

Washington University in St. Louis
Washington University Open Scholarship

Engineering and Applied Science Theses &
Dissertations

Engineering and Applied Science

Winter 12-15-2014

Accelerating Heuristic Search for AI Planning

You Xu

Washington University in St. Louis

Follow this and additional works at: http://openscholarship.wustl.edu/eng_etds



Part of the [Engineering Commons](#)

Recommended Citation

Xu, You, "Accelerating Heuristic Search for AI Planning" (2014). *Engineering and Applied Science Theses & Dissertations*. 67.
http://openscholarship.wustl.edu/eng_etds/67

This Dissertation is brought to you for free and open access by the Engineering and Applied Science at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS
School of Engineering and Applied Science
Department of Computer Science and Engineering

Thesis Examination Committee:
Yixin Chen, Chair
Lee Benham
Jeremy Buhler
Chenyang Lu
Yinjie Tang
Weixiong Zhang

Accelerating Heuristic Search for AI Planning

by

You Xu

A dissertation presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

December 2014
Saint Louis, Missouri

© 2014, You Xu

Contents

List of Figures	v
List of Tables	vii
Acknowledgments	ix
Abstract	xi
1 Introduction	1
1.1 AI Planning	2
1.2 Motivation	3
1.2.1 Limitations of Heuristic Search	3
1.3 Contributions	6
1.4 Dissertation Outline	8
2 Background and Related Works	10
2.1 SAS+ Planning	10
2.2 Heuristic Search for Planning	12
2.2.1 Notable Heuristics	13
2.2.2 Helpful Actions and Multiple Heuristics	14
2.3 Notable Non-heuristic Techniques	15
3 Accelerating Heuristic Search with Stratified Planning	17
3.1 Introduction	18
3.2 Stratified Planning	20
3.2.1 Stratification of Planning Problems	21
3.2.2 Stratified Planning Algorithm	24
3.2.3 Theoretical Analysis	27
3.3 Experimental Results	29
3.4 Discussions and Summary	30
4 Accelerating Heuristic Search with Partial Order Reduction	34
4.1 Partial Order Reduction Theory for Planning	35
4.1.1 Space Reduction for Planning	35
4.1.2 Stubborn Set Theory for Planning	37
4.1.3 Commutativity in SAS+ planning	39
4.1.4 Determining Commutativity	42

4.2	Stubborn-Set Theory for Existing POR Algorithms	43
4.2.1	Explanation of EC	44
4.2.2	Explanation of SP	48
4.3	A New POR Algorithm for Planning	52
4.3.1	SAC vs. EC	58
4.4	System Implementation	59
4.5	Experimental Results	61
4.6	Summary	74
5	Accelerating Heuristic Search with Random Walks	75
5.1	Background	76
5.2	Local Properties of Search Space	77
5.2.1	Approaches for Accelerating Plateau Exploration	82
5.3	Random Walk Assisted Best-First Search	82
5.3.1	Algorithm Framework	83
5.3.2	Performance Analysis	85
5.3.3	Parameter Settings	90
5.4	Experimental Results	91
5.4.1	Part I: Results on IPC 6 (2008) Domains	91
5.4.2	Part II: Results on IPC 7 (2011) Domains	95
5.5	Summary	103
6	Accelerating Heuristic Search with Cloud Computing	105
6.1	Background	106
6.1.1	Parallel Computing	106
6.1.2	Cloud Computing	107
6.1.3	Parallel Search Algorithms	107
6.1.4	Stochastic Search	108
6.1.5	Portfolio Search	108
6.2	Portfolio Stochastic Search Framework	108
6.2.1	Monte-Carlo Random Walk (MCRW)	109
6.2.2	Variability in MCRW Searches	110
6.2.3	Portfolio Stochastic Search (PoSS) Algorithm	111
6.2.4	Enhanced PoSS with Dovetailing	113
6.3	System Implementation	114
6.4	Experimental Results	117
6.4.1	Results for Parallel Computing	118
6.4.2	Evaluation in Windows Azure	121
6.5	Summary	124
7	Conclusion	126
7.1	Future Works	127
7.1.1	Symmetry in State Space	127

7.1.2	Helpful Actions	127
7.1.3	Probabilistic Models for Random Walks Guided by Heuristics	128
7.1.4	Cloud-Based Deterministic Search	128
	References	129

List of Figures

1.1	Number of generated nodes by the FastDownward planner on the Driverlog domain	4
3.1	The causal graph, contracted causal graph and stratification of Truck-02.	22
3.2	The CCGs of some instances of several planning domains. Each SCC is labelled by $x\langle y \rangle$, where “ x ” is the index for the SCC and “ y ” is the number of state variables in the SCC. The SCCs that contain goals are in a darker color.	23
4.1	Illustration of condition A1 in Definition 5. The big circle on the left stands for the set of all applicable actions at state s , while the small circle stands for the stubborn set $T(s)$ of s . Action $b \in T(s)$ can always be swapped to the beginning of a path consisting of b_i s without affecting the final state.	38
4.2	Illustration of commutative set.	41
4.3	A SAS+ task with four DTGs. The dashed arrows show preconditions (prevailing and transitional) of each edge (action). Only dashed arrows between DTGs are shown. Actions are marked with letters a to f. We see that b and e are associated with more than one DTG.	44
4.4	The search spaces for a simple SAS+ planning task with two state variables and four states when using SP and EC. SP (on the left) expands all four states while EC (on the right) only expands three. The dashed link on the left graph is the action that is not expanded by SP. Gray nodes are the goal states.	52
4.5	Search spaces of EC and SAC	59
4.6	System architecture of FD and SAC	60
5.1	Random exploration as a concatenation of random advancements. . .	84
5.2	Number of problem instances that are solvable by LAMA, RW-BFS $_s$, and RW-BFS $_p$ in 300 seconds.	92
5.3	Structure of Roamer-p.	97
5.4	Number of problems solved and quality score over time for Roamer, Lama 2008 and Arvand. The x-axis is shown on a logarithmic scale.	99
5.5	Number of problems solved and quality score over time for multi-core planners. The x-axis is shown on a logarithmic scale.	102

6.1	The run-time distribution of 500 MCRW runs with different random seeds on six planning problems.	111
6.2	The average running time of the MCRW algorithm with different parameter settings on problem Airport-17.	114
6.3	System architecture for PoSS in Windows Azure.	115
6.4	A simple Web UI for users to submit planning tasks to PoSS running in Windows Azure.	116

List of Tables

3.1	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.	32
3.2	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.	33
4.1	Supplementary table for Figure 4.3: list of actions and related DTGs.	45
4.2	Comparison of FD, EC, and SAC using A* with h_{max} heuristic on IPC domains. We show numbers of expanded and generated nodes. “-” means timeout after 300 seconds. For each problem, we also highlight the best values of expanded and generated nodes among three algorithms, if there is any difference.	64
4.3	Comparison of FD, EC and SAC with no-preferred operators on IPC’s domains. We show numbers of expanded and generated nodes. “-” means timeout after 1800 seconds. For each problem, we also highlight the best values of expanded and generated nodes among three algorithms if there is any difference.	67
4.4	Comparison of SP and SAC to FD on IPC domains without preferred operators. We show ratios in the table for SP and SAC compared to FD. We use 2-stratification for SP. Numbers in the parentheses after domain names are the number of problems in that domain that we ran experiments on. Smaller values in the table indicate better performance. Domains marked with a * are not stratifiable, which causes SP to roll back to FD.	73
5.1	List of parameters used in RANDOM EXPLORATION	85
5.2	Comparison of the search time (“T”), solution length (“L”), number of heuristic evaluations (“E”) of LAMA, RW-BFS _s and RW-BFS _p . For RW-BFS _p , “E” is the total number of heuristic evaluations of all threads.	93
5.3	Number of solved instances for three planners on IPC 6 domains (Time limit is 300s).	95
5.4	A comparison of Roamer and Roamer-p for IPC 7.	96
5.5	Results of IPC 7 for all planners in the sequential satisficing track. Domain names are shortened to fit the table in a page.	100

5.6	Results of IPC 7 for all planners in the multi-core satisficing track after applying the strict plan validating rule. Domain names are shortened to fit the table in a page.	101
6.1	A comparison between Random Exploration and MCRW.	110
6.2	The mean (μ), the variance (σ^2) and the standard deviation (σ) of 500 MCRW runs with different random seeds on six planning problems.	111
6.3	Comparison of MCRW and PoSS in different number of processors and strategies for the Airport domain. Problems with super linear speedups are highlighted.	118
6.4	Comparison of MCRW and PoSS in different number of processors and strategies for the Pipesworld Tankage domain. Problems with super linear speedups are highlighted.	119
6.5	Comparison of MCRW and PoSS in different number of processors and strategies for the Pipesworld NoTankage domain.	119
6.6	Comparison of MCRW and PoSS in different number of processors and strategies for the Philosophier domain. Problems with super linear speedups are highlighted.	120
6.7	Comparison of MCRW and PoSS in different number of processors and strategies for the Satellite domain.	120
6.8	Comparison of MCRW and PoSS in different number of processors and strategies for the Power Supply Restoration domain. Problems with super linear speedups are highlighted.	121
6.9	Comparison of the running time and cost of PoSS algorithms using different number of nodes in Windows Azure. “T” is the running time in seconds, “S” is the speedup and “C” is the average total cost in US cents.	122

Acknowledgments

First of all, I would like to thank my advisor, Yixin Chen. He convinced me to pursue a Ph.D. in computer science when I was perplexed by the uncertainty ahead. It was the best decision I have ever made. I will forever be thankful to all his guidance, kindness and encouragement that make my Ph.D. experience productive and stimulating.

We have a wonderful research group working on AI planning here at Dr. Chen's lab. I'd like to thank my friends and colleagues, Ruoyun Huang, Qiang Lu, Guohui Yao, Jianxia Chen, Guobin Zou and all other members in our research group for providing an incredible environment for me to work in. Our research group has always been a source of insights and enthusiasm for my research.

I would also like to thank Xueyang Feng, Abusayeed Saifullah, Yinjie Tang, Chengyang Lu and other researchers I have collaborated with. They have broadened my scope and reminded me that intellectual curiosity has no boundary.

I want to express my heartfelt thank you to my better half, Rachel. Her love for life has made me a better person.

You Xu

*Washington University in Saint Louis
December 2014*

Dedicated to my grandparents.

ABSTRACT OF THE DISSERTATION

Accelerating Heuristic Search for AI Planning

by

You Xu

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2014

Professor Yixin Chen, Chair

AI Planning is an important research field. Heuristic search is the most commonly used method in solving planning problems. Despite recent advances in improving the quality of heuristics and devising better search strategies, the high computational cost of heuristic search remains a barrier that severely limits its application to real world problems. In this dissertation, we propose theories, algorithms and systems to accelerate heuristic search for AI planning.

We make four major contributions in this dissertation.

First, we propose a state-space reduction method called Stratified Planning to accelerate heuristic search. Stratified Planning can be combined with any heuristic search to prune redundant paths in state space, without sacrificing the optimality and completeness of search algorithms.

Second, we propose a general theory for partial order reduction in planning. The proposed theory unifies previous reduction algorithms for planning, and ushers in

new partial order reduction algorithms that can further accelerate heuristic search by pruning more nodes in state space than previously proposed algorithms.

Third, we study the local structure of state space and propose using random walks to accelerate plateau exploration for heuristic search. We also implement two state-of-the-art planners that perform competitively in the Seventh International Planning Competition.

Last, we utilize cloud computing to further accelerate search for planning. We propose a portfolio stochastic search algorithm that takes advantage of the cloud. We also implement a cloud-based planning system to which users can submit planning tasks and make full use of the computational resources provided by the cloud.

We push the state of the art in AI planning by developing theories and algorithms that can accelerate heuristic search for planning. We implement state-of-the-art planning systems that have strong speed and quality performance.

Chapter 1

Introduction

Planning is an integral part of human intelligence. It is the conscious process of organizing activities to achieve certain goals. As part of the human cognitive process, planning serves as a fundamental connection between goal setting and action. Through planning, actions are organized toward a clear objective. Acting without planning, to a degree, is no different than monkeys trying to type Shakespeare's Hamlet by hitting random keys on a typewriter.

