Faster Optimal Planning with Partial-Order Pruning

David Hall, Alon Cohen, David Burkett, and **Dan Klein**
EECS CS Division
University of California Berkeley
Berkeley, California 94720
{dlwh,dburkett,klein}@cs.berkeley.edu, aloni.cohen@gmail.com

Abstract

When planning problems have many kinds of resources or high concurrency, each optimal state has exponentially many minor variants, some of which are "better" than others. Standard methods like A* cannot effectively exploit these minor relative differences, and therefore must explore many redundant, clearly suboptimal plans. We describe a new optimal search algorithm for planning that leverages a partial order relation between states. Under suitable conditions, states that are dominated by other states with respect to this order can be pruned while provably maintaining optimality. We also describe a simple method for automatically discovering compatible partial orders in both serial and concurrent domains. In our experiments we find that more than 98% of search states can be pruned in some domains.

Introduction

Planning problems differ from other search problems in a number of ways. One important distinction is that there are typically many planning states that are "similar" along one or more dimensions. For instance, consider a job shop domain in which we are trying to assemble a set of products from a set of components. Here, one search state might have the same number of widgets—but fewer sprockets—than another. Assuming in this example that more is always better and that the two states can be reached at the same time, we can safely discard the latter, *dominated* state, instead focusing our search on the better state.

These kinds of similar, but clearly suboptimal, states are particularly common in concurrent domains, where the planner must make decisions about whether or not to expend all resources now, or wait some amount of time for a currently running action to finish. Because of the sheer number of decisions to be made, many states end up falling irreparably behind their optimal variants.

In this paper, we seek to formalize this notion, using Pareto dominance to prune states that are strictly dominated by some other state. More specifically, we give conditions under which we can expand only those states in the *skyline*, that is, states that are not dominated by any other state. Our system, Skyplan, is a refinement of Uniform Cost Search or A* (Hart, Nilsson, and Raphael, 1968) that expands only those states that are in the skyline. Because heuristics like those used in A* cannot directly compare two states to one another, Skyplan speeds up search in a way that is orthogonal to the traditional heuristics used in A*.

The central idea underlying our approach is to define a partial order relationship between states in the search space. This partial order has an intuitive interpretation: one state dominates another if it has no fewer "good" resources (e.g. job shop outputs) than another, no more "bad" resources (e.g. labor expended or time taken), and it is better in one or more ways.

In our analysis, we prove that our algorithm is both complete and optimal with a suitable partial order. Moreover, we prove that Skyplan is optimally efficient in the same sense as A*. That is, given a fixed partial order, there is no complete and optimal algorithm that can expand fewer search nodes (up to breaking ties).

In addition to proving the correctness of our approach, we also show how to automatically infer a compatible partial order from a problem specification such as PDDL (Ghallab et al., 1998; Fox and Long, 2003). This procedure is fairly intuitive: one simply needs to determine which resources or propositions are uniformly good or bad, and—for domains with durative actions—which actions have uniformly good or bad final effects.

In our experiments, we compare Skyplan to a similar implementation of A* in several standard domains. In addition, we introduce a new domain, based on the popular video game StarCraft. Skyplan performs especially well in this latter domain, cutting the branching factor by more than 90% compared to A*, and run times by more than a factor of 80.

Skyplan

In this work, we focus on planning with additive costs, using the conventional formulation as search within a weighted directed graph. A planning problem consists of a directed graph G with planning states n. States are linked by directed edges with an associated cost function $\text{cost}(n_s, n_t) \in \mathbb{R}^+$. There is also a privileged initial state s, along with a set of goal states F. We define $g(n)$ to be the cost of the shortest path from s to a state n. Our goal is to find a path from s to some state $n_f \in F$ with the lowest g cost.

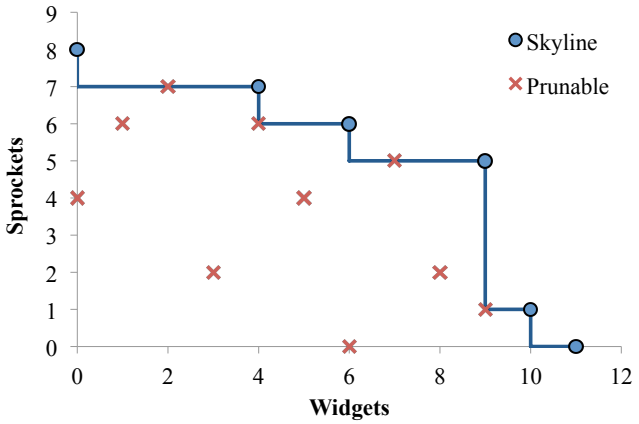

Figure 1: An example set of states in a two-resource planning domain. If all pictured states have the same cost, then only those shown as circles are worth exploring; the X's can be safely pruned.

Partial Orders

We will further assume that our graph is endowed with a partial order $\preceq$ that relates states to one another, with the intuitive semantics that $m \preceq n$ if n is at least as good as m in every way. For example, in our sprockets and widgets example, $m \preceq n$ if m has no more widgets and no more sprockets than n and if $g(m) \geq g(n)$. For any set of states N, we define the skyline of that set as skyline$(N) = \{m : \neg\exists n \in N : m \preceq n\}$.

For example, in Figure 1, we consider a domain with two resources, widgets and sprockets. If having more widgets and more sprockets is always better, then all states that are not at one of the upper-right corners of the skyline are strictly dominated by one or more of the states that are. Thus, if all states have the same cost, we can safely remove these dominated states (represented by X's) from search.

