Report Number: WUCSE-2008-14

2008-01-01

Faster Optimal State-Space Search with Graph Decomposition and Reduced Expansion

Yixin Chen

Traditional AI search methods, such as BFS, DFS, and A*, look for a path from a starting state to the goal in a state space most typically modelled as a directed graph. Prohibitively large sizes of the state space graphs make optimal search difficult. A key observation, as manifested by the SAS+ formalism for planning, is that most commonly a state-space graph is well structured as the Cartesian product of several small subgraphs. This paper proposes novel search algorithms that exploit such structure. The results reveal that standard search algorithms may explore many redundant paths. Our algorithms provide an... **Read complete abstract on page 2.**

Faster Optimal State-Space Search with Graph Decomposition and Reduced Expansion

Yixin Chen

Complete Abstract:

Traditional AI search methods, such as BFS, DFS, and A*, look for a path from a starting state to the goal in a state space most typically modelled as a directed graph. Prohibitively large sizes of the state space graphs make optimal search difficult. A key observation, as manifested by the SAS+ formalism for planning, is that most commonly a state-space graph is well structured as the Cartesian product of several small subgraphs. This paper proposes novel search algorithms that exploit such structure. The results reveal that standard search algorithms may explore many redundant paths. Our algorithms provide an automatic and mechanical way to remove such redundancy. Theoretically we prove the optimality and complexity reduction of the proposed algorithms. We further show that the proposed framework can accommodate classical planning. Finally, we evaluate our algorithms on various planning domains and report significant complexity reduction.

2008-14

Faster Optimal State-Space Search with Graph Decomposition and Reduced Expansion

Authors: Yixin Chen

Corresponding Author: chen@cse.wustl.edu

Web Page: http://www.cse.wustl.edu/~chen

Abstract: Traditional AI search methods, such as BFS, DFS, and A*, look for a path from a starting state to the goal in a state space most typically modelled as a directed graph. Prohibitively large sizes of the state space graphs make optimal search difficult.

A key observation, as manifested by the SAS+ formalism for planning, is that most commonly a state-space graph is well structured as the Cartesian product of several small subgraphs. This paper proposes novel search algorithms that exploit such structure. The results reveal that standard search algorithms may explore many redundant paths. Our algorithms provide an automatic and mechanical way to remove such redundancy. Theoretically we prove the optimality and complexity reduction of the proposed algorithms. We further show that the proposed framework can accommodate classical planning. Finally, we evaluate our algorithms on various planning domains and report significant complexity reduction.

Type of Report: Other

Faster Optimal State-Space Search with Graph Decomposition and Reduced Expansion

Yixin Chen
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
chen@cse.wustl.edu

Abstract

Traditional AI search methods, such as BFS, DFS, and A^*, look for a path from a starting state to the goal in a state space most typically modelled as a directed graph. Prohibitively large sizes of the state space graphs make optimal search difficult. A key observation, as manifested by the SAS+ formalism for planning, is that most commonly a state-space graph is well structured as the Cartesian product of several small subgraphs. This paper proposes novel search algorithms that exploit such structure. The results reveal that standard search algorithms may explore many redundant paths. Our algorithms provide an automatic and mechanical way to remove such redundancy. Theoretically we prove the optimality and complexity reduction of the proposed algorithms. We further show that the proposed framework can accommodate classical planning. Finally, we evaluate our algorithms on various planning domains and report significant complexity reduction.

Introduction

State-space search is a fundamental approach to planning. The state space is typically modelled as a directed graph in which the search tries to find a minimum-cost path from an initial state to a goal state.

A key observation that motivates this paper is that the state-space is not a random graph. Rather, in most domains, the state space graph is the Cartesian product of multiple smaller graphs. This point is best manifested by the SAS+ formalism (Bäckström & Nebel 1995) of planning, in which a state is represented by the assignments to a set of variables, and the state-space graph is composed as the Cartesian product of the domain transition graphs (DTGs), one for each variable.

For example, in the Airport domain, each plane traverses in a graph modelling the airport runway under certain non-blocking constraints. If the airport graph has 50 gates and there are 20 planes, the search graph has up to 50^{20} states, as a product of 20 small graphs.

The key question is, *if a state-space graph is a Cartesian product of graphs, can we make the search faster by exploiting such structure?* We propose novel algorithms to answer this question.

Much research has focused on the search control mechanisms, such as BFS, DFS, A^*, and the heuristic functions. Our algorithm, however, focuses on another important component, node expansion, of search. When searching in a Cartesian product of subgraphs, expanding a node amounts to expanding all the subgraphs, which we reveal is often wasteful. Instead, we find that we can expand only a subset of subgraphs that form a dependency closure and still guarantee optimality of search. We prove that the proposed algorithm is a general, theoretically sound principle that can be used to speedup search for a wide range of problems.

In the following, we first introduce several constrained problems whose search space is a Cartesian product of graphs. Then, we develop new algorithms that exploit the structure of search space and prove their optimality. We further show that the constrained search problems we study is general enough to model SAS+ classical planning. Finally, we perform experimental study on planning domains from the recent planning competitions and show significant improvements.

Graph Models

We consider the following model of state-space search.

Definition 1 *A* ***state-space graph*** *is a directed graph G. We use $V(G)$ to denote the set of vertices and $E(G)$ the set of edges in G. A search problem defines an* ***initial state*** *$u \in V(G)$ and a* ***goal test*** *$\pi(s) : V(G) \mapsto \{0,1\}$ which takes a state $s \in V(G)$ as input and outputs 1 if s is a* ***goal state****.*

Definition 2 *The* ***Cartesian product*** *$G \times H$ of two graphs G and H is a graph such that $V(G \times H) = V(G) \times V(H)$ and there is an edge between two vertices (u, u') and (v, v') in $G \times H$ if and only if either $u = v$ and $(u', v') \in E(H)$, or $u' = v'$ and $(u, v) \in E(G)$. An example of the operation is shown in Figure 1.*

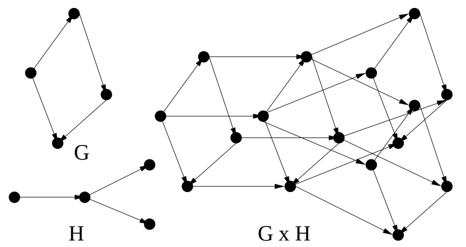

Figure 1: Cartesian products of two graphs. Currently most search algorithms work directly on $G \times H$. We investigate the problem if the search can be made faster given the decomposition into much smaller subgraphs.