The ability to plan is also a direct measurement of the intelligence of a machine. In fact, any intelligent machine that exhibits rational behaviors to outside observers must possess planning abilities, for without which actions and behaviors become less purposeful and rational.

There are an abundance of planning problems in our daily life. On a personal level, we plan for activities throughout the day. Modern society also relies on the solutions to various planning problems to function. Our fire department, police and postal services all rely on transportation planning to arrange routes efficiently. Airports, buses and assembly lines also depend on our ability to plan and schedule tasks in complicated systems. Planning has also been applied to many small to medium scale real-world problems, including controlling autonomous robots and unmanned vehicles, scheduling space telescope observations, modeling interventions of biological processes, and scheduling individual and organizational activities [59, 9, 19, 13, 65, 61].

1.1 AI Planning

In Artificial Intelligence (AI) research, planning is formulated as the process of arranging a course of *actions* (sometimes called *activities*) to achieve certain *goals* under given *constraints*. Solving planning problems requires taking into account constraints, action orders, dependencies and plan efficiencies.

While the correct action sequence is easy to get for simple problems, large-scale planning tasks are daunting and sometimes not feasible to tackle without the help of modern computers, as the scale of the problem is beyond our cognitive ability. Specifically, real-world planning problems oftentimes involve complex constraints and tangled action dependencies, rendering intuition and reasoning inadequate. Large-scale real-world problems can also easily lead to combinations that are beyond simple enumeration or intuitive guessing.

Planning problems (especially classic planning problems) can be formulated in a way that is particularly amiable for computers, as modern computers are capable of enumerating and checking rules and states quickly. Once a planning problem is formulated and represented in the way that computers can process, AI planners can leverage the computational power of modern computers to find solutions efficiently.

Domain knowledge has proven to be helpful in problem solving. However, unlike human planners, AI planners usually do not have a prior knowledge about the problem domain. While there are approaches that have domain experts in the problem-solving loop, in this thesis, we focus on *automated planning*, a family of planning approaches that automatically solve general planning problems without requiring problem-specific domain knowledge or human intervention.

Automated planning remains a challenging problem for AI researchers. For instance, classic planning, (i.e. problems without temporal constraints), one of the simplest categories of planning problems, has been proven to be PSPACE-complete. As the problem size grows, the computational resources required to solve the planning problems can grow exponentially.

Automated planning is at the core of AI research. Many important AI problems, such as the discrete time scheduling problem, the constraint satisfactory problem (CSP)

and the general state space search problem, can be formulated as planning problems. Accordingly, solving automated planning problems efficiently would help advancing other AI fields as well. Thus, it is critical to solve automated planning problems efficiently.

1.2 Motivation

Much research on classical planning has focused on the design of better heuristic functions. Despite the success of using domain-independent heuristics for classic planning, state-of-the-art heuristic planners still face scalability challenges for large-scale planning problems, due to the limitation of deterministic search and heuristic functions. As shown by recent work, search even with almost perfect heuristic guidance may still lead to very high search cost [37] for optimal planning. Therefore, it is important to improve other components of heuristic search that are orthogonal to the development of heuristics.

1.2.1 Limitations of Heuristic Search

Heuristic search is an important and pervasive technique for AI planning. Using heuristic search, an automated planning problem is mapped to a search problem guided by heuristics. Under this mapping, the initial state of the planning problem becomes the starting state of a search process. Starting from the initial state, the search algorithm would iteratively examine states that are reachable by applying actions to existing states, and terminating when a goal state is found.

Figure 1.1 shows the size of the search space with respect to the number of states in a planning domain called Driverlog. Problems in this domain model the route planning of delivery trucks. We used *Fast Downward*, a state-of-the-art heuristic search planner to generate Figure 1.1. As we can see from the figure, heuristic search would explore millions of nodes as the problem size increases. For real-world applications, heuristic search would become prohibitively expensive once the problem reaches certain size.

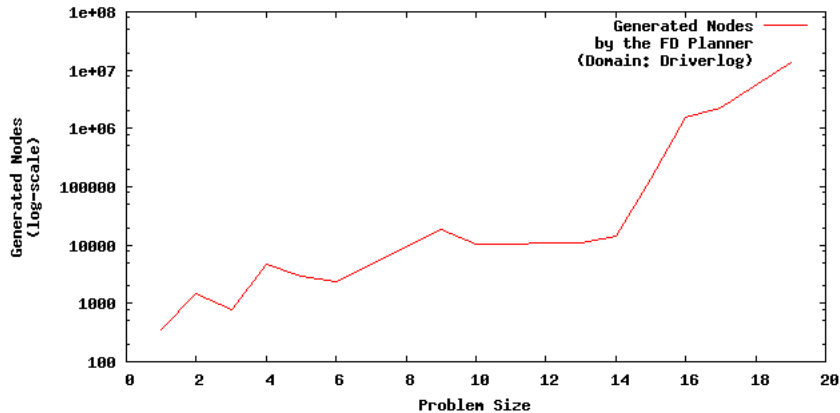


Figure 1.1: Number of generated nodes by the FastDownward planner on the Driverlog domain

Bloated State Space

Heuristic search algorithms map states in planning tasks to states in search space. Under this mapping, states that are inherently equivalent can be mapped to different states, leading to an excessive amount of equivalent states in the state space. This can be illustrated by the Gripper domain in the Third International Planning Competition (IPC 3) [1]. In these problems, a robot with two hands can move balls between two rooms. In many of the problems, balls are symmetric in terms of initial state and goal conditions. For example, if the goal is to have n balls in room 2, and the initial state is to have all n balls in room 1, then any intermediate states with $k \leq n$ balls in room 1 are effectively equivalent from the perspective of the robotic hand. However, when the problem is mapped to state space search, this equivalent class maps to $\binom{n}{k}$ states in state space, such that ball 1 in room 1 and ball 2 in room 2 is different than ball 1 in room 2 and ball 2 in room 1.

The bloated search space resulting from the equivalent states poses a fundamental challenge for state space search. Large-scale planning problems have similar structures

that can be exploited to reduce the size of the search space. In this thesis, *I will study a family of space reduction techniques called partial order reduction that utilizes the relation between states, such that heuristic search only needs to examine one state in each equivalent class.* The partial order reduction techniques presented in this thesis would accelerate heuristic search by pruning out states that do not need to be examined by heuristic search.

Imperfect Heuristics

Heuristic functions, in essence, estimate distances from any state to goal. Many heuristic functions have been proposed for solving automated planning problems. In general, heuristics that estimate distances more accurately lead to smaller search spaces. In the ideal case, the perfect heuristic function can guide the search procedure to goal states directly since it reveals the best successor to visit at each state along the path to goal. However, it is practically unattainable to construct the perfect heuristic function for general planning problems, as finding the perfect heuristic function for any state in the search space is equivalent to solving the planning problem itself.

It is a well-known phenomena that the number of states examined by heuristic search grow exponentially with the problem size when imperfect heuristics are used. As we take a closer look at the search process, we find that heuristic search does not progress at an even speed towards the goals. Instead, heuristic search hits plateau when heuristic functions are not informative. The resulted plateau exploration constitutes most computation in heuristic search, and that leads to high computational costs in heuristic search.

If we can reduce the amount of computation spent on plateau exploration during heuristic search, we can reduce the overall computational cost of heuristic search and accelerate heuristic search. *In this thesis, I will study approaches to accelerate plateau exploration for heuristic search, such that the overall computational cost for heuristic search is reduced.*

Hight Computational Costs

The high computational cost heuristic search severely limits its applicability to large-scale problems. A natural way to improve the efficiency of heuristic search is to utilize advanced, more powerful computing platforms. To this end, parallel heuristic search algorithms that are suitable for parallel and multi-core machines have been long and extensively studied. However, expensive computing infrastructures, such as supercomputers and large-scale clusters, are traditionally available to only a limited number of projects and researchers. As a result, many users with access to only commodity computers and clusters cannot benefit from the efficiency improvements of high-performance heuristic search algorithms to solve large-scale planning problems.

Cloud computing provides an attractive, highly accessible alternative to other traditional high-performance computing platforms. In cloud computing, resources can be leased from large data centers on a pay-as-you-go basis. This allows small teams and even a single user to routinely have access to the same large-scale computing facilities used by large companies and organizations.

Given the high accessibility of cloud computing and the fact that combinatorial search is ubiquitous in engineering and in various applications involving decision making, if we can significantly improve the efficiency of heuristic search in the cloud, the cloud-based algorithms can be routinely used by all users and may fundamentally change the landscape of AI planning applications. *In this thesis, I will study search algorithms that take advantage of cloud computing. I will also implement tools and systems for solving planning problems in the cloud such that users and planning researchers can take advantage of the cloud computing infrastructure.*

1.3 Contributions

This thesis is based on the premise that we do not have the perfect heuristic functions in heuristic search. By recognizing the imperfection of heuristic functions, we focus on techniques to *reduce the size of the search space* in heuristic search. Search-space

reduction (i.e. space reduction) approaches proposed in this thesis can be classified into two categories: partial order reduction techniques that takes advantage of the overall problem structure, and local space reduction techniques that utilizes the heuristic information in a local region.

In this thesis, we study the equivalence relation between states, based on partial order analysis of actions and of states. Partial order analysis enables search algorithms to explore only a subset of states, without compromising the completeness or optimality of search. We propose theories and algorithms that can automatically discover the partial order relations between states, even before search starts. During the heuristic search process, we utilize these partial order relations to impose orders between states and their corresponding actions, such that only a subset of the search space is explored, regardless of the heuristic functions used. We show that the proposed approaches can prune search space efficiently and effectively, without sacrificing the completeness of search nor the quality of the final solutions.

To reduce the search space in a local region during heuristic search, we study the behavior of heuristic search in local regions when heuristic functions are not informative. We propose a random walk based search framework that can work together with any existing deterministic search to escape from local plateau. Using the proposed algorithms, we participated in the Seventh International Planning Competition (IPC 7), with results showing that our algorithms perform competitively among other state-of-the-art heuristic search planners.

Finally, we present the portfolio stochastic search framework that takes advantage of cloud computing. Cloud typically has an abundance of computing cores but has high communication latency between nodes. We implement the portfolio stochastic search algorithm in both a local cluster, as well as the Windows Azure cloud platform. We show that our algorithms achieve superior, in many cases super linear, speedups in the cloud platform. We also show that our scheme is accessible and economically sensible for planning users and researchers.

This dissertation contains material from, and extends, the following publications:

- Y. Chen, Y. Xu, and G. Yao, **Stratified Planning**, *Proc. International Joint Conference on Artificial Intelligence (IJCAI-09)*, 2009.

- I refined the algorithm, implemented all the code and proved the completeness of the algorithm. I also conducted the experiments and gathered the results.

- Y. Xu , Y. Chen, Q. Lu, and R. Huang, **Theory and Algorithms for Partial Order Based Reduction in Planning**, *CoRR*, *abs/1106.5427*, 2011.
- Y. Xu, **Partial Order Reduction for Planning**, *Master's Thesis, Washington University in St. Louis*, 2010.
- Q. Lu, Y. Xu, R. Huang, and Y. Chen, **The Roamer Planner: Random Walk Assisted Best-First Search**, *Proc. International Planning Competition (IPC-2011)*, 2011
- I implemented the random walk part of the algorithm, the communication mechanism between the deterministic search and random walk, and pair-programmed with Lu during the development process.
- Y. Xu, Q. Lu, R. Huang, and Y. Chen, **The Roamer-p Planner**, *Proc. International Planning Competition (IPC-2011)*, 2011.
- Y. Xu, Q. Lu, R. Huang, and Y. Chen, **Enhancing Heuristic Search for Planning by Stochastic Plateau Escape**, *in preparation*, 2014.
- Q. Lu, Y. Xu, R. Huang, Y. Chen, and G. Chen, **Can Cloud Computing be Used for Planning? An Initial Study**, *Proc. IEEE CloudCom (CloudCom-11)*, 2011.
- I implemented the algorithm in Windows Azure and conducted all experiments for Azure.