Our goal is to define an optimal search algorithm that can exploit this partial order to reduce the search space by only expanding states n that are *weakly Pareto optimal*. That is, we wish to expand only those states that are on the skyline. In the next section, we will define this algorithm, and in the subsequent sections, we will give a useful sufficient condition under which we can exploit a partial order while preserving correctness of optimal graph search algorithms.

We are not the first to suggest the use of Pareto optimality or skyline queries in the context of planning. For example, the non-optimal Metric-FF planner (Hoffmann, 2003) employed a more limited notion of dominance, requiring that all propositions be the same between two states, the domain to be monotonically "good" in all resources, and that the dominating state has no fewer resources along any axis. Our notion of dominance is more general. That is, we are able to find more powerful partial orders in more general domains.

In another satisficing planner, Röger and Helmert (2010) combined multiple heuristics by only expanding those states which were currently Pareto optimal with respect to the current open set. By contrast, we seek to *prune* search states while still maintaining optimality.

Still others have employed fairly different notions of partial orders while maintaining completeness and optimality.

Algorithm 1 Skyplan based on uniform cost search

```
1:  procedure UCSSkyplan(s, G, F, ⪯)
2:      Initialize Open = {⟨s, 0⟩}, Closed = {}
3:      while Open is not empty do
4:          Pop the min cost state m = ⟨m, c⟩ ∈ Open
5:          m → Closed
6:          if m ∈ F then
7:              return the path to m following back pointers
8:          end if
9:          if ∄n ∈ Open ∪ Closed, m ⪯ n then
10:             for m′ ∈ succ(m) do
11:                 m′ ← ⟨m′, c + cost(m, m′)⟩
12:                 m′ → Open
13:             end for
14:         end if
15:     end while
16:     fail
17: end procedure
```

These methods largely fall into a category known as "partial order reduction techniques," which have their origins in computer aided verification. This class includes expansion cores (Chen and Yao, 2009; Xu et al., 2011), which select only "relevant" actions for any given state, commutativity pruning (Geffner and Haslum, 2000), which prunes redundant permutations of partial plans, and stratified planning (Chen, Xu, and Yao, 2009), which identifies layers of the planning problem and only considers actions appropriate to a given layer. Wehrle and Helmert (2012) recently showed that all of these techniques are special cases of more general techniques from computer aided verification, namely sleep sets (Godefroid, 1996) and stubborn sets (Valmari, 1992). These methods are largely orthogonal to our approach; indeed, we use a version of expansion cores in our system.

Algorithm

Skyplan is a fairly easy modification of a standard optimal search algorithm like Uniform Cost Search or A*. For the sake of exposition, we focus on Uniform Cost Search, though A* or any optimal graph search algorithm can be modified in the same way.[1]

Skyplan is defined in Algorithm 1. Essentially, we run Uniform Cost Search as normal, except that we only expand states that are not strictly dominated by another state that we have either expanded or enqueued for expansion. That is, we only expand nodes that are in the skyline of the nodes we know about so far.

Compatibility

Not just any partial order on states can be used to preserve optimality. As a perverse example, we can define a partial order under which all states on an optimal path are dominated by non-optimal states. Thus, we need to define a property under which a partial order is *compatible* with a search

[1] In our experiments we use either A* or Hierarchical A* (Holte et al., 1996).

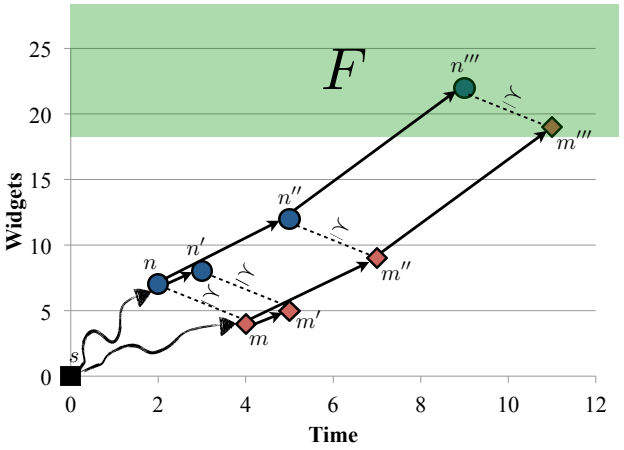

Figure 2: Abstract representation of search with compatible partial orders. The shaded area represents the goal region. Both m and n are reachable from the start state. Because n is a better state than m, and n's successors "stay ahead" of m's, we can safely avoid expanding m while still preserving optimality.

graph. While a broad class of partial orders might work, we have identified a sufficient condition that is especially applicable to planning problems. Figure 2 illustrates compatibility for a simple graph. If our goal is to accumulate at least 18 widgets, then n is clearly better than m as it has more widgets in less time. Moreover, none of m's successors reach a point better than all of n's successors. Because paths through n continue to "stay ahead" of those passing through m, we can safely prune m.

We formalize this intuition as follows:

Definition 1 (Compatibility)**.** A partial order $\preceq$ is *compatible* if for all states $m \preceq n$,

1. $g(n) \leq g(m)$, and
2. $\forall m' \in \text{succ}(m)\ \exists n' \in \text{succ}(n)$ such that $m' \preceq n'$ and $\text{cost}(n, n') \leq \text{cost}(m, m')$, and
3. If $m \in F$, then $n \in F$

This recursive definition of compatibility essentially means that if a state m is dominated by a state n, then it must be the case that m is no cheaper than n and that n's successors "stay ahead" of m's. Finally, F must be closed with respect to $\preceq$: every state that dominates a goal state must also be a goal. It is worth mentioning that there is always a compatible partial ordering: the trivial order with $m \not\preceq n$ for all states m and n with $m \neq n$.

