Angelic Hierarchical Planning: Optimal and Online Algorithms (Revised)[*]

Bhaskara Marthi BHASKARA@CSAIL.MIT.EDU
MIT/Willow Garage Inc.

Stuart Russell RUSSELL@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720

Jason Wolfe[†] JAWOLFE@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720

Abstract

High-level actions (HLAs) are essential tools for coping with the large search spaces and long decision horizons encountered in real-world decision making. In a recent paper, we proposed an "angelic" semantics for HLAs that supports proofs that a high-level plan will (or will not) achieve a goal, without first reducing the plan to primitive action sequences. This paper extends the angelic semantics with cost information to support proofs that a high-level plan is (or is not) *optimal*. We describe the Angelic Hierarchical A* algorithm, which generates provably optimal plans, and show its advantages over alternative algorithms. We also present the Angelic Hierarchical Learning Real-Time A* algorithm for situated agents, one of the first algorithms to do *hierarchical lookahead* in an online setting. Since high-level plans are much shorter, this algorithm can look much farther ahead than previous algorithms (and thus choose much better actions) for a given amount of computational effort. This is a revised, extended version of a paper by the same name appearing in ICAPS '08.

1. Introduction

Humans somehow manage to choose quite intelligently the twenty trillion primitive motor commands that constitute a life, despite the large state space. It has long been thought that hierarchical structure in behavior is essential in managing this complexity. Structure exists at many levels, ranging from small (hundred-step?) motor programs for typing characters and saying phonemes up to large (billion-step?) actions such as writing an ICAPS paper, getting a good faculty position, and so on. The key to reducing complexity is that one can choose (correctly) to write an ICAPS paper without first considering all the character sequences one might type.

Hierarchical planning attempts to capture this source of power. It has a rich history of contributions (to which we cannot do justice here) going back to the seminal work of Tate (1977). The basic idea is to supply a planner with a set of high-level actions (HLAs) in addition to the primitive actions. Each HLA admits one or more *refinements* into sequences of (possibly high-level) actions that implement it. Hierarchical planners such as SHOP2 (Nau *et al.*, 2003) usually consider only plans that are refinements of some top-level HLAs for achieving the goal, and derive power from constraints placed on the search space by the refinement hierarchy.

One might hope for more; consider, for example, the *downward refinement property*: every plan that claims to achieve some condition does in fact have a primitive refinement that achieves it. This property would enable the derivation of provably correct *abstract plans* without refining all the way to primitive actions, providing potentially exponential speedups. This requires, however, that HLAs have clear precondition–effect semantics, which have until recently been unavailable (McDermott, 2000). In a recent paper (Marthi, Russell,

*. This document is a revised version of Marthi, Russell, & Wolfe (2008a), which was itself an extended version of (Marthi, Russell, & Wolfe, 2008b). Major differences from Marthi, Russell, & Wolfe (2008a) include an extended discussion of pruning rules, and revised and extended empirical results. Results reported in the two previous papers were erroneous, due to a programming error.

†. The authors appear in alphabetical order.

& Wolfe, 2007) — henceforth (MRW '07) — we defined an "angelic semantics" for HLAs, specifying for each HLA the set of states reachable by *some* refinement into a primitive action sequence. The angelic approach captures the fact that the *agent* will choose a refinement and can thereby choose which element of an HLA's reachable set is actually reached. This semantics guarantees the downward refinement property and yields a sound and complete hierarchical planning algorithm that derives significant speedups from its ability to generate and commit to provably correct abstract plans.

Our previous paper ignored action costs and hence our planning algorithm used no heuristic information, a mainstay of modern planners. The first objective of this paper is to rectify this omission. The angelic approach suggests the obvious extension: the exact cost of executing a high-level action to get from state s to state s' is the *least* cost among all primitive refinements that reach s'. In practice, however, representing the exact cost of an HLA from each state s to each reachable state s' is infeasible, and we develop concise lower and upper bound representations. From this starting point, we derive the first algorithm capable of generating *provably optimal* abstract plans. Conceptually, this algorithm is an elaboration of A*, applied in hierarchical plan space and modified to handle the special properties of refinement operators and use both upper and lower bounds. We also provide a satisficing algorithm that sacrifices optimality for computational efficiency and may be more useful in practice. Preliminary experimental results show that these algorithms outperform both "flat" and our previous hierarchical approaches.

The paper also examines HLAs in the *online* setting, wherein an agent performs a limited lookahead prior to selecting each action. The value of lookahead has been amply demonstrated in domains such as chess. We believe that *hierarchical* lookahead with HLAs can be far more effective because it brings back to the present value information from far into the future. Put simply, it's better to evaluate the possible outcomes of writing an ICAPS paper than the possible outcomes of choosing "A" as its first character. We derive an angelic hierarchical generalization of Korf's LRTA* (1990), which shares LRTA*'s guarantees of eventual goal achievement on each trial and eventually optimal behavior after repeated trials. Experiments show that this algorithm substantially outperforms its nonhierarchical ancestor.

2. Background

2.1 Planning Problems

Deterministic, fully observable planning problems can be described in a representation-independent manner by a tuple $(S, s_0, t, \mathcal{L}, T, g)$, where S is a set of states, s_0 is the initial state, t is the goal state,[1] $\mathcal{L}$ is a set of primitive actions, and $T : S \times \mathcal{L} \rightarrow S$ and $g : S \times \mathcal{L} \rightarrow \mathbb{R}_+$ are transition and cost functions such that doing action a in state s leads to state $T(s, a)$ with cost $g(s, a)$.[2] These functions are overloaded to operate on sequences of actions in the obvious way: if $\mathbf{a} = (a_1, \ldots, a_m)$, then $T(s, \mathbf{a}) = T(\ldots T(s, a_1) \ldots, a_m)$ and $g(s, \mathbf{a})$ is the total cost of this sequence. The objective is to find a *solution* $\mathbf{a} \in \mathcal{L}^*$ for which $T(s_0, \mathbf{a}) = t$ and $g(s_0, \mathbf{a}) < \infty$.

Definition 1. A solution $\mathbf{a}^*$ is *optimal* iff it reaches the goal with minimal cost:

$$\mathbf{a}^* = \arg\min_{\mathbf{a} \in \mathcal{L}^* : T(s_0, \mathbf{a}) = t} g(s_0, \mathbf{a}).$$

We assume the state and action spaces are finite. To ensure that optimal solutions exist, we also assume that there is at least one finite-cost solution, and every cycle in the state space has positive cost. In this paper, we will represent S as the set of truth assignments to some set of ground propositions, and T using the STRIPS language (Fikes & Nilsson, 1971).

As a running example, we introduce a simple "nav-switch" domain. This is a grid-world navigation domain with locations represented by propositions X(x) and Y(y) for $x \in \{0, ..., x_{max}\}$ and $y \in \{0, ..., y_{max}\}$, and actions U, D, L, and R that move between them. There is a single global "switch" that can face horizontally (H) or vertically (¬H); move actions cost 2 if they go in the current direction of the switch and 4 otherwise.

1. A problem with multiple goal states can easily be translated into an equivalent problem with a single goal state.
2. $\mathbb{R}_+$ denotes the set $\mathbb{R} \cup \{\infty\}$.

The switch can be toggled by action F with cost 1, but only from a subset of designated squares. The goal is always to reach a particular square with minimum cost. Since these goals correspond to 2 distinct states (H, $\neg$H), we add a dummy action Z with cost 0 that moves from these (pseudo-)goal states to the single terminal state t. For example, in a 2x2 problem ($x_{max} = y_{max} = 1$) where the switch can only be toggled from the top-left square $(0, 0)$, if the initial state s_0 is X(1) $\wedge$ Y(0) $\wedge$ H, the optimal plan to reach the bottom-left square $(0, 1)$ is $(\mathsf{L}, \mathsf{F}, \mathsf{D}, \mathsf{Z})$ with cost 5.

2.2 High-Level Actions

In addition to a planning problem, our algorithms will be given a set $\mathcal{A}$ of *high-level actions*, along with a set $I(a)$ of allowed *immediate refinements* for each HLA $a \in \mathcal{A}$. Each immediate refinement consists of a finite sequence $\mathbf{a} \in \tilde{\mathcal{A}}^*$, where we define $\tilde{\mathcal{A}} = \mathcal{A} \cup \mathcal{L}$ as the set of all actions. Each HLA and refinement may have an associated precondition, which specifies conditions under which its application is allowed. To make a high-level sequence more concrete we may *refine* it, by replacing one of its HLAs by one of its immediate refinements, and we call one plan a *refinement* of another if it is reachable by any sequence of such steps. A *primitive refinement* consists only of primitive actions, and we define $I^*(\mathbf{a}, s)$ as the set of all primitive refinements of $\mathbf{a}$ that obey all HLA and refinement preconditions when applied from state s. We assume no plan is a refinement of itself. Finally, we assume a special *top-level* action $\mathsf{Act} \in \mathcal{A}$, and restrict our attention to plans in $I^*(\mathsf{Act}, s_0)$.

This definition of hierarchy is quite general. For instance, ordinary state-space search can be emulated by a "*flat hierarchy*" in which Act refines either to the empty sequence, or any primitive action followed by Act.

Definition 2. (Parr & Russell, 1998) A plan $\mathbf{a}^{h*}$ is *hierarchically optimal* iff

$$\mathbf{a}^{h*} = \arg\min_{\mathbf{a} \in I^*(\mathsf{Act}, s_0) : T(s_0, \mathbf{a}) = t} g(s_0, \mathbf{a}).$$

Remark. Because the hierarchy may constrain the set of allowed sequences, $g(s_0, \mathbf{a}^{h*}) \geq g(s_0, \mathbf{a}^*)$.

When equality holds from all possible initial states, the hierarchy is called *optimality-preserving*.

The hierarchy for our running example has three HLAs: $\mathcal{A} = \{\mathsf{Nav}, \mathsf{Go}, \mathsf{Act}\}$. $\mathsf{Nav}(x, y)$ navigates directly to location (x, y); it can refine to the empty sequence iff the agent is already at (x, y), and otherwise to any primitive move action followed by a recursive $\mathsf{Nav}(x, y)$. $\mathsf{Go}(x, y)$ is like Nav, except that it may flip the switch on the way; it either refines to $(\mathsf{Nav}(x, y))$, or to $(\mathsf{Nav}(x', y'), \mathsf{F}, \mathsf{Go}(x, y))$ where (x', y') can access the switch. Finally, Act is the top-level action, which refines to $(\mathsf{Go}(x_g, y_g), \mathsf{Z})$, where (x_g, y_g) is the goal location. This hierarchy is optimality-preserving for any instance of the nav-switch domain.[3]

3. Cost-Based Descriptions of HLAs

As mentioned in the introduction, our angelic semantics (MRW '07) describes the outcome of a high-level plan by its *reachable set* of states (by some refinement). However, these reachable sets say nothing about *costs* incurred along the way. This section describes a novel extension of the angelic approach that includes cost information. This will allow us to find *good* plans *quickly* by focusing on better-seeming plans first, and pruning provably suboptimal high-level plans without refining them further.

