abstract states until a fully primitive solution is reached. Chatterjee and Russell (2011) recently extended this idea to incorporate temporal abstraction, for hidden Markov models where some parts of the state tend to change much more frequently than others. This approach is analogous to the top-down angelic algorithms considered in this thesis, where the bounds between abstract states are essentially simplified forms of optimistic descriptions. However, the target applications are considerably simpler than planning, typically including a state space that can be completely enumerated and a fixed time horizon.

Holte et al. (1996, 2005) describe Hierarchical A* and IDA*, which apply the idea "bottom-up": when the heuristic value for a state is needed by A* at level l, it is computed recursively via search at level $l-1$. This is essentially a lazy, hierarchical application of pattern databases (Culberson and Schaeffer, 1996). Bulitko et al. (2007) apply related ideas in the *online* setting to speed up real-time pathfinding.

Finally, Felzenszwalb and McAllester (2007) describe the Hierarchical A* Lightest Derivation (HA*LD) algorithm, which generalizes and improves upon previous work in several ways. After deriving A*LD (see Section 2.2.2.2), they show how to generalize the above observation to compute admissible heuristics for *contexts* in AND/OR search problems. The resulting HA*LD algorithm uses a single priority queue in a novel way, which both reduces the overhead associated with multiple queues and allows for lazy *partial* computations of heuristics at higher levels. Thus, the special case of this algorithm for state-space search algorithms is also novel, and can improve substantially on previous algorithms such as HIDA*.

Despite the apparent similarities to our framework (hierarchy is analogous to AND/OR refinements, and implicit reachable sets are analogous to state abstraction), it does not seem that our hierarchical planning problems can be usefully encoded and solved by HA*LD. For one, as discussed in Section 2.2, bottom-up approaches are typically better-suited to inference problems where an easily enumerable set of evidence lies at the bottom of the refinement hierarchy, not planning problems where the bottom-up branching factor is typically vast. For another, the basic assumptions about the underlying structures are different. HA*LD assumes a given, fixed state abstraction, whereas in our framework the state abstraction is *generated* by the reachable sets of the actions. Moreover, HA*LD *computes* heuristics top-down, whereas our algorithms *assume* bounds produced by optimistic descriptions, which both guide and are informed by search at lower levels of the hierarchy.

7.3 Hierarchical Reinforcement Learning

There has also been extensive work on hierarchical decision-making in the probabilistic planning and learning fields, situated in the framework of Markov Decision Processes (MDPs, (Bellman, 1957)). For our purposes, it is sufficient to consider MDPs as probabilistic planning problems, where the outcome of each primitive action from a state is sampled independently from a given probability distribution over next states. Thus, solutions are (partial) policies that specify which action to take from any state (reachable under this policy). Beginning with Forestier and Varaiya (1978), a wide variety of hierarchical methods have been proposed to improve performance in this setting.

Reinforcement learning adds to this problem, assuming that the agent is deposited in an unknown MDP and must simultaneously learn about the environment and exploit knowledge from past experiences to minimize the total cost incurred (e.g., over many repeated trials). Many reinforcement learning approaches are based at least in part on Q-learning (Watkins, 1989), a non-hierarchical method wherein the mapping from $(state, action)$ pairs to expected future rewards is learned directly from experience, without any explicit model of the transition dynamics of the MDP.

Perhaps the simplest hierarchical reinforcement learning (HRL) method is *MAXQ* (Dietterich, 2000), which assumes a hierarchy much like those in this thesis, except that the HLAs are defined by *sets* of lower-level actions that can be applied in any combination, plus a termination condition (thus, sequencing information captured by our refinements cannot be encoded). MAXQ learns a *recursively optimal* policy from experience, which specifies which lower-level task to invoke from each state within a higher-level task (without regard for which reachable state might actually be best in a given circumstance). Efficiency is gained by using *state abstraction*, and sharing the learned policies between states that differ only in variables irrelevant to the task at hand (much as in this thesis — our method was largely inspired by MAXQ). Diuk et al. (2006) combined MAXQ with transition model learning in deterministic domains, producing a recursively optimal hierarchical planning algorithm for unknown environments.

Options (Sutton et al., 1999) are perhaps the best-studied approach to hierarchical reinforcement learning. Each option is an HLA defined by a *policy* that can be executed until a state in some target set is reached (e.g., LeaveTheRoom). Because an option corresponds to a fixed policy, like the macros of Section 2.3.1.2, it is easy to construct a transition model for an option that looks almost exactly like a primitive action model. This enables reinforcement learning agents designed for ordinary MDPs to transparently learn and plan in MDPs with options added. However, unlike MAXQ (in which all HLA policies are learned simultaneously), options must be learned in a bottom-up fashion, where the policies at all lower levels are fixed before the next-higher-level is learned. Another disadvantage of options is that they are fixed policies (like macros), so it may be difficult to find a small set of options

that are useful across a wide range of situations.