In planning problems, defining a compatible partial order is usually quite easy, so long as cost functions are additive. Actions are often defined in terms of partial states with constant costs (e.g. time elapsed), and so compatibility is easily checked. After proving the correctness and optimality of Skyplan, we discuss a standard structure for these partial orders, as well as how to automatically infer a compatible partial order from the specification of a planning problem.

Analysis

In this section, we show that Skyplan is complete, optimal, and optimally efficient for any compatible partial order when used in conjunction with an optimal search algorithm like Uniform Cost Search or A*. Before proving the main results of this section, we prove a useful lemma.

Lemma 1. *Let p be a path to a goal, let m be one state in p, and let $p_{m:}$ be the suffix of the path starting from m, and $|p|$ be the length of p in terms of the number of states. If n is a state with $m \preceq n$, then there is a corresponding path q that reaches a goal state through n with $|q_{n:}| \leq |p_{m:}|$ and* $\text{cost}(q) \leq \text{cost}(p)$.

Proof. $p_{m:} = (m, m_1, \ldots, m_k)$ with m_k a goal state. By condition (2) of compatibility, $\exists n_1 \in \text{succ}(n)$ with $n_1 \succeq m_1$ and $\text{cost}(n, n_1) \leq \text{cost}(m, m_1)$. Similarly, $\exists n_2, \ldots, n_k$ with $n_i \succeq m_i$ and $n_{i+1} \in \text{succ}(n_i)$ with total cost $\leq \text{cost}(p_{m:})$. By condition (3) of compatibility, $n_k \in F$. By condition (1), there is a path from s to n with $g(n) \leq g(m)$. The above argument extends this path to the goal state n_k and completes the proof of the lemma. □

This lemma states that if $m \preceq n$, then there is a path from n to a goal state that is no longer and no more costly than any such path from m. While we usually consider states being pruned by other states, the lemma allows us to equivalently reason about paths being pruned by other paths. In the following proof, we use these notions interchangeably.

Claim 1. *Completeness. If $\preceq$ is a compatible partial order, Skyplan is* complete.

Proof. Suppose otherwise, then all paths to all goals must have been pruned. This would require a cycle of paths pruning each other: paths $p^1, \ldots, p^k$ such that $p^{(i+1) \mod k}$ pruned p^i. For all paths, let $n_{(i+1) \mod k} \in p^{(i+1) \mod k}$ be the state which pruned p^i and $m_i \in p^i$ the corresponding state which was pruned ($\forall i : m_i \preceq n_{(i+1) \mod k}$).

Because each m_i was pruned before being expanded, $n_i \neq m_i$ nor any state succeeding m_i in $p^i_{m_i:}$. Hence $p^i_{n_i:} \supseteq p^i_{m_i:} \implies |p^i_{n_i:}| > |p^i_{m_i:}|$. By the lemma, we also have $|p^i_{m_i:}| \geq |p^{(i+1) \mod k}_{n_{(i+1) \mod k}:}|$. Combining the inequalities yields: $|p^1_{m_1:}| \geq |p^2_{n_2:}| > |p^2_{m_2:}| > \cdots > |p^k_{m_k:}| > |p^1_{m_1:}|$, a contradiction. □

Claim 2. *Optimality. If $\preceq$ is a compatible partial order, Skyplan is* optimal.

Proof. Let p^* be the last optimal path that was pruned during a run of Skyplan. By the proof of Claim 1 and the lemma there is some path p' that is not pruned with $\text{cost}(p') \leq \text{cost}(p^*)$. Thus, p' is optimal as well. □

By leveraging the information contained in the partial order, Skyplan is able to achieve optimal plans while expanding fewer states in general. But is Skyplan using this added information in the best possible manner? It is easy to see that any state expanded by Skyplan must not be dominated by any other state in the graph: otherwise $Closed$ would contain such a dominating state when the about-to-be-expanded state was popped, thereby pruning it. This turns out to be enough to prove the following:

Claim 3. *Optimal Efficiency. For any compatible partial order $\preceq$, Skyplan is* optimally efficient.

Proof. (We prove the claim on the set of graphs for which all paths have unique costs: $n \neq m \implies g(n) \neq g(m)$. This proof is similar to the analogous proof for A* in Hart, Nilsson, and Raphael (1968). The generalization to graphs with ties is also analogous, and we omit it in the interest of space.)

Suppose otherwise. Then, there must exist an optimal algorithm B and a search problem $(G, \preceq)$ on which B expands (calls the successor function on) strictly fewer states than Skyplan. Let n^* be a state expanded in Skyplan but not by B on the problem $(G, \preceq)$, and let f_B be the goal state returned by B. We construct a new graph G' by adding a single goal state f^* to G and an associated edge from n^* to f^* of cost ϵ (with $\epsilon < g(f_B) - g(n^*)$). On G' we apply the original partial order $\preceq$ with $n \not\preceq m$ if either n or m is f^*.

