Stratified Planning

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

You Xu
Department of Computer Science
Washington University
youxu@wustl.edu

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

Abstract

Most planning problems have strong structures. They can be decomposed into subdomains with causal dependencies. The idea of exploiting the domain decomposition has motivated previous work such as hierarchical planning and factored planing. However, these algorithms require extensive backtracking and lead to few efficient general-purpose planners. On the other hand, heuristic search has been a successful approach to automated planning. The domain decomposition of planning problems, unfortunately, is not directly and fully exploited by heuristic search.

We propose a novel and general framework to exploit domain decomposition. Based on a structure analysis on the SAS+ planning formalism, we stratify the sub-domains of a planning problem into dependency layers. By recognizing the stratification of a planning structure, we propose a space reduction method that expands only a subset of executable actions at each state. This reduction method can be combined with state-space search, allowing us to simultaneously employ the strength of domain decomposition and high-quality heuristics. We prove that the reduction preserves completeness and optimality of search and experimentally verify its effectiveness in space reduction.

1 Introduction

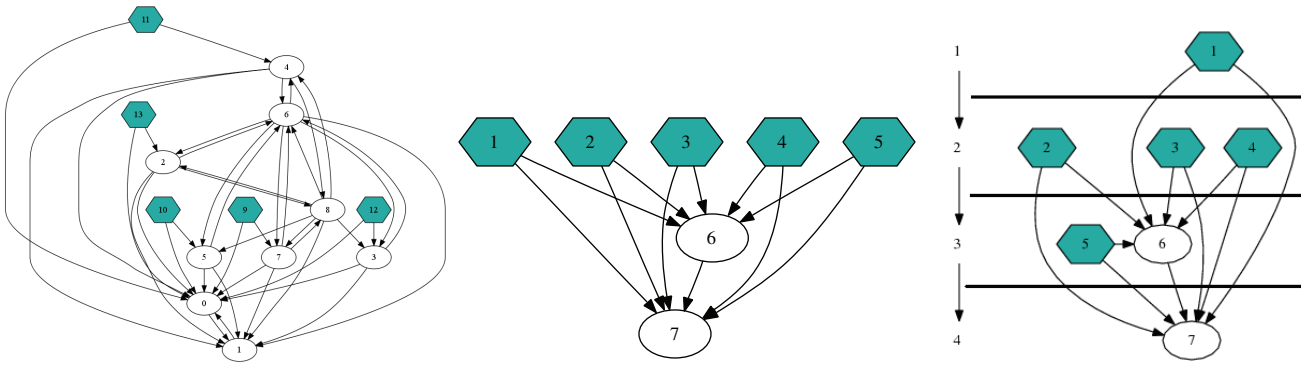
We have witnessed significant improvement of the capability of automated planners in the past decade. Heuristic search remains one of the key, general-purpose approaches to planning. The performance improvement is largely due to the development of high-quality heuristics. However, as shown by recent work, developing better heuristics only has some fundamental limitations [Helmert and Röger, 2008]. Heuristic planners still face scalability challenges for large-scale problems. It is important to develop new, orthogonal ways to improve the efficiency, among which domain decomposition has been an attractive idea to planning researchers.

A representative work based on domain decomposition is the automated hierarchical planning methods [Knoblock, 1994; Lansky and Getoor, 1995] that utilize hierarchical factoring of planning domains. However, they typically do not scale well since they require extensive backtracking across subdomains. Another work is the factored planning approach [Amir and Engelhardt, 2003; Brafman and Domshlak, 2006; Kelareva *et al.*, 2007] that finds subplans for each subproblem before merging some of them into one solution plan. However, the method requires either enumerating all subplans for each subproblem, which is very expensive, or extensive backtracking. Also, it faces difficulties involved with the length bound of the subplans. It is yet to be investigated if factored planning can give rise to a practically competitive approach for general-purpose planning.

In this paper, we propose a novel way to utilize the domain structure. Our key observation is that normally a planning problem P can be *stratified* into multiple sub-domains $P_1, P_2, \ldots, P_k$ in such a way that, for $i < j$, the actions in P_i may require states in P_j as preconditions, but not vice versa. We then investigate the intriguing problem: *given a stratification of a planning problem, can we make the search faster*?

