Completeness and Optimality Preserving Reduction for Planning

Yixin Chen
Department of Computer Science
Washington University
chen@cse.wustl.edu

Guohui Yao *
Department of Computer Science
Washington University
yaog@cse.wustl.edu

Abstract

Traditional AI search methods search in a state space typically modelled as a directed graph. Prohibitively large sizes of state space graphs make complete or optimal search expensive.

A key observation, as exemplified by the SAS+ formalism for planning, is that most commonly a state-space graph can be decomposed into subgraphs, linked by constraints. We propose a novel space reduction algorithm that exploits such structure. The result reveals that standard search algorithms may explore many redundant paths. Our method provides an automatic way to remove such redundancy. At each state, we expand only the subgraphs within a dependency closure satisfying certain sufficient conditions instead of all the subgraphs. Theoretically we prove that the proposed algorithm is completeness-preserving as well as optimality-preserving. We show that our reduction method can significantly reduce the search cost on a collection of planning domains.

1 Introduction

State-space search is a fundamental and pervasive approach for AI. The state space is typically modelled as a directed graph. A cost can be associated with each edge in the graph and the search tries to identify the path with the minimum total cost from an initial state to a goal state.

A key observation that motivates this paper is that most often the state-space is not a random graph. Rather, in most domains, the state space graph can be viewed as part of the Cartesian product of multiple smaller subgraphs. In SAS+ formalism [Bäckström & Nebel, 1995; Jonsson & Bäckström, 1998] of planning, a state is represented by the assignments to a set of multi-valued variables, and the state-space graph is derived from the Cartesian product of the domain transition graphs (DTGs), one for each variable.

The key question is, *if a state-space graph can be decomposed as the Cartesian product of subgraphs, can we make the search faster by exploiting such structure?*

Given the large size of the search space, exploiting decomposition of planning domains has long been of interests to planning researchers. There are several lines of existing work:

1) The automated hierarchical planning [Knoblock, 1994; Lansky & Getoor, 1995] methods utilize hierarchical factoring of planning domains, but they typically do not scale well since they require extensive backtracking across subdomains.

2) To avoid the cost of backtracking, the factored planning approach [Amir & Engelhardt, 2003; Brafman & Domshlak, 2006] finds all the subplans for each subproblem before merging some of them into one solution plan. However, as enumerating all subplans requires huge memory and time complexity, the practicability of factored planning has not been fully established [Kelareva *et al.*, 2007]. Our own empirical results suggest that the cost of factored planning can be exceedingly high. Also, the iterative deepening scheme that increments the length of the subplans poses additional complexity and compromises optimality.

3) Decomposition of planning domains has also been used to derive better heuristics [Helmert, 2006; Helmert, Haslum, & Hoffmann, 2007] inside a heuristic search planner. However, as revealed by recent work [Helmert & Röger, 2008], an admissible search with even almost perfect heuristics may still have exponential complexity. Hence, it is important to improve other components of the search algorithm that are orthogonal to the design of better heuristics.

The main contribution of this paper is a completeness-preserving reduction technique, the **expansion core (EC) method**, that can be combined with search algorithms on SAS+ planning domains. The key idea of EC is that, by restricting the set of actions to be expanded at each state, we are in effect removing a lot of redundant ordering of actions. For example, for a given state s, if for any solution path from s that places an action a before b, there exists another solution path from s that places b before a, then it reduces the search space to remove the ordering $a \rightarrow b$ and only consider those plans that place b before a.

Technically, the EC method modifies the way each node is expanded during the search. When searching in a Cartesian product of subgraphs, expanding a node amounts to expanding executable actions in all the subgraphs, which we reveal is often wasteful. EC expands a subset of subgraphs that form a dependency closure and still maintains the completeness and

optimality of search.

The partial order based reduction idea behind EC has been studied in the model checking community to address the state space explosion problem [Valmari, 1998]. However, clean and domain-independent reduction ideas for planning are few. Symmetry detection [Fox & Long, 1999] is one such idea that can be used to reduce redundant partial orderings in planning. We discuss later that our algorithm is different from symmetry detection.

This paper is organized as follows. We first give basic definitions in Section 2. We then propose the EC method in Section 3. We report experimental results in Section 4 and give conclusions in Section 5.

2 Background

A SAS+ planning task is defined on a set of **state variables** $X = \{x_1, \cdots, x_N\}$, each with a domain $Dom(x_i)$; and a set of actions $\mathcal{O}$, where each action $o \in \mathcal{O}$ is a tuple $(pre(o), eff(o))$, where $pre(o)$ defines some partial assignments of the state variables and $eff(o)$ defines a set of value transitions (v_i, v_i'), $v_i \in Dom(x_i) \cup \{\text{unknown}\}$, $v_i' \in Dom(x_i)$.