The basic problem $\mathcal{P}_0$

Now consider a state-space search on a graph G that is a Cartesian product of multiple subgraphs:

$$G = G_1 \times G_2 \times \cdots \times G_N \tag{1}$$

The basic problem ($\mathcal{P}_0$) is given below. In $\mathcal{P}_0$, the initial state is $u = (u_1, \cdots, u_N) \in V(G)$, where $u_i \in V(G_i)$ for $i = 1, \cdots, N$. In the paper, we assume that the goal test is **decomposable**, which means a state $g = (g_1, \cdots, g_N) \in V(G)$ is a goal state if and only if $\pi_i(g_i) = 1, \forall i = 1, \cdots, N$, where $\pi_i(\cdot)$ is the goal test function for G_i.

A path is a feasible **solution path** if it contains vertices $\{v^1, v^2, \cdots, v^k\}$, where $v^j \in V(G)$, $(v^{j-1}, v^j) \in E(G), j = 2, \cdots, k$, $v^1 = u$ and $\pi(v^k) = 1$.

The problem $\mathcal{P}_0$ is to find an **optimal solution path** that minimizes an objective function $F(v^1, v^2, \cdots, v^k)$. We assume that the **objective function** is separable:

$$F(v^1, v^2, \cdots, v^k) = \sum_{j=2}^{k} f(v^{j-1}, v^j), \tag{2}$$

where $f : E(G) \mapsto R^+$ is a non-negative weight assigned to each edge.

We note that each of $v^j, j = 1, \cdots, k$ is composed of N subgraph states, written as $v^j = (v_1^j, \cdots, v_N^j)$ where $v_i^j \in V(G_i), \forall i = 1, \cdots, N$. According to the definition of Cartesian product, for each $j = 2, \cdots, k$, v^{j-1} and v^j differ by exactly one of the subgraph states. We call the edge (v^{j-1}, v^j) an **l-transition** if v^{j-1} and v^j differ by the l^{th} subgraph state, i.e. $v_i^{j-1} = v_i^j$ if $i \neq l, i = 1, \cdots, N$ and $(v_l^{j-1}, v_l^j) \in V(G_l)$.

Definition 3 *The **contraction mapping** operation* $\varpi : E(G) \mapsto E(G_l)$ *is defined as:*

$$\varpi\big[(v^{j-1}, v^j)\big] = (v_l^{j-1}, v_l^j), \tag{3}$$

where (v^{j-1}, v^j) is an l-transition.

Given an initial vertex $u \in V(G)$, we can define the ***contraction mapping for a path*** as:

$$\varpi[(v^1, \cdots, v^k)] = (e_1, \cdots, e_{k-1}), \tag{4}$$

where $e_i = \varpi[(v^i, v^{i+1})], i = 1, \cdots, k-1$.

We further assume the edge weight is **context-free** in the sense that the weight of an l-transition $e \in E(G)$ depends only on the transition in G_l. That is, each edge in a subgraph $G_i, 1 \leq i \leq N$ has a weight f and we have: $f(e) = f(\varpi[e])$ for any $e \in E(G)$.

Conversely, given a transition $(u_l, w_l) \in E(G_l)$ in a subgraph $G_l, l = 1, \cdots, N$, we can map it back to a state (vertex) in the graph G for a given state $v = (v_1, \cdots, v_N)$ in G.

Definition 4 *The **expansion mapping*** $\rho : V(G) \times E(G_i) \mapsto V(G)$ *is defined as follows.* $\rho[v, (u_l, w_l)]$ *is undefined when $v_l \neq u_l$. When $v_l = u_l$, we have:*

$$\rho[v, (u_l, w_l)] = v', \tag{5}$$

where $v' \in V(G)$ is the vertex where $v'_l = w_l$, and $v'_i = v_i$ for $1 \leq i \leq N, i \neq l$.

Similarly, we can expand a sequence of subgraph transitions into a path in G. That is, given a number of edges $(e_1, \cdots, e_{k-1})$, where each e_i is an edge in a subgraph, given an initial vertex $u \in V(G)$, we can define the ***expansion mapping for a path*** as:

$$\rho[u, (e_1, \cdots, e_{k-1})] = (v^1, \cdots, v^k), \tag{6}$$

where $v^1 = u$ and for $j = 2, \cdots, k$,

$$v^j = \rho[v^{j-1}, e_j]. \tag{7}$$

The operation in (6) is undefined if any of the mappings in (7) is undefined according to Definition 4.

If we apply a complete search directly to the space modelled by G, the worst-case time complexity will be

$$\begin{aligned} & \Theta(|V(G)| + |E(G)|) \\ = \quad & \Theta(\prod_{i=1}^{N} |V(G_i)| + \prod_{i=1}^{N} |E(G_i)|). \end{aligned} \tag{8}$$

$\mathcal{P}_0$ can be solved faster by a decomposed search that finds an optimal path from u_i to g_i, $\pi_i(g_i) = 1$, in subgraph G_i, for $i = 1, \cdots, N$. The solution to $\mathcal{P}_0$ is to simply merge all the paths in an arbitrary way. The complexity of the decomposed search is

$$\Theta(\sum_{i=1}^{N} |V(G_i)| + \sum_{i=1}^{N} |E(G_i)|). \tag{9}$$

Comparing (9) against (8), we see that the complexity of the decomposed search is much lower than direct search. $\mathcal{P}_0$ is simple to solve since there is no inter-graph constraints so that the search can be completely decomposed. Inter-graph constraints are essential for modeling complex real-world problems.