We develop a completeness-preserving space reduction method based on stratification. Our observation is, in standard search algorithms, each state is composed of the states of the sub-domains $P_1, \ldots, P_k$, and the search will expand the applicable actions in all the sub-domains $P_1, \ldots, P_k$, which is often unnecessary. We propose a fundamental principle for systems that can be stratified into layers of sub-domains. Due to the oneway-ness of the dependencies across the stratified layers, the search can expand only those actions in a subset of the sub-domains. In principle, if the preceding action a of a state is at layer j, $1 \leq j \leq k$, we only expand actions at layer j to k and those actions that have direct causal relations with a at other layers. We prove that such a reduced expansion scheme preserves completeness (and consequently, optimality) of search algorithms.

The proposed scheme has a number of advantages. The reduction method is embedded inside a heuristic search. Therefore, 1) since most domains can be stratified, the method can effectively prune a lot of redundant paths, leading to significant reduction of search costs; 2) in the worst-case when the method cannot give any reduction (such as when all the subdomains are in one dependency closure and cannot be stratified), the search will expand the same number of nodes as the original search; 3) the method can leverage on the



a) Causal graph (CG) b) Contracted causal graph (CCG) c) Stratification

Figure 1: The causal graph, contracted causal graph and stratification of Truck-02.

highly sophisticated heuristic functions. Thus, this scheme seems more practical than those methods that explicitly use a decomposition-based search, such as factored planning and hierarchical planning. It combines the strength of both heuristic search and domain decomposition and adapts to the domain structure.

In summary, our main contributions are:

- We propose an automatic and domain-independent stratification analysis that gives vital structural information of a planning problem.
- We tackle the problem of reducing search cost from a novel perspective. We propose a space reduction method that can be embedded seamlessly to existing search algorithms. Our approach is orthogonal to the development of more powerful search algorithms and more accurate heuristics.
- We prove that a search algorithm combined with our reduction method is complete (respectively optimal) if the original search is complete (respectively optimal).
- We show that two implementations of the proposed framework can improve the search efficiency on various planning domains.

2 Background

We present our work based on the SAS+ formalism [Bäckström and Nebel, 1995] of planning, a natural representation of domain decomposition. The idea behind our work is general enough to be applied to other systems whose components can be stratified into dependency layers.

A SAS+ planning task Π is defined as a tuple $\Pi = \{X, O, S, s_{\mathcal{I}}, s_{\mathcal{G}}\}$.

- $X = \{x_1, \cdots, x_N\}$ is a set of multi-valued *state variables*, each with an associated finite domain $Dom(x_i)$.
- $\mathcal{O}$ is a set of actions and each action $o \in \mathcal{O}$ is a tuple $(pre(o), eff(o))$, where both $pre(o)$ and $eff(o)$ define some partial assignments of variables in the form $x_i = v_i, v_i \in Dom(x_i)$. $s_{\mathcal{G}}$ is a partial assignment that defines the goal.
- S is the set of states. A **state** $s \in S$ is a full assignment to all the state variables. $s_{\mathcal{I}} \in S$ is the initial state. A state s is a goal state if $s_{\mathcal{G}} \subseteq s$.

For a given state s and an action o, when all variable assignments in $pre(o)$ are met in state s, action o is *applicable* at state s. After applying o to s, the state variable assignment will be changed to a new state s' according to $eff(o)$. We denote the resulting state of applying an applicable action o to s as $s' = apply(s, o)$.

For a SAS+ planning task, for an action $o \in \mathcal{O}$, we define the following:

- The **dependent variable set** $dep(o)$ is the set of state variables that appear in the assignments in $pre(o)$.
- the **transition variable set** $trans(o)$ is the set of state variables that appear in both $pre(o)$ and $eff(o)$.
- the **affected variable set** $aff(o)$ is the set of state variables that appear in the assignments in $eff(o)$.

Note that $trans(o)$ might be $\emptyset$, and it is always true that $trans(o) \subseteq dep(o)$ and $trans(o) \subseteq aff(o)$.