The final HRL method, developed by Parr and Russell (1998) and Andre and Russell (2002), allows for general *partial policies* with nondeterministic choice points built in, which can be expressed in a programming language called ALISP. Novel reinforcement learning strategies (including MAXQ-style state abstraction) are used to learn to a *hierarchically optimal* policy that fills in these choice points in the best possible ways. Marthi (2006) extended ALISP to efficiently handle domains with multiple *concurrent* actors, using multithreaded programs and a novel Q-function decomposition. A main omission compared to options is that it is not obvious how to *plan* in ALISP-style hierarchies (to help find better policies before learning has converged). The issue is that ALISP HLAs admit multiple refinements, and thus are effectively a probabilistic generalization of the HLAs considered here. In fact, this connection was the initial motivation behind this thesis work. We hope that future probabilistic generalizations of the angelic approach will successfully be combined with ALISP, yielding a powerful system that uses planning to bridge the gaps in its current knowledge, while simultaneously learning to fill these gaps and thus free up future planning cycles for other tasks.

7.4 Hierarchies in Robotics

Hierarchy has long been a staple of robotics, to help bridge the gap between high-level task descriptions (e.g., clean a room) and the thousands or millions of low-level primitive actions required to implement them (e.g, "move left arm to joint configuration X", or even "apply voltage Y to motor Z"). Traditionally, such problems have been attacked top-down, strictly separating high-level task planning (e.g., sequencing pick-and-place operations) from lower-level planning (e.g., finding feasible paths for the arm). For instance, an agent might first search for a task-level STRIPS solution, and then attempt to execute a lower-level controller corresponding to each STRIPS action in turn (see Section 4.4.2). Task planning is simplified by ignoring low-level details, but the resulting plans may be inefficient or even infeasible due to missed lower-level synergies and conflicts.

In recent work (Wolfe et al., 2010), we demonstrated an encoding for robotic manipulation problems as vertically integrated hierarchical task networks (HTNs). In this application, we directly extended the domain and hierarchy of Sections 2.1.2.3 and 2.3.2.2 to include state variables corresponding to real-valued locations of the objects to be manipulated and joint angles for the robot, as well as discrete predicates such as On and Holding. Continuous choices (e.g,. arm trajectory refinements of Reach) were made finite via sampling, sometimes incorporating external solvers such as rapidly-exploring random trees (RRTs) and inverse kinematics. Given a known initial world state, the algorithms of Chapter 3 can be applied to generate guaranteed-feasible, high-quality solutions for manipulating several objects, at the level of paths through configuration space for the robot base and arms, in tens of seconds.

Preliminary work has added angelic descriptions (Chapter 4) for the HLAs and applied the algorithms of Chapter 5 to these problems, yielding significant further speedups.

We briefly summarize alternative proposals for hierarchical robot planning and control. A variety of recent robotic planning algorithms integrate information from the task level into a sampling-based motion planning framework. The configuration space can then be viewed as consisting of regions (one per instantiation of the discrete variables), connected by lower-dimensional submanifolds. aSyMov (Gravot et al., 2003) decomposes the configuration space for manipulation into transit and transfer manifolds, taking advantage of stability constraints for free objects. Hauser and Latombe (2009) use a geometrically motivated heuristic function to focus sampling on those subtasks that are likely to be feasible. HYDICE (Plaku et al., 2009), a hybrid system verification tool, also uses a heuristic estimate of the likely feasibility of each subproblem to focus sampling. Şucan and Kavraki (2011) propose a task motion multigraph, which extends the above approaches by explicitly reasoning about which set of joints to use for each motion. Choi and Amir (2009) propose a factored planning approach, which automatically partitions and decomposes the search space to speed up planning. An advantage of the above methods is that they interleave sampling between motion planning subproblems. However, they do not deal with general action hierarchies, and do not attempt to find optimal plans.

7.5 Interleaving Planning and Execution

Hierarchical reasoning about actions can also be incorporated into systems that *interleave* planning and execution steps. Advantages include faster decision-making, the ability to start acting before a complete plan is fleshed out, and the ability to defer decisions about what to do in uncertain future states until more information is gathered. Of course, the generated behaviors may be end up being highly suboptimal or even unsuccessful, if future difficulties are not properly anticipated.