A complete assignment of the state variables is a **state**. We can write a state s as $s = (s_1, \ldots, s_N)$, where each $s_i \in Dom(x_i)$ is the assignment to variable x_i. For each state s, an action o is **executable** if s agrees to the partial assignments in $pre(o)$ and the first component of each element of $eff(o)$. Applying o to s leads to a new state $s' = result(s, o)$ where the value of x_i is changed to v_i' for each $(v_i, v_i') \in eff(o)$ and other state variables keep their value. $result(s, o)$ is undefined if o is not executable at S. v_i can take value from $Dom(x_i)$ or be "unknown", which means the action will lead to state v_i' regardless of the value of x_i. The planning task is to find an executable sequence of actions that transits a state (s_0) to a partial assignment to state variables (a goal state).

A task can also specify a **optimality criterion**. In this paper we assume that each action has a positive cost and the task is to minimize the total action costs. Optimal search algorithms such as A* can optimize this preference using admissible heuristics.

Each state variable $x_i, i = 1, \cdots, N$ is associated with a **domain transition graph (DTG)** G_i, a directed graph with vertex set $V(G_i) = Dom(x_i)$ and edge set $E(G_i)$. An edge (v_i, v_i') belongs to $E(G_i)$ if and only if there is an action o with $(v_i, v_i') \in eff(o)$, in which case we say that o is associated with the edge $e_i = (v_i, v_i')$ (denoted as $o \vdash e_i$). A DTG G_i is **goal-related** if the partial assignments that define the goal states include an assignment in G_i.

Definition 1 *An action o is **associated** with a DTG G_i (denoted as $o \vdash G_i$) if o is associated with any edge in G_i.*

3 Expansion Core (EC) Method

Now we propose our search space reduction method. We characterize standard search algorithms by Algorithm 1. Different search algorithms differ by the remove-first() operation which fetches one node from the *open* list. Algorithm 1 is depth-first search (DFS) when *open* is a FILO queue, breadth

---

Algorithm 1: State_space_search

Input: State space graph G
Output: p, an optimal solution path

1   $closed \leftarrow$ an empty set;
2   insert the initial state s_0 into *open*;
3   **while** *open is not empty* **do**
4      $s \leftarrow$ remove-first(*open*);
5   **if** *s is a goal state* **then**   process solution_path(s);
6   **if** *s is not in closed* **then**
7   add s to *closed*;
8         $open \leftarrow$ insert(expand(s), *open*);

---

first search (BFS) when *open* is a FIFO queue, best-first search (including A*) when *open* is a priority queue ordered by an evaluation function.

We define a **reduction method** as a method to expand only a subset of the set $expand(s)$ in Algorithm 1 for each state s during the search. Normally a reduction method can be combined with any implementation of remove-first() and any heuristic function to form various algorithms. While the strategies for remove-first() and the design of heuristic functions have been heavily studied, the possible reduction of $expand()$ has been rarely studied in planning.

The standard definition of $expand(s)$, for a state s in a SAS+ task, is the set of states reachable from s by executing one action. Let the set of executable actions at s be $exec(s)$, we have $expand(s) = \{result(s, o) | o \in exec(s)\}$.

Definition 2 *For a SAS+ task, for each DTG $G_i, i = 1, \ldots, N$, for a vertex $v \in V(G_i)$, an edge $e \in E(G_i)$ is a **potential descendant edge** of v (denoted as $v \lhd e$) if 1) G_i is goal-related and there exists a path from v to the goal state in G_i that contains e; or 2) G_i is not goal-related and e is reachable from v. A vertex $w \in V(G_i)$ is a **potential descendant vertex** of v (denoted as $v \lhd w$) if 1) G_i is goal-related and there exists a path from v to the goal state in G_i that contains w; or 2) G_i is not goal-related and w is reachable from v.*

In a preprocessing phase, for each DTG G_i, we may decide for all vertex-edge pair (v, e), $v \in V(G_i)$, $e \in E(G_i)$, whether $v \lhd e$ in time polynomial to $|V(G_i)| + |E(G_i)|$. The algorithm checks, for each (v, e), $e = (u, w)$, whether v can reach u and w can reach the goal state, if any. We have $v \lhd e$ if both can be achieved. The $v \lhd w$ relationship of all vertex pairs can be decided similarly.

Definition 3 *For a SAS+ task, for each action $o \in \mathcal{O}$, we define $need(o)$ as the set*
$$need(o) = pre(o) \cup \{v_i | (v_i, v_i') \in eff(o)\}.$$

Intuitively, $pre(o)$ is the set of **prevailing preconditions**, while $need(o)$ is the set of all preconditions, including the second subset in the equation which we call **transitional preconditions**. Note that, we do not include "unknown" in $need(o)$ since it does not really require a precondition.