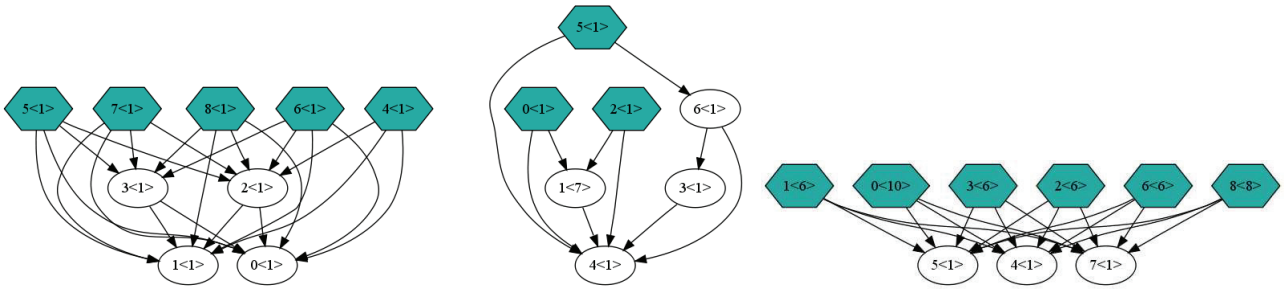
Definition 1. *Given a SAS+ planning task Π with state variable set X, its **causal graph** (CG) is a directed graph $CG(\Pi) = (X, E)$ with X as the vertex set. There is an edge $(x, x') \in E$ if and only if $x \neq x'$ and there exists an action o such that $x \in trans(o)$ and $x' \in dep(o)$, or, $x \in aff(o)$ and $x' \in trans(o)$.*

Intuitively, the nodes in the CG are state variables and the arrows in CG describe the dependency relationships between variables. If the CG contains an arc from x_i to x_j, then a value change of x_j will possibly affect the applicability of some action o that involves a transition of x_i. Figure 1a shows the CG of an instance (Truck-02) of the Truck planning domain used in the 5th International Planning Competition (IPC5). State variables that define the goal state are in darker color.

3 Stratification of Planning Problems

Now we propose our stratification analysis. Given a SAS+ task, usually its CG is not acyclic, which leads to cyclic causal dependencies among some of (but often not all) the state variables. We propose a strongly connected component analysis on CG.

A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. For a directed graph, a **strongly connected component (SCC)**



a) Satellite02 b) Rover01 c) TPP11

Figure 2: The CCGs of some instances of several planning domains. Each SCC is labelled by x<y>, where x is the ID and y is the number of state variables in the SCC. The SCCs that contain goals are in a darker color.

is a maximal strongly connected subgraph. A directed graph can be uniquely decomposed into several SCCs. A **partition** of a set X is a set of nonempty subsets of X such that every element x in X is in exactly one of these subsets.

Definition 2 (Component Set). *Given a SAS+ planning task* Π *and its causal graph* $CG(\Pi) = (X, E)$*, the component set* $\mathcal{M}(\Pi)$ *is the partition of* X *such that all the elements in each* $m \in \mathcal{M}(\Pi)$ *are in the same SCC of* $CG(\Pi)$*.*

Definition 3 (Contracted Graph). *Given a directed graph* $G = (V, E)$*, a contracted graph of* G *is a directed graph* $G' = (V', E')$*, where each* $v' \in V'$ *is a subset of* V *and* V' *is a partition of* V*. There is an arc* $(v', w') \in E'$ *if and only if there exist* $v \in v'$ *and* $w \in w'$ *such that* $(v, w) \in E$*.*

Definition 4 (Contracted Causal Graph (CCG)). *Given a SAS+ planning task* Π*, its contracted causal graph* $CCG(\Pi) = (V, E)$ *is a contracted graph of* $CG(\Pi)$ *such that* $V = \mathcal{M}(\Pi)$*.*

Figure 1b shows the corresponding CCG of Figure 1a. Each vertex in the CCG may contain more than one state variable. Intuitively, given the CG of a graph, we find its SCCs and contract each SCC into a vertex. The resulting graph is the CCG. Figure 2 shows the CCG of some other domains. The CCG plays the central role in our structural analysis. It has an important property.

Proposition 1. *For any SAS+ planning task* Π*,* $CCG(\Pi)$ *is a directed acyclic graph (DAG).*

The above statement is true because if the CCG contains a cycle, then all the vertices on that cycle are strongly connected and should be contracted into one SCC. Therefore, we see that although there are dependency cycles in the CG, there is no cycle after we contract each SCC to one vertex. A topological sort on the CCG gives an list of the SCCs, ordered by dependency relations. Stratification can be viewed as a generalization of topological sort.