We must show that $\preceq$ is compatible on G'. Let $m, n \in G'$ such that $m \preceq n$. By construction, neither m nor n is f^*. Conditions (1) and (3) are clearly true in G' because they are true in G. Furthermore, if $m \neq n^*$ condition (2) is true because $\text{succ}_{G'}(m) = \text{succ}_G(m)$. But as we observed above, no state expanded in Skyplan is dominated by any other state. Thus, $m \neq n^*$.

Except for the fact that n^* has an additional successor, G' is otherwise identical to G. By assumption, B never examined the successors of n^*, and must therefore follow the same execution path on the graph G', returning the goal state f_B. But by construction, f^* is a lower-cost goal in G', contradicting the assumed optimality of B. □

Inferring Partial Orders

Now that we have a definition of compatible partial orders and an algorithm that can take advantage of them, all we need is a method for finding them. Of course, one can construct compatible partial orders by hand using domain knowledge, but that is not always desirable. In this section, we show how to automatically infer a compatible partial order from a sequential planning problem domain's specification, such as those available in PDDL (Ghallab et al., 1998; Fox and Long, 2003). In the next section, we will extend this procedure to a broad class of concurrent domains with durative actions.

Planning States

When planning with resources, a basic planning state n can be viewed as a mapping from resource types to values. That is, for a given resource r, $n(r) \in \mathbb{R}$ is the quantity of that resource available at state n. Most planning state representations also require some notion of boolean propositions, but in order to simplify the exposition, for the purposes of defining a partial order we view these as special resources that always take values in $\{0, 1\}$.

Also note that for planning problems, the *cost* of a state is just a resource like any other. For example, in many domains, the cost is equivalent to time, which most actions increment as one of their effects. Thus, though we have been so far using $g(n)$ to refer to the cost of the *optimal* path to a state, in this section we will abuse notation slightly and treat $g(n)$ as an observable property of the state itself, with the understanding that a path with a different cost would have reached a different state.

Basic Partial Orders

If we want to determine whether a state n dominates m, then we need to know two things: first, does n have lower cost; and second, does n have a more "useful" set of resources? The first criterion is easy to check, but the second requires more careful consideration of what it means for a resource to be useful or not.

There are basically two ways in which the value of a particular resource can matter: it can either contribute to satisfying the goal test $n \in F$, or it can contribute to satisfying the precondition of some action that might get us closer to the goal. In both cases, we are concerned with the relationship between the value of a particular resource r, and a particular boolean predicate b. In general, we want to know when it helps to have more (or less) of a particular resource, leading us to the following classification of predicates based on how they interact with a given resource. Let $b(n)$ be the value of the predicate b in state n.

Definition 2 (r-Dependent)**.** A predicate b is *r-dependent* if $\exists n, n'$ with $b(n) =$ true and $b(n') =$ false, but where n and n' are identical except that $n(r) \neq n'(r)$.

Definition 3 (r-Positive)**.** A predicate b is *r-positive* if b is r-dependent and $\forall n, n'$ identical except $n(r) < n'(r)$, $b(n) \Rightarrow b(n')$.

Note that this additionally implies the contrapositive: $\forall n, n'$ identical except $n(r) > n'(r)$, $\neg b(n) \Rightarrow \neg b(n')$

Definition 4 (r-Negative)**.** A predicate b is *r-negative* if b is r-dependent and $\forall n, n'$ identical except $n(r) > n'(r)$, $b(n) \Rightarrow b(n')$.

For example, the predicate "$n(r) \geq 1$" is r-positive, the predicate "$n(r) \leq 0$" is r-negative, the predicate "$n(r_1) \geq n(r_2)$" is r_1-positive, but r_2-negative, and the predicate "$1 \leq n(r) \leq 2$" is r-dependent, but neither r-positive nor r-negative.

Using these definitions, we can construct a partial order based on three parts of the problem definition:

1. The set of (grounded, where applicable) resources.
2. The set of possible (grounded) actions, particularly their preconditions.
3. The goal test: $n \in F$.

Let B be the set of all relevant predicates, that is, the set of preconditions for each action plus the goal test. We partition all resources into four disjoint sets: R^+, R^-, $R^=$, and $R^\emptyset$, according to the following criteria:

- $r \in R^+$ if $\forall b \in B, b$ is r-dependent implies that b is r-positive, and $\exists$ at least one $b \in B$ that is r-positive.
- $r \in R^-$ if $\forall b \in B, b$ is r-dependent implies that b is r-negative, and $\exists$ at least one $b \in B$ that is r-negative.

- $r \in R^=$ if $\exists b \in B$ that is r-dependent but not r-positive or r-negative, *or* if $\exists b_1, b_2 \in B$ with b_1 r-positive and b_2 r-negative.
- $r \in R^\emptyset$ if $\forall b \in B, b$ is *not* r-dependent.

These sets are fairly intuitive. For instance, resources in R^+ are better the more you have of them (e.g. widgets), while resources in $R^=$ have a complex relationship with the problem, and so it is not safe to prune a state n unless it has the exact same value as its potential dominator for that resource. $R^\emptyset$ consists of those resources that have no effect on any condition in the problem. With these sets in mind, we can define the partial order $\preceq_R$:

Definition 5 ($\preceq_R$)**.** For planning states n and m, we say that $m \preceq_R n$ if and only if:

1. $g(m) \geq g(n)$,
2. $\forall r \in R^+, m(r) \leq n(r)$,
3. $\forall r \in R^-, m(r) \geq n(r)$, and
4. $\forall r \in R^=, m(r) = n(r)$

Claim 4. *If action costs and effects do not depend on the current state, then $\preceq_R$ is compatible.*

Proof. Condition 1 of Definition 1 follows directly from condition 1 of Definition 5.