1.4 Dissertation Outline

This dissertation is organized as follows.

In Chapter 2, we give a brief introduction to the background and terminology. In particular, we introduce the SAS+ formalism for classic planning problems, as well as domain transition graphs.

In Chapter 3, we introduce Stratified Planning, one of the first attempts to conduct partial order reduction for planning. We explain the stratified planning algorithm as well as the correctness of it in pruning search spaces. In Chapter 4, we go further and introduce a general theory for conducting partial order reduction for planning. We establish the connections between the existing partial order reduction techniques for planning and the stubborn set theory for model checking. We also present a new algorithm based on the new theory, and show its effectiveness in accelerating heuristic search for planning through experimental results.

In Chapter 5, we focus on accelerating heuristic search when it is stuck on plateau exploration. We employ random walks as a way to assist heuristic search in escaping from traps or blocks during search. Competition results from the Seventh International Planning Competition are also presented to show the effectiveness of our algorithms.

In Chapter 6, we apply the sequential stochastic search algorithms to the cloud environment, and discuss the advantages of dovetailing parameter settings in the portfolio stochastic search algorithm. We also present our system implementations in the Windows Azure platform and report the experimental results by running the portfolio search algorithm using up to 120 computational nodes in the cloud.

Chapter 2

Background and Related Works

2.1 SAS+ Planning

This thesis is focused on solving classic planning problems. Classical planning is the most fundamental form of planning, which deals with only propositional logic. In addition to classical planning, there are temporal planning (problems with temporal conditions) and probabilistic planning (problems with probabilistic instead of deterministic action effects).

We work on the SAS+ formalism [42] of classical planning. SAS+ formalism has recently attracted attention due to a number of advantages it has over the traditional STRIPS [24] formalism for classic planning. In the following, we review this formalism and introduce the notations used in this thesis.

Definition 1 *A SAS+ planning task Π is defined as a quintuple*

$$\Pi = \{X, \mathcal{O}, S, s^J, s^G\}.$$

- $X = \{x_1, \dots, x_N\}$ 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 state variables in the form $x_i = v_i, v_i \in Dom(x_i)$.

- S is the set of states. A **state** $s \in S$ is a full assignment to all state variables. $s^J \in S$ is the **initial state**. s^G is a partial assignment that defines the goal. A state s is a **goal state** if $s^G \subseteq s$.

For a SAS+ planning task, a given state s and an action o , when all variable assignments in $pre(o)$ are met in state s , action o is *applicable* in state s . After applying o to s , the state variable assignment will be changed to a new state s' according to $eff(o)$: the state variables that appear in $eff(o)$ will be changed to the assignments in $eff(o)$ while other state variables remain the same. We denote the resulting state after applying an applicable action o to s as $s' = apply(s, o)$. $apply(s, o)$ is undefined if o is not applicable in s . The planning task is to find a **path**, or a sequence of actions, that transition the initial state s^J to a goal state that includes s^G .

An important structure for a given SAS+ task is the domain transition graph (DTG) defined as follows:

Definition 2 For a SAS+ planning task, each state variable x_i ($i = 1, \dots, N$) corresponds to a **domain transition graph (DTG)** G_i , a directed graph with a vertex set $V(G_i) = Dom(x_i) \cup v_0$, where v_0 is a special vertex, and an edge set $E(G_i)$ determined by the following.

- If there is an action o such that $(x_i = v_i) \in pre(o)$ and $(x_i = v'_i) \in eff(o)$, then (v_i, v'_i) belongs to $E(G_i)$ and we say that o is **associated** with the edge $e_i = (v_i, v'_i)$ (denoted as $o \vdash e_i$). It is conventional to call the edges in DTGs **transitions**.
- If there is an action o such that $(x_i = v'_i) \in eff(o)$ and no assignment to x_i is in $pre(o)$, then (v_0, v'_i) belongs to $E(G_i)$ and we say that o is **associated** with the transition $e_i = (v_0, v'_i)$ (denoted as $o \vdash e_i$).

Intuitively, a SAS+ task can be decomposed into multiple objects, each corresponding to one DTG, which models the transitions of the possible values of that object.

2.2 Heuristic Search for Planning

As mentioned in the previous chapter, heuristic search is one of the most popular approaches to automated planning. A heuristic function h that maps any state to a real number is used in search to estimate the distance from a state to goal. In other words, for any state s , its heuristic value $h(s)$ is an approximation for $d(s)$, the distance from state s to goal. Namely, $h(s) \approx d(s)$, or $h(s) = d(s) + \omega$ where ω is a random variable whose distribution Ω is solely determined by the heuristic algorithm. A heuristic function is admissible if and only if $h(s) \leq d(s)$ for any s , or $\omega \leq 0$ for all ω in Ω .

For a classical planning task, its **state space** is a directed graph \mathcal{S} in which each state s is a vertex and each directed edge (s, s') represents an action. There is an edge (s, s') if and only if there exists an action o such that s' is the resulting state after applying action o to s . State s' is also called a *successor* state of s .

Heuristic search uses heuristic functions to guide the exploration of search space to arrive at goal states. A data structure called the *open* list, usually implemented as a priority queue, is used to store any states that are ready to be explored. At the beginning of a search, the initial state s_0 , along with its heuristic value $h(s_0)$, is inserted into the *open* list. At each step of the heuristic search, the state with the smallest heuristic value is explored, meaning it is removed from the *open* list to check if it is a goal state. If not, all of its successors, along with their heuristic values, are inserted into the *open* list for later exploration. The state itself is inserted into the *closed* list.

Algorithm 1 shows the general framework of heuristic search. In this algorithm, heuristic function is used to decide which state REMOVE-FIRST should yield. For instance, the above algorithm becomes **best-first search** if nodes in the *open* list are sorted by their heuristic values. Algorithm 1 becomes A^* **search** when the heuristic function is admissible and nodes in the *open* list are stored by $f(s) = g(s) + h(s)$, where g is the distance from the initial state to s , and $h(s)$ is the heuristic value for state s .

Algorithm 1: Heuristic Search Procedure

Input: problem, *open*

Output: found or failure

closed $\leftarrow \emptyset$;

insert INITIAL STATE to *open* ;

while *True* **do**

if *open* is empty **then**

return failure

end

 node \leftarrow REMOVE-FIRST (*open*) ;

if node is GOAL **then**

return found

end

if node is not in *closed* **then**

 add node to *closed* ;

 insert SUCCESSOR(node) and their heuristics to *open* ;

end

end

2.2.1 Notable Heuristics

Most of the award-winning planners of the International Planning Competitions (IPCs) are using the heuristic search framework. The success of heuristic search planners is largely linked to the development of high quality heuristic functions, here follows a brief overview.

Deletion Relaxation

In SAS+ planning tasks, when an operator o is applied to state s , it changes the value of state variable x_i from v_i to v'_i according to $eff(o)$. By allowing state variable x_i to be both v_i and v'_i after applying o , planning task $\Pi(s)$ is relaxed to a new task $\Pi^+(s)$ such that every solution to $\Pi(s)$ is also a solution to $\Pi^+(s)$. This way, the optimal solution cost of $\Pi^+(s)$, denoted by $h^+(s)$, can thus be used as an admissible heuristic for $\Pi(s)$. However, calculating h^+ itself is NP-hard and thus impractical. Heuristic functions such as h_{add} and h_{max} approximate h^+ by estimating the cost of achieving certain goals [10]. The heuristic function used in the Fast Forward [39] planner, h_{ff} ,

approximates h^+ by extracting an explicit solution. Other heuristic functions based on deletion relaxation include additive h_{max} [21] and h_{LM-cut} [34].

Causal Graph Relaxation

The h_{cg} heuristic used in the Fast Downward planning system utilizes a data structure called *causal graph* for heuristic calculation [33]. It relaxes the planning task from $\Pi(s)$ to $\Pi_{cg}(s)$ by ignoring certain dependencies between state variables, such that the final causal graph is acyclic. Similar to h_{ff} , the goal distance at s is estimated by finding a plan for the relaxed task $\Pi_{cg}(s)$. An improved version of h_{cg} , h_{sea} , is able to handle causal graphs with cyclic links. Since both heuristics assume that the cost function is additive when multiple goals are present, they are not guaranteed to be admissible.

Landmark Relaxation

The landmark counting heuristic used in the LAMA planner [63], h_{lm} , relies on the counting of landmarks that have not been visited so far. Heuristic h_{lm} is inadmissible and path-dependent because it relies on the path so far to determine the number of landmarks that have not yet been reached. An admissible version of h_{lm} is proposed by Karpas et al. [43], which uses action cost partitioning. Yet another admissible heuristic called “merge-and-shrink” was developed based on abstraction of domain transitions [36], which dominates the admissible landmark heuristics [35].

2.2.2 Helpful Actions and Multiple Heuristics

Heuristics such as h_{ff} and h_{cg} not only give estimations of goal distances, but also provide solutions to the relaxed problems. It is likely that actions appearing in the solution to the relaxed problems are also part of the solution to the original problems. Search algorithms can use these actions as extra information to improve search efficiency. Applicable actions that are part of the solution are considered *helpful actions* and can be given preference during search. This technique is used in both

Fast Forward and Fast Downward planners. Helpful actions are extremely useful for non-optimal planners [62], and they are helpful when heuristic calculation is deferred to when the state is explored [40, 33].

Since the debut of Fast Downward, it has also become commonplace for planners to utilize multiple heuristics during search. The premise is that certain heuristics may become uninformative in certain regions of the search space. By having multiple open lists ordered using different heuristics, when a heuristic function becomes uninformative or misleading, search can still make progress using other heuristics.

2.3 Notable Non-heuristic Techniques

In this section we review techniques that are orthogonal to the design of better heuristics.

Symmetry. Symmetry detection is a way to reduce search space [25]. It finds symmetric objects (DTGs in SAS+ formalism) and actions that are indistinguishable with respect to initial state and goal. However, this method proposed by Fox and Long [25] can only detect symmetry from the specification of initial and goal states, and may miss many symmetries.

Factored planning. Factored planning [6, 11, 44] is a class of search algorithm that exploits the decomposition of state space. Factored planning finds all the subplans for each individual subgraph and tries to merge them. There are some limitations of factored planning. The most notable is that search becomes prohibitively expensive when there are many subplans in each subgraph, and not every subgraph has goal states. Although factored planning has shown potential on some domain-dependent studies, its practicality for general domain-independent planning has not been established yet.

Partial order reduction. Partial order reduction (POR) is a way to reduce the search cost for classical planning [17, 18]. It allows a search to explore only part of the entire search space while still maintaining completeness and/or optimality. The idea is to enforce partial orders between states during search. POR algorithms have

been extensively studied for model checking [74, 20], which also requires examining a state space in order to prove certain properties. Model checking is not practical without POR due to its time complexity [27, 28, 76, 26, 58, 41].

Stochastic search. An alternative to deterministic search is stochastic search. One representative stochastic search algorithm is called the “Monte-Carlo Random Walk algorithm (MCRW)”. A *random walk* in state space is a trajectory of states that are linked by random actions. An MCRW starts from a known state, usually the initial state, by applying random actions to known states, generates a random walk in the search space, and terminates when a goal state is found in the walk. Stochastic search has been used by some of the leading planners in the Seventh International Planning Competition (IPC 7) [56, 52, 77].

Chapter 3

Accelerating Heuristic Search with Stratified Planning

Most planning problems have strong structures. They can be decomposed into sub-domains with causal dependencies. The idea of exploiting the domain decomposition has motivated previous work such as hierarchical planning and factored planning. 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.

3.1 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, only developing better heuristics has some fundamental limitations [37]. 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 method [46, 49] that utilizes hierarchical factoring of planning domains. However, it typically does not scale well since it requires extensive backtracking across subdomains. Another work is the factored planning approach [6, 11, 44] 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 chapter, 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, \dots, 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, \dots, P_k , and the search will expand the applicable actions in all the sub-domains P_1, \dots, 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 leverages the 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.