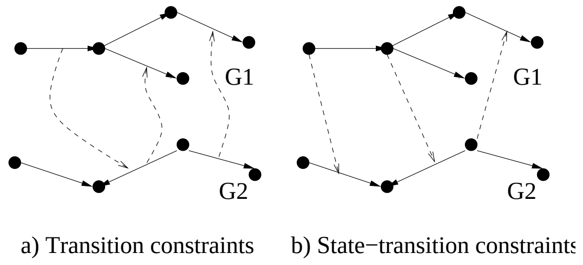

Figure 2: Illustration of constrained problems on Cartesian products of two subgraphs G_1 and G_2. In a) the dashed arrows show the partial orders among subgraph edges. In b) the arrows show the preconditions of subgraph edges.

Constrained problems $\mathcal{P}_1$-$\mathcal{P}_3$

We comment that there could be many types of inter-graph constraints. The goal of this paper is not to find which type of constraints is the best in practice, as there is always the trade-off between expressiveness and efficiency. Rather, we present three what we believe to be representative constraint models and show that: 1) the structure of a search space being a Cartesian product of subgraphs can provide a fundamental mechanism for reducing the search complexity, and 2) The models we consider here are expressive enough to encode classical planning.

$\mathcal{P}_1$ **with transition constraints.** On top of $\mathcal{P}_0$, $\mathcal{P}_1$ defines a set of transition constraints, which specify partial orders between edges in different subgraphs.

Formally, there is a set Φ of partial orders:

$$\Phi = \{((v, v'), (u, u'))\}, \tag{10}$$

where $(v, v') \in E(G_i), (u, u') \in E(G_j)$, $1 \le i, j \le N, i \ne j$. The problem is to find an optimal path $(v^1, \cdots, v^k)$ for which there does not exist $i, j, 2 \le i < j \le k$ such that

$$\left(\varpi[(v^{j-1}, v^j)], \varpi[(v^{i-1}, v^i)]\right) \in \Phi. \tag{11}$$

Figure 2a illustrates the problem.

$\mathcal{P}_2$ **with state-transition constraints.**

On top of $\mathcal{P}_0$, $\mathcal{P}_2$ defines a set of preconditioning subgraph states for some edges in the subgraphs. For each subgraph $G_i, i = 1, \cdots, N$, each transition $(v, u) \in E(G_i)$, we define a **precondition vector**

$$\Omega[(v, u)] = (w_1, \cdots, w_N), \tag{12}$$

where each $w_l, 1 \le l \le N$ is either a vertex in $V(G_l)$ or NULL when $l \ne i$, and $w_i =$ NULL.

For an edge $(v, u) \in E(G_i)$, $i = 1, \cdots, N$, we define its **precondition set** $pre(v, u)$ to be the set of subgraph vertices in $\Omega[(v, u)]$ that are not NULL.

Algorithm 1: Subpath_composition_search (SCS)

```
  Input: a constrained problem $\mathcal{P}_1$, $\mathcal{P}_2$, or $\mathcal{P}_3$
  Output: an optimal solution path
1 for $i = 1, \cdots, N$ do
2     find the transition path set $P_i$ of subgraph $G_i$;
3 foreach $(p_1, \cdots, p_N), p_i \in P_i, i = 1, \cdots, N$ do
4     if compose_feasible_plan($p_1, \cdots, p_N$) then
5         update the incumbent solution path;
6 return the best solution path;
```

The problem is to find an optimal path $(v^1, \cdots, v^k)$ such that, for each $2 \le j \le k$, let vector $\mathbf{w} = \Omega[(v^{j-1}, v^j)]$, $\forall i = 1, \cdots, N$,

$$v_i^{j-1} = w_i \text{ or } w_i = \text{ NULL }. \tag{13}$$

As shown in Figure 2b, those non-NULL elements in a precondition vector define a set of "preconditions" that have to be satisfied when an edge in a subgraph is "executed" in a path.

$\mathcal{P}_3$ **with state constraints.** $\mathcal{P}_3$ is an extension of $\mathcal{P}_0$. On top of $\mathcal{P}_0$, $\mathcal{P}_3$ defines a set of mutual exclusion constraints between two nodes in different subgraphs. Formally, there is a set χ of mutual exclusion pairs:

$$\chi = \{(v, u)\}, \ v \in V(G_i), u \in V(G_j), \tag{14}$$

where $1 \le i, j \le N, i \ne j$.

A state $v \in V(G) = (v_1, \cdots, v_N)$ is an **invalid state** if and only if there exist $1 \le i, j \le N, i \ne j$ such that $(v_i, v_j) \in \chi$. The problem is to find an optimal solution path with no invalid state in it.

Fast Search Algorithms for $\mathcal{P}_1$-$\mathcal{P}_3$

We note that $\mathcal{P}_2$ and $\mathcal{P}_3$ can be solved by a direct search on the overall graph G. The complexity of a standard search algorithm, such as BFS, DFS, or A^*, will be bounded by (8).

To exploit the space structure, we propose two algorithms. The first algorithm, subpath composition search (SCS), has certain limitations, but offers insight into why graphs composed by subgraphs can be searched faster. The second algorithm, reduced expansion search (RES), is our main contribution which safely prunes the search space without sacrificing optimality.

Subpath composition search (SCS)

We outline the framework of the SCS algorithm in Algorithm 1. The idea of the algorithm is simply to find all subpaths from the initial state to a goal state for each individual subgraph and merge them.

For each subgraph G_i, $i = 1, \cdots N$, the transition path set P_i in Line 2 of Algorithm 1 contains all such subpaths that starts from the initial state u_i to a goal state. Namely, we have:

$P_i = \{p | p$ is a path in G_i from u_i to a goal state$\}$.

The compose_feasible_plan() procedure tries to merge the N subpaths into a single solution path to the original problem. The procedure depends on the constraints. We give an algorithm for each of $\mathcal{P}_1$ and $\mathcal{P}_2$.