Definition 5 (Stratification). *Given a DAG* $G = (V, E)$*, a stratification* $Str(G)$ *of* G *is a tuple* $(\mathcal{U}, \mathcal{L})$*, where* $\mathcal{U} = \{u_1, \cdots, u_k\}$ *is a partition of* V*.* $\mathcal{U}$ *satisfies that there do not exist* $i, j, 1 \leq i < j \leq k$*,* $v_i \in u_i$*, and* $v_j \in u_j$ *such that* $(v_j, v_i) \in E$*. The function* $\mathcal{L} : V \mapsto \mathbb{N}^+$ *is called the* ***layer function****.* $\mathcal{L}(v) = k$ *for any* $v \in V$ *and* $v \in u_k$*.*

The stratification of a DAG $G = (V, E)$ is not unique. A stratification $Str(G) = (\mathcal{U}, \mathcal{L})$ is called a k**-stratification** if $|\mathcal{U}| = k$. The upper bound of k is $|V|$. When $k = |V|$, $\mathcal{U}$ must be a topological sort of V.

Definition 6 (Stratification of a SAS+ Task). *Given a SAS+ planning task* Π*, a stratification of* Π*, denoted by* $Str(\Pi)$*, is a stratification of* $CCG(\Pi)$*.*

Intuitively, a stratification of a CCG gives the flexibility to cluster state variables while pertaining to the topological order. There can be one-way dependency or no dependency, but no two-way dependency, between any two state variables at different layers under a stratification.

Figure 1c shows an example of stratification of the Truck-02 problem. Each SCC in Figure 1b is now assigned a layer and the topological order is maintained in the stratification. Basicly, the requirement is that there is no arrow pointing from a larger-numbered layer to a smaller-numbered one.

The stratification defines the layer function for any state variable $x \in X$. Based on that, we can define the layer function $\mathcal{L}(o)$ for each action $o \in \mathcal{O}$.

Definition 7 (Action Layer). *For a SAS+ task* Π*, given a stratification* $Str(\Pi) = (\mathcal{U}, \mathcal{L})$*, for an action* $o \in \mathcal{O}$*,* $\mathcal{L}(o)$ *is defined as* $\mathcal{L}(x)$*, for an arbitrary* $x \in trans(o)$*, if* $trans(o)$ *is nonempty; and* $\mathcal{L}(o) = \infty$ *if* $trans(o) = \emptyset$*.*

We prove that $\mathcal{L}(o)$ is well-defined by showing that all $x \in trans(o)$ has the same $\mathcal{L}(x)$, for any action $o \in \mathcal{O}$.

Proposition 2. *For a SAS+ task* Π*, for an action* $o \in \mathcal{O}$ *with* $trans(o) \neq \emptyset$*, we have* $\mathcal{L}(x_i) = \mathcal{L}(x_j), \forall x_i, x_j \in trans(o)$*.*

Proof. Since $x_j \in trans(o) \subseteq dep(o)$ and $x_i \in trans(o)$, by Definition 1, there is an arc from x_j to x_i in $CG(\Pi)$. Similarly, $x_i \in trans(o) \subseteq dep(o)$ and $x_j \in trans(o)$, an arc exists from x_i to x_j in $CG(\Pi)$. This implies that x_i and x_j are strongly connected in $CG(\Pi)$ and are elements in a same vertex of $CCG(\Pi)$. By Definition 5, we have $\mathcal{L}(x_i) = \mathcal{L}(x_j)$. □

4 Stratified Planning Algorithm

Now we propose our stratified planning algorithm. In fact, it is not a standalone algorithm but rather a space reduction method that can be combined with other search algorithms. It reduces the number of actions that need to be expanded at each state.

We outline stratified planning in Algorithm 1. The input is a SAS+ task Π and a stratification $Str(\Pi)$. It is a general framework where the *open* list can be implemented as a stack, queue or priority queue. The *open* list contains a list of states that are generated but not expanded.