3.2 Stratified Planning

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 $x \in aff(o)$ and $x' \in dep(o)$, or, $x \in aff(o)$ and $x' \in aff(o)$.*

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 3.1a shows the CG of an instance (Truck-02) of the Truck planning domain used in the 5th International Planning Competition (IPC5) [3]. State variables that define the goal state are in a darker color.

3.2.1 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)** 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.

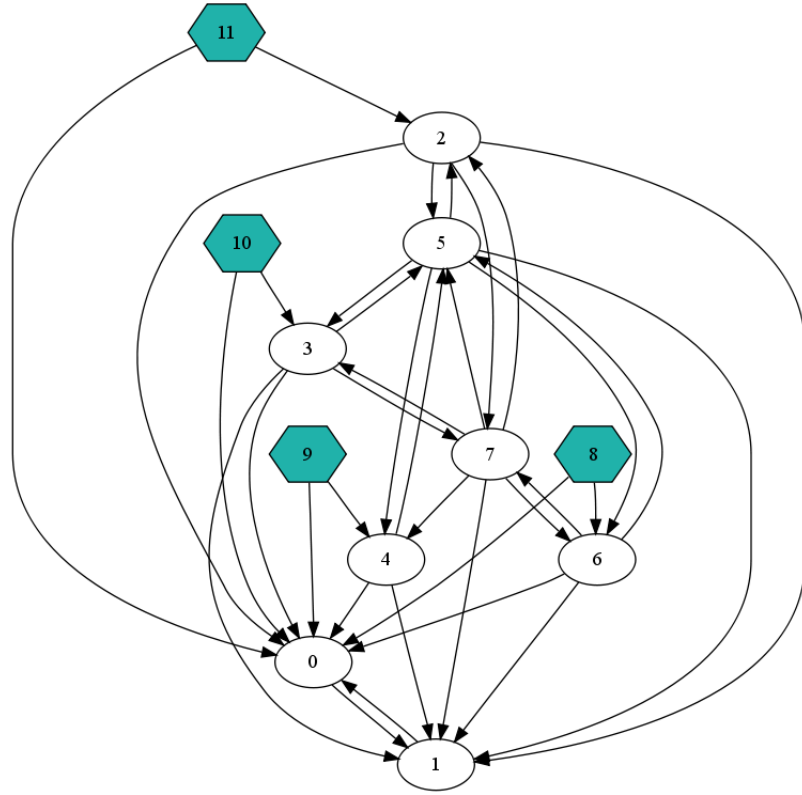
Definition 2 (Component Set) *Given a SAS+ planning task Π and its causal graph $CG(\Pi) = (X, E)$, the component set $\mathcal{M}(\Pi)$ is the partition of X such that all the elements in each $m \in \mathcal{M}(\Pi)$ are in the same SCC of $CG(\Pi)$.*

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 $(v, w) \in E$.*

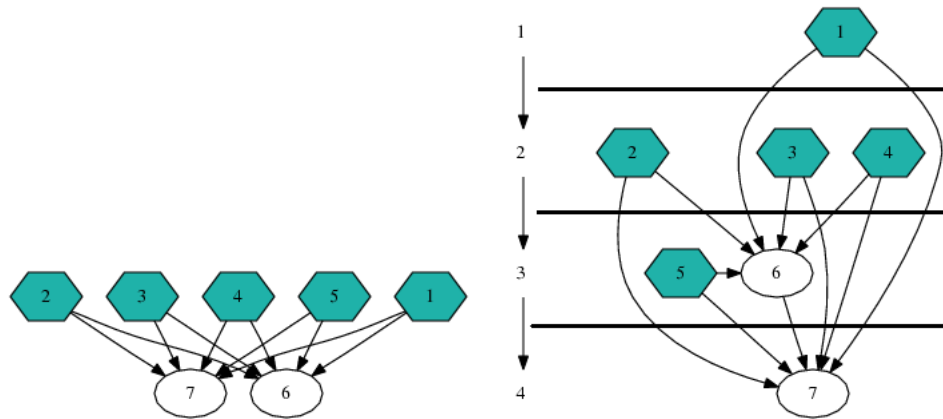
Definition 4 (Contracted Causal Graph (CCG)) *Given a SAS+ planning task Π , its contracted causal graph $CCG(\Pi) = (V, E)$ is a contracted graph of $CG(\Pi)$ such that $V = \mathcal{M}(\Pi)$.*

Figure 3.1b shows the corresponding CCG of Figure 3.1a. 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 3.2 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).*



(a) Causal graph (CG)



(b) Contracted causal graph (CCG)

(c) Stratification

Figure 3.1: The causal graph, contracted causal graph and stratification of Truck-02.

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.

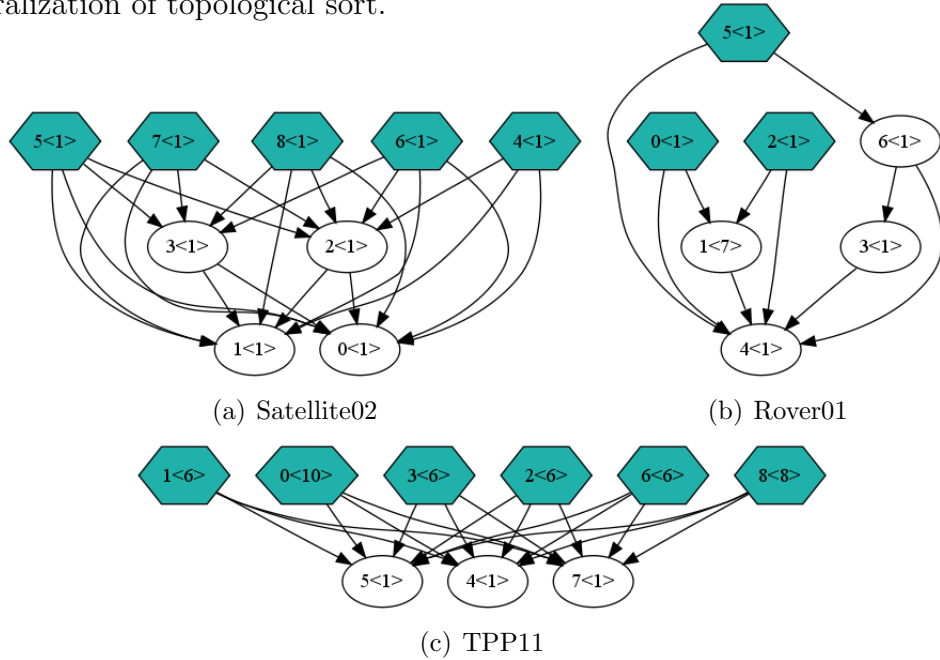


Figure 3.2: The CCGs of some instances of several planning domains. Each SCC is labelled by $x\langle y \rangle$, where “ x ” is the index for the SCC and “ y ” is the number of state variables in the SCC. The SCCs that contain goals are in a darker color.

Definition 5 (Stratification) *Given a DAG $G = (V, E)$, a stratification $Str(G)$ of G is a tuple $(\mathcal{U}, \mathcal{L})$, where $\mathcal{U} = \{u_1, \dots, u_k\}$ 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 3.1c shows an example of stratification of the Truck-02 problem. Each SCC in Figure 3.1b is now assigned a layer and the topological order is maintained in the stratification. Basically, 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 the same vertex of $CCG(\Pi)$. By Definition 5, we have $\mathcal{L}(x_i) = \mathcal{L}(x_j)$. ■

3.2.2 Stratified Planning Algorithm

Algorithm 2: Stratified Planning($\Pi, Str(\Pi)$)

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

Output: A solution plan

```
1  $closed \leftarrow$  an empty set;  
2 insert the initial AS pair (no-op, $s_j$ ) to  $open$ ;  
3 while  $open$  is not empty do  
4    $(a, s) \leftarrow$  REMOVE-FIRST( $open$ );  
5   if  $s$  is a goal state then return solution;  
6   ;  
7   if  $s$  is not in  $closed$  then  
8     add  $s$  to  $closed$ ;  
9      $\Psi(a, s, \mathcal{L}) =$  STRATIFIED-EXPANSION( $a, s, \mathcal{L}$ ) ;  
10     $open \leftarrow open \cup \Psi(a, s, \mathcal{L})$ ;  
11  end  
12 end
```

Now we propose Stratified Planning in Algorithm 2. In fact, it is not a stand-alone 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. The input of Algorithm 2 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.

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-FIRST operation fetches one AS pair (a, s) from the $open$ list, 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 the $open$ list, 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.

Algorithm 3: 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

```
1  $\Psi \leftarrow \emptyset$ ;  
2 foreach applicable action  $b$  at  $s$  do  
3   | if  $\mathcal{L}(b) \geq \mathcal{L}(a)$  then  
4   |   | compute  $s' = \text{apply}(s, b)$ ;  
5   |   |  $\Psi \leftarrow \Psi \cup \{(b, s')\}$  ;  
6   | else if  $a \triangleright b$  then  
7   |   | compute  $s' = \text{apply}(s, b)$ ;  
8   |   |  $\Psi \leftarrow \Psi \cup \{(b, s')\}$  ;  
9 end  
10 return  $\Psi$  ;
```

Since the initial state s_j 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 $a \triangleright b$) if $\text{aff}(a) \cap \text{dep}(b) \neq \emptyset$ or $\text{aff}(a) \cap \text{aff}(b) \neq \emptyset$. Any action is a follow-up action of no-op.*

Given this definition, we can describe the STRATIFIED-EXPANSION operation, shown in Algorithm 3. Given a stratification $\text{Str}(\Pi) = (\mathcal{U}, \mathcal{L})$, 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 ($a \triangleright b$).

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

3.2.3 Theoretical Analysis

Here, we show that stratified planning search preserves the completeness and optimality properties of the original search strategy, decided by the implementation of the *open* list and the evaluation function. For example, if the *open* list 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 $p = (o_1, \dots, o_n)$ is a valid path if, let $s_i = \text{apply}(s_{i-1}, o_i), i = 1, \dots, n$, o_i is applicable at s_{i-1} for $i = 1, \dots, n$. 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 $\text{str}(\Pi) = (\mathcal{U}, \mathcal{L})$ and a state s_0 , a sequence of actions $p = (o_1, \dots, o_n)$ is a stratified path if it is a valid path and, let $s_i = \text{apply}(s_{i-1}, o_i), i = 1, \dots, n$, $(o_i, s_i) \in \Psi(o_{i-1}, s_{i-1}, \mathcal{L})$ for $i = 1, \dots, n$, where $o_0 = \text{no-op}$.

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

Lemma 1 For path $p = (a_1, \dots, a_n)$ that is a valid path but not a stratified path,

$(a_i, s_i) \notin \Psi(a_{i-1}, s_{i-1}, \mathcal{L})$. Since p is not a stratified path, such an i must exist. Now we perform a **swapping operation** and obtain a path

$$p' = (a_1, \dots, a_{i-2}, a_i, a_{i-1}, a_{i+1}, \dots, a_n).$$

We show that p' is also a valid path from s_0 .

Proof. According to Algorithm 3, We must have that $\mathcal{L}(a_i) < \mathcal{L}(a_{i-1})$ and that a_i is not a follow-up action of a_{i-1} . Since a_i is not a follow-up action of a_{i-1} , according to Definition 9, $\text{eff}(a_{i-1})$ contains no assignment in $\text{pre}(a_i)$. Therefore, since a_i is applicable at s_{i-1} , which is $\text{apply}(s_{i-2}, a_{i-1})$, 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 $pre(a_{i-1})$ is satisfied at s_{i-2} , it is also satisfied at $s' = apply(s_{i-2}, a_i)$. Hence, a_{i-1} is applicable at s' .

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 . ■

Theorem 1 *Given a SAS+ planning task Π and a stratification $Str(\Pi)$, for any state s_0 and any valid path $p_a = (a_1, \dots, a_n)$ from s_0 , there exists a stratified path $p_b = (b_1, \dots, b_n)$ 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 $p^0 = (a_1, \dots, a_{k+1})$, by our induction hypothesis, we can permute the first k actions to obtain a stratified path (a_1^1, \dots, a_k^1) .