In the following, we assume the input to compose_feasible_plan() is $(p_1, \cdots, p_N), p_i \in P_i, i = 1, \cdots, N$, where each p_i a subpath in G_i. Let

$$p_i = (\xi_{i,1}, \xi_{i,2}, \cdots, \xi_{i,k_i}). \tag{15}$$

We also assume that any subpath in P_i is canonical, i.e. each vertex appears at most once in a subpath.

Compose subpaths for $\mathcal{P}_1$. For $\mathcal{P}_1$, a possible implementation of compose_feasible_plan() is the following. Given $(p_1, \cdots, p_N), p_i \in P_i, i = 1, \cdots, N$, we construct a directed graph H in which there is one vertex for each edge $(\xi_{i,j-1}, \xi_{i,j})$, for all $1 \le i \le N$ and $j = 2, \cdots, k_i$. In H, there is an edge from $(\xi_{i,j-1}, \xi_{i,j})$ to $(\xi_{i,j}, \xi_{i,j+1})$ for all $1 \le i \le N$ and $j = 2, \cdots, k_i - 1$. Plus, as illustrated in Figure 3b, for each partial order pair in the Φ set defined in (10), there is an edge from (v, v') to (u, u').

Next, we perform a topological sort on the graph H to get a sorted list of subgraph transitions. Finally, we use the expansion mapping defined in (6) to get a solution path. The time complexity of topological sort on a graph H is $\Theta(|V(H)| + |E(H)|)$. Let L be the maximum length of the subpaths p_1 to p_N. Since in H there are no more than NL vertices and N^2L^2 edges, the time complexity of compose_feasible_plan() is $T_{comp} = \Theta(N^2L^2)$. Note that L cannot exceed the maximum $|V(G_i)|$ for $1 \le i \le N$.

Compose subpaths for $\mathcal{P}_2$. For $\mathcal{P}_2$, a possible implementation of compose_feasible_plan() is the following. Given $(p_1, \cdots, p_N), p_i \in P_i, i = 1, \cdots, N$, we construct a directed graph H in which there is one vertex for each edge $(\xi_{i,j-1}, \xi_{i,j})$, for all $1 \le i \le N$ and $j = 2, \cdots, k_i$. In H, there is an edge from $(\xi_{i,j-1}, \xi_{i,j})$ to $(\xi_{i,j}, \xi_{i,j+1})$ for all $1 \le i \le N$ and $j = 2, \cdots, k_i - 1$.

In addition, for each edge (u, v) with an nonempty precondition set, for each vertex w in $pre(u, v)$, suppose $w \in G_i$, we check if w is on p_i. If not, then compose_feasible_plan() returns no solution. Otherwise, let $w = \xi_{i,j}$, we add an edge from $(\xi_{i,j-1}, \xi_{i,j})$ to (u, v) (if $j > 1$) and an edge from (u, v) to $(\xi_{i,j}, \xi_{i,j+1})$ (if $j < k_i$) in H. It is illustrated in Figure 3d.

Finally, we perform a topological sort on the graph H to get a sorted list of subgraph transitions and use the expansion mapping defined in (6) to get a solution path. The way that H is constructed ensures that every edge has all their preconditions satisfied when the solution path is followed. Like $\mathcal{P}_1$, the cost of compose_feasible_plan() for $\mathcal{P}_2$ is bounded by $T_{comp} = \Theta(N^2L^2)$.

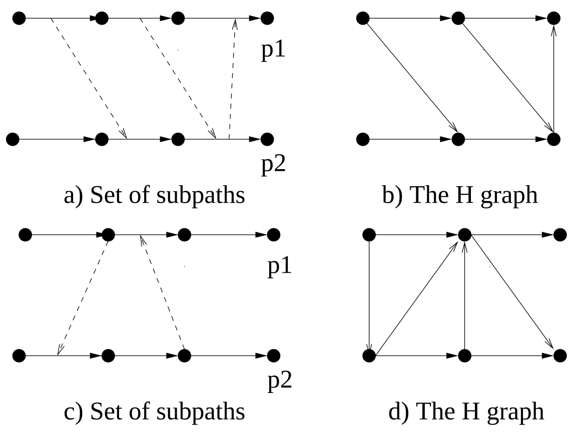

Figure 3: Illustration of the H graphs generated for compose_feasible_plan(). a) and c) show example subpaths for $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. b) and d) show the corresponding H graphs. Note that the vertices in the H graphs correspond to edges in subpaths. The dashed lines in a) and c) show transition constraints and state-transition constraints, respectively.

Compose subpaths for $\mathcal{P}_3$. For $\mathcal{P}_3$, given $(p_1, \cdots, p_N), p_i \in P_i, i = 1, \cdots, N$, we construct a directed graph H in which there is one vertex for each edge $(\xi_{i,j-1}, \xi_{i,j})$, for all $1 \le i \le N$ and $j = 2, \cdots, k_i$. In H, there is an edge from $(\xi_{i,j-1}, \xi_{i,j})$ to $(\xi_{i,j}, \xi_{i,j+1})$ for all $1 \le i \le N$ and $j = 2, \cdots, k_i - 1$. Further, for any two edges (v, v') and (u, u') at different subgraphs, there is an edge from (v, v') to (u, u') in H if $(v', u') \notin \chi$. Finally, in H, introduce a new vertex v_0, add edges from v_0 to the first edge in each subpath, and add edges from the last edge in each subgraph to v_0. Compose_feasible_plan() then tries to find a Hamiltonian path in H. The complexity T_{comp} is exponential in NL.