We begin with the notion of an *exact description* E_a of an HLA a, which specifies, for each pair of states (s, s'), the *minimum cost* of any primitive refinement of a that leads from s to s' (this generalizes the original definition from (MRW '07)).

Definition 3. The *exact description* of HLA a is a function $E_a(s)(s') = \min_{\mathbf{b} \in I^*(a,s) : T(s, \mathbf{b}) = s'} g(s, \mathbf{b})$.

Remark. Note that the set of primitive refinements may be infinite. The minimum must still be attained, however, due to the finiteness and positive-cycle assumptions.

3. Technically, Go and Nav also take the current x and y as parameters; to simplify notation, these extra parameters are suppressed in the text.

Remark. Definition 3 implies that if s' is not reachable from s by any refinement of a, $E_a(s)(s') = \infty$.

Definition 4. A *valuation* is a function $v : S \rightarrow \mathbb{R}_+$. The *initial valuation* v_0 has $v_0(s_0) = 0$ and $v_0(s) = \infty$ for all $s \neq s_0$.

We will use valuations to represent outcomes of high-level sequences, mapping each state to the cost of reaching it by some refinement (or ∞, if a state is not reachable). The initial valuation represents the initial situation of the agent (i.e., the outcome of doing the empty sequence).

We can think of descriptions as functions from states to valuations (see Figure 1(b)). Then, descriptions can be extended to functions from valuations to valuations, defining $\bar{E}_a(v)(s') = \min_{s \in S} v(s) + E_a(s)(s')$; for each final state s', the agent chooses the initial state s that enables reaching s' with minimal total cost. Finally, these extended descriptions can be composed to produce descriptions for high-level *sequences*.

Definition 5. Given a sequence $\mathbf{a} = (a_1, \ldots, a_N)$, the *exact transition function of* $\mathbf{a}$ is a function mapping valuations to valuations: $\bar{E}_\mathbf{a} = \bar{E}_{a_N} \circ \ldots \circ \bar{E}_{a_1}$.

Theorem 1. *For any integer N, final state s_N, and action sequence $\mathbf{a} \in \tilde{\mathcal{A}}^N$, the minimum over all state sequences $(s_1, ..., s_{N-1})$ of total cost $\sum_{i=1}^N E_{a_i}(s_{i-1})(s_i)$ equals $\bar{E}_\mathbf{a}(v_0)(s_N)$. Moreover, for any such minimizing state sequence, concatenating the primitive refinements of each HLA a_i that achieve the minimum cost $E_{a_i}(s_{i-1})(s_i)$ for each step yields a primitive refinement of* $\mathbf{a}$ *that reaches s_N from s_0 with minimal cost.*

Proof. The proof is by induction. When $N = 1$, the theorem follows trivially from Definitions 3 and 4. When $N > 1$,

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

□

By this theorem, an efficient, compact representation for E_a would (under mild conditions) lead to an efficient optimal planning algorithm. Unfortunately, since deciding even simple plan existence is PSPACE-hard (Bylander, 1994), we cannot hope for this in general. We will therefore consider principled *compact approximations* to E_a that still allow for precise inferences about the effects and costs of high-level plans.

3.1 Optimistic and Pessimistic Bounds on Descriptions

Definition 6. Valuation v_1 *dominates* valuation v_2, written $v_1 \preceq v_2$, iff $(\forall s \in S)\ v_1(s) \leq v_2(s)$.

Definition 7. Valuation v_1 *strictly dominates* v_2, written $v_1 \prec v_2$, iff $(\forall s \in S)\ v_1(s) < v_2(s) \ \vee \ v_2(s) = \infty$.

Definition 8. Valuation v_1 *weakly dominates* v_2, written $v_1 \leqq v_2$, iff $v_1 \preceq v_2 \ \wedge \ v_1 \not\prec v_2$.

(The latter two definitions will be used later, when we discuss pruning.)

Definition 9. An *optimistic description* O_a of HLA a satisfies $(\forall s)\ O_a(s) \preceq E_a(s)$.

For example, our optimistic description of Go (see Figure 1(a/c)) specifies that the cost for getting to the target location (possibly flipping the switch on the way) is at least twice its Manhattan distance from the current location; moreover, all other states are unreachable by Go.

(a) Properties of HLA Go(x_s, y_s, x_t, y_t)   (precondition $X(x_s) \wedge Y(y_s)$)

refs	(Nav(x_s, y_s, x_t, y_t)) (Nav(x_s, y_s, x, y), F, Go(x, y, x_t, y_t)) $(\forall x, y)$ s.t. a switch at (x, y)
optimistic	$-X(x_s)$, $-Y(y_s)$, $+X(x_t)$, $+Y(y_t)$, $\tilde{\pm}H$ $cost \geq 2 * (\lvert x_t - x_s \rvert + \lvert y_t - y_s \rvert)$

pessimistic	**if** H: $-X(x_s)$, $-Y(y_s)$, $+X(x_t)$, $+Y(y_t)$
	 $cost \leq 2\lvert x_t - x_s \rvert + 4\lvert y_t - y_s \rvert$
	 if ¬H: $-X(x_s)$, $-Y(y_s)$, $+X(x_t)$, $+Y(y_t)$
	 $cost \leq 4\lvert x_t - x_s \rvert + 2\lvert y_t - y_s \rvert$

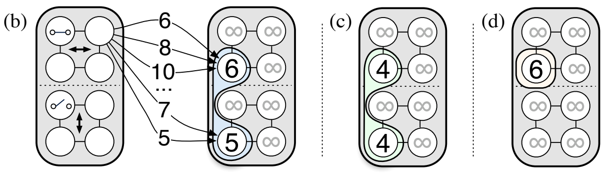


Figure 1: Some examples taken from our example nav-switch problem. (a) Refinements and NCSTRIPS descriptions of the Go HLA. The extra source position parameters (x_s, y_s) omitted elsewhere in the text are shown here, for precision. (b) Exact valuation from s_0 for Go$(1, 0, 0, 1)$. Gray rounded rectangles represent the state space; in the top four states (circles) the switch is horizontal, and in the bottom four it is vertical. Each arrow represents a primitive refinement of Go$(1, 0, 0, 1)$; the cost assigned to each state is the min cost of any refinement that reaches it. The exact reachable set corresponding to this HLA is also outlined. (c) Optimistic simple valuation $X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) : 4$ for the example in (b), as would be produced by the description in (a). (d) Pessimistic simple valuation $X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) \wedge H : 6$.

Definition 10. A *pessimistic description* P_a of HLA a satisfies $(\forall s)\ E_a(s) \preceq P_a(s)$.

For example, our pessimistic description of Go specifies that the cost to reach the destination is at most the cost incurred by directly navigating there without any further switch flips.

Remark. Primitive action descriptions are always exact. Formally, for primitive actions $a \in \mathcal{L}$, $O_a(s)(s') = P_a(s)(s') = g(s, a)$ iff $s' = T(s, a)$, ∞ otherwise.

Optimistic and pessimistic descriptions generalize our previous complete and sound descriptions (MRW '07). In this paper, we will assume that the descriptions are given along with the hierarchy. However, we note that it is theoretically possible to derive them automatically from the structure of the hierarchy.

As with exact descriptions, we can extend optimistic and pessimistic descriptions and then compose them to produce bounds on the outcomes of high-level sequences, which we call *optimistic* and *pessimistic valuations* (see Figure 1(c/d)).

Theorem 2. *Given any sequence* $\mathbf{a} \in \tilde{\mathcal{A}}^N$ *and state* s*, the cost* $c = \min_{\mathbf{b} \in I^*(\mathbf{a}, s_0) | T(s_0, \mathbf{b}) = s} g(s_0, \mathbf{b})$ *of the best primitive refinement of* $\mathbf{a}$ *that reaches* s *from* s_0 *satisfies* $\bar{O}_{a_N} \circ \ldots \circ \bar{O}_{a_1}(v_0)(s) \leq c \leq \bar{P}_{a_N} \circ \ldots \circ \bar{P}_{a_1}(v_0)(s)$.

Proof. The theorem is equivalent to the assertion that $\bar{O}_{a_N} \circ \ldots \circ \bar{O}_{a_1}(v_0) \preceq \bar{E}_{a_N} \circ \ldots \circ \bar{E}_{a_1}(v_0) \preceq \bar{P}_{a_N} \circ \ldots \circ \bar{P}_{a_1}(v_0)$. When $N = 1$, this follows trivially from Definitions 9 and 10. When $N > 1$, for optimistic descriptions (the pessimistic case is symmetric):

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

□

Moreover, following Theorem 1, these are the tightest bounds derivable from a set of optimistic and pessimistic descriptions.

The reader might wonder what descriptions are appropriate for Act. Since the agent cannot stop acting until it reaches the goal state, Act's pessimistic descriptions cannot assign finite cost to any outcome other than t. Moreover, the optimistic cost to t for Act will be our normal notion of an *admissible heuristic*, which could be automatically derived from a relaxed version of the problem (e.g., a planning graph).

3.2 Representing and Reasoning with Descriptions

Whereas the results presented thus far are representation-independent, to utilize them effectively we require compact representations for valuations and descriptions as well as efficient algorithms for operating on these representations.