To verify condition 3 of Definition 1, we only need to consider resources r for which the goal test is r-dependent, as changes in other resources will not affect the outcome of the test. If the goal test is r-positive, then $r \in R^+$ or $r \in R^=$, so $m(r) \leq n(r)$. If the goal test is r-negative, then $r \in R^-$ or $r \in R^=$, so $m(r) \geq n(r)$. If the goal test is neither (but is r-dependent), then $r \in R^=$, so $m(r) = n(r)$. Taken together, these conditions ensure that $n \in F \Rightarrow m \in F$.

Finally, to verify condition 2 of Definition 1, observe that some particular action results in the transition from n to n'. By the argument above, if the precondition of this action holds for m, then it also holds for n. Let n' be the result of taking this action from n. By assumption, the cost is identical and the change in resources from n to n' is also identical to that from m to m'. Thus, for all $r, n'(r) - m'(r) = n(r) - m(r)$. Therefore, $m' \preceq_R n'$. □

A few remarks are in order with regard to this inferred ordering. First, there are occasionally resources that are usually beneficial, but they might have some high upper limit. For instance, we might like to have as many widgets as possible, except that we have only a large but finite amount of storage space for them. In this case, we would not be able to prune based on the number of widgets: its resource would be in $R^=$, because there are in some sense two resources: "widgets" and "free space for widgets." However, if the maximum number of widgets is indeed so high that we suspect that we will not reach it in search, we can treat it as though there was no limit. If we do hit the limit, we then have to reenqueue all nodes pruned based on widget count, and back off to the safer partial order.

Second, it may not be the best ordering for a given problem. Consider a domain in which we can use sprockets to make a transmogrifier, but not widgets. However, suppose there is a free conversion from widgets to sprockets, and vice versa—because sprockets are in fact just upside-down widgets. In this case, sprockets and widgets are interchangeable, and so we can actually prune based on the sum of their count. However, the method just described is incapable of realizing this relationship. In general, it is intractable to find the optimal partial ordering. That said, we suspect that such perverse examples are uncommon.

Partial Orders and Concurrency

In classical planning domains, constructing a search graph is straightforward, with transitions corresponding directly to the simple actions available to the agent. However, in planning domains that permit multiple simultaneous actions, the state space and the search graph need to be augmented slightly for planning to still work properly as a search problem. In this work, we use the concurrent planning framework described in Bacchus and Ady (2001). First, we briefly describe this representation, and then describe the extension of the partial order inference to concurrent domains.

Modeling Concurrency

The main modification is that each action no longer necessarily makes only atomic modifications to a state. Instead, action effects now have three parts: effects that take place immediately upon beginning the action, the duration required before the action completes, and effects that take place when the action ends. In order to keep track of actions whose completion effects haven't occurred yet, basic states are augmented with an *action queue*. An action queue is a multiset of (a, t) pairs, where a is an in-progress action, and t is the length of time until that action completes.

The search graph also needs to be modified to handle in-progress actions. First, in this work, we assume that the cost function used in concurrent planning domains is $C_a + \lambda T$, where C_a is the accumulated cost of all basic actions taken, T is the total amount of time elapsed, and $\lambda > 0$ is a scaling constant. The search graph still contains transitions for all basic actions whose preconditions are satisfied, each with the same cost. However, the construction of the ending state n' is different: when performing action a, only the *immediate* effects are applied, and then the pair (a, d_a) is added to the action queue, where d_a is the duration of a. In addition to the transitions for basic actions, we also need a set of special transitions $\mathrm{elapse\_time}(k)$. The cost of this transition is λk, and the effects are:

1. For every pair (a, t) on the action queue with $t \leq k$, the *completion* effects of a are applied.
2. All pairs (a, t) with $t \leq k$ are removed from the action queue.
3. All pairs (a, t) with $t > k$ are replaced with $(a, t - k)$.

Inferring Partial Orders with Action Queues

In concurrent domains, in addition to a collection of resources, the state contains a queue of in-progress actions. Here, we'll construct a new partial order $\preceq_Q$ that takes the

action queue into account. To simplify the exposition, we assume that all actions have *beneficial* completion effects. Formally, this means that any resource whose value is increased at completion is in R^+ or $R^\emptyset$, and any resource whose value is decreased is in R^- or $R^\emptyset$. The extension to more general action effects is straightforward.

Intuitively, a state n can only be better than a state m if its action queue finishes faster than m's, or if n already has more resources than m will have. We formalize this intuition as follows:

Definition 6 ($\preceq_Q$)**.** Let $q(m)$ denote the action queue for state m. For planning states m and n, we say that $m \preceq_Q n$ if and only if $m \preceq_R n$, and also $\forall (a,t) \in q(m)$, at least one of the following conditions holds:

1. $\exists$ a distinct $(a,t') \in q(n)$ with $t' \leq t$.
2. $\forall r$ affected by the completion effects of a, the change from $m(r)$ to $n(r)$ is at least as big as that produced by all actions currently enqueued that are not satisfied by condition 1.

The notion of a *distinct* pair (a,t') is important, as the same action can appear multiple times in the same action queue. For example, let a be an action whose completion effect is "Gain 5 r". If $q(m) = \{(a,10),(a,20)\}$ then $q(n) = \{(a,10),(a,15)\}$ meets this condition, but $q(n) = \{(a,5)\}$ does not, unless $n(r) \geq m(r) + 5$. Similarly, $q(n) = \{\}$ does not meet the condition unless $n(r) \geq m(r) + 10$.