Definition 4 *For a SAS+ task, given a state $s = (s_1, \cdots, s_N)$, for any $1 \leq i, j \leq N, i \neq j$, we call s_i a*
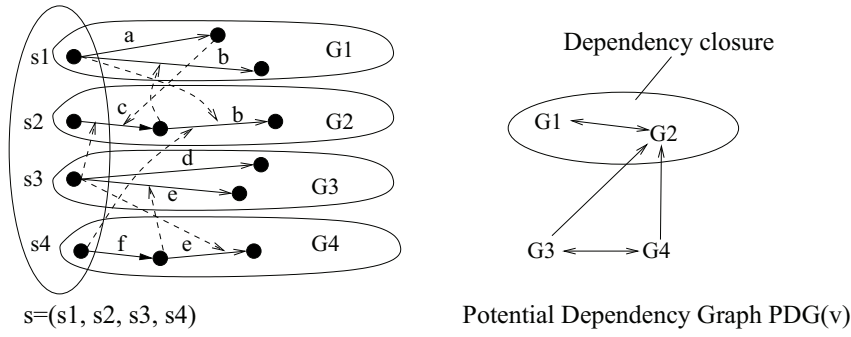
Figure 1: A SAS+ task with four DTGs. The dashed arrows show preconditions (prevailing and transitional) of each edge (action). Actions are marked with letters a to f. We see that b and e are associated with more than one DTG.

potential precondition of the DTG G_j if there exist $o \in \mathcal{O}$ and $e_j \in E(G_j)$ such that

$$s_j \lhd e_j, \; o \vdash e_j, \; and \; s_i \in need(o) \qquad (1)$$

Definition 5 *For a SAS+ task, given a state $s = (s_1, \ldots, s_N)$, for any $1 \le i, j \le N, i \ne j$, we call s_i a* **potential dependent** *of the DTG G_j if there exists $o \in \mathcal{O}$, $e_i = (s_i, s_i') \in E(G_i)$ and $w_j \in V(G_j)$ such that*

$$s_j \lhd w_j, \; o \vdash e_i, \; and \; w_j \in need(o) \qquad (2)$$

Definition 6 *For a SAS+ task, for a state $s = (s_1, \ldots, s_N)$, its* **potential dependency graph** *PDG(s) is a directed graph in which each DTG $G_i, i = 1, \cdots, N$ corresponds to a vertex, and there is an edge from G_i to G_j, $i \ne j$, if and only if s_i is a potential precondition or potential dependent of G_j.*

Definition 7 *For a directed graph H, a subset C of $V(H)$ is a* **dependency closure** *if there do not exist $v \in C$ and $w \in V(H) - C$ such that $(v, w) \in E(H)$.*

Figure 1 illustrates the above definitions. In PDG(s), G_1 points to G_2 as s_1 is a potential precondition of G_2 and G_2 points to G_1 as s_2 is a potential dependent of G_1. We also see that G_1 and G_2 form a dependency closure of PDG(s).

Given PDG(s), we first find the strongly connected components (SCCs) of it. If each SCC is contracted to a single vertex, the resulting graph is a directed acyclic graph $\mathcal{S}$. We observe that each SCC in $\mathcal{S}$ with a zero out-degree forms a dependency closure.

The **expansion core (EC)** method can be described as follows. In the *expand(s)* operation, instead of expanding actions in all the DTGs, we only expand actions in DTGs that belong to a dependency closure of PDG(s) under the condition that not all DTGs in the dependency closure are at a goal state. The operation is described as follows.

Definition 8 *The EC method modifies the $expand(s)$ operation in Algorithm 1 to:*

$$expand_r(s) = \bigcup_{i \in \mathcal{C}(s)} \left\{ result(s, o) \middle| o \in exec(s) \land o \vdash G_i \right\},$$

where $\mathcal{C}(s) \subseteq \{1, \cdots, N\}$ is an index set satisfying:

1) The DTGs $\{G_i, i \in \mathcal{C}(s)\}$ form a dependency closure in PDG(s); and

2) there exists $i \in \mathcal{C}(s)$ such that G_i is goal-related and s_i is not the goal state in G_i.

The set $\mathcal{C}(s)$ can always be found for any non-goal state s since PDG(s) itself is always such a dependency closure. If there are more than one such closure, theoretically any dependency closure satisfying the above conditions can be used in EC without losing completeness. In practice, we choose the one with the minimum number of DTGs.