In particular, we consider *simple valuations* of the form $\sigma : c$ where $\sigma \subseteq S$ and $c \in \mathbb{R}_+$, which specify a *reachable set* of states along with a single numeric bound on the cost to reach states in this set (all other states are assigned cost ∞). As exemplified in Figure 1(c/d), an optimistic simple valuation asserts that states in σ *may* be reachable with cost at least c, and other states are *unreachable*; likewise, a pessimistic simple valuation asserts that each state in σ *is* reachable with cost at most c, and other states *may* be reachable as well.[4]

Simple valuations are convenient, since we can reuse our previous machinery (MRW '07) for reasoning with reachable sets represented as DNF (disjunctive normal form) logical formulae and HLA descriptions specified in a language called NCSTRIPS (Nondeterministic Conditional STRIPS). NCSTRIPS is an extension of ordinary STRIPS that can express a set of possible effects with mutually exclusive, conjunctive preconditions. Each effect consists of four lists of propositions: add (+), delete (−), possibly-add ($\tilde{+}$), and possibly-delete ($\tilde{-}$). Added propositions are always made true in the resulting state, whereas possibly-added propositions may or may not be made true; in a pessimistic description, the agent can force either outcome, whereas in an optimistic one the outcome may not be controllable. By extending NCSTRIPS with cost bounds (which can be computed by arbitrary code), we produce descriptions suitable for the approach taken here. Figure 1(a) shows possible descriptions for Go in this extended language (as is typically the case, these descriptions could be made more accurate at the expense of conciseness).

With these representational choices, we require an algorithm for progressing a simple valuation represented as a DNF reachable set plus numeric cost bound through an extended NCSTRIPS description. If we perform this progression exactly, the output may not be a simple valuation (since different states in the reachable set may produce different cost bounds). Thus, we will instead consider an approximate progression algorithm that projects results back into the space of simple valuations. Applying this algorithm repeatedly will allow us to compute optimistic and pessimistic simple valuations for entire high-level sequences.

The algorithm is a simple extension of that given in (MRW '07), which progresses each (conjunctive clause, conditional effect) pair separately and then disjoins the results. This progression proceeds by (1) conjoining effect preconditions onto the clause (and skipping this clause 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). With our extended NCSTRIPS descriptions, each (clause, effect) pair also produces a cost bound. When progressing optimistic (resp. pessimistic) valuations, we simply take the min (resp. max) of all these bounds plus the initial bound to get the cost bound for the final valuation.[5] Hierarchical HLA and refinement preconditions (described in more detail below) are enforced by simply conjoining them onto the preconditions of each effect of an HLA.

As a concrete example, consider progressing the valuation $X(0) \wedge \neg X(1) \wedge Y(0) \wedge \neg Y(1) : 1$ through the optimistic and pessimistic descriptions of Go(0, 0, 0, 1). The optimistic description has only a single possible effect, yielding an optimistic result of $X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) : 3$. The pessimistic description has two

4. More interesting tractable classes of valuations are possible; for instance, rather than using a single numeric bound, we could allow linear combinations of indicator functions on state variables.

5. A more accurate algorithm for pessimistic progression sorts the clauses by increasing pessimistic cost, computes the minimal prefix of this list whose disjunction covers all of the remaining clauses, and then restricts the max over cost bounds to clauses in this prefix. We did not implement this version, since it requires many potentially expensive subsumption checks.

possible effects; after conjoining the preconditions and applying the effects, these give the output clauses and rewards $X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) \wedge H : 5$ and $X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) \wedge \neg H : 3$. Combining these and projecting back to a simple valuation gives the final result $(X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) \wedge H) \vee (X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) \wedge \neg H) : 5$, which could be simplified to $X(0) \wedge \neg X(1) \wedge \neg Y(0) \wedge Y(1) : 5$.

Our above definitions need some minor modifications to allow for approximate progression algorithms. For simplicity, we will absorb any additional approximation into our notation for the descriptions themselves:

Definition 11. An approximate progression algorithm corresponds to, for each extended optimistic and pessimistic description $\bar{O}_a$ and $\bar{P}_a$, (further) approximated descriptions $\tilde{O}_a$ and $\tilde{P}_a$. Call the algorithm *correct* if, for all actions a and valuations v, $\tilde{O}_a(v) \preceq \bar{O}_a(v)$ and $\bar{P}_a(v) \preceq \tilde{P}_a(v)$.

Intuitively, a progression algorithm is correct as long as the errors it introduces only further weaken the descriptions.

Theorem 3. *Theorem 2 still holds if we use any correct approximate progression algorithm, replacing each $\bar{O}_a$ and $\bar{P}_a$ with their further approximated counterparts $\tilde{O}_a$ and $\tilde{P}_a$.*

The proof is similar to that of Theorem 2.

4. Offline Search Algorithms

This section describes algorithms for the *offline* planning setting, in which the objective is to quickly find a low-cost sequence of actions leading all the way from s_0 to t.

Because we have *models* for our HLAs, our planning algorithms will resemble existing algorithms that search over primitive action sequences. Such algorithms typically operate by building a *lookahead tree* (see Figure 2(a)). The initial tree consists of a single node labeled with the initial state and cost 0, and computations consist of leaf node *expansions*: for each primitive action a, we add an outgoing edge labeled with that action and its cost $g(s, a)$, whose child is labeled with the state $s' = T(s, a)$ and total cost to s'. We also include at leaf nodes a heuristic estimate $h(s')$ of the remaining cost to the goal. Paths from the root to a leaf are potential plans; for each such plan $\mathbf{a}$, we estimate the total cost of its best continuation by $f(s_0, \mathbf{a}) = g(s_0, \mathbf{a}) + h(T(s_0, \mathbf{a}))$, the sum of its cost and heuristic value. If the heuristic h never overestimates, we call it *admissible*, and this f-cost will also never overestimate. If h also obeys the triangle inequality $h(s) \leq g(s, a) + h(T(s, a))$, we call it *consistent*, and expanding a node will always produce extensions with greater or equal f-cost. These properties are required for A* and its graph version (respectively) to efficiently find optimal plans. In this framework, some sort of repeated-state checking is often essential for good performance. For instance, in grid-world environments, A* graph search is exponentially faster than A* tree search (Russell & Norvig, 2003).

In hierarchical planning, we will consider algorithms that build *abstract lookahead trees* (ALTs). In an ALT, edges are labeled with (possibly high-level) actions and nodes are labeled with optimistic and pessimistic valuations for corresponding partial plans. For example, in the ALT in Figure 2(b), by doing $(\mathsf{Nav}(0, 0), \mathsf{F}, \mathsf{Go}(0, 1))$, state s_{01v} is definitely reachable with cost 5, s_{01h} may be reachable with cost at least 5, and no other states are possibly reachable. Since our planning algorithms will try to find low-cost solutions, we will be most concerned with finding optimistic (and pessimistic) bounds on the cost of the best primitive refinement of each high-level plan that reaches t. These bounds can be extracted from the final ALT node of each plan; for instance, the optimistic and pessimistic costs to t of plan $(\mathsf{Nav}(0, 0), \mathsf{F}, \mathsf{Go}(0, 1), \mathsf{Z})$ are 5.

In a generalization of the ordinary notion of consistency, we will sometimes desire *consistent* HLA descriptions, under which we never lose information by refining.[6] As in the flat case, when descriptions are consistent, the optimistic cost to t (i.e., f-cost) of a plan will never decrease with further refinement. Similarly, its best pessimistic cost will never increase.

6. Specifically, a set of optimistic descriptions (plus approximate progression algorithm, if applicable) is consistent iff, when we refine any high-level plan, its optimistic valuation dominates the optimistic valuations of its refinements. A set of pessimistic descriptions (plus progression algorithm) is consistent iff the state-wise minimum of a set of refinements' pessimistic valuations always dominates the pessimistic valuation of the parent plan.

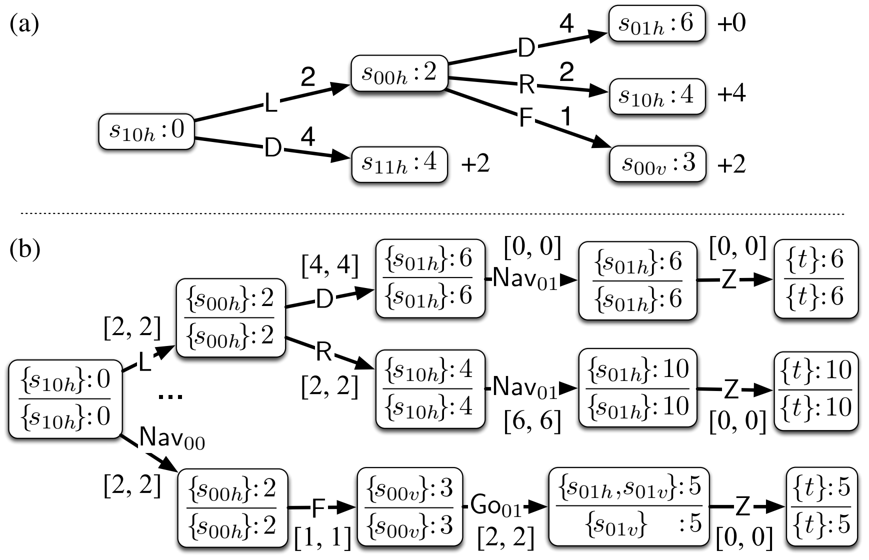

(a)
$s_{10h}:0$
L
2
$s_{00h}:2$
D
4
$s_{11h}:4$
+2
D
4
$s_{01h}:6$
+0
R
2
$s_{10h}:4$
+4
F
1
$s_{00v}:3$
+2
(b)
$\{s_{10h}\}:0$ / $\{s_{10h}\}:0$
[2, 2]
L
$\{s_{00h}\}:2$ / $\{s_{00h}\}:2$
[4, 4]
D
$\{s_{01h}\}:6$ / $\{s_{01h}\}:6$
[0, 0]
Nav$_{01}$
$\{s_{01h}\}:6$ / $\{s_{01h}\}:6$
[0, 0]
Z
$\{t\}:6$ / $\{t\}:6$
R
[2, 2]
$\{s_{10h}\}:4$ / $\{s_{10h}\}:4$
Nav$_{01}$
[6, 6]
$\{s_{01h}\}:10$ / $\{s_{01h}\}:10$
Z
[0, 0]
$\{t\}:10$ / $\{t\}:10$
...
Nav$_{00}$
[2, 2]
$\{s_{00h}\}:2$ / $\{s_{00h}\}:2$
F
[1, 1]
$\{s_{00v}\}:3$ / $\{s_{00v}\}:3$
Go$_{01}$
[2, 2]
$\{s_{01h}, s_{01v}\}:5$ / $\{s_{01v}\}:5$
Z
[0, 0]
$\{t\}:5$ / $\{t\}:5$

Figure 2: (a) A standard lookahead tree for our example. Nodes are labeled with states (written $s_{xy(h/v)}$) and costs-so-far, edges are labeled with actions and associated costs, and leaves have a heuristic estimate of the remaining distance-to-goal. (b) An *abstract* lookahead tree (ALT) for our example. Nodes are labeled with optimistic and pessimistic simple valuations and edges are labeled with (possibly high-level) actions and associated optimistic and pessimistic costs.