The Procedural Reasoning System (PRS) is one well-known example of such a system (Georgeff and Lansky, 1987). PRS represents high-level actions using "knowledge areas" (KAs), which are finite-state machines that encode procedural knowledge about how to do a task. KAs can be given descriptions of the form (c,P,g), which means that "in any state satisfying c, a successful execution of P will result in a state satisfying g." This is similar to our notion of optimistic descriptions, with an additional caveat about success of the execution, which must also be defined by the KA. Thus, the descriptions cannot be used to guarantee at the high level that a plan will achieve a particular goal. Another difference is that PRS does not explicitly do lookahead planning, though such planning may be effected by meta-level KAs.

Another proposal by Barrett (2010) combines structured representations of concurrent be-

haviors using hierarchies of Petri nets with the probabilistic inference machinery of Bayesian networks. This combination provides a rich language for programming robust agent behaviors, which has been successfully applied to control players in RoboCup soccer. The system does not include explicit planning; however, in some cases its programs can be proved to correctly accomplish their desired behaviors across a class of environments, as discussed further in the next section.

Finally, we have proposed an approach for interleaving planning and execution based on angelic hierarchical lookahead (Marthi et al., 2008). This system conducts a hierarchical search much like the algorithms in Chapter 5, but can be applied in very large environments in which (even bounded suboptimal) planning might take far too long to be useful. Instead, the search space is only partially expanded, focusing on actions in the more immediate future, and this partial search is used to select a promising first primitive action. This action is executed in the world, updated bounds discovered through planning are stored, and then the process is repeated until the goal is eventually reached. Helwig and Haddawy (1996) proposed a similar approach for probabilistic environments, by extending the DRIPS system (Doan and Haddawy, 1995) to the online setting.

7.6 Formal Verification

Researchers in a variety of areas analyze reachable sets to reason about and prove properties about formal systems. Examples include program synthesis, automatic program verification, model checking, and hybrid systems. Because the state spaces involved are often vast or infinite, a combination of state abstraction and symbolic reasoning techniques is commonly used to make reasoning tractable.

Angelic nondeterminism can be a useful tool for writing and reasoning about programs. For instance, while in the process of writing a program, one is on the right track iff there exist implementations for the remaining functions that make the final program correct. Bodik et al. (2010) implemented a system that directly captures this idea, alerting the programmer if it can prove that no such implementation exists, using a combination of abstraction and symbolic model checking techniques. Angelic nondeterminism is also frequently used to succinctly describe search algorithms, and exists as an explicit language primitive in ALISP (Andre and Russell, 2002) for writing programs corresponding to partial MDP policies.

More frequently, however, formal verification is concerned with ordinary (demonic) nondeterminism. For instance, we would like to be able to prove that a given program does not deadlock or perform unsafe operations, over *all* possible executions. Again, a successful approach is to use model checking, along with state abstractions of different granularity (Jhala, 2004). Because programs are typically structured as hierarchies of functions, each subroutine behaves like a high-level action of sorts. Such methods may prove useful for inducing and

proving correctness of HLA descriptions of the sort we consider.

Researchers in *hybrid systems* attempt to prove properties (and discover effective controllers) for systems with both discrete and continuous state components, where each discrete state may have different continuous dynamics. Hybrid systems embody a two-level hierarchy: each execution of the system corresponds to a sequence of discrete states at the high level, with continuous trajectories within each state at the low level. Unlike the planning problems considered here, the discrete structure is typically simple (not combinatorially large), and the principal difficulty involves the continuous dynamics. Moreover, the focus is on *controllers*, which are typically reactive policies that specify how to act in any state, rather than on online planning to discover an (optimal) action sequence from a particular initial state.[1]

Given a deterministic controller, proving that it always satisfies a given safety property (for some set of initial conditions) is equivalent to proving that its reachable set does not intersect an unsafe set of states. Because computing exact reachable sets is typically intractable for realistic dynamic models, numeric methods are often used to soundly approximate reachability while maintaining correctness (Tomlin et al., 2003). The task of finding a counterexample (i.e., unsafe execution) is thus related to the angelic planning tasks considered here. Given a particular hybrid system, it may also be desirable to automatically find a stable (or optimal) controller that satisfies some properties. Because of numerical difficulties, optimal control is typically only studied for subclasses of systems (e.g., with linear dynamics) (Henzinger et al., 1997; Gokbayrak and Cassandras, 2000; Cho et al., 2001).

[1] Work in hybrid model-predictive control does invoke some online planning, but this planning is typically over a fixed receding horizon (like the algorithms discussed in the previous section).