3.1 Theoretical analysis

We define some generic properties of reduction methods before analyzing the EC method. For a SAS+ task, its **state-space graph** is a directed graph $\mathcal{G}$ in which each state s is a vertex and there is an edge (s, s') if and only if there exists an action o such that $result(s, o) = s'$. For a reduction method, we can get a reduced state-space graph in which there is an edge (s, s') only if s' will be expanded as a child of s by the reduction method. For EC, the **EC-reduced state-space graph** $\mathcal{G}_r$ is a directed graph in which each state s is a vertex and there is an edge (s, s') if and only if $s' \in expand_r(s)$.

For a SAS+ task, a **solution sequence** in a state-space graph $\mathcal{G}$ is a pair (s^0, p), where s^0 is a non-goal state, $p = (a_1, \ldots, a_k)$ is a sequence of actions, and, let $s^i = result(s^{i-1}, a_i), i = 1, \ldots, k$, (s^{i-1}, s^i) is an edge in $\mathcal{G}$ for $i = 1, \ldots, k$ and s^k is a goal state.

Definition 9 *A reduction method is* **completeness-preserving** *if for any solution sequence (s^0, p) in the state-space graph, there also exists a solution sequence (s^0, p') in the reduced state-space graph.*

Definition 10 *A reduction method is* **cost-preserving** *(respectively,* **action-preserving***) if, for any solution sequence (s^0, p) in the state-space graph, there also exists a solution sequence (s^0, p') in the reduced state-space graph satisfying that p' has the same total action cost as p does (respectively, p' is a re-ordering of the actions in p).*

Clearly, action-preserving implies cost-preserving, which implies completeness-preserving.

Theorem 1 *The EC method is action-preserving.*

Proof. We prove that for any solution sequence (s^0, p) in the state-space graph $\mathcal{G}$, there exists a solution sequence (s^0, p') in the EC-reduced state-space graph $\mathcal{G}_r$ such that p' is a re-ordering of actions in p. We prove this fact by induction on

1661

k, the length of the path p. In our proof, s_j^i denotes the value of the DTG G_j at state s^i.

When $k = 1$, let a be the only action in p, for any goal-related DTG G_i such that s_i^0 is not the goal in G_i, we must have $a \vdash G_i$. According to the second condition in Definition 8, since $\mathcal{C}(s^0)$ must include the index of a goal-related DTG whose state is not the goal (let it be j), we have that $a \vdash G_j$ and that $result(s^0, a) \in expand_r(s^0)$. Thus, (s^0, p) is also a solution sequence in $\mathcal{G}_r$. The EC method is action-preserving in the base case.

When $k > 1$, consider a solution sequence (s^0, p) in $\mathcal{G}$: $p = (a_1, \ldots, a_k)$. Let $s^i = result(s^{i-1}, a_i), i = 1, \ldots, k$.

We consider two cases.

a) If there exists $l \in \mathcal{C}(s^0)$ such that $a_1 \vdash G_l$, then $s^1 \in expand_r(s^0)$ and (s^0, s^1) is in $\mathcal{G}_r$. Let $p^* = (a^2, \cdots, a^k)$, then (s^1, p^*) is also a solution sequence in $\mathcal{G}$. According the induction assumption, there exists a solution sequence (s^1, p^{**}) in $\mathcal{G}_r$ such that p^{**} is a re-ordering of the actions in p^*. Therefore, a_1 followed by p^{**} is a solution sequence (starting from s^0) in $\mathcal{G}_r$ and is a re-ordering of p.

b) If there exists no $l \in \mathcal{C}(s^0)$ such that $a_1 \vdash G_l$, let a_j be the first action in p such that there exists $m \in \mathcal{C}(s^0)$ and $a_j \vdash G_m$. Such an action must exist because of the second condition in Definition 8. Consider the sequence $p^* = (a_j, a_1, \cdots, a_{j-1}, a_{j+1}, \cdots, a_k)$.

For any $h \in \{1, \cdots, N\}$ such that $s_h^0 \in need(a_j)$, G_m will point to G_h in PDG(s^0) since s_m^0 is a potential dependent of G_h. Since $m \in \mathcal{C}(s^0)$, we have $h \in \mathcal{C}(s^0)$. Hence, none of the preconditions of a_j is in $G_i, i \notin \mathcal{C}(s^0)$. On the other hand, $s_i^0 = s_i^{j-1}, \forall i \in \mathcal{C}(s^0)$ and all the preconditions of a_j are satisfied at s^{j-1}. Thus, all the preconditions of a_j are satisfied at s^0 and a_j can be executed at s^0.