We first describe our ALT data structures and how they address some of the issues that arise in our hierarchical planning framework in novel ways. We then present our optimal planning algorithm, AHA*, and briefly describe an alternative "satisficing" algorithm, AHSS.

4.1 Abstract Lookahead Trees

Our ALT data structures support our search algorithms by efficiently managing a set of candidate high-level plans and associated valuations. The issues involved differ from the primitive setting because nodes store valuations rather than single states and exact costs, and because (unlike node expansion) plan refinement is "top-down" and may not correspond to simple extensions of existing plans.

Algorithm 1 shows pseudocode for some basic ALT operations. Our search algorithms work by first creating an ALT containing some initial set of *live* plans using MakeInitialALT, and then repeatedly refining live plans using RefinePlanEdge. A plan is considered live if it has not been pruned, and it has not yet been refined (at any HLA). When an HLA is refined, its preconditions and the preconditions of the refinement taken are added to the first action in the refinement, ensuring that lower-level sequences respect the higher-level conditions on their generation.[7] New plans are added to the ALT by the AddPlan operation, which starts at the existing node corresponding to the longest prefix shared with any existing plan, and creates nodes for the remaining plan suffix by progressing its valuations through the corresponding action descriptions. In the process, partial plans that cannot possibly reach the goal and those that are prunable by other plans (according to Definitions 12 and 14 below) are recognized and skipped over.

7. The preconditions are considered to be part of the identity of an action, and are taken into account during progression as described above. If a plan cannot respect the preconditions, progression will produce an empty optimistic reachable set and the plan will be automatically skipped in AddPlan. To ensure that preconditions are not forgotten, our implementation transparently adds a "noop" action to empty refinements.

Algorithm 1 : Abstract lookahead tree (ALT) operations

```
function AddPlan($n, (a_1, ..., a_k)$)
    for $i$ from 1 to k do
        if node $n[a_i]$ does not exist then
            create $n[a_i]$ from $n$ and the descriptions of $a_i$
            for node $n'$ where $a_{i+1}, ..., a_k$ is an existing extension from $n'$ do
                if $P(n') \prec O(n[a_i])$ then return          /* strict pruning */
                if $P(n') \leqq O(n[a_i])$ and
                  Prefix$(n')$ $+(a_{i+1}, ..., a_k)$ not an ancestor of $(a_1, ..., a_k)$ then
                    make $(a_1, ..., a_k)$ a parent of Prefix$(n')$ $+(a_{i+1}, ..., a_k)$
                    return                                 /* weak pruning */
        $n \leftarrow n[a_i]$
    If $O(n)(t) < \infty$ then mark $n$ as a live plan

function MakeInitialALT($s_0$, $plans$)
    $root \leftarrow$ a new node with $O(root) = P(root) = v_0$
    for each $plan \in plans$ do AddPlan($root, plan$)
    return $root$

function RefinePlanEdge($root, (a_1, ..., a_k), i$)
    mark node $root[a_1]...[a_k]$ as not live
    $p_h \leftarrow$ preconditions of $a_i$
    $parentPlan \leftarrow (a_1, ..., a_k)$
    for $(b_1...b_j) \in I(a_i)$ w/ refinement precondition $p_r$ do
        $plan \leftarrow (a_1, ..., a_{i-1},$ AddPreconditions$(b_1, p_h \wedge p_r), b_2, ..., b_j, a_{i+1}, ..., a_k)$
        make $parentPlan$ a parent of $plan$
        AddPlan($root, plan$)
```

A key requirement of the ALT operations is that they should be *optimality-preserving*. This corresponds to the invariant that, after executing any sequence of refinements and legal pruning operations,

$$\min_{\mathbf{a} \in \text{Live-Plans}(root)} \bar{E}_{\mathbf{a}}(v_0)(t) \quad = \quad c^* := \min_{\mathbf{a} \in \text{Initial-Plans}} \bar{E}_{\mathbf{a}}(v_0)(t) \tag{1}$$

Clearly, as long as each individual operation preserves at least one best-cost live plan, the invariant will remain satisfied. Without pruning, this invariant is preserved by refinement, since by definition pure refinement only splits a plan's set of primitive refinements. But, as mentioned above, some form of repeated-state detection or pruning is often essential for good performance. In particular, at minimum we would like our optimal search algorithm AHA* to reduce to ordinary A* graph search, when passed a "flat hierarchy" as described above. The pruning rules implemented in Algorithm 1 accomplish this goal; the rest of this section is devoted to describing them, and proving that they preserve invariant (1).

Lemma 1. *A live plan* **a** *can be pruned (i.e., marked non-live) while preserving invariant* (1)*, as long as a* different *live plan* **b** *exists such that* $\bar{E}_{\mathbf{b}}(v_0)(t) \leq \bar{E}_{\mathbf{a}}(v_0)(t)$*.*

This lemma provides a simple necessary and sufficient condition for pruning, but is by itself of little practical value since it assumes access to exact descriptions (that our algorithms lack). The next several definitions provide complementary, *sufficient* (but not necessary) conditions under which the existence of a suitable **b** can be *efficiently* proven, which we use in our actual implementation.

Definition 12. A live plan **a** passing through node n can be *strictly pruned* by another plan **b** passing through node n' when $P(n') \prec O(n)$ and Suffix$(\mathbf{b}, n')$ = Suffix$(\mathbf{a}, n)$.

Theorem 4. *Strict pruning preserves the invariant* (1).

Proof. First, show that $\bar{E}_{\mathbf{b}}(v_0)(t) < \bar{E}_{\mathbf{a}}(v_0)(t)$. Split $\mathbf{b}$ into $\mathbf{b}_1 = \textsc{Prefix}(n')$ and $\mathbf{b}_2 = \textsc{Suffix}(\mathbf{b}, n')$, and similarly divide $\mathbf{a}$ into $\mathbf{a}_1$ and $\mathbf{a}_2$. Now, the pruning condition ensures that for every primitive refinement of $\mathbf{a}_1$ that reaches some state s, there exists a strictly better primitive refinement of $\mathbf{b}_1$ that reaches s. Moreover, the continuation condition (that $\mathbf{b}_2 = \mathbf{a}_2$) ensures that any primitive refinement of $\mathbf{a}_2$ that can be done from s after a primitive refinement of $\mathbf{a}_1$ can also be done from s after a refinement of $\mathbf{b}_1$. Together, these facts demonstrate the above strict inequality. (Note that while $\mathbf{b}$ may have even better primitive refinements that use states reachable by $\mathbf{b}_1$ but not $\mathbf{a}_1$, this fact is not needed for the proof.)

Now, since the invariant holds before pruning, there must exist a live, hierarchically optimal plan $\mathbf{c}$ s.t. $\bar{E}_{\mathbf{c}}(v_0)(t) = c^*$. But, $c^* \leq \bar{E}_{\mathbf{b}}(v_0)(t) < \bar{E}_{\mathbf{a}}(v_0)(t)$. Thus, $\mathbf{c}$ is strictly better than $\mathbf{a}$, which entails that $\mathbf{c} \neq \mathbf{a}$ and $\mathbf{a}$ can be pruned by Lemma 1. □

Remark. The continuation requirement (that the plans have identical plan suffixes) is needed since the hierarchy might allow better continuations from node n than n'.

For example, the plan $(\mathsf{L}, \mathsf{R}, \mathsf{Nav}(0, 1), \mathsf{Z})$ in Figure 2(b) is prunable since its optimistic valuation is strictly dominated by the pessimistic valuation above it, and the empty continuation is allowed from that node.

Unfortunately, by itself this condition misses out on many opportunities for pruning. In particular, it cannot exploit weak (non-strict) domination, which results in all plans of possibly-equal cost being considered, leading to potentially exponential slowdown. For instance, in the nav-switch domain, a Nav action that travels h horizontal steps and v vertical steps has $\frac{(h+v)!}{h!\,v!}$ *optimal* primitive refinements. Thus, we also consider another pruning condition that allows some pruning based on weak domination, by maintaining an explicit graph recording plan provenance and pruning relationships.

In what follows, to simplify the presentation we assume that no plan will be generated more than once. Many useful hierarchies (including the two we consider in this work) have this property. For hierarchies that do not, a simple extension to the algorithms described is needed.

Definition 13. In the context of a particular abstract lookahead tree, let plan $\mathbf{a}$ be a *parent* of $\mathbf{b}$, if $\mathbf{b}$ is an immediate refinement of $\mathbf{a}$, *or* if $\mathbf{a}$ was *weakly pruned* by $\mathbf{b}$.

Definition 14. A live plan $\mathbf{a}$ passing through node n can be *weakly pruned* by another plan $\mathbf{b}$ passing through node n' when $P(n') \leqq O(n)$, $\textsc{Suffix}(\mathbf{b}, n') = \textsc{Suffix}(\mathbf{a}, n)$, *and* $\mathbf{b}$ is not an ancestor of $\mathbf{a}$.

For some intuition behind this rule, first consider how weak pruning could go wrong if the ancestor condition was not included. As a simplest example, a solitary live plan $\mathbf{a}$ might claim that it could achieve the goal with cost c^*. Upon refining $\mathbf{a}$, its sole refinement $\mathbf{b}$ could then be pruned on $\mathbf{a}$, thus sacrificing hierarchical optimality (and perhaps even completeness). The ancestor condition prevents this pathological case, as well as more subtle cases that can arise with several applications of weak pruning. In particular, by ensuring that the "ancestor graph" remains cycle-free, it guarantees the existence of at least one sink node corresponding to a hierarchically optimal, live plan.[8]

Theorem 5. *Weak pruning (alone, or in combination with strict pruning) preserves invariant* (1).

Proof. In fact, weak and strict pruning satisfy a stronger invariant, which entails (1). In order to define this invariant, we must add a small bookkeeping step to strict pruning: suppose that whenever a plan $\mathbf{a}$ is *strictly pruned* on $\mathbf{b}$, $\mathbf{a}$ becomes a *strict parent* of $\mathbf{b}$. Say that plan $\mathbf{a}$ is a *general ancestor* of $\mathbf{b}$ if $\mathbf{b}$ can be reached from $\mathbf{a}$ by following any combination of ordinary and strict edges, and reserve the term *ancestor* for relationships that exist without considering strict edges.