Chapter 8

Conclusion

8.1 Summary

The primary contributions of this thesis are as follows.

First, Chapter 4 proposed an *angelic semantics*, which attempts to answer a decades-old problem in hierarchical planning. This semantics describes correct transition models for high-level actions, enabling the identification of *high-level solutions* that can provably be extended to at least one concrete solution. We also proposed practical methods for implementing this semantics within a hierarchical planner, including compact representations for principled approximations to the exact angelic semantics, and efficient algorithms for operating on these representations. These methods can be used to efficiently derive bounds on the outcomes and costs of high-level plans, in domains ranging from discrete "toy" problems to encodings of real-world robotic mobile manipulation problems.

Second, Chapters 3 and 5 introduced several novel families of algorithms for hierarchically optimal search. The algorithms of Chapter 3 require only the ability to simulate the primitive actions in a domain, plus an action hierarchy expressing the ways in which each high-level task can be broken down into lower-level subtasks. These algorithms apply *decomposition* and *state abstraction* techniques to expose the existence of isomorphic subproblems in the search space. By sharing bounds and solutions between these isomorphic subproblems, the effective size of the search space and thus planning time can be reduced exponentially compared to previous hierarchically optimal planning algorithms. Then, Chapter 5 described hierarchically optimal search algorithms that also make use of angelic bounds for high-level plans produced by the machinery of Chapter 4. This chapter concluded with the Decomposed, Angelic, State-abstracted Hierarchical A* (DASH-A*) algorithm, which incorporates angelic bounds, decomposition, state abstraction, and a number of additional techniques in a novel AND/OR search framework, enabling agents that can effectively reason about the

outcomes and costs of high-level plans from abstract sets of states. These inferences can quickly eliminate large swaths of provably suboptimal plans from the search space, leading to exponential speedups in more cases than the algorithms of Chapter 3.

8.2 Future Work

This thesis focused on sequential hierarchical planning for STRIPS-style classical planning domains. There is still much to be done in this simple setting, including more general and expressive angelic representations, improved meta-level search control, and automatic hierarchy and angelic description induction. Beyond classical planning, applying the results and algorithms in this thesis to many real-world problems will require extensions to incorporate reasoning about continuous state and action spaces, uncertainty, partial knowledge, multiple agents, and concurrency.

8.2.1 Classical Planning

The empirical results reported in Chapter 6 seem to indicate that the simple factored representations of Section 4.3 and state abstraction of Section 3.1.2 can perform quite well in practice. Nevertheless, there is probably much to be gained by considering more expressive representations, which could more compactly and accurately capture descriptions, reachable sets, and costs. The simple variable-dropping state abstraction could no doubt be improved significantly as well; for example, the descriptions and valuations for higher-level actions could be described in terms of *derived predicates* that compactly capture complex combinations of lower-level variables. Finally, *relational* state abstraction could allow information sharing between HLAs applied to identical objects.

Next, as our search algorithms become more complex and structured, the use of simple heuristics for selecting what computation to do next (e.g., always EXPAND the highest-level eligible node) becomes a more and more glaring omission. For example, when executing a bounded-suboptimal variant of DASH-A*, we have a vast array of knowledge at our disposal that could be useful for this purpose, including the structure of the graph (better to expand nodes appearing in many near-optimal contexts), upper and lower bounds on the costs of each action (better to expand more uncertain things), a global lower cost bound (are we close to finding a w-optimal solution?), and so on. This situation is ripe for *meta-level control* (Russell and Wefald, 1991), wherein the selection of a next computation is treated as an search and/or learning problem in its own right.

Finally, in this thesis we have assumed that the structure of the hierarchy (and angelic descriptions, where applicable) were provided by a human. If our only desire is to effectively solve a real-world problem, this may be satisfactory (assuming that the burden of doing so

is not much greater than writing down the primitive domain in the first place). Nevertheless, methods for automatically learning hierarchical structures and descriptions would be a significant advance, allowing for the fully autonomous application of angelic hierarchical planning techniques given only a primitive domain description. A number of promising approaches for learning the structure of a hierarchy have been proposed in the hierarchical reinforcement learning literature (e.g., (McGovern and Barto, 2001; Hengst, 2002; Simsek and Barto, 2004; Grounds and Kudenko, 2005; Marthi et al., 2007a)), but none have yet approached the abilities of a human designer.

It may also be of interest to apply the algorithms in this thesis to problems beyond standard HTN planning.