Algorithm 1: Stratified_planning_search(Π, $Str(\Pi)$)

Input: A SAS+ planning task Π and a stratification $Str(\Pi) = (\mathcal{U}, \mathcal{L})$
Output: A solution plan

```
closed ← an empty set;
insert the initial AS pair (no-op,s_I) to open;
while open is not empty do
    (a, s) ← remove-next(open);
    if s is a goal state then return solution;
    if s is not in closed then
        add s to closed;
        Ψ(a, s, L) = stratified_expansion(a, s, L) ;
        open ← open ∪ Ψ(a, s, L);
```

Algorithm 2: stratified_expansion($a, s, \mathcal{L}$)

Input: An AS pair (a, s) and the $\mathcal{L}$ function
Output: The set $\Psi(a, s, \mathcal{L})$ of successor AS pairs

```
Ψ ← ∅;
foreach applicable action b at s do
    if L(b) ≥ L(a) then
        compute s' = apply(s, b);
        Ψ ← Ψ ∪ {(b, s')} ;
    else if a ▷ b then
        compute s' = apply(s, b);
        Ψ ← Ψ ∪ {(b, s')} ;
end
return Ψ ;
```

Definition 8. *For the purpose of stratified planning, for each generated state s, we record an **action-state (AS) pair** (a, s) in the open list, where a is the action that leads to s during the search. a is called the **leading action** of s.*

Each time during the search, a remove-next() operation fetches one AS pair (a, s) from $open$, checks if the state s is a goal state or is in the $closed$ list. If not, the stratified_expansion() operation will generate a set of AS pairs (b, s') to be inserted to $open$, where s' is the resulting state of applying b to s, i.e. $s' = apply(s, b)$.

The difference between stratified planning and a standard search is that, in standard search, we will expand all the actions that are applicable at s, while stratified_expansion() may not expand every applicable action.

Since the initial state $s_{\mathcal{I}}$ has no leading action, a special action no-op is defined as its leading action and its layer is defined as 0.

Definition 9 (Follow-up Action). *For a SAS+ task Π, for two actions $a, b \in \mathcal{O}$, b is a follow-up action of a (denoted as $a \rhd b$) if $aff(a) \cap dep(b) \neq \emptyset$ or $aff(a) \cap aff(b) \neq \emptyset$. Any action is a follow-up action of no-op.*

Given this definition, we can describe the **stratified_expansion()** operation, shown in Algorithm 2. Given a stratification $Str(\Pi) = (\mathcal{U}, \mathcal{L})$, the procedure of stratified_expansion() is quite simple. For any AS pair (a, s) to be expanded, for each action $b \in \mathcal{O}$ that is applicable at s, we consider two cases.

- If $\mathcal{L}(b) \geq \mathcal{L}(a)$, we expand b.
- If $\mathcal{L}(b) < \mathcal{L}(a)$, we expand b only if b is a follow-up action of a ($a \rhd b$).

For any AS pair (a, s), all the AS pairs expanded by stratified_expansion() forms the set $\Psi(a, s, \mathcal{L})$.

5 Theoretical Analysis

In this section, we show that stratified planning search preserves the completeness and optimality property of the original search strategy, decided by the implementation of the $open$ list and the evaluation function. For example, if $open$ is a priority queue and the evaluation function is admissible, then the original search, with a full expansion at each state, is both complete and optimal.

Definition 10 (Valid Path). *For a SAS+ task Π and a state s_0, a sequence of actions $p = (o_1, \ldots, o_n)$ is a valid path if, let $s_i = apply(s_{i-1}, o_i), i = 1, \ldots, n$, o_i is applicable at s_{i-1} for $i = 1, \ldots, n$. We also say that applying p to s results in the state s_n.*

Definition 11 (Stratified Path). *For a SAS+ task Π, for a stratification $str(\Pi) = (\mathcal{U}, \mathcal{L})$ and a state s_0, a sequence of actions $p = (o_1, \ldots, o_n)$ is a stratified path if it is a valid path and, let $s_i = apply(s_{i-1}, o_i), i = 1, \ldots, n$, $(o_i, s_i) \in \Psi(o_{i-1}, s_{i-1}, \mathcal{L})$ for $i = 1, \ldots, n$, where o_0 = no-op.*

Intuitively, a stratified path is a sequence of actions that can possibly be generated by the stratified search.

Lemma 1. *For a SAS+ task Π, a stratification $str(\Pi) = (\mathcal{U}, \mathcal{L})$ and a state s_0, for any valid path $p = (a_1, \ldots, a_n)$, if there exists $2 \leq i \leq n$, such that $\mathcal{L}(a_i) < \mathcal{L}(a_{i-1})$ and that a_i is not a follow-up action of a_{i-1}, then $p' = (a_1, \ldots, a_{i-2}, a_i, a_{i-1}, a_{i+1}, \ldots, a_n)$ is also a valid path and leads to the same state from s_0 as p does.*

Proof. Let $s_j = apply(s_{j-1}, a_j), j = 1, \cdots, n$. Since a_i is not a follow-up action of a_{i-1}, according to Definition 9, $eff(a_{i-1})$ contains no assignment in $pre(a_i)$. Therefore, since a_i is applicable at s_{i-1}, which is $apply(s_{i-2}, a_{i-1})$, we know a_i is also applicable at s_{i-2}.