Now we consider a new path $p^1 = (a_1^1, \dots, a_k^1, a_{k+1})$. 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 where $\mathcal{L}(a_{k+1}) > \mathcal{L}(a_k^1)$ and a_{k+1} is not a follow-up action of a_k^1 . Consider a new path $p^2 = (a_1^1, \dots, a_{k-1}^1, a_{k+1}, a_k^1)$. 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, \dots, a_k^2) . Define $p^3 = (a_1^2, \dots, a_k^2, a_k^1)$.

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, \dots , as long as $p^j, (j \in \mathbb{Z}^+)$ 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

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 $p^m = (a'_1, \dots, a'_k, a'_{k+1}), m \geq 1$. 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 we prove the induction step. ■

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. ■

3.3 Experimental Results

We test on STRIPS problems in the recent International Planning Competitions (IPCs). We implement our stratification analysis and stratified search on top of the Fast Downward planner [33] which gives SAS+ encoding of planning problems. We still use the causal graph heuristic h_{cg} 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.

In practice, how to determine the granularity of stratification is an important issue. We test two extreme 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 3.1 and 3.2. We did not include certain 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.

3.4 Discussions and Summary

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 3.1a, the SCCs numbered 1 to 5 are siblings in the CCG. However, after we stratify the CCG as in Figure 3.1c, 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.

Stratified search may also incorporate symmetry removal in some situations. For example, if three trucks $T1, T2, T3$ are symmetric and can support the delivery of a package. If the stratification places the three trucks at different layers, then it effectively provides symmetry removal since we will try using $T1$ before $T2$ and before $T3$ whenever possible. Also, stratified search seems to have an advantage in that it does not require the three objects to be absolutely symmetric for all the states and can adjust dynamically to the situation. For example, if at certain state $T2$ has to be used due to certain constraints, stratified search will find such a solution since it is completeness preserving.

In summary, we have proposed stratified planning, a reduction method that exploits the domain structure. We have defined the stratification of a SAS+ task on top of its contracted causal graph. We have then proposed a space reduction method that exploits the oneway-ness of the dependencies between stratified layers and proved that the reduction method is optimality and completeness preserving. Our experimental results on recent IPC domains show that search can be made more efficient when the search space can be stratified into dependency layers.

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	8.45
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 3.1: 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.

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
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
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
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 3.2: 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.

Chapter 4

Accelerating Heuristic Search with Partial Order Reduction

Stratified Planning discussed in the last chapter is a type of partial order based reduction (POR) technique for search. Stratified Planning, together with the Expansion Core [18] algorithm, showed a new direction of research where we can reduce the search space that is an orthogonal and complementary approach to improving heuristics. POR has shown promise in speeding up heuristic searches.

Partial order reduction has been extensively studied in model checking research and is a key technique for enabling scalability of model checking systems. Although the POR theory has been extensively studied in model checking, it has never before been developed systematically for planning. In addition, the conditions for POR in the model checking theory are abstract and not directly applicable in planning. Previous works on POR algorithms for planning did not establish the connection between these algorithms and existing theory in model checking.

In this chapter, we develop a theory for POR in planning. The new theory we develop connects the stubborn set theory in model checking to POR methods in planning. We show that previous POR algorithms in planning can be explained by the new theory. Based on the new theory, we propose a new, stronger POR algorithm. Experimental results using the new algorithm show further search cost reduction in various planning domains.

4.1 Partial Order Reduction Theory for Planning

In this section, we will first introduce the concept of search reduction. Then, we will present a general POR theory for planning, which gives sufficient conditions that guide the design of practical POR algorithms.

4.1.1 Space Reduction for Planning

We first introduce the concept of search reduction that previously has been informally introduced [18, 17]. A standard search algorithm, such as breadth-first search, depth-first search, or A^* search, needs to explore a state space graph. A reduction algorithm is an algorithm that reduces the state space graph into a subgraph, so that a search will be performed on the subgraph instead of the original state space graph. We first define the state space graph. In our presentation, for any graph G , we use $V(G)$ to denote the set of vertices and $E(G)$ the set of edges. For a directed graph G , for any vertex $s \in V(G)$, a vertex $s' \in V(G)$ is its **successor** if and only if $(s, s') \in E(G)$.

For a SAS+ planning task, its **original state space graph** is a directed graph \mathcal{G} in which each state s is a vertex and there is a directed edge (s, s') if and only if there exists an action o such that $apply(s, o) = s'$. We say that action o **marks** the edge (s, s') .

Definition 3 For a SAS+ planning task, for a state space graph \mathcal{G} , the **successor set** of a state s , denoted by $succ_{\mathcal{G}}(s)$, is the set of all the successor states of s . The **expansion set** of a state s , denoted by $expand_{\mathcal{G}}(s)$, is the set of actions

$$expand_{\mathcal{G}}(s) = \{o \mid o \text{ marks } (s, s'), (s, s') \in E(\mathcal{G})\}.$$

Intuitively, the successor set of a state s includes all the successor states that shall be generated by a search upon expanding s , while the expansion set includes all the actions to be expanded at s .

In general, a **reduction method** is a method that maps the original state space graph \mathcal{G} for a planning task to a subgraph of \mathcal{G} called the **reduced state space**

graph. POR algorithms remove edges from \mathcal{G} . More specifically, each state s is only connected to a subset of all its successors in the reduced state space graph. We note that, by removing edges, a POR algorithm may also reduce the number of vertices that are reachable from the initial state, hence reducing the number of nodes examined during the search process. The decision as to whether a successor state s' would still be a successor in the reduced state space graph can be made locally by checking certain conditions related to the current state and some precomputed information. Hence, a POR algorithm can be combined with various search algorithms.

For a SAS+ planning task, a **solution sequence** in its state space graph \mathcal{G} is a pair (s^0, p) , where

- s^0 is a non-goal state,
- $p = (a_1, \dots, a_k)$ is a sequence of actions, and,
- let $s^i = \text{apply}(s^{i-1}, a_i), i = 1, \dots, k$, (s^{i-1}, s^i) is an edge in \mathcal{G} for $i = 1, \dots, k$ and s^k is a goal state.

We now define a property of reduction methods.

Definition 4 *For a SAS+ planning task, 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.*

Similarly, a reduction method is **optimality-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 objective function value as p does. In addition, a reduction method is **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 the actions in p' are a permutation of the actions in p .

Clearly, being action-preserving is a sufficient condition for being completeness-preserving. When the objective function is action set invariant (such as optimizing plan length

or total action cost), being action-preserving is also a sufficient condition for being optimality-preserving. As a result, we prove the completeness-preserving or optimality-preserving attributes of POR algorithms by proving they are action-preserving.

4.1.2 Stubborn Set Theory for Planning

Among the many variations of POR methods in model checking, a popular and representative POR algorithm is the stubborn set method [67, 68, 69, 72, 70, 71]. We briefly introduce the idea of stubborn set and stubborn set method in model checking without distracting readers with technical details. In model checking, a stubborn set is a subset of applicable transitions for a state that satisfies a set of conditions.

For each state, a stubborn set method finds the stubborn set of each state and expands only the actions in the stubborn set during search. Conditions that define stubborn sets guarantee that important properties (such as deadlock preserving) are preserved under reduction. By expanding a small subset of applicable actions in each state, stubborn set methods can reduce the search space without compromising correctness for model checking. Since planning also examines a large search space, we develop a stubborn set theory for planning. To achieve this, we need to handle various subtle issues arising from the differences between model checking and planning. We first adapt the definition of stubborn set in model checking and define the concept of stubborn sets for planning.

Definition 5 (Stubborn Set for Planning) *For a SAS+ planning task, a set of actions $T(s)$ is a stubborn set at a non-goal state s if and only if*

- A1) For any action $b \in T(s)$ and actions $b_1, \dots, b_k \notin T(s)$, if (b_1, \dots, b_k, b) is a prefix of a path from s to a goal state, then (b, b_1, \dots, b_k) is a valid path from s and leads to the same state that (b_1, \dots, b_k, b) does; and*
- A2) Any valid path from s to a goal state contains at least one action in $T(s)$.*

The above definition is schematically illustrated in Figure 4.1. Once we define the stubborn set $T(s)$ in each state s , we in effect reduce the state space graph to a

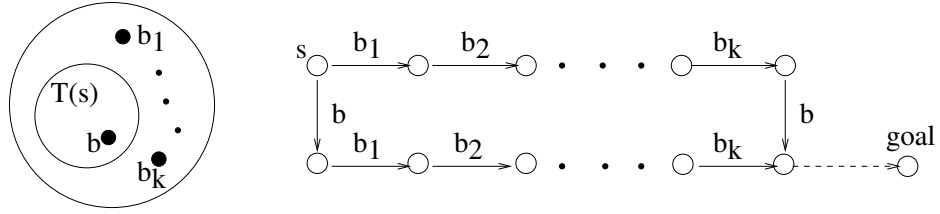


Figure 4.1: Illustration of condition A1 in Definition 5. The big circle on the left stands for the set of all applicable actions at state s , while the small circle stands for the stubborn set $T(s)$ of s . Action $b \in T(s)$ can always be swapped to the beginning of a path consisting of b_i s without affecting the final state.

subgraph: only the edges corresponding to actions in the stubborn sets are kept in the subgraph. Also note that conditions for the stubborn set do not directly lead to an efficient procedure to decide whether a given set is a stubborn set or not. For instance, given a planning task and a state s , unless we have more information about paths from s to a goal state, we cannot verify condition A2 in the above definition.

Definition 6 *For a SAS+ planning task, given a stubborn set $T(s)$ defined at each state s , the stubborn set method reduces its state space graph \mathcal{G} to a subgraph \mathcal{G}_r such that $V(\mathcal{G}_r) = V(\mathcal{G})$ and there is an edge (s, s') in $E(\mathcal{G}_r)$ if and only if there exists an action $o \in T(s)$ such that $s' = \text{apply}(s, o)$.*

A **stubborn set method for planning** is a reduction method that reduces the original state space graph \mathcal{G} to a subgraph \mathcal{G}_r according to Definition 6. In other words, a stubborn set method expands actions only in a stubborn set in each state. We will now show that such a reduction method preserves actions, hence, it also preserves completeness and optimality.

Lemma 2 *Any stubborn set method for planning is action-preserving.*

Proof: We prove that for any solution sequence (s^0, p) in the original state space graph \mathcal{G} , there exists a solution sequence (s^0, p') in the reduced state space graph \mathcal{G}_r resulting from the stubborn set method, such that p' is a permutation of actions in p . We prove this fact by induction on k , the length of p .

When $k = 1$, let a be the only action in p , according to the second condition in Definition 5, a is in $T(s^0)$. Thus, (s^0, p) is also a solution sequence in \mathcal{G}_r . Thus, a stubborn set method is action-preserving in the base case.

When $k > 1$, the induction assumption is that any solution path in \mathcal{G} with length less than or equal to $k - 1$ has a permutation in \mathcal{G}_r that leads to the same final state. Now we consider a solution sequence (s^0, p) in \mathcal{G} : $p = (a_1, \dots, a_k)$. Let $s^i = \text{apply}(s^{i-1}, a_i), i = 1, \dots, k$. If $a_1 \in T(s)$, we can invoke the induction assumption for the state s^1 and prove our induction assumption for k .

We now consider the case where $a_1 \notin T(s)$. Let a_j be the first action in p such that $a_j \in T(s)$. Such an action must exist because of condition A2 in Definition 5.