In recent work, we proposed a novel method for solving classical planning instances hierarchically, which exploits the causal structure of a domain (Wolfe and Russell, 2011). The original goal of that work was to apply angelic planning on top of the hierarchy; surprisingly, it turned out that simple search algorithms (akin to H-UCS) were sufficient to achieve exponential speedups, and our investigation thus far focused on exploring this simple idea further. In light of the results reported here, the application of angelic and decomposed search techniques to this problem may merit further exploration.

Even farther afield, some structured probabilistic inference tasks are isomorphic to classical planning tasks, and thus it may be possible to apply the algorithms reported here to solve them faster.

8.2.2 Beyond Classical Planning

Natural hierarchies for some (even purely sequential) planning domains seem to require *concurrency*. For instance, if the robot in our discrete manipulation domain had two grippers, it would be natural to consider interleaving actions from simultaneous REACH HLAs for each gripper. This complicates the problem of writing correct descriptions substantially, however, as the description of each HLA must account for potential interactions caused by primitives interspersed from other HLAs.

It should also be possible to extend the results and algorithms of this thesis to cover more realistic planning domains that include *probabilistic uncertainty* and *partial observability* (i.e., MDPs and POMDPs, respectively). Hierarchical planning has even more promise in such domains, because of the potential to abstract over future uncertain outcomes and information-gathering behaviors. Effective angelic planning agents in these settings will be based on *hierarchical lookahead*, planning in detail for only for the immediate future (and perhaps key future junctures), leaving the remainder of their plans abstract until more information is gathered. Especially exciting is the potential for combination with *hierarchical reinforcement learning* algorithms such as ALISP, enabling agents that gain task-specific

expertise as they act and compute, and use planning to fill in the gaps in their experience when faced with novel tasks or circumstances.

8.3 Outlook

Generating effective behaviors in real-world environments often requires planning over long time scales, in vast spaces of possibilities. Given limited computational resources, coping with this complexity seems to require interleaving reasoning at various levels of abstraction. Doing so effectively requires a detailed understanding of the meaning of and relationships between abstraction levels, and methods for efficiently searching through them to locate high-quality solutions. The tools provided in this thesis represent small but important steps towards these goals.

Bibliography

D. Andre and S. Russell. State Abstraction for Programmable Reinforcement Learning Agents. In *AAAI*, pages 119–125, 2002.

F. Bacchus and Q. Yang. The Downward Refinement Property. In *IJCAI*, pages 262–292, 1991.

C. Bäckström. Equivalence and Tractability Results for SAS+ Planning. In *KR*, pages 126–137, 1992.

A. Bagchi and A. Mahanti. Admissible Heuristic Search in And/Or Graphs. *Theoretical Computer Science*, 24:207–219, 1983.

A. Barrett and D. Weld. Task-Decomposition via Plan Parsing. In *AAAI*, pages 1117–1122, 1994.

L. Barrett. *An Architecture for Structured, Concurrent, Real-Time Action.* PhD thesis, UC Berkeley, Berkeley, CA, 2010.

A. Barto, S. Bradtke, and S. Singh. Learning to Act Using Real-Time Dynamic Programming. *Artificial Intelligence*, 72:81–138, 1995.

R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

P. Bertoli, A. Cimatti, M. Pistore, and M. Roveri. MBP: a Model Based Planner. In *IJCAI Workshop on Planning under Uncertainty*, 2001.

S. Biundo and B. Schattenberg. From Abstract Crisis to Concrete Relief – A Preliminary Report on Combining State Abstraction and HTN Planning. In *ECP*, pages 157–168, 2001.

A. Blum and M. Furst. Fast Planning through Planning Graph Analysis. In *IJCAI*, pages 1636–1642, 1995.

R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with Angelic Nondeterminism. In *POPL*, pages 339–352, 2010.

B. Bonet and H. Geffner. An Algorithm Better than AO*? In *AAAI*, pages 1343–1347, 2005.

A. Botea, M. Enzenberger, M. Muller, and J. Schaeffer. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *JAIR*, 24:581–621, 2005.

A. Botea, M. Muller, and J. Schaeffer. Near-Optimal Hierarchical Pathfinding. *Journal of Game Development*, 1(1):7–28, 2004.

R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

V. Bulitko, N. Sturtevant, J. Lu, and T. Yau. Graph Abstraction in Real-time Heuristic Search. *JAIR*, 30:51–100, 2007.

T. Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69:165–204, 1994.

D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32(3):333–377, 1987.

S. Chatterjee and S. Russell. A Temporally Abstracted Viterbi Algorithm. In *UAI*, pages 96–104, 2011.

Y. Cho, C. Cassandras, and D. Pepyne. Forward Decomposition Algorithms for Optimal Control of a Class of Hybrid Systems. *International Journal of Robust and Nonlinear Control*, 11(5):497–513, 2001.