Now, the invariant is: at every point, from every previously considered plan $\mathbf{a}$ (live or not) with at least one primitive refinement that reaches the goal, there exists a *general path* to a *live, general descendant* $\mathbf{b}$ with

8. A simpler condition for weak pruning (implied by the above) is that $\mathbf{b}$ is also a live plan. Theoretically and empirically, this performs substantially worse, however; for instance, given a flat hierarchy, it cannot prune many plans that A* graph search would avoid.

non-increasing exact costs-to-goal along the path. Formally, $\bar{E}_{\mathbf{b}}(v_0)(t) \leq ... \leq \bar{E}_{\mathbf{a}}(v_0)(t)$ where the ellipses represent the exact costs of nodes on the path from **a** to **b**.

This invariant is trivially preserved by refinement without pruning. Thus, it suffices to show that the invariant is preserved by a single pruning operation following some arbitrary sequence of previous refinement and strict/weak pruning operations that preserved the invariant.

In particular, suppose that one of the pruning conditions allows pruning of plan **a** by **b**. Define $c^*(\mathbf{a}) = \bar{E}_{\mathbf{a}}(v_0)(t)$. By the pruning conditions, we must have $c^*(\mathbf{b}) \leq c^*(\mathbf{a})$, with strict inequality if pruning was strict. Moreover, before pruning, the invariant assures the existence of a general path with non-increasing exact cost from **b** to **c**, a live, general descendant of **b**.

Now, there are several cases. Since $c^*(\mathbf{c}) \leq c^*(\mathbf{b}) \leq c^*(\mathbf{a})$, if either $c^*(\mathbf{c}) < c^*(\mathbf{b})$ or $c^*(\mathbf{b}) < c^*(\mathbf{a})$, we have $c^*(\mathbf{c}) < c^*(\mathbf{a})$ by transitivity, so $\mathbf{a} \neq \mathbf{c}$. Otherwise, $c^*(\mathbf{c}) = c^*(\mathbf{b}) = c^*(\mathbf{a})$, and so we must be pruning **a** *weakly* on **b**. By the weak pruning condition, **b** cannot be an ancestor of **a** (but it may be a *general* ancestor). In other words, every path from **b** to **a** must include at least one strict edge. Now, when following a strict edge, exact costs strictly decrease; since $c^*(\mathbf{b}) = c^*(\mathbf{a})$, every path from **b** to **a** must include a strict cost increase as well. But, by assumption, there exists a path from **b** to **c** that does not include any cost increases, so again $\mathbf{a} \neq \mathbf{c}$.

Finally, since $\mathbf{a} \neq \mathbf{c}$, after pruning **a**, **c** will remain a live plan. Moreover, **c** will become a (possibly general) descendant of **a** (and its ancestors), via the (possibly strict) edge added from **a** to **b** after pruning, thus maintaining the invariant at **a** and its ancestors. □

We note that while these conditions are sufficient for pruning, they are not necessary, in that there exist situations where a (correctly) prunable plan will not be pruned using these rules alone. For instance, the suffix condition can be relaxed, and some pruning will be missed in the presence of zero-cost actions. Nevertheless, together these conditions are sufficient to reproduce A* graph search in certain conditions:

Theorem 6. *So long as all primitive actions have strictly positive cost, when applied to a "flat hierarchy", strict and weak pruning can prune any plan that would be skipped by A* graph search when executed on the corresponding primitive search problem.*

Proof. Although it is usually described differently, A* graph search can be understood as simply A* tree search plus the following pruning rule: a (primitive) plan **a** that reaches state s with cost c can be pruned, if another *live* or *refined* (i.e., not pruned) plan **b** reaches state s with cost $\leq c$.

In the "flat hierarchy" setting, plans are the same, except that they are suffixed by Act, whose optimistic description embodies the heuristic function and whose pessimistic description is vacuous. Thus, **a** and **b** will always have the same plan suffix: (Act). Moreover, primitive action descriptions are exact, so weak domination corresponds to reaching the same state with equal cost, and strict domination means reaching the same state with strictly better cost. So, if **b** has strictly lower cost than **a**, the strict pruning rule will prune **a**. If the cost is equal, and **b** is live, it cannot have any descendants and thus the weak pruning rule can prune **a**. Finally, if **b** has been refined, its immediate descendants must have strictly greater g-cost (by the positive-cost assumption). Moreover, when one plan weakly prunes another, the two plans must have equal g-costs. Thus, every descendant of **b** has strictly greater g-cost. This implies that **a** cannot be a descendant of **b**, and can again be pruned by the weak pruning rule. □

Since detecting all subsumption relationships can be very expensive, our implementation uses hashing to consider pruning only for pairs of nodes with identical plan suffixes and reachable set representations. Testing of ancestor relationships is implemented with a naive algorithm; we note that efficient incremental algorithms exist for this problem (e.g., (Haeupler *et al.*, 2008)). We have also developed stronger but more complex rules that can do more pruning, as well as weaker but more efficiently checkable rules. These issues are beyond the scope of this paper, and will be expanded on in future work.

4.2 Angelic Hierarchical A*

Our first offline algorithm is *Angelic Hierarchical A** (AHA*), a hierarchically optimal planning algorithm that takes advantage of the semantic guarantees provided by optimistic and pessimistic descriptions. AHA*

Algorithm 2 : Angelic Hierarchical A*

```
function FindOptimalPlan($s_0, t$)
    $root$ ← MakeInitialALT($s_0$, {(Act)})
    while ∃ a live plan do
        $\mathbf{a}$ ← a live plan with min optimistic cost to $t$ (tiebreak by pessimistic cost, then depth)
        if $\mathbf{a}$ is primitive then return $\mathbf{a}$
        RefinePlanEdge($root$, $\mathbf{a}$, index of any HLA in $\mathbf{a}$)
    return failure
```

(see Algorithm 2) is essentially A* in *refinement space*, where the initial node is the plan (Act), possible "actions" are *refinements* of a plan at some HLA, and the goal set consists of the primitive plans that reach t from s_0. The algorithm repeatedly expands a node with smallest optimistic cost bound, until a goal node is chosen for expansion, which is returned as an optimal solution.

More concretely, at each step AHA* selects a high-level plan $\mathbf{a}$ with minimal optimistic cost to t (e.g., the bottom plan in Figure 2(b)). Then it *refines* $\mathbf{a}$, selecting some HLA a and adding to the ALT all plans obtained from $\mathbf{a}$ by replacing a with one of its immediate refinements.

We will make the technical assumption that for every c, there are only finitely many high-level plans with optimistic cost less than c. This is essentially a positive-cost-cycle condition on the optimistic costs, and is not hard to ensure in practice. Under this assumption, we have the following theorem.

Theorem 7. *AHA* eventually terminates, and returns a hierarchically optimal plan.*

Proof. We will show that at the beginning of each iteration of the loop, the lookahead tree contains a plan $\mathbf{b}$ which can be refined to an hierarchically optimal primitive plan. This is certainly true at the first iteration. By induction, suppose it is true at the k^{th} iteration. Now, if there exists such a plan that is not chosen for refinement, then it will continue to be in the tree on the next iteration. So we only need to worry about the case when there is a unique such plan, and it is chosen for refinement. By definition, no matter which action in the plan is refined, at least one refinement will continue to be refinable to an optimal plan. By Theorems 4 and 5, the first such refinement added to the tree will not be pruned.

In particular, the invariant above holds when the loop terminates. At this point, the returned plan has optimistic cost lower than all other plans in the tree. Since its own optimistic cost is exact (as it is primitive), it in fact has minimal cost among all refinements of plans currently in the tree, and is therefore hierarchically optimal.

Finally, by assumption on the optimistic costs, all plans whose cost is at most the optimal cost will eventually be considered, including the hierarchically optimal one, which is primitive. Thus the algorithm eventually terminates. □

We now make concrete the connection between AHA* and standard A* search. AHA* clearly differs from A* over the state space, since the set of candidate plans and expansion operations differ. However, it is closely related to A* or greedy best-first search in the space of abstract plans. This search space consists of all sequences of high-level or primitive actions together with a dummy terminal state t. The initial state is the plan Act. Given a rule for choosing which HLA of a given plan to refine next, the successors of a nonprimitive plan are obtained by substituting the refinements of that HLA into the original plan, and the associated cost is 0. A primitive plan's only successor is the terminal state, and this move's cost equals the cost of the primitive plan. The heuristic value of a plan is its optimistic cost.

Theorem 8. *If the optimistic descriptions are consistent, then the sequence of plans refined by AHA* is a subsequence of the sequence of plans expanded by A* over the corresponding plan space, for some sequence of tiebreaking choices.*

Proof. Let $\mathbf{a}_1, \ldots, \mathbf{a}_k$ be the sequence of plans refined by AHA*, and S_t its set of live plans at step t. We show inductively the stronger statement that we can construct tiebreaking choices for A* such that if $\mathbf{b}_1, \ldots, \mathbf{b}_l$

denotes its sequence of expanded plans and S'_t denotes its open list at step t, there exist times $1 = t_1 < \ldots < t_k = l$ such that $\mathbf{a}_i = \mathbf{b}_{t_i}$ and $S_i \subseteq S'_{t_i}$. The statement is clearly true for t_1. Consider step i of AHA*, which corresponds to step t_i of A*. Following this step, the live plans in the ALT are S_{i+1}. At this point, the plan $\mathbf{a}_{i+1}$ is tied for the lowest cost in S_{i+1}. S'_{t_i+1} contains this plan, and possibly other ones with lower cost. However, none of those can be in S_{i+1}. We can therefore, by making appropriate tiebreaking choices in A*, ensure that, if t_{i+1} is the next time at which A* expands a plan in S_{i+1}, $\mathbf{b}_{t_{i+1}} = \mathbf{a}_{i+1}$, and furthermore, $S_{i+1} \subseteq S'_{t_{i+1}}$, completing the induction. □

While AHA* might thus seem like an obvious generalization of A* to the hierarchical setting, we believe that it is an important contribution for several reasons. First, its effectiveness hinges on our ability to generate nontrivial cost bounds for high-level sequences, which did not exist previously. Second, it derives additional power from our ALT data structures, which provide caching, pruning, and other novel improvements specific to the hierarchical setting.

The only free parameter in AHA* is the choice of which HLA to refine in a given plan; our implementation chooses the first HLA with nonzero gap between its optimistic and pessimistic costs (defined below).

Finally, we note that with consistent descriptions, as soon as AHA* finds an optimal high-level plan with equal optimistic and pessimistic costs, it will find an optimal primitive refinement very efficiently. Consistency ensures that after each subsequent refinement, at least one of the resulting plans will also be optimal with equal optimistic and pessimistic costs; moreover, all but the first such plan will be pruned. Further refinement of this first plan will continue until an optimal primitive refinement is found *without backtracking*.