Let $s' = result(s^0, a_j)$. We know $(a_1, \cdots, a_{j-1})$ is an executable action sequence starting from s'. This is true because none of $a_1, \cdots, a_{j-1}$ has a precondition in a DTG that a_j is associated with (otherwise G_m will point to that DTG, forcing that DTG to be in $\mathcal{C}(s^0)$, which is a contradiction to the way we choose a_j. Therefore, moving a_j before $(a_1, \cdots, a_{j-1})$ will not make any of their preconditions unsatisfied. Further, since executing $(a_1, \cdots, a_j)$ and $(a_j, a_1, \cdots, a_{j-1})$ from s^0 lead to the same state, $a_{j+1}, \cdots, a_k$ is an executable action sequence after $(a_j, a_1, \cdots, a_{j-1})$ is executed.

From the above, we see (s^0, p^*) is a solution sequence in $\mathcal{G}$. Let $p^{**} = (a_1, \cdots, a_{j-1}, a_{j+1}, \cdots, a_k)$, (s', p^{**}) is a solution sequence in $\mathcal{G}$. From the induction assumption, we know there is a sequence p' which is a re-ordering of p^{**}, such that (s', p') is a solution sequence in $\mathcal{G}_r$. Since $s' \in expand_r(s^0)$, we have that a_j followed by p' is a solution sequence from s^0 and is a re-ordering of p^*, which is a re-ordering of p. Thus, the EC method is action-preserving. ∎

Since the EC method is action-preserving, it is also cost-preserving and completeness-preserving.

Example. An example is shown in Figures 2 and 3. Figure 2 shows a SAS+ task, where $a1$ is the initial state and $e5$ is the goal state. In Figure 3, a) and b) show the original and EC-reduced search space, respectively. We see that EC significantly reduces the space. For example, at node $a1$, since
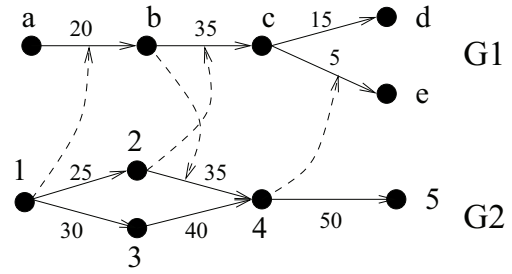

Figure 2: A SAS+ task with two DTGs. Action costs are shown on edges. All edges are distinct actions, except that (b, c) and $(2, 4)$ are the same action. The dashed arrows show precondition of actions.

a is not a potential dependent or precondition of G_2, we can only expand G_1 instead of expanding both DTGs. As another example, at $b4$, 4 is a potential precondition of G_1, but b is not a potential dependent or precondition of G_2. So at $b4$, G_1 is a dependency closure and we only expand G_1. In our case, there is no executable action in G_1 at $b4$ and we can safely conclude that $b4$ cannot reach the goal although there is an executable action in G_2. The reason is that, if expanding G_2 can make true certain preconditions for actions in G_1, then b would be a potential dependent of G_2 and G_2 would be in the dependency closure.

In the original space, the optimal cost 145. In the EC-reduced space we can still find an optimal path (in bold) with cost 145, which validates the correctness of the EC method.

EC allows us to search on the reduced graph $\mathcal{G}_r$. Since $expand_r()$ is always a subset of $expand()$, we have $|E(\mathcal{G}_r)| \leq |E(\mathcal{G})|$. Further, although $|V(\mathcal{G}_r)| = |V(\mathcal{G})|$, the vertices reachable from the initial state may be much fewer in $\mathcal{G}_r$, as shown in Figures 3. The number of vertices is reduced from 16 to 10 and number of edges from 21 to 9.

Theorem 2 *For a SAS+ task, a complete search with the EC method is still complete; an optimal search with the EC method is still optimal.*

Proof. Since the EC method is completeness-preserving, if there is a solution path in $\mathcal{G}$, there is also a solution path in $\mathcal{G}_r$, which will be found by a complete search with the EC method. Since the EC method is cost-preserving, an optimal search with the EC method will find an optimal path in $\mathcal{G}_r$, which is also an optimal path in $\mathcal{G}$. ∎

Theorem 3 *For an admissible A^* search on a SAS+ task, for nodes with $f < f^*$ where f^* is the cost of the optimal plan, the A^* search with the EC method expands no more such nodes than the original A^* search does.*

Theorem 3 can be easily seen. For each node v that A^* with the EC method expands, if its $f = g + h$ is less than the optimal cost, it must also be expanded by the original A^*. A node with $f < f^*$ will always be expanded by A^* but may not be expanded when EC is used, as shown in Figure 3. Hence, an A^* search with the EC method is guaranteed to expand at most as many nodes as A^* except possibly through tie-breaking among nodes with $f = f^*$.

For inadmissible search, although it is hard to give a similar result as Theorem 3, empirical study shows that EC typically also reduces the search cost.