J. Choi and E. Amir. Combining Planning and Motion Planning. In *ICRA*, pages 238–244, 2009.

A. Cimatti, E. Giunchiglia, and F. Giunchiglia. Planning via Model Checking: A Decision Procedure for AR. In *ECP*, 1997.

B. J. Clement and E. H. Durfee. Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. In *AAAI*, pages 495–502, 1999.

I. A. Şucan and L. E. Kavraki. Mobile Manipulation: Encoding Motion Planning Options Using Task Motion Multigraphs. In *ICRA*, pages 5492–5498, 2011.

J. C. Culberson and J. Schaeffer. Searching with Pattern Databases. In *Canadian Conference on AI*, pages 402–416, 1996.

T. Dietterich and N. Flann. Explanation-Based Learning and Reinforcement Learning: A Unified View. *Machine Learning*, 28:176–184, 1997.

T. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *JAIR*, 13:227–303, 2000.

E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

C. Diuk, A. L. Strehl, and M. L. Littman. A Hierarchical Approach to Efficient Reinforcement Learning in Deterministic Domains. In *AAMAS*, pages 313–319, 2006.

A. Doan and P. Haddawy. Decision-Theoretic Refinement Planning: Principles and Application. Technical Report TR-95-01-01, Univ. of Wisconsin-Milwaukee, 1995.

M. Drummond and K. Currie. Goal Ordering in Partially Ordered Plans. In *IJCAI*, pages 960–965, 1989.

K. Erol, J. Hendler, D. Nau, and R. Tsuneto. A Critical Look at Critics in HTN Planning. In *ICAPS*, pages 1592–1598, 1995.

K. Erol, J. Hendler, and D. Nau. Semantics for Hierarchical Task-Network Planning. Technical Report UMIACS-TR-94-31, Univ. of Maryland at College Park, 1994a.

K. Erol, J. Hendler, and D. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. In *AIPS*, pages 249–254, 1994b.

K. Erol, J. Hendler, and D. Nau. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, 18:69–93, 1996.

T. Estlin, S. Chien, and X. Wang. An Argument for a Hybrid HTN/Operator-Based Approach to planning. In *ECP*, pages 182–194, 1997.

P. F. Felzenszwalb and D. McAllester. The Generalized A* Architecture. *JAIR*, 29(1): 153–190, 2007.

R. Fikes, P. Hart, and N. Nilsson. Learning and Executing Generalized Robot Plans. Technical Report 70, AI Center, SRI International, 1972.

R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

J. Forestier and P. Varaiya. Multilayer Control of Large Markov Chains. *Automatic Control*, 23:298–305, 1978.

M. Fox and D. Long. Hierarchical Planning Using Abstraction. *Control Theory and Applications*, 142(3):197–210, 1995.

J. G. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1979.

M. Genesereth and I. Nourbakhsh. Time-Saving Tips for Problem Solving with Incomplete Information. In *AAAI*, pages 724–730, 1993.

M. Georgeff and A. Lansky. Reactive Reasoning and Planning. In *AAAI*, pages 677–682, 1987.

A. Gerevini, U. Kuter, D. Nau, A. Saetti, and N. Waisbrot. Combining Domain-Independent Planning and HTN Planning: The Duet Planner. *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling*, 2008.

M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

F. Giunchiglia and T. Walsh. The Inevitability of Inconsistent Abstract Spaces. *Journal of Automated Reasoning*, pages 23–41, 1993.

K. Gokbayrak and C. Cassandras. A Hierarchical Decomposition Method for Optimal Control of Hybrid Systems. In *CDC*, pages 1816–1821, 2000.

R. Goldman. A Semantics for HTN Methods. In *ICAPS*, pages 146–153, 2009.

F. Gravot, S. Cambon, and R. Alami. aSyMov: A Planner That Deals with Intricate Symbolic and Geometric Problems. In *ISRR*, pages 100–110, 2003.

M. Grounds and D. Kudenko. Combining Reinforcement Learning with Symbolic Planning. In *AAMAS*, pages 75–86, 2005.

B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Faster Algorithms for Incremental Topological Ordering. In *ICALP*, pages 421–433, 2008.

E. A. Hansen and S. Zilberstein. LAO*: A Heuristic Search Algorithm that Finds Solutions with Loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

P. Hart, N. Nilsson, and B. Raphael. Correction to A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *SIGART Bulletin*, 37:28–29, 1972.

K. Hauser and J. C. Latombe. Integrating Task and PRM Motion Planning. In *ICAPS Workshop on Bridging the Gap between Task and Motion Planning*, 2009.