Complexity analysis for $\mathcal{P}_1$ and $\mathcal{P}_2$. SCS is not efficient for solving $\mathcal{P}_3$ since it requires finding Hamiltonian paths. Now let us check the overall complexity of SCS on $\mathcal{P}_1$ and $\mathcal{P}_2$. The complexity for finding all the subpaths in subgraph G_i is $\Theta(|V(G_i)| + |E(G_i)|)$. The total complexity of SCS is

$$\Theta\left(\sum_{i=1}^{N}(|V(G_i)| + |E(G_i)|) + T_{comp}\prod_{i=1}^{N}|P_i|\right). \tag{16}$$

We compare the complexity of SCS in (16) to that of (8), the complexity of searching directly on the full graph G. We see that the complexity in (16) is better than (8) when the number of paths in each subgraph is less than the size of the subgraph. For each subgraph G_i, define the ratio

$$R_i = \frac{|P_i|}{|V(G_i)| + |E(G_i)|}. \tag{17}$$

We see that, ignoring the first term of ((16) since it is asymptotically negligible comparing to (8), we have:

$$\frac{(16)}{(8)} = T_{comp} \prod_{i=1}^{N} R_i \tag{18}$$

We see that if $R_i < 1$ then SCS saves time for large N, as T_{comp} is only polynomial in N for $\mathcal{P}_1$ and $\mathcal{P}_2$ but $\prod_{i=1}^{N} R_i$ is exponential in N. For example, if there are $N = 30$ subgraphs, each subgraph has 50 vertices and 150 edges, each subgraph generates 10 subpaths from the initial state to the goal state, then $R_i = 1/20$ and the time reduction is of the order 20^{30}, which is significant and dominates T_{comp} (which is of the order $50^2 \cdot 30^2$). In fact, SCS is effective in reducing the time when $R_i < 1$. Such a property is observed in some planning domains, such as Airport in which each plane has only a few possible routes to reach the destination and the number of subpaths is much less than the number of edges and vertices in the airport topology graph.

There are possible ways to further improve SCS. For example, the search for a solution path in Lines 3-5 in Algorithm 1 can be replaced by a branch-and-bound style algorithm which branches on the selection of p_i sequentially. For some given subpaths $p_1, \cdots, p_i$, $i < N$, if there is a loop in the partially constructed graph H, then there is no valid solution to the topological sort, regardless of the choice of $p_{i+1}, \cdots, p_N$. Therefore, we can prune the search. Another technique is to use the cost of the incumbent solution to prune branches that can not yield better solution paths. Finally, it is possible to use existing heuristics to order the subpaths, with the hope that high-quality solution paths can be found earlier and provide stronger pruning.

Limitations of SCS. The SCS algorithm and the above analysis give insights as to how a Cartesian-product structure of a state-space graph may lead to significant reduction of search complexity. However, the SCS algorithm is not very practical for two reasons.

First, for some problems with dense subgraphs, the number of subpaths in each subgraph may be larger than the size of the subgraph, leading to $R_i > 1$ for some i so that SCS becomes less favorable.

Second, SCS makes the canonicality assumption that any subpath can pass each vertex at most once, which can be restrictive for some real-world problems, especially those with cycles in the subgraphs.

To overcome the limitations of SCS, we next propose a practical algorithm, reduced expansion search (RES),

Algorithm 2: State_space_search

```
Input: State space graph $G$
Output: $p$, an optimal solution path
1 closed $\leftarrow$ an empty set;
2 insert the initial vertex $u$ into $open$;
3 while $open$ is not empty do
4     $v \leftarrow$ remove-first($open$);
5     if $\pi(v) = 1$ then process solution_path($v$);
6     if $v$ is not in closed then
7         add $v$ to $closed$;
8         $open \leftarrow$ insert(expand($v$), $open$);
```

that still exploits the Cartesian-product structure and reduces the complexity for both dense and sparse subgraphs, without requiring the canonicality assumption.

Reduced expansion search (RES)

The idea of RES is to safely prune the search space by recognizing the structure of Cartesian products. We need the following definition first.

Definition 5 *Given a state-space graph G defined in Definition 1, for a vertex $v \in V(G)$, an edge $e \in E(G)$ ($resp.$ vertex $w \in V(G)$) is a* ***potential descendent edge*** *($resp.$ vertex) of v if there exists a feasible solution path from v to a goal state that contains e ($resp.$ w). We denote this relationship as $v \lhd e$ ($resp.$ $v \lhd w$).*

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

Node expansion. We characterize a wide range of standard search algorithms by Algorithm 2. Different search algorithms differ by the remove-first() operation which gets one node from the *open* list. Algorithm 2 is DFS when *open* is a FIFO queue, BFS when *open* is a FILO queue, best-first search (including A*) when *open* is a priority queue ordered by a heuristic function.

Algorithm 2 is always optimal if we record all solution paths and finally report the best path. However, in some cases, the first solution is optimal. For example, BFS is optimal when all edges have a same cost and A^* is optimal when the heuristic is admissible. In the following, we regard Algorithm 2 as optimal, assuming appropriate schemes are adopted according to the remove-first() implementation.

The RES algorithm we propose modifies the expand() operation. So it can be *combined* with any implementation of remove-first() and any heuristic function

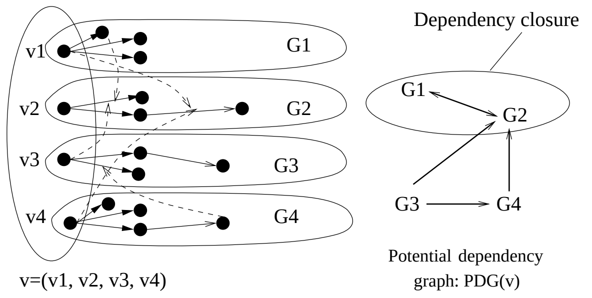

Figure 4: Illustration of expand() on $G = G_1 \times G_2 \times G_3 \times G_4$. The dashed arrows show preconditions of edges.

to form various algorithms. The idea is, by recognizing the structure of G, we reduce the nodes that expand() will generate for some states.

Given a vertex $v \in V(G)$, its set of **children** $ch(v)$ is defined as $ch(v) = \{u|(v,u) \in E(G)\}$.

We define $expand(v)$ as a subset of $ch(v)$ that only expands feasible children with respect to given constraints. For $\mathcal{P}_2$, we check if all the preconditions of an edge are in v and only expand those valid edges.