4.3 Angelic Hierarchical Satisficing Search

This section presents an alternative algorithm, Angelic Hierarchical Satisficing Search (AHSS), which attempts to find a plan that reaches the goal with at most some pre-specified cost α. AHSS can be much more efficient than AHA*, since it can commit to a plan without first proving its optimality.

At each step, AHSS (see Algorithm 3) begins by checking if any primitive plans succeed with cost $\leq \alpha$. If so, the best such plan is returned. Next, if any (high-level) plans succeed with pessimistic cost $\leq \alpha$, the best such plan is committed to by discarding other potential plans. Finally, a plan with maximum *priority* is refined at one of its HLAs. Priorities can be assigned arbitrarily; our implementation uses the negative average of optimistic and pessimistic costs,[9] to encourage a more depth-first search and favor plans with smaller pessimistic cost.

Theorem 9. *If there exist primitive plans consistent with the hierarchy, with cost $\leq \alpha$, AHSS eventually returns one of them. Otherwise, when α is finite, it eventually returns failure.*

Proof. The algorithm eventually terminates since there are only finitely many plans with optimistic cost $\leq \alpha$. Since optimistic costs are exact for primitive plans, it will never falsely report success. Suppose there do exist primitive plans with cost $\leq \alpha$. It suffices to show that at the beginning of each iteration, the tree contains a plan one of whose primitive refinements has cost $\leq \alpha$. The invariant holds by assumption at the first iteration. Suppose it is true at the beginning of the k^{th} iteration. It will continue to hold after the if-statement, by definition of pessimistic costs. We only need to consider the case when there is a single such plan in the tree after the if-statement, and it is selected for refinement. Regardless of which action in the plan is refined, one of the refinements will also have a primitive refinement with cost $\leq \alpha$. At least one such refinement will not be pruned. Thus the invariant holds at the next iteration. □

5. Online Search Algorithms

In the *online* setting, an agent must begin executing actions without first searching all the way to the goal. The agent begins in the initial state s_0, performs a fixed amount of computation, then selects an action a.[10]

9. Except for Act, which contributes 3 times its optimistic cost to the total. For the purpose of computing priorities, infinite pessimistic costs are replaced with twice the corresponding optimistic costs.

10. More interesting ways to balance real-world and computational cost are possible, but this suffices for now.

Algorithm 3 : Angelic Hierarchical Satisficing Search

```
function FindSatisficingPlan($s_0, t, \alpha$)
    $root$ ← MakeInitialALT($s_0$, {(Act)})
    while ∃ a live plan with optimistic cost ≤ $\alpha$ to $t$ do
        if any live plan has pessimistic cost ≤ $\alpha$ to $t$ then
            if any such plans are primitive then return a best one
            else delete all plans other than one with min pessimistic cost
        a ← a live plan with optimistic cost ≤ $\alpha$ to $t$ with max priority
        RefinePlanEdge($root$, a, index of any HLA in a)
    return failure
```

It then does this action in the environment, moving to state $T(s_0, a)$ and paying cost $g(s_0, a)$. This continues until the goal state t is reached. Performance is measured by the total cost of the actions executed. We assume that the state space is *safely explorable*, so that the goal is reachable from any state (with finite cost), and also assume positive action costs and consistent heuristics/descriptions from this point forward.

This section presents our next contribution, one of the first *hierarchical lookahead* algorithms. Since it will build upon a variant of Korf's (1990) Learning Real-Time A* (LRTA*) algorithm, we begin by briefly reviewing LRTA*.[11]

At each environment step, LRTA* uses its computation time to build a lookahead tree consisting of all plans **a** whose cost $g(s_0, \mathbf{a})$ just exceeds a given threshold. Then, it selects one such plan $\mathbf{a}_{min}$ with minimal f-cost and does its first action in the world. Intuitively, looking farther ahead should increase the likelihood that $\mathbf{a}_{min}$ is actually good, by decreasing reliance on the (error-prone) heuristic. The choice of candidate plans is designed to compensate for the fact that the heuristic h is typically biased (i.e., admissible) whereas g is exact, and thus the f-cost of a plan with higher h and lower g may not be directly comparable to one with higher g and lower h.

This core algorithm is then improved by a learning rule. Whenever a partial plan **a** leading to a previously-visited state s is encountered during search, further extensions of **a** are not considered; instead, the remaining cost-to-goal from s is taken to be the value computed by the most recent search at s. This augmented algorithm has several nice properties:

Theorem 10. *(Korf, 1990) If g-costs are positive, h-costs are finite, and the state space is finite and safely explorable, then LRTA* will eventually reach the goal.*

Theorem 11. *(Korf, 1990) If, in addition, h is admissible and ties are broken randomly, then given enough runs, LRTA* will eventually learn the true cost of every state on an optimal path, and act optimally thereafter.*

However, as described thus far, LRTA* has several drawbacks. First, it wastes time considering obviously bad plans. (Korf prevented this with "alpha pruning"). Second, a cost threshold must be set in advance, and picking this threshold so that the algorithm uses a desired amount of computation time may be difficult. Both drawbacks can solved using the following *adaptive LRTA** algorithm, a relative of Korf's "time-limited A*": (1) Start with the empty plan. (2) At each step, select an unexpanded plan with lowest f-cost. If this plan has greater g-cost than any previously expanded plan, "lock it in" as the current return value. Expand this plan. (3) When computation time runs out, return the current "locked-in" plan.

Theorem 12. *At any point during the operation of this algorithm, let* **a** *be the current locked-in plan, c_2 be its corresponding "record-setting" g-cost, and c_1 be the previous record g-cost ($c_1 < c_2$). Given any threshold in $[c_1, c_2)$, LRTA* would choose* **a** *for execution (up to tiebreaking).*

Proof. First, note that given any threshold $c \in [c_1, c_2)$, LRTA* would definitely have constructed and expanded all of the ancestors of **a**. Consider any ancestor of **a**. By consistency and positive action costs, it must

11. To be precise, Korf focused on the case of unit action costs; we present a natural generalization to positive real-valued costs.

have $\leq g$-cost and f-cost than **a**. Because **a** was "record-setting", the g-cost must actually be strictly <. Now, suppose that this ancestor was not expanded. Then, its g-cost must be > c. But, $c_1 < c$ was the previous record-setting cost, so we have a contradiction. Thus, LRTA* would have generated but not expanded **a**.

Now, suppose that LRTA* with threshold c chooses some other plan over **a** for execution. This plan must have cost $> c$ to be present and unexpanded, and f-cost < that of **a** to be selected. But, if this was the case, this plan would have been selected for expansion by the adaptive algorithm before **a**, and would have been the previous record-setting plan. But, its cost is $> c \geq c_1$, the cost of the previous record-setting plan, a contradiction. □

Thus, this modified algorithm can be used as an efficient, anytime version of LRTA*. Since its behavior reduces to the original version for a particular (adaptive) choice of cost thresholds, all of the properties of LRTA* hold for it as well.

5.1 Angelic Hierarchical Learning Real-Time A*

This section describes Angelic Hierarchical Learning Real-Time A* (AHLRTA*, see Algorithm 4), which bears (roughly) the same relation to adaptive LRTA* as AHA* does to A*. Because a single HLA can correspond to many primitive actions, for a given amount of computation time we hope that AHLRTA* will have a greater effective lookahead depth than LRTA*, and thus make better action choices. However, a number of issues arise in the generalization to the hierarchical setting that must be addressed to make this basic idea work in both theory and practice.

First, while AHLRTA* searches over the space of high-level plans, when computation time runs out it must choose a *primitive* action to execute. Thus, if the algorithm initializes its ALT with the single plan (Act), it will have to consider its refinements carefully to ensure that in its final ALT, at least one of the (hopefully better) high-level plans begins with an executable primitive. To avoid this issue (and to ensure convergence of costs, as described below), we instead choose to initialize the ALT with the set of all plans consisting of a primitive action followed by Act.[12] With this set of plans, the choice of which HLA to refine in a plan is open; our implementation uses the policy described above for AHA*.

Second, as we saw earlier, an analogue of f-cost can be extracted from our optimistic valuations. However, there is no obvious breakdown of f into g and h components, since a high-level plan can consist of actions at various levels, each of whose descriptions may make different types and degrees of characteristic errors. For now, we assume that a set of higher-level HLAs (e.g., Act and Go) has been identified, let h be the sum of the optimistic costs of these actions, and let $g = f - h$ be the cost of the primitives and remaining HLAs.

Finally, whereas the outcome of a primitive plan is a particular concrete state whose stored cost can be simply looked up in a hash table, the optimistic valuations of a high-level plan instead provide a *sequence* of *reachable sets* of states. In general, for each such set we could look up and combine the stored costs of its elements; instead, however, for efficiency our implementation only checks for stored costs of singleton optimistic sets (e.g., those corresponding to a primitive prefix of a given high-level plan). If the state in a constructed singleton set has a stored cost, progression is stopped and this value is used as the cost of the remainder of the plan. This functionality is added by modifying REFINEPLANEDGE and ADDPLAN accordingly (not shown).

Given all of these choices, we have the following:

Theorem 13. *AHLRTA* reduces to adaptive LRTA*, given a "flat hierarchy" (in which* Act *refines to the empty sequence, or any primitive action followed by* Act*).*

Proof. Trivial; simply note that refining a plan in the "flat" hierarchy is the same as expanding a plan in the primitive LRTA* setting. □

12. Note that with this choice, the plans considered by the agent may not be valid hierarchical plans (i.e., refinements of Act). However, since the agent can change its mind on each world step, the actual sequence of actions executed in the world is not in general consistent with the hierarchy anyway.

Algorithm 4 : Angelic Hierarchical Learning Real-Time A*

```
function HierarchicalLookaheadAgent($s_0, t$)
    $memory$ ← an empty hash table
    while $s_0 \neq t$ do
        $root$ ← MakeInitialALT($s_0, \{(a, \mathsf{Act}) \mid a \in \mathcal{L}\}$)
        $(g, a, f) \leftarrow (-1, nil, 0)$
        while ∃ live plans from $root$ and time remains do
            $\mathbf{a}$ ← a live plan w/ min $f$-cost
            if $\mathbf{a}$ is primitive or reaches a known state, or the $g$-cost of $\mathbf{a} > g$ then
                $(g,a,f)$ ← ($g$-cost of $\mathbf{a}$, $a_1$, $f$-cost of $\mathbf{a}$)
                if $\mathbf{a}$ is primitive or reaches a known state then break
            RefinePlanEdge($root$, $\mathbf{a}$, some index, $memory$)
        do $a$ in the world
        $memory[s_0] \leftarrow f$
        $s_0 \leftarrow T(s_0, a)$
```

(In fact, this is how we have implemented LRTA* for our experiments.) Moreover, the desirable properties of LRTA* also hold for AHLRTA* in general hierarchies. This follows because AHLRTA* behaves identically to LRTA* in neighborhoods in which every state has been visited at least once.