Since $\mathcal{L}(a_i) < \mathcal{L}(a_{i-1})$, the SCC in $CCG(\Pi)$ that contains a_{i-1} has no dependencies on the SCC that contains a_i. Therefore, $eff(a_i)$ contains no assignment in $pre(a_{i-1})$. Since the variable assignments in $pre(a_{i-1})$ are satisfied at s_{i-2}, it is also satisfied at $s' = apply(s_{i-2}, a_i)$.

From the above, we see that (a_i, a_{i-1}) is an applicable action sequence at s_{i-2}. Further, since a_i is not a follow-up action of a_{i-1}, we have that $aff(a_i) \cap aff(a_{i-1}) = \emptyset$. Hence, applying (a_i, a_{i-1}) to s_{i-2} leads to the same state as applying (a_{i-1}, a_i), which is s_i. Therefore, p' is a valid path from s_0 and leads to the same state as p does. □

Theorem 1. *Given a SAS+ planning task Π and a stratification $Str(\Pi)$, for any state s_0 and any valid path $p_a = (a_1, \ldots, a_n)$ from s_0, there exists a stratified path $p_b = (b_1, \ldots, b_n)$ from s_0 such that p_a and p_b result in the same state when applied to s_0.*

Proof. We prove by induction on the number of actions. When $n = 1$, since the only action in the path p is a follow-up

action of no-op, p is also a stratified path. Now we assume the proposition is true for $n = k, k \geq 1$ and prove the case when $n = k + 1$.

For a valid path $p^0 = (a_1, \ldots, a_{k+1})$, by our induction hypothesis, we can permute the first k actions to obtain a stratified path $(a_1^1, \ldots, a_k^1)$.

Now we consider a new path $p^1 = (a_1^1, \ldots, a_k^1, a_{k+1})$. If we have $\mathcal{L}(a_{k+1}) \leq \mathcal{L}(a_k^1)$, or $\mathcal{L}(a_{k+1}) > \mathcal{L}(a_k^1)$ and a_{k+1} is a follow-up action of a_k^1, then p^1 is already a stratified path.

Now we focus on the case when $\mathcal{L}(a_{k+1}) > \mathcal{L}(a_k^1)$ and a_{k+1} is not a follow-up action of a_k^1. Consider a new path $p^2 = (a_1^1, \ldots, a_{k-1}^1, a_{k+1}, a_k^1)$. From Lemma 1, we know that p^2 is a valid path leading to the same state as p^1 does.

By our induction hypothesis, we can permute the first k actions of p^2 to obtain a stratified path $(a_1^2, \ldots, a_k^2)$. Define $p^3 = (a_1^2, \ldots, a_k^2, a_k^1)$.

Comparing p^2 and p^3, we know that $\mathcal{L}(a_{k+1}) > \mathcal{L}(a_k^1)$, namely, the level of the last action in p^2 is strictly larger than that in p^3. We can repeat the above process to generate p^4, $p^5, \cdots$, as long as p^j, $(j \in Z^+)$ is not a stratified path. For each p^j, the first k actions is a stratified path. Also, every p^j is a valid path that leads to the same state as p^0 does.

Since we know that the level of the last action in p^j is monotonically decreasing as j increases, such a process must stop in a finite number of iterations. Suppose it stops at $p^m = (a'_1, \ldots, a'_k, a'_{k+1}), m \geq 1$. We must have that $\mathcal{L}(a'_{k+1}) \leq \mathcal{L}(a'_k)$, or $\mathcal{L}(a'_{k+1}) > \mathcal{L}(a'_k)$ and a'_{k+1} is a follow-up action of a'_k. Hence p^m is a stratified path and the induction step is proved. □

Theorem 2. *For a SAS+ task, a complete search is still complete when combined with stratified_expansion(), and an optimal search is still optimal when combined with stratified_expansion().*

Proof. For any search algorithm, we define its search graph as a graph where each vertex is a state and there is an arc from s to s' if and only if s' is expanded as a successor state of s during the search. For a complete search, if it can find a solution path p in the original search graph, then according to Theorem 1, there is another path p' in the search graph of the stratified search. Therefore, the complete search combined with stratified_expansion() will find p'.