Consider the sequence $p^* = (a_j, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_k)$. According to condition A1 in Definition 5, $(a_j, a_1, \dots, a_{j-1})$ is also a valid sequence from s^0 which leads to the same state that (a_1, \dots, a_j) does. Hence, we know that (s^0, p^*) is also a solution path. Therefore, let $s' = \text{apply}(s^0, a_j)$, we know (a_1, \dots, a_{j-1}) is an executable action sequence starting from s' . Let $p^{**} = (a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_k)$, (s', p^{**}) is a solution sequence in \mathcal{G} . From the induction assumption, we know there is a sequence p' which is a permutation of p^{**} , such that (s', p') is a solution sequence in \mathcal{G}_r . Since $a_j \in T(s^0)$, we know that a_j followed by p' is a solution sequence from s^0 and is a permutation of actions in p^* , which is a permutation of actions in p . Thus, the stubborn set method is action-preserving. ■

Since being action-preserving is a sufficient condition for being completeness-preserving and optimality-preserving, when the object function is action set invariant, we have the following theorem.

Theorem 3 *A stubborn set method for planning is completeness-preserving. In addition, it is optimality-preserving when the objective function is action set invariant.*

4.1.3 Commutativity in SAS+ planning

Theorem 3 shows that we can use a stubborn set method to reduce the search space. However, as we mentioned earlier, conditions for stubborn sets defined in Definition 5

are only necessary for a given stubborn set. These conditions do not directly lead to efficient algorithms for finding stubborn sets. In turn, we want to find efficient procedures for finding stubborn sets that facilitate state space search algorithms. In the following, we define several concepts that can lead to sufficient conditions for stubborn sets.

Definition 7 (State-Dependent Commutativity) *For a SAS+ planning task, an ordered action pair (a, b) , $a, b \in O$ is commutative in state s , if (a, b) is a valid path at s implies that (b, a) is also a valid path at s that ends at the same state. We denote such a relationship by $s : b \Rightarrow a$.*

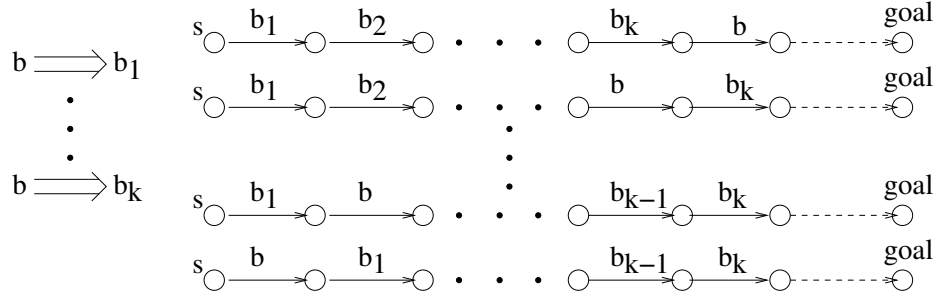
Definition 8 (State-Independent Commutativity) *For a SAS+ planning task, an ordered action pair (a, b) , $a, b \in O$ is commutative if, for any state s , it is true that $s : b \Rightarrow a$. We denote such a relationship by $b \Rightarrow a$.*

The order in the notation $b \Rightarrow a$ suggests that we should always try only (b, a) during the search instead of trying both (a, b) and (b, a) . Also, not every state-dependent commutative action pair is state-independent commutative. For instance, in a SAS+ planning task with three state variables $\{x_1, x_2, x_3\}$, action a with $pre(a) = \{x_1 = 0\}$, $eff(a) = \{x_2 = 1\}$ and action b with $pre(b) = \{x_2 = 1, x_3 = 2\}$, $eff(b) = \{x_3 = 3\}$ are commutative in state $s^1 = \{x_1 = 0, x_2 = 1, x_3 = 2\}$ but not in state $s^2 = \{x_1 = 0, x_2 = 0, x_3 = 2\}$ as b is not applicable in state s^2 . That is to say, $b \Rightarrow a$ is only true in state s^1 but not in state s^2 .

Although the conditions for state-independent commutativity are stronger, they greatly simplify the derivation of sufficient conditions for finding stubborn sets. Our ultimate results, however, only need to assume the state-dependent commutativity of action pairs.

Definition 9 (State-Independent Commutative Set) *For a SAS+ planning task, a set of actions $T(s)$ is a commutative set at state s if and only if*

- L1) For any action $b \in T(s)$ and any action $a \in O - T(s)$, if there exists a valid path from s to a goal state that contains both a and b , then it is the case that $b \Rightarrow a$; and*



In this diagram, the left part plots the condition L1 in Definition 9 and the right part plots the strategy in the proof to Theorem 4. We swap action b with each b_i during the constructive proof.

Figure 4.2: Illustration of commutative set.

A2) Any valid path from s to a goal state contains at least one action in $T(s)$.

Theorem 4 For a SAS+ planning task, for a state s , if a set of actions $T(s)$ is a state-independent commutative set, then it is also a stubborn set.

Proof: We only need to prove that L1 in Definition 9 implies A1 in Definition 5. The proof strategy is schematically shown in Figure 4.2.

For an action $b \in T(s)$ and actions $b_1, \dots, b_k \notin T(s)$, if (b_1, \dots, b_k, b) is a prefix of a path from s to a goal state, then according to L1, we see that $b \Rightarrow b_i$, for $i = 1, \dots, k$. According to the definition of commutativity, we see that b_k and b can be swapped and that the resulting path (b_1, \dots, b, b_k) is still a valid path that leads to the same state that (b_1, \dots, b_k, b) does. We can subsequently swap b with b_{k-1} , \dots , and b_1 to obtain equivalent paths, before finally obtaining (b, b_1, \dots, b_k) , as shown in the schematic illustration in the right part of Figure 4.2. Hence, we have shown that if $p = (b_1, \dots, b_k, b)$ is a prefix of a path from s to a goal state, then $p' = (b, b_1, \dots, b_k)$ is also a valid path from s that leads to the same state that p does, which is exactly condition A1 in Definition 5. ■

From the proof above, we see that the requirement of state-independent commutativity in Definition 9 is unnecessarily strong. Instead, only certain state-dependent commutativity is necessary. In fact, when we change (b_1, \dots, b_k, b) to (b_1, \dots, b, b_k) , we only require $s' : b \Rightarrow b_k$ where s' is the state after b_{k-1} is executed. Similarly, when

we change (b_1, \dots, b_k, b) to $(b_1, \dots, b, b_{k-1}, b_k)$, we only require $s'' : b \Rightarrow b_{k-1}$ where s'' is the state after b_{k-2} is executed. Based on the above analysis, we can refine the sufficient conditions.

Definition 10 (State-Dependent Commutative Set) *For a SAS+ planning task, a set of actions $T(s)$ is a commutative set at state s if and only if*

L1') For any action $b \in T(s)$ and actions $b_1, \dots, b_k \notin T(s)$, if (b_1, \dots, b_k, b) is a prefix of a path from s to a goal state, then $s' : b \Rightarrow b_k$, where s' is the state after (b_1, \dots, b_{k-1}) is executed; and

A2) Any valid path from s to a goal state contains at least one action in $T(s)$.

We only need to slightly modify the proof to Theorem 4 in order to prove the following theorem.

Theorem 5 *For a SAS+ planning task, for state s , if a set of actions $T(s)$ is a state-dependent commutative set, it is also a stubborn set.*

The above result gives sufficient conditions for finding stubborn sets in planning. The concept of state-dependent commutative sets requires a less stringent condition than the state-independent commutative set. Such a nuance actually leads to different previous POR algorithms with varying performances. Therefore, it will result in smaller $T(s)$ sets and stronger reduction. Next, we present our algorithm for finding such a set at each state to satisfy these conditions.

4.1.4 Determining Commutativity

Theorem 5 provides a key result for POR. However, the conditions in Definition 7 are still abstract and not directly implementable. The key issue is to efficiently find commutative action pairs. Now we give necessary and sufficient conditions for Definition 7 that can practically determine commutativity and facilitate the design of reduction algorithms.

Theorem 6 *For a SAS+ planning task, for a valid action path (a, b) in state s , we have $s : b \Rightarrow a$ if and only if $pre(a)$ and $eff(b)$, $pre(b)$ and $eff(a)$, $eff(a)$ and $eff(b)$ are all conflict-free and b is applicable at s .*

Proof: First, from the definition of $s : b \Rightarrow a$, we know that action b is applicable in state $apply(s, a)$. This implies that $pre(b)$ and $eff(a)$ are conflict-free. Symmetrically, since action a is applicable in state $apply(s, b)$, $pre(a)$ and $eff(b)$ are also conflict-free. Now we prove $eff(a)$ and $eff(b)$ are conflict-free by contradiction. If $eff(a)$ and $eff(b)$ are not conflict-free, without loss of generality, we can assume that $eff(a)$ contains $x_i = v_i$ and $eff(b)$ contains $x_i = v'_i \neq v_i$. Thus, the value of x_i is v_i for state $s^{ab} = apply(apply(s, a), b)$ and v'_i for state $s^{ba} = apply(apply(s, b), a)$, i.e., s^{ab} is different than s^{ba} . This contradicts our assumption that a and b are commutative. Thus, $eff(a)$ and $eff(b)$ are conflict-free.

Second, if b is applicable in s , $apply(s, b)$ is well-defined, and a is also applicable in state $apply(s, b)$ as $pre(a)$ and $eff(b)$ are conflict-free. Hence, (b, a) is a valid path at s . Also, for any state variable x_i , its value in states $s^{ab} = apply(apply(s, a), b)$ and $s^{ba} = apply(apply(s, b), a)$ are the same, because $eff(a)$ and $eff(b)$ are conflict-free. Therefore, we have $s^{ab} = s^{ba}$. Hence, we have $s : b \Rightarrow a$. ■

Theorem 6 gives necessary and sufficient conditions for deciding whether two actions are commutative or not. Based on this result, we later develop practical POR algorithms that find stubborn sets using commutativity.

4.2 Stubborn-Set Theory for Existing POR Algorithms

Previously, we have proposed two POR algorithms for planning: expansion core (EC) [18] and stratified planning (SP) [17], both of which showed good performance in reducing the search space. However, we did not have a unified theoretical background for them. We now explain how these two algorithms can be explained by our theory.

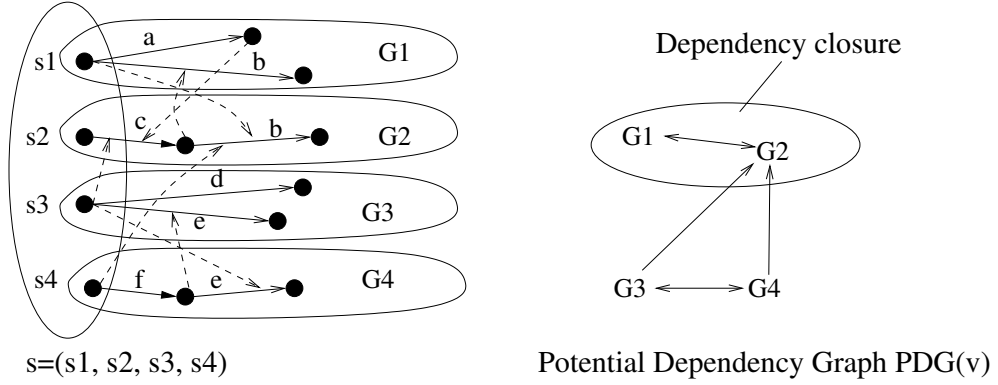


Figure 4.3: A SAS+ task with four DTGs. The dashed arrows show preconditions (prevailing and transitional) of each edge (action). Only dashed arrows between DTGs are shown. Actions are marked with letters a to f. We see that b and e are associated with more than one DTG.

4.2.1 Explanation of EC

Expansion core (EC) algorithm is a POR-based reduction algorithm for planning. We will see that, in essence, the EC algorithm exploits the SAS+ formalism to find a commutative set for each state. To describe the EC algorithm, we need the following definitions.

Definition 11 For a SAS+ task, for each DTG $G_i, i = 1, \dots, N$, for a vertex $v \in V(G_i)$, an edge $e \in E(G_i)$ is a **potential descendant edge** of v (denoted as $v \triangleleft e$) if 1) G_i is goal-related and there exists a path from v to the vertex that stands for a goal assignment in G_i that contains e ; or 2) G_i is not goal-related and e is reachable from v .