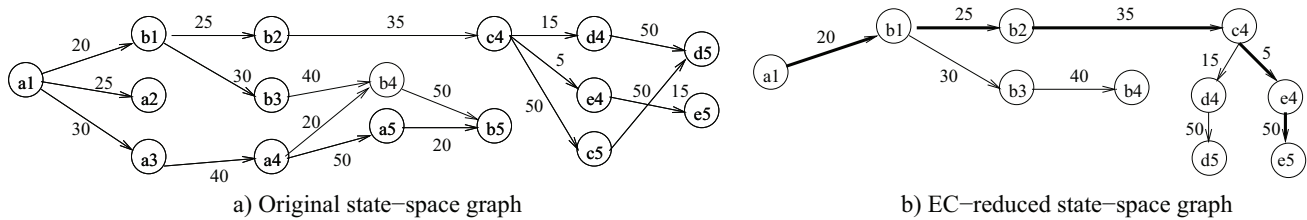We give a couple of additional comments below.

Figure 3: Comparison of the original and EC-reduced search space for solving the problem in Figure 2.

Symmetry. Symmetry detection is another way for reducing the search space [Fox & Long, 1999]. From the view of node expansion on a SAS+ formalism for planning, we can see that symmetry removal is different from EC. For example, consider a domain with three objects A_1, A_2, and B, where A_1 and A_2 are symmetric, and B has no interaction with A_1 or A_2. In this case, symmetry removal will expand DTGs for (A_1 and B) or (A_2 and B), whereas EC will only expand the DTG for B since the DTG for B forms a dependency closure. Intuitively, symmetric removal finds that it is not important whether A_1 or A_2 is used since they are symmetric, whereas EC finds that it is not important whether B is used before or after $A_i, i = 1, 2$ since there is no dependency.

Limitations. EC is most effective for the case where the DTGs are "directional" and the inter-DTG dependencies are not dense. It may not be useful for problems where the DTGs are strongly connected *and* there is a high degree of inter-DTG dependencies. For example, it is not useful for the 15-puzzle where each piece corresponds to a DTG which is a clique. Since each move excludes any other piece on the target position, any action in a DTG depends on all the other DTGs. Thus, the potential dependency graph is a clique for all the states and EC cannot give any reduction.

4 Experimental Results

We test on STRIPS problems in the recent International Planning Competitions (IPCs): IPC3, IPC4, and IPC5. We use a preprocessor in Fast Downward (FD) [Helmert, 2006] to convert a STRIPS problem into a SAS+ instance with multiple DTGs. Each instance is assigned 300 seconds to solve.

In our implementation, to save time for computing the $expand_r()$ set for each state, we use a preprocessing phase to precompute some reachability information of those DTGs that do not depend on others. For every pair (v_i, G_j), $v_i \in Dom(x_i)$, we precompute if v_i is a potential dependent or potential precondition of G_j.

Table 1 shows the results of applying EC to an A^* search with the admissible HSP heuristic [Bonet & Geffner, 2001]. Table 2 shows the results of applying EC to the Fast Downward planner with the casual graph (CG) heuristic and without the helper action heuristic. Both HSP and HSP+EC give the same plan length since they are both optimal (according to Theorem 2). For CG and CG+EC, their solution lengths are similar. For instances in Table 2, they give the same plan length in 21% of the cases, CG gives shorter plans in 41% of the cases, and CG+EC gives shorter plans in 38% of the cases.

From the tables, we see that EC can reduce the search cost, in terms of time and the number of generated nodes, significantly for most domains, although the degree of reduction is domain-dependent. The numbers of expanded nodes are also largely reduced, although not shown in the table due to space limit. The reduction is very significant for some domains and is more than 100 times for some instances. We do not show results on three domains where the EC method gives no reduction: pipesworld, freecell, and storage.

To gain some insights on why the EC method performs better for certain domains, we examine the causal graph (CG) [Helmert, 2006] of each domain. The CG is a graph where each vertex is a DTG and the edges represent dependencies among DTGs. For each CG, we find its strongly connected components (SCCs) and contract each SCC into a vertex. We find that for domains where EC is very effective (such as rovers and satellite), their CG can be decomposed into multiple SCCs (for example, 9 SCCs for satellite02), while for domains where EC is not useful (such as pipesworld and freecell), their CG is just one single SCC.