Claim 5. *If action costs and effects do not depend on the state, and all action completion effects are beneficial, then $\preceq_Q$ is compatible.*

Proof. The proof of Claim 4 already covers conditions 1 and 3 of Definition 1. There are two types of transitions we need to handle for condition 2: basic action transitions, and $\mathrm{elapse\_time}$ transitions. Basic actions are also mostly covered by the proof of Claim 4; the only difference is that the action queues are modified. $q(m)$ and $q(m')$ differ only in the addition of (a,d_a), and likewise for $q(n)$ and $q(n')$. This new addition is covered by the first case in Definition 6, and $m \preceq_Q n$ implies that the rest of $q(m')$ is matched by n', so we also have $m' \preceq_Q n'$.

As for the $\mathrm{elapse\_time}$ transitions, assume that m' is reached by following $\mathrm{elapse\_time}(k)$ from m. If we follow $\mathrm{elapse\_time}(k)$ from n, we incur the same cost and n' has the following two properties. First, for every $(a,t) \in q(m)$ with $t \leq k$ that is completed as part of the transition, either:

1. n already had a corresponding resource advantage over n and thus m' does not overtake n', or
2. $\exists (a,t') \in q(n)$ with $t' \leq t \leq k$, and so this action completes and n' gets a corresponding resource improvement.

Second, for every pair $(a,t) \in q(n)$ with $t > k$ there is now a pair $(a, t-k)$ in $q(n')$. But in m', one of three conditions holds:

1. n already had a corresponding resource advantage over m, so n' preserves this advantage over m', or
2. $\exists (a,t') \in q(n)$ with $k < t' \leq t$, so $(a, t'-k)$ is in $q(n')$, or
3. $\exists (a,t') \in q(n)$ with $t' \leq k \leq t$, so n' has gained the necessary resource advantage over m'.

Taken altogether, we once again get $m' \preceq_Q n'$.

□

Skyplan in Practice

Stepping back, the basic outline of our system is fairly straightforward. Given a planning problem, Skyplan infers the partial order and we then use that partial order in a standard graph search.

There are a few pragmatic details of our system that can substantially affect performance that are worth mentioning. One is our choice of Hierarchical A* (Holte et al., 1996) as base search algorithm. The other is the somewhat more mundane question of how best to implement a skyline query for our setting. In this section, we discuss each in turn.

Hierarchical A*

While skyline pruning works with any optimal search procedure, this pruning method has a particular synergy with the Hierarchical A* algorithm. Hierarchical A* works by identifying a series of relaxations to the search space, and uses each level of relaxation as the (admissible and consistent) heuristic for the corresponding lower level.

One particularly easy way to apply this algorithm to planning is to define the relaxations only in terms of relaxations to the goal test. That is, one can simply take a subset of the goal conditions at each level without any further relaxations of the state space.

If we use this series of projections, we can actually maintain a *single* skyline for all levels of the hierarchy so long as we use the partial order $\preceq_R$ defined on the original search problem (that is, with the full goal test). Thus, when expanding a state, we can prune any of its successors if they are dominated by a state at *any* level of the hierarchy. In particular, the combination of Hierarchical A* and skyline pruning means that the projected searches give us two sources of information: how far a state is from a (partial) goal, and which states are not possibly on the optimal path to the full goal.

Implementing Skyline Pruning

The implementation of the skyline is somewhat delicate. There is some work in this area in the database literature, but these are not designed for online applications (Börzsönyi, Kossmann, and Stocker, 2001; Tan, Eng, and Ooi, 2001) or are for fairly low dimensional states (Kossmann, Ramsak, and Rost, 2002), and so they are not useful here.

The obvious, naïve solution is to maintain a list of all elements in the current skyline, and to check all of these elements when deciding whether or not to add an element to the skyline. Unfortunately, this check is clearly linear in the number of states (which means we must make quadratically many checks). In practice, this runtime can overwhelm the benefit obtained from pruning the branching factor, especially if the number of states pruned is small.

However, note that in high dimensions most states are clearly not related to one another. They might both have different positive propositions, with one not having strictly

Problem	Nodes Popped		Time (s)	
	HA*	Skyplan	HA*	Skyplan
Pegsol 1	25	25	1.11	1.09
Pegsol 2	246	170	10.26	6.86
Pegsol 3	670	435	28.0	17.4
Pegsol 4	592	587	37.2	36.3
Pegsol 5	1553	636	86.4	31.2
Pegsol 6	5695	2450	312	128
Pegsol 7	589	328	31.3	16.8
Pegsol 8	199,361	113,848	12,813	7271
Pegsol 9	48,668	15,268	3750	1179
Pegsol 10	272,112	197,165	21,108	15,138
Openstacks 1	205	205	0.061	0.064
Openstacks 2	588	588	0.320	0.320
Openstacks 3	2022	2022	1.37	1.31
Openstacks 4	53,630	53,630	46.7	51.3
Openstacks 5	201,437	201,437	259	288

Table 1: Results on sequential domains. Times are averaged over 10 runs, except for Pegsol 10, which was averaged over 5 runs.

more than another. Therefore, we can quickly do a "coarse" version of a skyline query by having a list of sets, one for each resource or proposition in R^+ (which in practice includes most resources), that contains all states in the skyline that have a positive value for that resource. Then, when deciding whether or not a new state is dominated, we can simply intersect these sets, and then run the check on those states in the intersection. Because most states in practice are dominated by very few (or sometimes no) states, this intersection is much smaller than the entire skyline. So, while this optimization does not decrease the asymptotic runtime, in practice it is several orders of magnitude faster.