If a search is optimal, then when it is combined with stratified_expansion(), it will find an optimal path p' in the search graph of the stratified search. According to Theorem 1, if the length of the optimal path in the original search graph is n, there must exist a path in the search graph of the stratified search with length n. Hence, the length of p' is n and the new search is still optimal. □

6 Experimental Results

We test on STRIPS problems in the recent International Planning Competitions (IPCs). We implement our stratification analysis and stratified search on top the Fast Downward planner [Helmert, 2006] which gives SAS+ encoding of planning problems. We still use the causal graph heuristic and only modify the state expansion part. On a PC with a 2.0 GHz Xeon CPU and 2GB memory, we set a time limit of 300 seconds for all problem instances.

ID	Fast Downward		∞-stratification		2-stratification	
	Nodes	Time	Nodes	Time	Nodes	Time
zenotravel1	10	0	5	0	5	0
zenotravel1	122	0	23	0	23	0
zenotravel3	723	0	236	0	236	0
zenotravel4	455	0	194	0	194	0
zenotravel5	884	0	479	0	479	0
zenotravel6	1895	0	785	0	785	0
zenotravel7	1468	0	883	0	883	0
zenotravel8	1795	0.04	1828	0.02	1828	0.02
zenotravel9	2017	0.04	2938	0.04	2938	0.04
zenotravel10	4218	0.05	8708	0.04	8708	0.04
zenotravel11	3485	0.02	3429	0.02	3429	0.02
zenotravel12	5002	0.06	7671	0.04	7671	0.04
zenotravel13	9654	0.07	6911	0.12	6911	0.12
zenotravel14	495266	0.26	49623	0.18	49623	0.18
zenotravel15	23853	0.52	1254	0.39	1254	0.45
drivelog1	355	0	51	0	152	0
drivelog2	1450	0	633	0.01	743	0.01
drivelog3	774	0	297	0.01	433	0.01
drivelog4	4692	0.01	1549	0.03	3454	0.02
drivelog5	2879	0.01	576	0.01	957	0.01
drivelog6	2394	0	577	0	1442	0.01
drivelog7	1707	0.02	4341	0.03	1948	0.01
drivelog8	531	0	57006	0.35	4372	0.02
drivelog9	18920	0.02	3808	0.03	26991	0.24
drivelog10	10356	0.02	4317	0.04	8965	0.07
drivelog11	3755	0.05	2435	0.04	3616	0.06
drivelog12	64714	0.31	21252	0.28	135518	2.18
drivelog13	10995	0.13	5659	0.14	9333	0.22
drivelog14	14344	0.06	3195	0.1	7388	0.21
drivelog15	140305	1.25	14371	1.39	14369	0.79
drivelog16	1554010	44.49	180020	29.39	-	-
drivelog17	2218657	51.33	860136	33.81	3986057	167.69

Table 1: Comparison of Fast Downward and two stratification strategies. We give the number of generated nodes and CPU time in seconds. "-" means timeout after 300 seconds.

In practice, how to determine the granularity of stratification is an important issue. We test two extremal cases in our experiments. On one extreme, we test **∞-stratification**, which performs a topological sort on the CCG and treats each SCC as a layer. This represents the finest granularity of stratification. On the other extreme, we test **2-stratification**, which partitions the CCG into two layers and represents the coarsest granularity of stratification. We also implement a factor $\gamma, 0 < \gamma < 1$ for 2-stratification, which specifies the ratio of the number of state variables in Layer 1 to the total number of state variables. We topologically sort the CCG and find the dividing point that gives a ratio closest to γ. We use $\gamma = 0.7$ in our experiments.

The results are shown in Tables 1 and 2. We did not include the domains, such as pipesworld and freecell, where the CG is only one SCC and cannot be stratified. We can see that both ∞-stratification and 2-stratification can give reduction for most problem instances. The reduction of the number of generated nodes can be more than an order of magnitude. Comparing ∞-stratification to 2-stratification, we see that they give similar performance. Despite the reduction in number of generated nodes, the CPU time reduction is more modest. This is due to the fact that our preliminary implementation is not efficient. For example, we check whether an

action is a follow-up action of another one at each state, although a preprocessing phase will save much time. We will develop more efficient implementations in our future work.