From Table 1 and Table 2, we also see that the EC method gives significant reduction for both admissible (HSP) and inadmissible (CG) heuristics. It seems that *the degree of reduction that EC can give is more related to the domain structure than to the quality of heuristics*. We explain this observation by the following. The goal of the EC method is to avoid trying redundant partial orderings of actions. For example, at a state s, if the action orderings $b \rightarrow a$ and $a \rightarrow b$ are determined to be redundant by the EC method, then EC will choose to expand only one ordering, say $a \rightarrow b$. Using a better heuristic may not help prune one of those two orderings. Let $s_a = result(s, a)$, $s_b = result(s, b)$, and $s_{ab} = result(s_a, b) = result(s_b, a)$. Normally, we would have $f(s_a) \leq f(s_{ab})$ and $f(s_b) \leq f(s_{ab})$ if the heuristic is consistent. Hence, s_a and s_b will both be expanded before s_{ab} is expanded and the search will essentially try both $b \rightarrow a$ and $a \rightarrow b$ even if the heuristic is highly accurate. This is essentially the same reasoning used in [Helmert & Röger, 2008] to explain why an almost perfect heuristic may still lead to high search costs. The EC method is orthogonal to the design of heuristics and can reduce redundant orderings.

5 Conclusions

We have proposed EC, a completeness-preserving reduction method for SAS+ planning which provides an automatic way to reduce the search space without losing completeness and optimality. EC is a general principle for removing redundancy in search and does not require any parameter tuning.

ID	HSP+EC		HSP		ID	HSP+EC		HSP	
	Time	Node	Time	Node		Time	Node	Time	Node
zeno02	0.01	93	0.01	127	zeno03	0.08	4679	0.08	5762
zeno04	0.16	9585	0.14	10215	zeno05	1.59	157406	1.55	185715
zeno06	11.11	1395342	11.3	1624301	zeno07	11.4	1379764	11.69	1628619
rovers01	0.01	949	0.01	2043	rovers02	0.01	254	0.01	1019
rovers03	0.05	2869	0.03	5724	rovers04	0.02	935	0.01	2399
rovers05	44.51	3768608	-	-	rovers07	231.9	15093217	-	-
rovers5-01	0.01	949	0.01	2043	rovers5-02	0.01	254	0.01	1019
rovers5-03	0.05	2869	0.03	5724	rovers5-04	0.02	935	0.01	2399
rovers5-05	45.68	3768608	-	-					
tpp01	0.01	5	0.01	8	tpp02	0.01	8	0.01	34
tpp03	0.01	11	0.01	188	tpp04	0.01	14	0.01	1130
tpp05	1.35	53715	0.34	72689					
satellite01	0.01	367	0.01	437	satellite02	0.07	5820	0.04	6724
satellite03	0.54	31123	1.13	151351	satellite04	6.38	781393	16.13	3894047
depots01	0.01	725	0.01	1000	depots02	1.01	17613	1.07	17903
pathways1	0.01	146	0.01	162	pathways2	0.06	2600	0.03	2890
pathways3	1.48	47852	0.37	82588	pathways4	22.4	597628	12.86	2112481
driverlog01	0.01	293	0.01	373	driverlog02	2.32	192915	1.99	417835
driverlog03	0.09	15302	0.09	22685	driverlog04	12.87	1430792	26.29	4794197
driverlog05	135.78	14806514	144.22	24223525	driverlog06	6.22	650872	8.25	1289283
driverlog07	20.36	1960212	77.76	10115601					

Table 1: Comparison of an A^* search with the HSP heuristic and HSP+EC. We give CPU time in seconds and number of generated nodes. "-" means timeout after 300 seconds.

EC is orthogonal to and can be combined with the development of other components of search such as heuristics and search control strategies.

Acknowledgement

References

[Amir & Engelhardt, 2003] Amir, E., and Engelhardt, B. 2003. Factored planning. In *Proc. IJCAI*.

[Bäckström & Nebel, 1995] Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:17–29.

[Bonet & Geffner, 2001] Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129:5–33.

[Brafman & Domshlak, 2006] Brafman, R., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *Proc. AAAI*.

[Fox & Long, 1999] Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proc. IJCAI*.

[Helmert & Röger, 2008] Helmert, M., and Röger, G. 2008. How good is almost perfect. In *Proc. AAAI*.

[Helmert, Haslum, & Hoffmann, 2007] Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*.

[Helmert, 2006] Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

[Jonsson & Bäckström, 1998] Jonsson, P., and Bäckström, C. 1998. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence* 100(1-2):125–176.