M. Helmert. A Planning Heuristic Based on Causal Graph Analysis. In *ICAPS*, pages 161–170, 2004.

M. Helmert. The Fast Downward Planning System. *JAIR*, 26:191–246, 2006.

M. Helmert and G. Röger. How Good is Almost Perfect? In *AAAI*, pages 944–949, 2008.

J. Helwig and P. Haddawy. An Abstraction-Based Approach to Interleaving Planning and Execution in Partially-Observable Domains. In *AAAI Fall Symposium*, 1996.

B. Hengst. Discovering Hierarchy in Reinforcement Learning with HEXQ. In *ICML*, pages 243–250, 2002.

T. Henzinger, P. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

R. Holte, J. Grajkowski, and B. Tanner. Hierarchical Heuristic Search Revisited. In *SARA*, pages 121–133, 2005.

R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *AAAI*, pages 530–535, 1996.

R. Howard. *Dynamic Programming and Markov Processes*. MIT Technology Press, 1960.

R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic Non-Determinism in Concurrent Constraint Programming. Technical report, Xerox PARC, 1992.

R. Jhala. *Program Verification by Lazy Abstraction*. PhD thesis, UC Berkeley, Berkeley, CA, 2004.

P. Jiménez and C. Torras. An Efficient Algorithm for Searching Implicit AND/OR Graphs with Cycles. *Artificial Intelligence*, 124(1):1–30, 2000.

A. Jonsson. The Role of Macros in Tractable Planning. *JAIR*, 36:471–511, 2009.

P. Jonsson and C. Bäckström. State-Variable Planning under Structural Restrictions: Algorithms and Complexity. *Artificial Intelligence*, 100:125–176, 1998.

L. P. Kaelbling and T. Lozano-Perez. Hierarchical Task and Motion Planning in the Now. In *ICRA*, pages 1470–1477, 2011.

S. Kambhampati. Formalizing Dependency Directed Backtracking and Explanation Based Learning in Refinement Search. In *AAAI*, pages 757–762, 1996.

S. Kambhampati, A. Mali, and B. Srivastava. Hybrid Planning for Partially Hierarchical Domains. In *AAAI*, pages 882–888, 1998.

S. Kambhampati, C. Knoblock, and Q. Yang. Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning. *Artificial Intelligence*, 76:167–238, 1995.

H. A. Kautz and B. Selman. Planning as Satisfiability. In *ECAI*, pages 359–363, 1992.

D. Klein and C. D. Manning. Accurate Unlexicalized Parsing. In *ACL*, pages 423–430, 2003.

C. Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *AAAI*, pages 923–928, 1990.

C. Knoblock, J. D. Tenenberg, and Q. Yang. Characterizing Abstraction Hierarchies for Planning. In *AAAI*, pages 692–697, 1991.

D. E. Knuth. A Generalization of Dijkstra's Algorithm. *Information Processing Letters*, 6: 1–5, 1977.

R. E. Korf. Toward a Model of Representation Changes. *Artificial Intelligence*, 14(1):41–78, 1980.

R. E. Korf. Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, 26: 35–77, 1985.

R. E. Korf. Planning as Search: A Quantitative Approach. *Artificial Intelligence*, 33:65–88, 1987.

V. Kumar, D. Nau, and L. N. Kanal. A Generalization of the AO* Algorithm. In *COMPSAC*, 1985.

U. Kuter and D. Nau. Forward-Chaining Planning in Nondeterministic Domains. In *AAAI*, pages 513–518, 2004.

U. Kuter, D. Nau, M. Pistore, and P. Traverso. A Hierarchical Task-Network Planner based on Symbolic Model Checking. In *ICAPS*, pages 300–309, 2005.

J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: an Architecture for General Intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

E. L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, 1966.

M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *NIPS*, pages 1–8, 2003.

A. Martelli and U. Montanari. Additive AND/OR Graphs. In *IJCAI*, pages 1–11, 1973.

B. Marthi. *Concurrent Hierarchical Reinforcement Learning*. PhD thesis, UC Berkeley, Berkeley, CA, 2006.

B. Marthi, S. Russell, and J. Wolfe. Angelic Semantics for High-Level Actions. In *ICAPS*, pages 232–239, 2007a.

B. Marthi, S. Russell, and J. Wolfe. Angelic Semantics for High-Level Actions. Technical Report UCB/EECS-2007-89, EECS Department, University of California, Berkeley, 2007b.