Definition 12 For a SAS+ task, for each DTG $G_i, i = 1, \dots, N$, for a vertex $v \in V(G_i)$, a vertex $w \in V(G_i)$ is a **potential descendant vertex** of v (denoted as $v \triangleleft w$) if 1) G_i is goal-related and there exists a path from v to the goal vertex in G_i that contains w ; or 2) G_i is not goal-related and w is reachable from v .

Definition 13 For a SAS+ task, given a state $s = (s_1, \dots, s_N)$, for any $1 \leq i, j \leq N, i \neq j$, we call s_i a **potential precondition** of the DTG G_j if there exist $o \in \mathcal{O}$

Action	Associated with	Preconditions in
a	G_1	
b	G_1, G_2	G_2, G_1
c	G_2	G_3
d	G_3	
e	G_3, G_4	G_4, G_3
f	G_4	

Table 4.1: Supplementary table for Figure 4.3: list of actions and related DTGs.

and $e_j \in E(G_j)$ such that

$$s_j \triangleleft e_j, \quad o \vdash e_j, \quad \text{and } s_i \in \text{pre}(o) \quad (4.1)$$

Definition 14 For a SAS+ task, given a state $s = (s_1, \dots, s_N)$, for any $1 \leq i, j \leq N, i \neq j$, 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

$$s_j \triangleleft w_j, \quad o \vdash e_i, \quad \text{and } w_j \in \text{pre}(o) \quad (4.2)$$

Definition 15 For a SAS+ task, given a state $s = (s_1, \dots, s_N)$, its **potential dependency graph** $\text{PDG}(s)$ is a directed graph in which each DTG $G_i, i = 1, \dots, N$ corresponds to a vertex, and there is an edge from G_i to $G_j, i \neq j$, if and only if s_i is a potential precondition or potential dependent of G_j .

Figure 4.3 illustrates the above definitions. The dashed arrows on the left side of the figure are preconditions of each action. We show the DTGs where each actions preconditions are in in Table 4.1. For each action, by having arrows from the third columns of Table 4.1 to the second column, we obtain the $\text{PDG}(s)$. It is shown on the right side of the figure. $\text{PDG}(s)$ intuitively shows the dependencies between DTGs. For instance, G_3 has an arrow to G_2 in this graph, meaning actions associated with G_3 may affect the actions in G_2 . It is true because action d associated with G_3 will render action c associated with G_2 not applicable. However, since there is no arrow from G_2 to G_3 , we know that any actions associated with G_2 will not affect the applicability of actions associated with G_3 , as we can indeed tell from this example.

Definition 16 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 $(v, w) \in E(H)$.

Intuitively, a DTG in a dependency closure may depend on other DTGs in the closure but not those DTGs outside of the closure. In Figure 4.3, G_1 and G_2 form a dependency closure of $PDG(s)$.

The EC algorithm is defined as follows:

Definition 17 (Expansion Core Algorithm) For a SAS+ planning task, the EC method reduces its state space graph \mathcal{G} to a subgraph \mathcal{G}_r such that $V(\mathcal{G}_r) = V(\mathcal{G})$ and for each vertex (state) $s \in V(\mathcal{G})$, it expands actions in the following set $T(s) \subseteq O$:

$$T(s) = \bigcup_{i \in \mathcal{C}(s)} \left\{ o \mid o \in \text{exec}(s) \wedge o \vdash G_i \right\}, \quad (4.3)$$

where $\text{exec}(s)$ is the set of applicable actions in s and $\mathcal{C}(s) \subseteq \{1, \dots, N\}$ is an index set satisfying:

- EC1) The DTGs $\{G_i, i \in \mathcal{C}(s)\}$ form a dependency closure in $PDG(s)$; and
- EC2) There exists $i \in \mathcal{C}(s)$ such that G_i is goal-related and s_i is not the goal vertex in G_i .

Intuitively, the EC method can be described as follows. To reduce the original state-space graph, for each state, instead of expanding actions in all the DTGs, it only expands actions in DTGs that belong to a dependency closure of $PDG(s)$ under the condition that at least one DTG in the dependency closure is goal-related and not at a goal state.

The set $\mathcal{C}(s)$ can always be found for any non-goal state s since $PDG(s)$ itself is always a dependency closure. If there is more than one such closure, theoretically any dependency closure satisfying the above conditions can be used in EC. In practice, when there are multiple such dependency closures, EC picks the one with fewer actions in order to get stronger reduction. EC has adopted the following scheme to find the dependency closure for any state s :

Given a $PDG(s)$, EC first finds its strongly connected components (SCCs). If each SCC is contracted to a single vertex, the resulting graph is a directed acyclic graph \mathcal{S} . Note that each vertex in \mathcal{S} with a zero out-degree corresponds to a dependency closure. It then topologically sorts all the vertices in \mathcal{S} to get a sequence of SCCs: S_1, S_2, \dots , and picks the smallest m such that S_m includes a goal-related DTG that is not in its goal state. It chooses all the DTGs in S_1, \dots, S_m as the dependency closure.

Now we explain the EC algorithm using the POR theory we developed in Section 4.1. We show that the EC algorithm can be viewed as an algorithm for finding a state-dependent commutative set in each state.

Lemma 3 *For a SAS+ planning task, the EC algorithm defines a state-dependent commutative set for each state.*

Proof: Consider the set of actions $T(s)$ expanded by the EC algorithm in each state s , as defined in (4.3). We prove that $T(s)$ satisfies conditions L1' and A2 in Definition 10.

Consider an action $b \in T(s)$ and actions $b_1, \dots, b_k \notin T(s)$ such that (b_1, \dots, b_k, b) is a prefix of a path from s to a goal state, we show that $s' : b \Rightarrow b_k$, where s' is the state after (b_1, \dots, b_{k-1}) is applied to s .

Let $\mathcal{C}(s)$ be the index set of the DTGs that form a dependency closure, as used in (4.3). Since $b \in T(s)$, there must exist $m \in \mathcal{C}(s)$ such that $b \vdash G_m$. Let the state after applying (b_1, \dots, b_k) to s be s^* . We see that we must have $s_m^* = s_m$ because otherwise there must exist a $b_j, 1 \leq j \leq m$ that changes the assignment of state variable x_m . However, that would imply that $b_k \in T(s)$. Since b is applicable in s^* , we see that $s_m = s_m^* \in pre(b)$.

If there exists a state variable x_i such that an assignment to x_i is in both $eff(b_k)$ and $pre(b)$, then G_m will point to the DTG G_i as s_m is a potential dependent of G_i , forcing G_i to be included in the dependency closure, i.e. $i \in \mathcal{C}(s)$. However, as $b_k \vdash G_i$, it will violate our assumption that $b_k \notin T(s)$. Hence, none of the precondition assignments of b is added by b_k . Therefore, since b is applicable in $apply(s', b_k)$, it is also applicable in s' .

On the other hand, if b_k has a precondition assignment in a DTG that b is associated with, then G_m will point to that DTG since s_m is a potential precondition of b_k , forcing that DTG to be in $\mathcal{C}(s)$, which contradicts the assumption that $b_k \notin T(s)$. Hence, b does not alter any precondition assignment of b_k . Therefore, since b_k is applicable in s' , it is also applicable in the state $apply(s', b)$.

Finally, if there exists a state variable x_i such that an assignment to x_i is altered by both b and b_k , then we know $b \vdash G_i$ and $b_k \vdash G_i$. In this case, G_m will point to G_i since s_m is a potential precondition of G_i , making $b_k \in T(s)$, which contradicts our assumption. Hence, $eff(b)$ and $eff(b_k)$ correspond to assignments to distinct sets of state variables. Therefore, applying (b_k, b) and (b, b_k) to s' will lead to the same state.

From the above, we see that b is applicable in s' , b_k is applicable in $apply(s', b)$, and hence (b, b_k) is applicable in s' . Further we see that (b, b_k) leads to the same state that (b_k, b) does when applied to s' . We conclude that $s' : b \Rightarrow b_k$ and $T(s)$ satisfies L1'.

Moreover, for any goal-related DTG G_i and a state s , if its assignment s_i is not the goal vertex in G_i , then some actions associated with G_i have to be executed in any solution path from s . Since $T(s)$ includes all the actions in at least one goal-related DTG G_i , any solution path must contain at least one action in $T(s)$. Therefore, $T(s)$ also satisfies A2 and it is indeed a state-dependent commutative set. ■

From Lemma 3 and Theorem 5, we obtain the following result, which shows that EC fits our framework as a stubborn set method for planning.

Theorem 7 *For any SAS+ planning task, the EC algorithm defines a stubborn set in each state.*

4.2.2 Explanation of SP

The stratified planning (SP) algorithm is also a POR-based reduction algorithm that exploits commutativity of actions directly [17]. To describe the SP algorithm, we need the following definitions first.

Definition 18 Given a SAS+ planning task Π with state variable set X , the **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 $eff(o)$ has an assignment to x and either $pre(o)$ or $eff(o)$ has an assignment to x' .

Definition 19 For a SAS+ task Π , a **stratification** of the causal graph $CG(\Pi)$ as (X, E) is a partition of the node set X : $X = (X_1, \dots, X_k)$ in such a way that there exists no edge $e = (x, y)$ where $x \in X_i, y \in X_j$ and $i > j$.

By stratification, each state variable is assigned a level $L(x)$, where $L(x) = i$ if $x \in X_i, 1 \leq i \leq k$. Subsequently, each action o is assigned a level $L(o), 1 \leq L(o) \leq k$. $L(o)$ is the level of the state variable(s) in $eff(o)$. Note that all state variables in the same $eff(o)$ must be in the same level. Hence, $L(o)$ is well-defined.

Definition 20 (Follow-up Action) For a SAS+ task Π , an action b is a follow-up action of a (denoted as $a \triangleright b$) if $eff(a) \cap pre(b) \neq \emptyset$ or $eff(a) \cap eff(b) \neq \emptyset$.

The SP algorithm can be combined with standard search algorithms, such as breadth-first search, depth-first search, and best-first search (including A^*). During the search, for each state s that is going to be expanded, the SP algorithm examines the action a that leads to s . Then, for each applicable action b in state S , SP makes the following decisions.

Definition 21 (Stratified Planning Algorithm) For a SAS+ planning task, in any non-initial state s , assuming a is the action that leads directly to s , and b is an applicable action in s , then SP does not expand b if $L(b) < L(a)$ and b is not a follow-up action of a . Otherwise, SP expands b . In the initial state s^J , SP expands all applicable actions.

The following result shows the relationship between the SP algorithm and our new POR theory.

Lemma 4 If an action b is not SP-expandable after a , and state s is the state before action a , then $s : b \Rightarrow a$.

Proof: Since b is not SP-expandable after a , following the SP algorithm, we have $L(a) > L(b)$ and b is not a follow-up action of a . According to Definition 20, we have $eff(a) \cap pre(b) = eff(a) \cap eff(b) = \emptyset$. These imply that $eff(a)$ and $pre(b)$ are conflict-free, and that $eff(a)$ and $eff(b)$ are conflict-free. Also, since b is applicable in $apply(s, a)$ and $eff(a)$ and $pre(b)$ are conflict-free, b must be applicable in s (otherwise $eff(a)$ must change the value of at least one variable in $pre(b)$, which means $eff(a)$ and $pre(b)$ are not conflict-free).

Now we prove that $pre(a)$ and $eff(b)$ are conflict-free by showing $pre(a) \cap eff(b) = \emptyset$. If their intersection is non-empty, we assume a state variable x is assigned by both $pre(a)$ and $eff(b)$. By the definition of stratification, x is in layer $L(b)$. However, since x is assigned by $pre(a)$, there must be an edge from layer $L(a)$ to layer $L(x) = L(b)$ since $L(a) \neq L(b)$. In this case, we know that $L(a) < L(b)$ from the definition of stratification. Nevertheless, this contradicts with the assumption that $L(a) > L(b)$. Thus, $pre(a) \cap eff(b) = \emptyset$, and $pre(a)$ and $eff(b)$ are conflict-free.