Experiments

We empirically evaluated Skyplan in two settings: a collection of standard domains from the International Planning Competitions, and a new domain based on the video game StarCraft. In the first setting, we note that the competitions have not focused on optimal planning in time-sensitive domains with concurrency. Instead, the goals for problems in these competitions are either to find the optimal plan in a small domain without concurrency, or to find a good, satisficing plan in larger domains with concurrency.[2] We conduct experiments from domains of both types.

These standard domains are largely concerned with discrete objects rather than ever-growing resources. Therefore, even the concurrent domains require a still rather modest amount of concurrency, with even the most complex problems never permitting more than a dozen simultaneous actions. Nevertheless, Skyplan can perform quite well in some of these domains, while in others it does not perform as well.

Skyplan performs best in domains with a large amount of concurrency, much more than is available in most standard planning domains. Therefore, we also ran experiments in a more realistic resource acquisition setting modeled on the video game StarCraft. Because this domain involves manipulating large numbers of resources at the same time, Skyplan's strengths will be more readily apparent.

[2]Still other configurations exist, but these are further afield from our current interest.

Problem	Nodes Popped		Time (s)	
	HA*	Skyplan	HA*	Skyplan
Pegsol 1	29	29	0.697	0.643
Pegsol 2	320	244	5.92	4.65
Pegsol 3	983	694	19.0	13.5
Pegsol 4	6133	1766	176.7	51.8
Pegsol 5	2178	1147	51.5	25.1
Pegsol 6	25,171	10,448	727	287
Woodworking 1	722	427	0.103	0.051
Woodworking 2	5030	2838	0.527	0.431
Woodworking 3	19,279	11,395	1.99	1.96
Woodworking 4	272,863	132,487	51.1	49.9
Woodworking 5	2,626,520	1,312,511	588	2044
Openstacks 1	3,872,619	70,644	437.2	23.4

Table 2: Results on temporal satisficing domains. Times are averaged over 10 runs.

We implemented two versions of Skyplan and the relevant baselines, one for the generic PDDL-based domains, and one specialized to the Starcraft domain.[3] The two Skyplan and A* implementations differ primarily in the heuristics used. Our generic implementation uses a hierarchical A* implementation (with skyline pruning), with no heuristic at the base level. In StarCraft, we use a special heuristic crafted to this domain. The heuristic is essentially a critical path heuristic that ignores resource costs, taking into account only the time to complete actions. Both implementations use a variant of expansion cores (Chen and Yao, 2009; Xu et al., 2011) to prune irrelevant actions.

Finally, we note that no available open source planning system can handle both temporal problems and problems with resources. As these properties are key motivations for our approach, we instead constructed our own baselines. The most obvious baseline is to use A* without skyline pruning. In addition, for Starcraft we compare to a non-optimal greedy baseline. This planner determines which structures and units are required to reach the goal and builds the most costly structures and units first, taking dependencies into account.

The IPC Domains

We evaluated Skyplan on problems from three different domains that were released with the 2008 International Planning Competition: Pegsol (peg solitaire), Woodworking, and Openstacks. Of these, Pegsol and Openstacks each appeared in both sequential and temporal versions, and we evaluated on both. These particular domains were chosen because the initial problems from each domain were simple enough to be solved by basic optimal planners. For all problems, we measured the number of nodes popped (including those popped during the recursive searches used for heuristic computation) and the total time taken by each algorithm.[4] The results for the sequential variants are shown in Table **??**, and those for the temporal ones are in Table **??**.

[3]Source for the generic implementation is available at `http://overmind.cs.berkeley.edu/skyplan`.

[4]All these experiments were run on isolated 2.67GHz Intel Xeon processors.

Planner	7 SCVs			1 Marine			1 Firebat			1 Marine, 1 Firebat			2 Command Centers		
	Nodes	Time	Cost	Nodes	Time	Cost	Nodes	Time	Cost	Nodes	Time	Cost	Nodes	Time	Cost
A*	3.1k	0.27	1.0	11.7k	1.34	1.0	>809k	>1800	*	>655k	>1800	*	>768k	>1800	*
Skyplan	233	0.12	1.0	405	0.22	1.0	97k	21.9	1.0	509k	393	1.0	26k	209	1.0
Greedy		0.03	1.0		0.06	3.3		0.06	2.4		0.09	2.4		0.06	2.1

Table 3: Results on the StarCraft domain. Times are in seconds. Asterisks indicate that A* timed out after more than 30 minutes of execution.

Skyplan is typically faster than Hierarchical A*, pruning as many as 98% of the nodes that A* expands, though the overhead of maintaining the skyline means that for this problem (the first temporal Openstacks problem), expanding fewer than 1/50th of the nodes results in a slightly less than 20x speedup. The Pegsol domains also showed consistent improvement, usually expanding fewer than half the nodes, and achieving commensurate speedups, particularly in the temporal variant. Results from the sequential Openstacks domain demonstrate that when the partial order fails to successfully prune many (or any) nodes, Skyplan's overhead eventually results in a slowdown vs A*. The overhead is superlinear with respect to the size of the fringe, as exemplified in the Woodworking domain, where Skyplan's modest initial success gives way to the result in problem 5 where, due to the extremely high number of nodes, Skyplan's 50% pruning rate is inadequate to combat the overhead, and the net effect of using Skyplan is an almost 4x slowdown.