B. Marthi, S. Russell, and J. Wolfe. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *ICAPS*, pages 222–231, 2008.

B. Marthi, S. Russell, and J. Wolfe. Angelic Hierarchical Planning: Optimal and Online Algorithms (Revised). Technical Report UCB/EECS-2009-122, EECS Department, University of California, Berkeley, 2009.

D. Mcallester and D. Rosenblitt. Systematic Nonlinear Planning. In *AAAI*, pages 634–639, 1991.

D. McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2):35–55, 2000.

A. McGovern and A. Barto. Automatic Discovery of Subgoals in Reinforcement Learning Using Diverse Density. In *ICML*, pages 361–368, 2001.

S. McIlraith and R. Fadel. Planning with Complex Actions. In *NMR*, pages 356–364, 2002.

M. Minsky. Steps toward Artificial Intelligence. *Computers and Thought*, pages 406–450, 1963.

S. Minton. Quantitative Results Concerning the Utility of Explanation-Based Learning. In *AAAI*, pages 564–569, 1988.

T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1(1):47–80, 1986.

D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI*, pages 968–973, 1999.

D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order Planning with Partially Ordered Subtasks. In *IJCAI*, pages 425–430, 2001.

D. Nau, T. C. Au, O. Ilghami, U. Kuter, W. J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *JAIR*, 20:379–404, 2003.

B. Nebel. On the Compilability and Expressive Power of Propositional Planning Formalisms. *JAIR*, 12:271–315, 2000.

A. Newell and H. Simon. *Human Problem Solving*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1972.

R. Parr and S. Russell. Reinforcement Learning with Hierarchies of Machines. In *NIPS*, pages 1043–1049, 1998.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman, Boston, MA, 1984.

E. P. D. Pednault. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *KR*, pages 324–332, 1989.

E. Plaku, L. E. Kavraki, and M. Y. Vardi. Hybrid Systems: From Verification to Falsification by Combining Motion Planning and Discrete Search. *Formal Methods in System Design*, 34(2):157–182, 2009.

I. Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1: 193–204, 1970.

C. Raphael. Coarse-to-Fine Dynamic Programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:1379–1390, 2001.

S. Richter and M. Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39:127–177, 2010.

S. Russell and E. Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, Cambridge, MA, 1991.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 3rd edition, 2009.

E. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5(2): 115–135, 1974.

E. Sacerdoti. The Nonlinear Nature of Plans. Technical Report A726854, SRI, 1975.

S. Sanner and D. McAllester. Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference. In *IJCAI*, pages 1384–1390, 2005.

H. Simon. The Architecture of Complexity. *Transactions, American Philosophical Society*, 106(6):467–482, 1962.

O. Simsek and A. Barto. Using Relative Novelty to Identify Useful Temporal Abstractions in Reinforcement Learning. In *ICML*, pages 95–101, 2004.

R. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112:181–211, 1999.

P. Tadepalli and T. Dietterich. Hierarchical Explanation-Based Reinforcement Learning. In *ICML*, pages 358–366, 1997.

A. Tate. Generating Project Networks. In *IJCAI*, pages 888–893, 1977.

J. D. Tenenberg. *Abstraction in Planning*. PhD thesis, University of Rochester, Rochester, NY, 1988.

J. T. Thayer and W. Ruml. Finding Acceptable Solutions Faster Using Inadmissible Information. In *SOCS*, 2010.

C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi. Computational Techniques for the Verification of Hybrid Systems. *Proceedings of the IEEE*, 91(7):986–1001, 2003.

R. Tsuneto, J. Hendler, and D. Nau. Analyzing External Conditions to Improve the Efficiency of HTN Planning. In *AAAI*, pages 913–920, 1998.

C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

D. E. Wilkins. Representation in a Domain-Independent Planner. In *IJCAI*, pages 733–740, 1983.

D. E. Wilkins. Can AI Planners Solve Practical Problems? *Computational Intelligence*, 6 (4):232–246, 1990.

J. Wolfe, B. Marthi, and S. Russell. Combined Task and Motion Planning for Mobile Manipulation. In *ICAPS*, pages 254–258, 2010.

J. Wolfe and S. Russell. Exploiting Belief State Structure in Graph Search. In *ICAPS Workshop on Planning in Games*, 2007.

J. Wolfe and S. Russell. Bounded Intention Planning. In *IJCAI*, pages 2039–2045, 2011.

Q. Yang. Formalizing Planning Knowledge for Hierarchical Planning. *Computational Intelligence*, 6(1):12–24, 1990.

R. M. Young, M. E. Pollack, and J. D. Moore. Decomposition and Causality in Partial-order Planning. In *AIPS*, pages 188–193, 1994.