We examine closely the expand() operation on a graph $G = G_1 \times \cdots \times G_N$. For a current vertex $v = (v_1, \cdots, v_N)$, the next child vertex only differs from v by one of the $v_i, i = 1, \cdots, N$. Let $expand(v_i)$ be the set of valid children of v_i in G_i, we have:

$$\text{expand}(v) = \bigcup_{i=1}^{N} \left\{ \rho[v, (v_i, u_i)] \middle| u_i \in \text{expand}(v_i) \right\}, \quad (19)$$

which is illustrated in Figure 4.

Reduced expansion for $\mathcal{P}_2$. Now we consider $\mathcal{P}_2$.

Definition 6 *For an instance of $\mathcal{P}_2$, given a vertex $v \in V(G)$, let $v = (v_1, \cdots, v_N)$. For any $1 \le i, j \le N, i \ne j$, we call v_i a **potential precondition** of G_j if there exists $e \in E(G_j)$ such that*

$$v_j \lhd e \text{ and } v_i \in pre(e) \quad (20)$$

Definition 7 *For an instance of $\mathcal{P}_2$, given a vertex $v \in V(G)$, let $v = (v_1, \cdots, v_N)$. For any $1 \le i, j \le N, i \ne j$, we call v_i a **potential dependent** of G_j if there exists $e = (v_i, u) \in E(G_i)$ and $w \in V(G_j)$ such that*

$$v_j \lhd w \text{ and } w \in pre(e) \quad (21)$$

Definition 8 *For an instance of $\mathcal{P}_2$, for a vertex $v \in V(G)$, its associated **potential dependency graph** PDG(v) is a directed graph in which each of the subgraphs $G_i, i = 1, \cdots, N$ corresponds to a vertex, and there is an edge from G_i to G_j, $i \ne j$, if and only if v_i is a potential precondition or potential dependent of G_j.*

Definition 9 *For a directed graph H, a subset C of $V(H)$ is a **dependency closure** if there do not exist $v \in C$ and $u \in V(H) - C$ such that $(v,u) \in H$.*

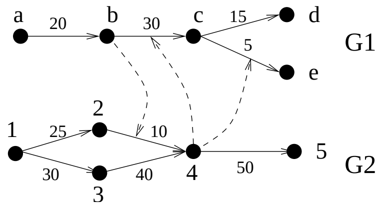

Figure 5: An example problem $\mathcal{P}_2$ with weights on edges.

Figure 4 shows the above definitions. In PDG(v), G_1 points to G_2 as v_1 is a potential precondition of G_2 and G_2 points to G_1 as v_2 is a potential dependent of G_1. We see that G_1 and G_2 form a dependency closure.

The RES algorithm can be described as follows. In the $expand(v)$ operation, instead of expanding all the children as in (19), we only expand the subgraphs that belong to a dependency closure of PDG(v) under the condition that not all subgraphs in the dependency closure are at a goal state and at least one subgraph in the closure has a valid child. If there are more than one such dependency closure, we choose the one with the minimum branching factor. More precisely, we find an index subset $\mathcal{C}(v) \subseteq \{1, \cdots, N\}$ such that the subgraph formed by $G_i, i \in \mathcal{C}(v)$ is a dependency closure in PDG(v) and that not every $v_i, i \in \mathcal{C}(v)$ satisfies $\pi(v_i) = 1$. Such a $\mathcal{C}(v)$ can always be found for any non-goal state v since PDG(v) itself is always a dependency closure. When the set $\mathcal{C}(v)$ is found, the RES algorithm modifies the $expand(v)$ operation to:

$$\text{expand}_r(v) = \bigcup_{i \in \mathcal{C}(v)} \left\{ \rho[v, (v_i, u_i))] \middle| u_i \in \text{expand}(v_i) \right\},$$

An example of RES is shown in Figures 5 and 6. Figure 5 shows two subgraphs and the precondition constraints, where $a1$ is the initial state and $d5$ and $e5$ are goal states. In Figure 6, a) and b) show the original and reduced search graph, respectively. We see that RES significantly reduces the space. For example, at node $a1$, since a is not a potential dependent or precondition of G_2, we can only expand G_1 instead of expanding both subgraphs. As another example, at $b4$, 4 is a potential precondition of G_1, but b is a not potential dependent or precondition of G_2. So at $b4$ G_1 is a dependency closure and we only expand G_1.

We see that the optimal path cost is 140. In the original space, there is redundancy since there are multiple paths with cost 140. In the RES-reduced space we can still find an optimal path (in bold) with cost 140, which shows the correctness of RES pruning.

Theorem 1 *Algorithm 2 using the $expand_r$() operation can optimally solve $\mathcal{P}_2$.*

Proof. We consider the directed graph G' where $V(G') = V(G)$. For each vertex v, there is an edge

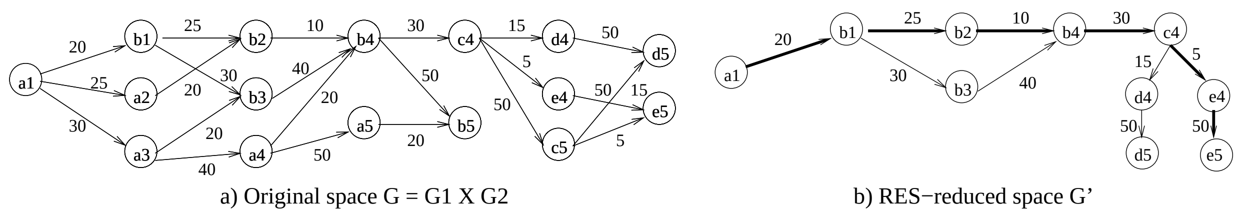

Figure 6: Comparison of the original and RES-reduced search space for solving the problem in Figure 5.

(v, w) if and only if $w \in expand_r(v)$. Using Algorithm 2 with $expand_r()$ to search on G is equivalent to using Algorithm 2 with $expand()$ to search on G'.