7 Discussions

The idea of stratified planning can be explained by looking at a simple 2-stratification. In a 2-stratification, all the state variables are divided into two groups, U_1 and U_2, where U_1 depends on U_2. Therefore, during the search, whenever we expand an action a in U_2, there are only two purposes: to transform a state in U_2 to a goal state, or to provide a precondition for an action in U_1. Therefore, we allow to further expand actions in U_2 but do not allow actions in U_1 except those directly supported by a. In other words, we do not expand any action in U_1 that is not a follow-up action of a because it is a "loose" partial order that can be pruned.

From the above, we see that stratified search can avoid redundant orderings between ancestor/offspring SCCs. Besides that, another source of reduction is that stratified planning imposes certain partial orders between sibling SCCs. For example, in Figure 1b, the SCCs numbered 1 to 5 are siblings in the CCG. However, after we stratify the CCG as in Figure 1c, we impose certain partial orders. For example, we are forced to place actions in SCC 1 before SCC 5 whenever possible. Such a reduction can be significant for many domains.

Acknowledgement

This work is supported by NSF grant IIS-0713109, a DOE ECPI award, and a Microsoft Research New Faculty Fellowship.

References

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

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

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

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

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

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

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

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

ID	Fast Downward		∞-stratification		2-stratification	
	Nodes	Time	Nodes	Time	Nodes	Time
depots1	117	0.01	34	0.01	46	0.01
depots2	1647	0.02	390	0.02	432	0.02
depots3	150297	3.28	42363	3.13	43238	2.65
depots4	295799	6.16	53191	5.37	185819	13.2
depots5	1754366	79.46	67146	9.98	70296	7.39
depots7	926076	21.79	64395	5.09	223034	10.32
depots10	-	-	21544464	211.1	27737	1.08
tpp1	8	0	5	0	3	0
tpp2	20	0	7	0	7	0
tpp3	40	0	15	0	17	0
tpp4	67	0	24	0	28	0
tpp5	139	0	51	0	52	0
tpp6	1081	0.04	1132	0.02	1949	0.01
tpp7	12444	0.11	1436	0.02	4282	0.08
tpp8	20536	0.19	14060	0.49	7373	0.14
tpp9	24641	0.3	4128	0.21	7926	0.48
tpp10	298225	3.14	175383	2.5	130502	2.12
truck1	356	0.01	426	0	426	0
truck2	1664	0.02	493	0.04	493	0.04
truck3	6676	0.02	5475	0.04	5475	0.04
truck4	991625	11.66	41066	0.48	41066	0.48
truck5	13313	0.65	6561	0.07	6561	0.07
truck6	238523	3.7	27267	0.30	27267	0.30
truck7	612647	4.89	74485	1.47	74485	1.47
truck8	22827	0.3	37491	0.5	37492	0.5
truck9	-	-	5090375	212.26	5090375	132.26
truck10	-	-	528454	13.44	528254	11.08
truck11	-	-	926217	11.12	926217	8.17
truck12	-	-	928489	16.55	928489	12.57
truck14	-	-	1631138	23.0	1631138	15.1
truck15	-	-	928489	16.63	-	-
satellite01	226	0	97	0	91	0
satellite02	512	0	240	0	233	0
satellite03	1551	0.01	2168	0.01	744	0.01
satellite04	5036	0.01	2470	0.01	2342	0.01
satellite05	7455	0.02	3674	0.01	3483	0.02
satellite06	20452	0.04	10049	0.02	9681	0.03
satellite07	52902	0.08	26212	0.08	26012	0.04
satellite08	54250	0.11	27118	0.1	26940	0.04
satellite09	104051	0.18	1268	0.12	51173	0.12
satellite10	318621	1.54	129971	1.01	531861	0.78
rover01	228	0	78	0	128	0
rover02	263	0	114	0	102	0
rover03	617	0	218	0	199	0
rover04	225	0	88	0	95	0
rover05	2106	0.01	763	0.01	890	0
rover06	419655	3.84	319500	11.16	572341	11.86
rover07	3659	0.02	1396	0.03	1420	0.05
rover08	20480	0.11	2736	0.06	6003	0.71
rover09	-	-	6003	0.04	29477	0.72
rover10	49789	0.3	22023	0.41	19024	0.38

Table 2: Comparison of Fast Downward and two stratification strategies. We give the number of generated nodes and CPU time in seconds. "-" means timeout after 300 seconds.