ID	CG+EC		CG		ID	CG+EC		CG	
	Time	Node	Time	Node		Time	Node	Time	Node
zeno02	0.01	55	0.01	79	zeno03	0.01	33	0.01	140
zeno04	0.04	1414	0.01	186	zeno05	0.01	325	0.01	520
zeno06	0.02	111	0.01	948	zeno07	0.01	79	0.04	6072
zeno08	0.4	36673	0.06	3728	zeno09	0.04	442	0.12	17571
zeno10	0.83	90180	0.08	30128	zeno11	0.63	64296	0.11	7486
zeno12	0.06	553	0.44	84042	zeno13	0.08	551	0.2	21576
zeno14	7.14	216755	-	-	zeno15	0.77	2394	-	-
rovers01	0.01	92	0.01	562	rovers02	0.01	47	0.01	518
rovers03	0.01	91	0.01	733	rovers04	0.01	83	0.01	315
rovers05	0.28	18681	11.59	4640634	rovers06	0.89	73788	-	-
rovers08	1.23	64696	-	-	rovers09	0.53	30970	-	-
rovers12	0.12	5454	2.75	472044	rovers14	1.99	86599	74.97	6800535
rovers15	5.93	375318	-	-					
rovers5-01	0.01	117	0.01	562	rovers5-02	0.01	47	0.01	518
rovers5-03	0.01	91	0.01	733	rovers5-04	0.01	83	0.01	315
rovers5-05	0.4	22789	12.17	4640634	rovers5-06	0.9	76385	-	-
rovers5-07	0.84	32795	6.7	507085	rovers5-08	1.37	59096	-	-
rovers5-09	0.59	31806	-	-	rovers5-11	5.43	172783	-	-
rovers5-12	0.05	4514	3.39	468894	rovers5-14	1.26	70280	47.01	7665779
rovers5-15	2.71	142651	-	-					
pathways1	0.01	146	0.01	162	pathways2	0.03	591	0.02	763
pathways3	0.25	3495	0.2	5280	pathways4	0.18	3314	0.24	10868
pathways5	35.67	711566	76.03	2327520					
trucks01	0.01	534	0.01	686	trucks02	0.02	1376	0.03	898
trucks03	0.04	2430	0.05	2524	trucks04	11.9	988472	22.79	2499556
trucks05	9.42	1221125	158.2	14362673	trucks07	0.47	45150	0.97	66456
trucks08	1.57	73822	0.31	10909					
driverlog01	0.01	460	0.01	669	driverlog02	0.12	24064	0.27	53426
driverlog03	0.01	364	0.01	471	driverlog04	0.05	3813	0.05	10570
driverlog05	0.17	17838	0.13	27458	driverlog06	0.2	16993	0.19	33262
driverlog07	0.05	2816	0.06	7609	driverlog08	0.79	73211	1	238640
driverlog09	0.37	54847	0.29	86047	driverlog10	0.05	2980	0.05	3197
driverlog11	0.52	9152	0.51	10361	driverlog13	3.7	112476	4.32	163153
driverlog14	126.84	4419057	173.2	6466593	driverlog15	170.56	2612581	179.17	2818507
tpp01	0.01	3	0.01	8	tpp02	0.01	9	0.01	27
tpp03	0.01	19	0.01	106	tpp04	0.01	33	0.01	292
tpp05	0.01	267	0.01	1857	tpp06	0.69	32828	4.89	1231832
tpp07	1.6	118115	-	-	tpp08	56.77	2958141	-	-
satellite01	0.01	7	0.01	182	satellite02	0.01	33	0.01	1394
satellite03	0.01	283	0.01	2298	satellite04	0.02	763	0.22	45107
satellite05	0.03	266	0.55	26460	satellite06	0.02	444	0.36	68135
satellite07	0.05	518	7.99	968897	satellite08	0.94	13892	41.45	3576497
satellite09	0.14	2915	36.68	3148465	satellite10	0.12	3970	92.92	7543625
satellite11	0.09	5216	-	-	satellite12	1.48	33226	-	-
satellite13	69.54	4060593	-	-	satellite14	4.11	111973	-	-
satellite15	3.41	116438	-	-	satellite16	10.24	150791	-	-
satellite17	4.85	117576	-	-	satellite18	0.45	15094	-	-
satellite19	12.63	247603	-	-	satellite20	51.58	728401	-	-
depots01	0.01	476	0.01	471	depots02	0.38	2868	0.12	1800
depots03	2.89	26953	1.72	35824	depots04	22.22	151188	10.79	153575
depots07	1.39	6944	2.9	52026	depots10	79.92	490742	139.11	1780267
depots13	10.03	68415	77.96	1001913	depots17	23.21	87025	-	-

Table 2: Comparison of search with the CG heuristic and CG+EC. We give number of generated nodes and CPU time in seconds. "-" means timeout after 300 seconds.

[Kelareva *et al.*, 2007] Kelareva, E.; Buffet, O.; Huang, J.; and Thiébaux, S. 2007. Factored planning using decomposition trees. In *Proc. IJCAI*.

[Knoblock, 1994] Knoblock, C. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68:243–302.

[Lansky & Getoor, 1995] Lansky, A., and Getoor, L. 1995. Scope and abstraction: two criteria for localized planning. In *Proc. AAAI*.

[Valmari, 1998] Valmari, A. 1998. The state explosion problem. *Lectures on Petri Nets I: Basic Models,* Lecture Notes in Computer Science 1491:429–528.