Since Algorithm 2 with $expand()$ is optimal, we only need to show that for any initial state u, the cost of an optimal path in G' is the same as the cost of an optimal path in G. We prove this fact by induction on k, the length of the optimal path from the initial state u to a goal state in G.

When $k = 1$, the optimal path in G is (u, g) where g is a goal state. There exists $i, 1 \le i \le N$ such that u_i and g_i are different. According to the definition of $\mathcal{C}(u)$, we have $i \in \mathcal{C}(u)$. Thus, we have that $g \in expand_r(u)$ and that the optimal path (u, g) is also in G'.

When $k > 1$, let an optimal path from u to a goal state g in G be $p^* = (v^0 = u, v^1, \cdots, v^k = g)$. Let (u, v^1) be an l-transition, i.e. $v^1 = \rho[u, (u_l, v^1_l)]$.

We consider two cases. 1) If $l \in \mathcal{C}(u)$, then $v^1 \in expand_r(u)$ and $(u, v^1) \in E(G')$. According to the Principle of Optimality, $p = (v^1, \cdots, v^k)$ is an optimal path from v^1 to a goal state in G. According to the induction assumption, there exists a path p' from v^1 to v^k in G' with the same cost as p. From (2), (u, v^1) followed by p' is a path in G' and has the same cost as p^*.

2) If $l \notin \mathcal{C}(u)$, let (v^{j-1}, v^j) be the first edge in p^* such that (v^{j-1}, v^j) is an m-transition and $m \in \mathcal{C}(u)$. Apply the contraction mapping to p^* and let $\varpi[p^*] = (e_1, \cdots, e_k)$. Consider the path

$$p^{**} = \rho[u, (e_j, e_1, \cdots, e_{j-1}, e_{j+1}, \cdots, e_k)].$$

Since $\mathcal{C}(u)$ is a dependency closure, u_m is not a potential dependent of any $G_i, i \notin \mathcal{C}(u)$. That is, none of the preconditions of e_j is in $G_i, i \notin \mathcal{C}(u)$. On the other hand, $u_i = v^{j-1}_i, \forall i \in \mathcal{C}(u)$ and all the preconditions of e_j are satisfied at at v^{j-1}. Thus, all the preconditions of e_j are satisfied at u.

Let $v' = \rho[u, e_j]$. We know $(e_1, \cdots, e_{j-1})$ is a valid transition sequence starting from v'. This is true because none of $e_1, \cdots, e_{j-1}$ has u_m as a precondition (according to the definition of dependency closure), thus moving e_j before $(e_1, \cdots, e_{j-1})$ will not make their preconditions unsatisfied. Further, since executing $(e_1, \cdots, e_j)$ and $(e_j, e_1, \cdots, e_{j-1})$ from u lead to the same state, $e_{j+1}, \cdots, e_k$ are still valid transitions after $(e_j, e_1, \cdots, e_{j-1})$ are executed.

From the above, we know p^{**} is a valid and optimal path from u in G. According to the Principle of Optimality, we have $p = \rho[v', (e_1, \cdots, e_{j-1}, e_{j+1}, \cdots, e_k)]$ is an optimal path from v' in G. From the induction assumption, we know there is a path p', from v' to v_k in G' with the same cost as p. Since $(u, v') \in E(G')$, we have that (u, v') followed by p' is an optimal path in G' with the same cost as p^* and p^{**}. ■

A note on the complexity reduction. In effect, the RES algorithm searches on the reduced graph G' defined in the above proof. Since $expand_r()$ is always a subset of $expand()$, we have $|E(G')| \le |E(G)|$. Further, although $|V(G')| = |V(G)|$, the vertices reachable from the initial state may be much less in G', as shown in Figures 5 and 6. The number of vertices is reduced from 16 to 10 and number of edges from 22 to 10.

Theorem 2 *For an A^* search on $\mathcal{P}_2$, if the heuristic function is admissible, then using $expand_r()$ always expands less nodes than using $expand()$.*

Theorem 2 can be easily seen. For each node v that A^* with $expand\_r()$ expands, since its $f + h$ is less than the optimal cost, it must also be expanded by the original A^*, but not vice versa. Comparing to SCS, RES does not need the canonicality assumption and can handle cycles in subgraphs.

RES for $\mathcal{P}_1$ and $\mathcal{P}_3$. $\mathcal{P}_1$ cannot be solved by a graph search with a *closed* list since the previous edges along a path need to be remembered. A tree search has to be employed. For the tree search, a RES pruning technique can be used. Given a vertex $v \in V(G)$, there is an edge (G_i, G_j) in PDG(v) iff there exist $(v_i, u) \in E(G_i)$ and $e \in E(G_j)$ such that $v_j \lhd e$ and $(e, (v_i, u)) \in \Phi$.

For $\mathcal{P}_3$, for a vertex $v \in V(G)$, there is an edge (G_i, G_j) in PDG(v) iff there exist $(v_i, v') \in E(G_j)$ and $(u, w) \in E(G_j)$ such that $v_j \lhd (u, w)$ and $(v', u) \in \chi$.

For both $\mathcal{P}_1$ and $\mathcal{P}_3$, with the modified ways to generate PDGs, the same way to form dependency closures and reduce expansions as for $\mathcal{P}_2$ can apply. The optimality can be proved using a similar argument as that for Theorem 1.

Reducing Classical Planning to $\mathcal{P}_2'$

Now we show that the proposed graph model is general enough to encode classical planning problems. There have been recent studies on automatically transforming STRIPS planning tasks into a SAS+ representation (Bäckström & Nebel 1995; Helmert 2006). We sketch the basic definition of SAS+.

A SAS+ planning task is defined over a set of multi-valued **state variables** $X = \{x_1, \cdots, x_N\}$, each with an associated finite domain $\mathcal{D}_x$; and a set of actions $\mathcal{O}$, where each action $o \in \mathcal{O}$ is a triple (pre, add, del) of partial assignments to state variables. The planning task to is find a shortest sequence of actions from a complete assignment to state variables (s_0) to a partial assignment to state variables (a goal state), under the well-known semantics of actions.