Theorem 14. *If primitive g-costs are positive, f-costs are finite, and the state space is finite and safely explorable, then AHLRTA* will eventually reach the goal.*

Proof. Note that AHLRTA* is very similar to single-step-lookahead LRTA*, where the heuristic is computed by a limited hierarchical search from each next state reachable by some primitive action. In fact, the only difference is that AHLRTA* may choose paths based on the fact that they reach low-cost previously visited states (or the goal) that are more than 1 step away. However, this situation can only occur when none of the earlier states on the path have been visited before. Thus, at each step AHLRTA* either behaves identically to LRTA*, or moves to a previously unvisited state (while preserving the admissibility of all stored costs). Since there are only finitely many states, AHLRTA* can only behave differently than LRTA* finitely many times, and must thus eventually reach the goal. □

Theorem 15. *If, in addition, f-costs are admissible, ties are broken randomly, and the hierarchy is optimality-preserving, then over repeated trials AHLRTA* will eventually learn the true cost of every state on an optimal path and act optimally thereafter.*

Proof. Same as previous theorem. □

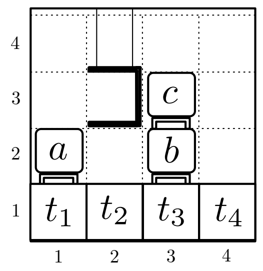
If f-costs are inadmissible or the hierarchy is not optimality-preserving, the theorem still holds if s_0 is sampled from a distribution with support on S in each trial.

6. Experiments

This section describes results for the above algorithms on two domains: our "nav-switch" running example, and the *warehouse world* (MRW '07).[13]

The warehouse world is an elaboration of the well-known blocks world, with discrete spatial constraints added. In this domain, a forklift-like gripper hanging from the ceiling can move around and manipulate blocks stacked on a 1-d table. The gripper and blocks occupy single (x, y) grid cells, where x represents position on the table and y represents height. The gripper can move to free squares in the four cardinal directions,

13. Empirical results differ from Marthi, Russell, & Wolfe (2008b,a) due to a programming error, which reduced the efficiency of all algorithms, but especially A* graph search.



HLA	**Goal**
Act	Achieve goal by MoveTo? Move*
Move($b, c, ...$)	Get block b and stack on c by (Nav to top, Turn)? Nav to one side of b, Pickup, MoveTo($c, ...$)
MoveTo($c, ...$)	Stack block held on c by (Nav to top, Turn)? Nav to one side of c, Putdown
Nav(x, y)	Go to (x, y)

Figure 3: Left: A 4x4 warehouse world problem with goal $\textsc{on}(c, t_2) \wedge \textsc{on}(a, c)$. Right: HLAs for warehouse world domain.

turn (to face the other way) when in the top row, and pick up and put down blocks from either side. Each primitive action has unit cost. Because of the limited maneuvering space, warehouse world problems can be rather difficult. For instance, Figure 3 shows a problem that cannot be solved in fewer than 50 primitive steps. The figure also shows our (optimality-preserving) hierarchy for the domain.[14] We consider 21 instances of varying difficulty.

For the nav-switch domain, we consider square grids of varying size where 20 randomly chosen squares can access the switch, and the goal is always to navigate from one corner to the other. We use the hierarchy and descriptions described above.

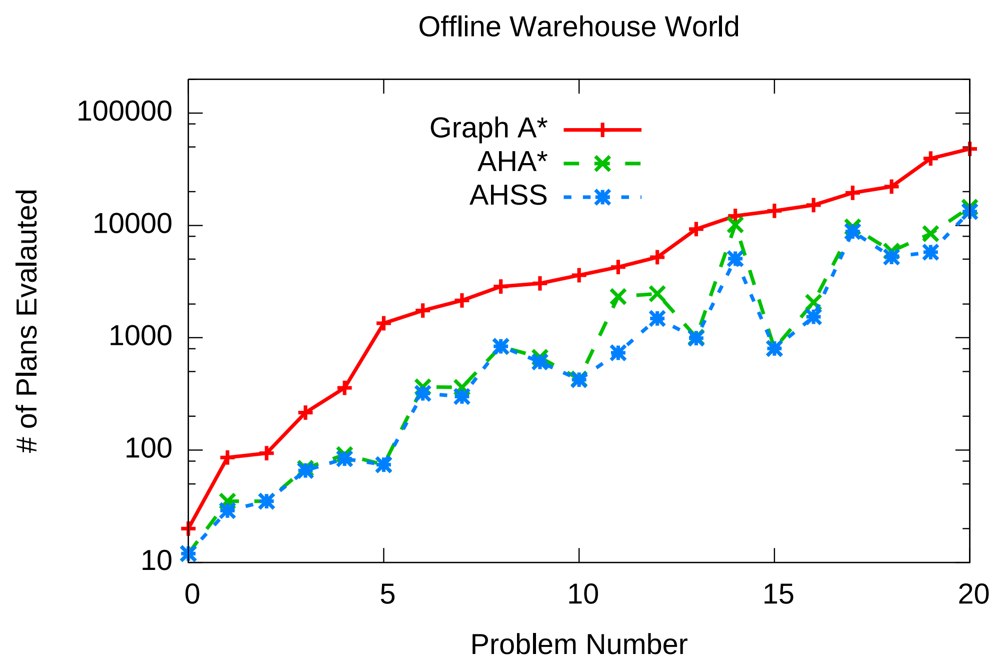
We first present results for our offline algorithms on these domains (see Figure 4). On the warehouse world instances, AHA* finds optimal solutions while evaluating between a half and full order of magnitude fewer plans than A*, in most cases. (Suboptimal) AHSS performs only slightly better. Results for runtimes are qualitatively similar, but with a 2x-3x smaller gap between A* graph search and the hierarchical algorithms. This gap is due to overhead introduced by progressing optimistic and pessimistic valuations, and multiple progressions required per new plan in the hierarchical setting.

On the nav-switch instances, differences between the algorithms are much greater. Runtime for A* graph search grows quadratically with the size of the board (proportional to the number of squares), while the hierarchical algorithms' scaling is closer to linear. The reason is that in this domain, the descriptions for Nav are exact, and thus AHA* can very quickly find a provably optimal high-level plan and refine it down to the primitive level without backtracking, as described earlier. AHSS is even faster, since it can omit the first step and immediately commit to a plan that simply navigates directly to the goal location.

We also collected results for an improved version of the Hierarchical Forward Search (HFS) algorithm (MRW '07) , which does not consider plan cost. Because it lacks the heuristic guidance provided by HLA cost estimates, HFS not only produces (sometimes very) suboptimal plans, it is also much slower than the heuristic algorithms we consider here: it could only solve the first 5 warehouse world problems, and nav-switch problems up to size 20, before exceeding the 512 MB memory limit. On the most difficult problems solved by HFS, runtime and plan counts (not shown in Figure 4) are several orders of magnitude higher than any of the other algorithms tested.

The obvious next step would be to compare AHA* with other optimal hierarchical planners, such as SHOP2 on its "optimal" setting. However, this is far from straightforward, for several reasons. First, useful hierarchies are often not optimality-preserving, and it is not at all obvious how we should compare different "optimal" planners that use different standards for optimality. Second, as described in the related work section below, the type and amount of problem-specific information provided to our algorithms can be very different

14. Descriptions are as follows. Only Nav has a non-vacuous pessimistic description; it states that the target location can definitely be reached when it and the top row is free of blocks, with cost corresponding to going up to the top row, then over and down to the destination (if the destination is in the current column, cost is just the vertical distance). Nav's optimistic cost is Manhattan distance, Move and MoveTo optimistically succeed (with the gripper on either side of the destination) with a lower cost bound based on Manhattan distance, and Act optimistically reaches the goal with cost lower-bounded by an adjusted bipartite matching heuristic.



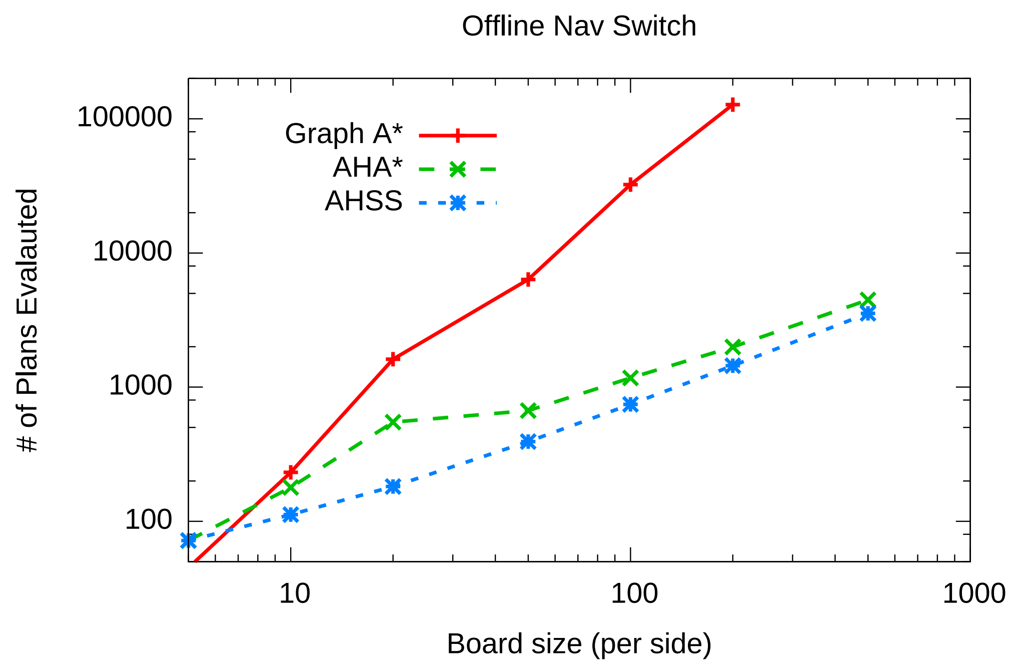
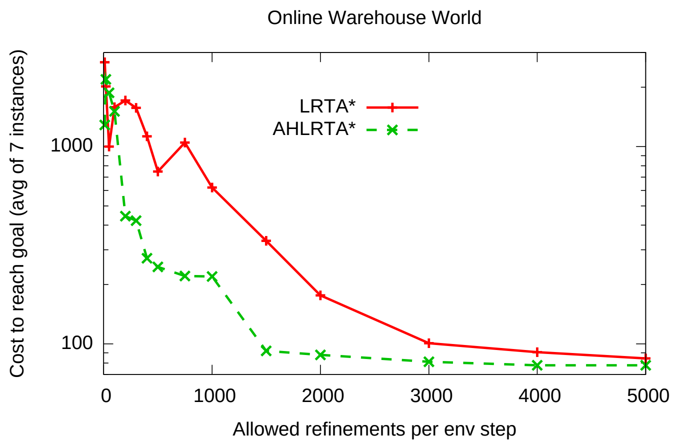
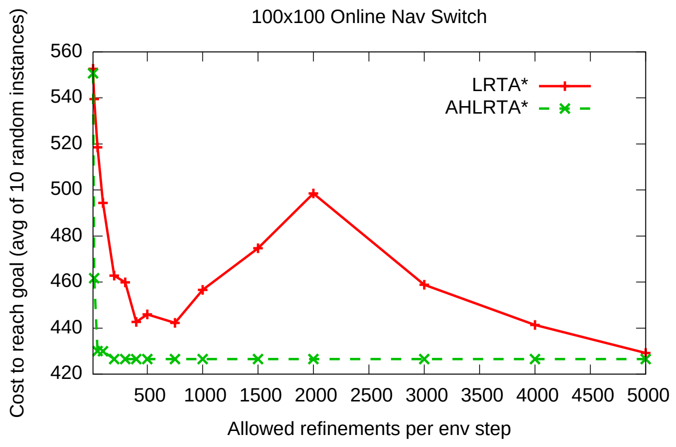