The StarCraft Domain

StarCraft® is a real time strategy game created by Blizzard Entertainment, and one of the most competitive and popular games in the world. The game involves the accumulation of resources (minerals and gas), which are gathered by workers. These resources are then spent to build structures (e.g. a barracks), which are then in turn used to build military units. The ultimate goal is to build an army to defeat an opponent. Skyplan is designed to take advantage of this sort of structure, suggesting that it will perform well in this environment.

There are many aspects of this game that are worth modeling from an artificial intelligence perspective in general and from planning in particular. The game lends itself to partial observability, planning against an adversary, highly concurrent plans, and online planning. Indeed, Churchill and Buro (2011) recently described an online planning system designed for use in an artificial intelligence agent, and Chan et al. (2007) earlier developed an online planner for a similar open source game called Wargus.

Here, we focus instead on planning opening build orders. Much like Chess, there are a large number of different openings, some of which are known to be better than others. In StarCraft, openings can be either more aggressive or more "economic" (i.e. aimed at producing lots of resources), and typically are designed to get a specific composition of buildings and units as quickly as possible. In practice, the first several minutes of the game are played largely independent of what the opponent is doing, unless certain surprise "cheese" openings are observed. Moreover, shaving seconds off an opening can be the difference between an easy early win and a loss. Thus, coming up with an optimal build order for a given composition is useful, if it is feasible.

One crucial aspect of StarCraft is that workers are both the resource gatherers and the builders of buildings. Combined with StarCraft being a "real time" game, this property results in an extremely high branching factor in the game tree. However, many of the possible combinations are clearly not optimal: it is usually (but not always) a bad idea to have your workers remain idle. This aspect of the game results in a large number of highly similar states, most of which are worse than others.

Our specification of StarCraft differs from that of Churchill and Buro (2011) precisely in how we handle gathering resources. For simplicity, they assume a linearization of resource gathering, with resources gathered at a continuous rate. However, resources are actually gathered in discrete amounts. By more accurately modeling resource collection, we can find more accurate plans, particularly in the early game, where seconds are so critical.

Results in the StarCraft Domain We selected a number of early game units and resource combinations as goals for our planners. We compared Skyplan against an A* planner and our nonoptimal Greedy planner, measuring plan length and total planning time taken by each algorithm, and the number of nodes expanded in Skyplan and A*.

The results are in Table 2. As mentioned above, the nature of the domain allows for aggressive pruning, reducing the branching factor by as much as 90%. This results in dramatic speedups over A* on all problems tested. In each experiment, Skyplan expands fewer than 10% of the nodes that A* expands, expanding as few as 4% as many nodes. In the simplest problems, Skyplan takes half as long as A*; on the most complex problems, Skyplan takes less than 2% the time of A*. In every case, the Greedy planner discovers a satisficing plan in less than 100ms, but these plans cost up to three times more than optimal.

Conclusion

Planning in highly concurrent domains can be particularly challenging for standard search-based planners, but by restricting attention to only those states that could conceivably be part of an optimal plan, Skyplan can efficiently solve problems that A* alone could not solve. Furthermore, since geometrically pruning the state space achieves speedups in a way that is orthogonal to heuristic domain knowledge, this gain should stack well with improved A* approaches. There is overhead introduced by the optimality-preserving skyline, but further innovations in data structures for state space pruning could obviate this problem, perhaps trading guaranteed optimality for improvements in asymptotic complexity.

References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *IJCAI*, volume 17, 417–424.

Börzsönyi, S.; Kossmann, D.; and Stocker, K. 2001. The skyline operator. In *ICDE*, 421–430.

Chan, H.; Fern, A.; Ray, S.; Wilson, N.; and Ventura, C. 2007. Online planning for resource production in real-time strategy games. In *ICAPS*, 65–72.

Chen, Y., and Yao, G. 2009. Completeness and optimality preserving reduction for planning. In *IJCAI*, 1659–1664.

Chen, Y.; Xu, Y.; and Yao, G. 2009. Stratified planning. In *International Joint Conference on Artificial Intelligence*.

Churchill, D., and Buro, M. 2011. Build order optimization in StarCraft. In *AIIDE*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Geffner, P., and Haslum, P. 2000. Admissible heuristics for optimal planning. In *AI Planning Systems*, 140–149.

Ghallab, M.; Aeronautiques, C.; Isi, C.; Wilkins, D.; et al. 1998. PDDL—the planning domain definition language. Technical report.

Godefroid, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics* SSC-4(2):100–107.

Hoffmann, J. 2003. Metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *Journal Of Artificial Intelligence Research* 20.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *National Conference on Artificial Intelligence*, 530–535.

Kossmann, D.; Ramsak, F.; and Rost, S. 2002. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, 275–286.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. *Alternation* 10(100s):1000s.

Tan, K.-L.; Eng, P.-K.; and Ooi, B. C. 2001. Efficient progressive skyline computation. In *VLDB*, 301–310.

Valmari, A. 1992. A stubborn attack on state explosion. *Formal Methods in System Design* 1:297–322.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In *ICAPS*.

Xu, Y.; Chen, Y.; Lu, Q.; and Huang, R. 2011. Theory and algorithms for partial order based reduction in planning. *CoRR* abs/1106.5427.