We can map a SAS+ planning task into $\mathcal{P}_2'$, a slight extension of $\mathcal{P}_2$. 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 $\mathcal{D}_x$ and edge set $\mathcal{A}_x$. An edge (v, v') belongs to $\mathcal{A}_x$ if and only if there is an action o with $v \in del(o)$, $v \in pre(o)$ and $v' \in add(o)$. Thus, the SAS+ task maps to an instance of $\mathcal{P}_2'$ where the subgraphs are $G_i, i = 1, \cdots, N$, and each edge $(v, v') \in G_i$ corresponding to action o has a set of preconditions as defined in pre(o). It is easy to see that the goal test is decomposable and the plan cost is separable and context free, as assumed by $\mathcal{P}_2$.

We note that $\mathcal{P}_2'$ has two subtle extensions to $\mathcal{P}_2$. First, between each vertex pair (v, v') in a subgraph G_i there may be multiple edges, corresponding to different actions. No change is required to make RES work on such graphs, so long as we expand all the edges between a vertex pair in $expand_r()$ and consider all edges when finding dependency closures. Second, an action o in G_i may also appear in other subgraphs, requiring simultaneous transitions. However, $expand_r()$ in RES still works correctly without compromising optimality. Suppose a subgraph G_j also contains o, then (G_i, G_j) and (G_j, G_i) are both in the PDG, and thus, G_i and G_j are in the same dependency closure. The proof to Theorem 1 still works since all subgraphs containing o are in a dependency closure. The details are omitted here.

Experimental Results

Using the reduction described in the last section, we apply RES to solve STRIPS planning problems in the recent International Planning Competitions (IPCs). We use a preprocessor in Fast Downward (FD) (Helmert 2006) to convert a STRIPS problem into a SAS+ instance with multiple DTGs.

FD contains its own heuristic which is not admissible. To test the effect of RES, we implement a BFS search by setting $h = 0$ in FD. Due to lack of available admissible heuristics, we also implement an A^* search based on $f + 0.2h$, where h is the FF heuristic in FD.

Problem-ID	BFS	BFS+RES	Problem-ID	BFS	BFS+RES
pathway2	17582	2770	pathway3	1.2e7	10046
pipesworld1	1103	1103	pipesworld2	4425	4425
rovers3	35684	5942	rovers4	11428	3152
storage5	1346	1272	storage6	3715	3322
trucks2	39317	19951	trucks3	639121	39317
TPP4	1275	14	TPP5	126322	5692
openstacks4	16389	16353	openstacks5	16389	16353
airport6	1483	1192	Airport7	1475	1188
depot1	2603	2212	depot3	-	153318
driverlog4	-	72891	driverlog5	-	222062
zenotravel2	391	148	zenotravel4	103630	7700

Table 1: Numbers of nodes expanded by BFS and RES on IPC domains. ”-” means no solution is found in 30 minutes.

ID	A^*	A^*+RES	ID	A^*	A^*+RES
open4	4937 (1)	1369 (2)	open8	-	33850 (225)
open9	-	31080(233)	path3	-	5023 (16)
path4	9.8e4 (83)	4225 (17)	pipe9	-	1197 (133)
pipe10	-	1953 (196)	rover4	11428 (1)	486 (1)
rover5	-	28724 (11)	rover6	-	27631 (13)
stor12	-	9.1e4 (723)	stor13	-	6.2e5 (820)
TPP5	26771 (4)	2124 (3)	TPP6	-	82181 (34)
TPP7	-	9.1e4 (44)	TPP9	-	2.6e5 (193)
airp8	685 (1)	334 (10)	airp9	2.2e4 (29)	840 (35)
psr19	5092(2)	3198 (5)	psr20	135 (1)	50 (1)
depot6	-	17014 (384)	depot9	-	4853 (315)
driver5	81630 (1)	23474 (44)	driver7	-	37582 (82)
driver9	-	61300 (152)	zeno5	24282 (2)	1818 (21)
zeno6	87812 (10)	3282 (41)	zeno7	24527 (2)	799 (9)

Table 2: Comparison of A^* and RES. ”-” means timeout after 30 minutes. We give both number of expanded nodes and the CPU time in seconds (in parentheses). We use some abbreviated domain names.

We have validated that $0.2h$ gives an admissible heuristics in the cases we study. Then we implement the RES algorithm, in which we only expand DTGs in a dependency closure at each state.

We test STRIPS domains in recent IPCs and report the results in Table 1. For each domain, we report the two highest numbered instances that can be solved by BFS or RES in 30 minutes. Currently, we use a simple enumeration-based implementation and have not optimized the efficiency of our code. It takes much more time to process a node in RES than BFS. However, finding the dependency closure can be done efficiently by finding strongly connected components (SCC) in the PDG, contracting each SCC into a node and finding zero in-degree nodes. We believe an efficient implementation will lead to little overhead in RES, which will be dominated by the computation of admissible heuristics.

From Table 1, we see that RES can reduce the search space very significantly for most domains. Moreover, for all the problems, RES can find a plan in the reduced space with the same length as the optimal plan found by BFS, which confirms Theorem 1.

In Table 2, we list the results of several highest numbered instances either solver can solve. We see that RES can dramatically reduce node expansion. Although due to the inefficiency of our simple implementation, expanding a node using RES is often much more expensive than before, A^* with RES can still solve much more problems within 30 minutes than without. For example, in the storage domain, A^* can only solve 3 instances while A^*+RES can solve 13.

We believe that RES is of significance because it is theoretically optimal and provides an automatic and mechanical way to reduce the space for optimal search. RES is a general principle for removing redundancy in search and does not require any parameter tuning. RES is orthogonal to and can be combined with the development of better admissible heuristics.

References

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

Helmert, M. 2006. The Fast Downward planning system. *J. Artificial Intelligence Research* 26:191–246.