Figure 4: Number of plans evaluated to find an (optimal) solution, on some warehouse world and nav-switch problem instances. The algorithms are (flat) A* graph search, AHA*, and (suboptimal) AHSS with threshold $\alpha=\infty$. Warehouse world problems are ordered by difficulty for A* graph search. Nav-switch data points are medians of three random domains of each size. Note log scale.





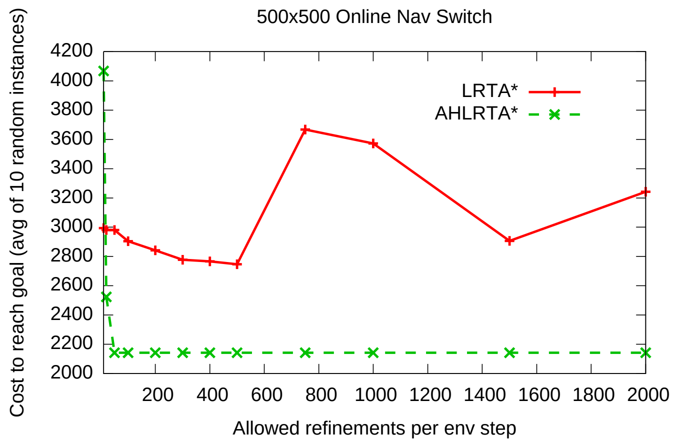


Figure 5: Total cost-to-goal for online algorithms as a function of the number of allowed refinements per environment step, averaged (using geometric mean) over seven medium-difficulty warehouse world instances (top), ten 100x100 nav-switch instances (middle), and ten 500x500 nav-switch instances (bottom). (Warehouse world costs shown in log-scale.)

than for HTN planners such as SHOP2. We have yet to find a way to perform meaningful experimental comparisons under these circumstances.

For the online setting, we compared flat LRTA* and AHLRTA*. The performance of an online algorithm on a given instance depends on the number of allowed refinements per step. Our graphs therefore plot total cost against refinements per step for LRTA* and AHLRTA*. Time per refinement is very similar for LRTA* and AHLRTA* in both domains, with hierarchical refinements at most 60% slower, usually much less.

The top graph of Figure 5 is averaged across seven medium-difficulty warehouse world instances from Figure 4.[15] This domain is fairly challenging for online lookahead, as the combinatorial structure of the problem makes the Act heuristic somewhat unreliable. AHLRTA* was able to perform reasonably given very few refinements per step, and quickly converged to near-optimal solutions. In contrast, flat lookahead required significantly more refinements to reach near-optimal behavior, and rarely achieved optimal behavior within the range of allowed refinements considered (note that the y-axis is on a log scale).

The bottom two graphs show online performance in the nav-switch domain, averaged over ten random 100x100 and ten random 500x500 instances (each with 20 switch locations). This domain is relatively easy as an online lookahead problem, because the heuristic for Act always points in roughly the right direction. In all cases, the hierarchical agent behaved optimally given very few refinements per step. With this number of refinements, the flat agent usually followed a reasonable, though suboptimal plan. As the number of allowed refinements was increased to intermediate numbers, performance actually decreased in most cases; it seems that flat LRTA* consistently suffers from lookahead pathologies on this class of problems. Finally, even given a large number of refinements, the flat agent did not converge to optimal behavior. In this domain, the hierarchical agent's ability to quickly find a good high-level sequence of switch flips and start heading for the first allows it to greatly outperform the flat agent's local lookahead strategy.

7. Related Work

We briefly describe work related to our specific contributions, deferring to (MRW '07) for discussion of relationships between this general line of work and previous approaches.

Most previous work in hierarchical planning (Tate, 1977; Yang, 1990; Russell & Norvig, 2003) has viewed HLA descriptions (when used at all) as constraints on the planning process (e.g., "only consider refinements that achieve *p*"), rather than as making true assertions about the effects of HLAs. Such HTN planning systems, e.g., SHOP2 (Nau *et al.*, 2003), have achieved impressive results in previous planning competitions and real-world domains—despite the fact that they cannot assure the correctness or bound the cost of abstract plans. Instead, they encode a good deal of domain-specific advice on which refinements to try in which circumstances, often expressed as arbitrary program code. For fairly simple domains described in tens of lines of PDDL, SHOP2 hierarchies can include hundreds or thousands of lines of Lisp code. In contrast, our algorithms only require a (typically simple) hierarchical structure, along with descriptions that logically follow from (and are potentially automatically derivable from) this structure.

The closest work to ours is by Doan and Haddawy (1995). Their DRIPS planning system uses action abstraction along with an analogue of our optimistic descriptions to find optimal plans in the probabilistic setting. However, without pessimistic descriptions, they can only prove that a given high-level plan satisfies some property when the property holds for *all of its refinements*, which severely limits the amount of pruning possible compared to our approach. Helwig and Haddawy (1996) extended DRIPS to the online setting. Their algorithm did not cache backed-up values, and hence cannot guarantee eventual goal achievement, but it was probably the first principled online hierarchical lookahead agent.

Several other works have pursued similar goals to ours, but using *state abstraction* rather than HLAs. Holte *et al.* (1996) developed Hierarchical A*, which uses an automatically constructed hierarchy of state abstractions in which the results of optimal search at each level define an admissible heuristic for search at

15. Medium-difficulty instances are defined as those where some algorithm consistently solved them optimally given 5000 refinements per step, but neither algorithm consistently solved them optimally given fewer than 1000 refinements per step.

the next-lower level. Similarly, Bulitko *et al.* (2007) proposed the PR LRTS algorithm, a real-time algorithm in which a plan discovered at each level constrains the planning process at the next-lower level.

Finally, other works have considered adding pessimistic bounds to the A* (Berliner, 1979) and LRTA* (Ishida & Shimbo, 1996) algorithms, to help guide search and exploration as well as monitor convergence. These techniques may also be useful for our corresponding hierarchical algorithms.

8. Discussion

We have presented several new algorithms for hierarchical planning with promising theoretical and empirical properties. There are many interesting directions for future work, such as developing better representations for descriptions and valuations, automatically synthesizing descriptions from the hierarchy, and generalizing domain-independent techniques for automatic derivation of planning heuristics to the hierarchical setting. One might also consider extensions to partially ordered, probabilistic, and partially observable settings, and better online algorithms that, e.g., maintain more state across environment steps.

9. Acknowledgements

Bhaskara Marthi thanks Leslie Kaelbling and Tomas Lozano-Perez for useful discussions. This research was also supported by DARPA IPTO, contracts FA8750-05-2-0249 and FA8750-07-D-0185 (subcontract 03-000219).

References

Berliner, H. 1979. The B* Tree Search Algorithm: A Best-First Proof Procedure. *Artif. Intell.* 12:23–40.

Bulitko, V.; Sturtevant, N.; Lu, J.; and Yau, T. 2007. Graph Abstraction in Real-time Heuristic Search. *JAIR* 30:51–100.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artif. Intell.* 69:165–204.

Doan, A., and Haddawy, P. 1995. Decision-theoretic refinement planning: Principles and application. Technical Report TR-95-01-01, Univ. of Wisconsin-Milwaukee.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.* 2:189–208.

Haeupler, B.; Kavitha, T.; Mathew, R.; Sen, S.; and Tarjan, R. E. 2008. Faster algorithms for incremental topological ordering. In *ICALP*, 421–433. Berlin, Heidelberg: Springer-Verlag.

Helwig, J., and Haddawy, P. 1996. An Abstraction-Based Approach to Interleaving Planning and Execution in Partially-Observable Domains. In *AAAI Fall Symposium*.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI*.

Ishida, T., and Shimbo, M. 1996. Improving the learning efficiencies of realtime search. In *AAAI*.

Korf, R. E. 1990. Real-Time Heuristic Search. *Artif. Intell.* 42:189–211.

Marthi, B.; Russell, S. J.; and Wolfe, J. 2007. Angelic Semantics for High-Level Actions. In *ICAPS*.

Marthi, B.; Russell, S. J.; and Wolfe, J. 2008a. Angelic Hierarchical Planning: Optimal and Online Algorithms. Technical Report UCB/EECS-2008-150, EECS Department, University of California, Berkeley.

Marthi, B.; Russell, S. J.; and Wolfe, J. 2008b. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *ICAPS*.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, W. J.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

Parr, R., and Russell, S. 1998. Reinforcement Learning with Hierarchies of Machines. In *NIPS*.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

Tate, A. 1977. Generating project networks. In *IJCAI*.

Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Comput. Intell.* 6(1):12–24.