With all three conflict-free pairs, we have $s : b \Rightarrow a$ according to Theorem 4. ■

Although SP reduces the search space by avoiding the expansion of certain actions, it is in fact not a stubborn set based reduction algorithm. We have the following theorem for the SP algorithm.

Definition 22 *For a SAS+ planning task S , a valid path $p_a = (a_1, \dots, a_n)$ is an **SP-path** if and only if p_a is a path in the search space of the SP algorithm applied to S .*

Theorem 8 *For a SAS+ planning task S , for any initial s^j and any valid path $p_a = (a_1, \dots, a_n)$ from s^j , there exists a path $p_b = (b_1, \dots, b_n)$ from s^j such that p_b is an SP-path, and both p_a and p_b lead to the same state from s^j , and p_b is a permutation of actions in p_a .*

Proof: We prove by induction on the number of actions.

When $n = 1$, since there is no action before s^j , any valid path (a_1) will also be a valid path in the search space of the SP algorithm.

Now we assume this proposition is true for $n = k, k \geq 1$ and prove the case when $n = k + 1$. For a valid path $p^0 = (a_1, \dots, a_k, a_{k+1})$, by our induction hypothesis, we can rearrange the first k actions to obtain a path $(a_1^1, a_2^1, \dots, a_k^1)$.

Now we consider a new path $p^1 = (a_1^1, \dots, a_k^1, a_{k+1})$. There are two cases. First, if $L(a_{k+1}) < L(a_k^1)$, or $L(a_{k+1}) > L(a_k^1)$ and a_{k+1} is a follow-up action of a_k^1 , then p^1 is already an SP-path. Otherwise, we have $L(a_{k+1}) > L(a_k^1)$ and a_{k+1} is not a follow-up action of a_k^1 . In this case, by Lemma 4, path $p^{1'} = (a_1^1, \dots, a_{k-1}^1, a_{k+1}, a_k^1)$ is also a valid path that leads s to the same state as p_a does.

By the induction hypothesis, if $p^{1'}$ is still not an SP-path, we can rearrange the first k actions in $p^{1'}$ to get a new path $p^2 = (a_1^2, \dots, a_k^2, a_{k+1})$. Otherwise we let $p^2 = p^{1'}$. Comparing p^1 and p^2 , we know $L(a_{k+1}) > L(a_k^1)$, namely, the level value of the last action in p^1 is strictly larger than that in p^2 . We can repeat the above process to generate p^3, \dots, p^m, \dots as long as $p^j (j \in Z^+)$ is not an SP-path. Our transformation from p^j to p^{j+1} also ensures that every p^j is a valid path from s and leads to the same state that p_a does.

Since we know that the layer value of the last action in each p_j is monotonically decreasing as j increases, such a process must stop after a finite number of iterations. Suppose it finally stops at $p^m = (a_1^m, a_2^m, \dots, a_k^m, a_{k+1}^m)$, we must have that $L(a_{k+1}^m) \leq L(a_k^m)$ or $L(a_{k+1}^m) > L(a_k^m)$ and a_{k+1}^m is a follow-up action of a_k^m . Hence, p^m now is an SP-path. We then assign p^m to p_b and the induction step is proved. ■

Theorem 8 shows that the SP algorithm cannot inherently reduce the number of states expanded in the search space. The reason is as follows: for any state in the original search space that is reachable from the initial state s^J via a path p , there is still an SP-path that reaches s . Therefore, every reachable state in the search space is still reachable by the SP algorithm. In other words, SP reduces the number of generated states, but not the number of expanded states.

SP is not a stubborn set based reduction algorithm. We illustrate this using Figure 4.4. Assuming a SAS+ planning task S that contains two state variables x_1 and x_2 , where both x_1 and x_2 have domain $\{0, 1\}$, with the initial state as $\{x_1 = 0, x_2 = 0\}$ and the goal as $\{x_1 = 1, x_2 = 1\}$. Actions a and b are two actions in S where $pre(a)$ is $\{x_1 = 0\}$ and $eff(a)$ is $\{x_1 = 1\}$ and $pre(b)$ is $\{x_2 = 0\}$ and $eff(b)$ is $\{x_2 = 1\}$. It is

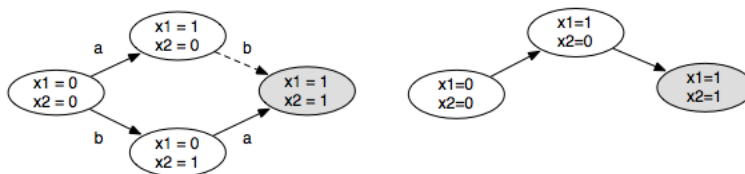


Figure 4.4: The search spaces for a simple SAS+ planning task with two state variables and four states when using SP and EC. SP (on the left) expands all four states while EC (on the right) only expands three. The dashed link on the left graph is the action that is not expanded by SP. Gray nodes are the goal states.

easy to see that a and b are not follow-up actions of each other, and that x_1, x_2 will be in different layers after stratification. Without loss of generality, we can assume $L(a) = L(x_1) > L(x_2) = L(b)$. Therefore, we know that action b will not be expanded after action a in state $s : \{x_1 = 1, x_2 = 0\}$. However, $apply(s, b)$ is the goal state. Not expanding b in state s violates condition $A2$ in Definition 5 where any valid path from s to a goal state has to contain at least one action in the expansion set of s .

Assuming we are using best-first search for solving the above example problem, and the heuristic function values of the initial state s^j , $apply(s^j, a)$, $apply(s^j, b)$ and the goal state are all the same. In this case, the search space explored by SP contains four states: namely, the initial state s^j , $apply(s^j, a)$, $apply(s^j, b)$ and the goal state. Meanwhile, under the EC algorithm, in state s^j , the DTGs for x_1 and x_2 are not in each other's dependency closures. This implies that in s^j , EC expands either action a or b , but not both. Therefore, EC expands three states while SP expands four. This illustrates our conclusion in Theorem 8 that the SP algorithm cannot inherently reduce the number of expanded states, if used with best-first search.

4.3 A New POR Algorithm for Planning

We have developed a POR theory for planning and explained two previous POR algorithms using the theory. Now, based on the theory, we propose a new POR algorithm that leads to stronger reductions than the previous EC algorithm.

Our theory shows in Theorem 5 that the condition for enabling POR reduction is strongly related to commutativity of actions. In fact, constructing a stubborn set

can be reduced to finding a commutativity set. As we show in Theorem 7, the EC algorithm follows this idea. However, the basic unit of reduction in EC is DTG (i.e., either all actions in a DTG are expanded or none of them are), which is not necessary according to our theory. Based on this insight, we propose a new algorithm that operates with the granularity of actions instead of DTGs.

Definition 23 For a state s , an action set L is a **landmark action set** if and only if any valid path starting from s to a goal state contains at least one action in L [60].

Definition 24 For a SAS+ task, an action $a \in \mathcal{O}$ is **supported** by an action b if and only if $pre(a) \cap eff(b) \neq \emptyset$.

Definition 25 For a state s , its **action support graph (ASG)** at s is defined as a directed graph in which each vertex is an action, and there is an edge from a to b if and only if a is not applicable in s and a is supported by b .

The above definition of ASG is a direct extension of the definition of a causal graph. Instead of having domains as basic units, here we directly use actions as basic units. We utilize this action support graph to define the action closure that exhibits helpful attributes for our later algorithm design.

Definition 26 For an action a and a state s , the **action closure** of a at s , denoted by $C_s(a)$, is the set of actions that are in the transitive closure of a in $ASG(s)$. The action closure for a given set of actions A is the union of action closures of every action in A .

The above definition also gives a straightforward way to find action closure given an action – finding a transitive closure on the ASG. In addition, action closure has the following attributes.

Lemma 5 For a state s , if an action a is not applicable in s and there is a valid path p starting from s whose last action is a , then p contains an action $b, b \neq a, b \in C_s(a)$.

Proof: We prove this by induction on the length of p .

In the base case where $|p| = 2$, we assume $p = (b, a)$. Since a is not applicable in s , it must be supported by b . Thus, $b \in C_s(a)$. Suppose this lemma is true for $2 \leq |p| \leq k - 1$, we prove the case for $|p| = k$. For a valid path $p = (o_1, \dots, o_k)$, again there exists an action b before a that supports a . If b is applicable in s , then $b \in C_s(a)$. Otherwise, we consider the subpath $p' = (o_1, \dots, b)$ of p , with $2 \leq |p'| \leq k - 1$. Since b is not applicable, and according to our inductive assumption, there is an action b' in p' that is also in $C_s(b)$, which is a subset of $C_s(a)$, according to Definition 25 and 26. Thus, our proposition is true for $|p| = k$. By induction principle, our lemma is true for any path p . ■

We give some remarks here to motivate our later discussion. In the assumption of this lemma, if there is a nonempty action path p from s such that a is applicable at $apply(s, p)$, we say a is eventually applicable in s . The above lemma essentially ensures that for any eventually applicable action a in s , there is at least one corresponding applicable action b in both the path to a and the action closure $C_s(a)$.

Though the actual conditions are more complicated, informally, we can say that actions in $C_s(a)$ are the only actions of interest as far as the eventual applicability of a is concerned. For instance, if we have a valid action path $p = (c, b, a)$ in s where both b and c are applicable in s , $b \in C_s(a)$ and $c \notin C_s(a)$, we can easily see that $p' = (b, a)$ is still a valid path in s as c doesn't provide any effects or preconditions for a or b (otherwise c would be in $C_s(a)$). That is to say, the eventual applicability of a is unaffected if we ignore c , regardless of its applicability in s . A search procedure can take advantage of this and choose not to expand action c in s . However, two questions arise naturally: 1) how can we ensure the chosen subset of applicable actions are sufficient in terms of not ignoring certain paths to a goal? 2) how can we pick the initial starting point a such that the resulting applicable action set is small? We address these two questions in the following sections. Specifically, we introduce the concept of action core as an amendment to action closure to address question 1, and adopt action landmarks to address question 2.

Definition 27 Given a SAS+ planning task Π with O as set of all actions O , for a state s and a set of actions A , the **action core** of action set A at s , denoted by $AC_s(A)$, satisfies the following conditions:

- $AC_s(A)$ is a subset of O and a superset of $C_s(A)$, the action closure of A ;
- for any applicable action $a \in AC_s(A)$ at s and any action $b \in O \setminus AC_s(A)$, $eff(a)$ and $eff(b)$ are conflict-free and
- if $pre(b) \subset S$, $eff(a)$ and $pre(b)$ are conflict-free.

Intuitively, given a set A as a “seed”, actions in action core $AC_s(A)$ can be executed without affecting the completeness and optimality of search. Specifically, because any applicable action in $AC_s(A)$ and any action not in $AC_s(A)$ will not assign different values to the same state variable, for action $a \in AC_s(A)$ and action $b \in O \setminus AC_s(A)$ at s , path (a, b) will lead to the same state that (b, a) does. Additionally, because $pre(b)$ and $eff(a)$ are conflict-free when $pre(b) \subset s$, executing action a will not affect the applicability of action b in the future. Therefore, actions in $AC_s(A)$ can be safely expanded first during the search, while actions outside it can be expanded later.

A simple procedure, shown in Algorithm 4, can be used to find the action core for a given action set A .

The new POR algorithm, called stubborn action core (SAC), works as follows: at any given state s , the expansion set $E(s)$ of state s is determined by Algorithm 5.

There are various ways to implement the FIND-LANDMARKS procedure for finding landmarks. Richter [63] and Porteous [60] both gave in-depth discussions on the technical details of finding landmarks. Here we give one example that is used in our current implementation. To find a landmark action set L at s , we utilize the DTGs associated with the SAS+ formalism. We first find a transition set that includes all possible transitions (s_i, v_i) in an unachieved goal-related DTG G_i where s_i is the current state of G_i in s . It is easy to see that all actions that mark transitions in this set make up a landmark action set, because G_i is unachieved and at least one action starting from s_i has to be performed in any solution plan.

Algorithm 4: A procedure to find the action core of an action set.

input : A SAS+ task with action set O , an action set $A \subseteq O$, and a state s

output: An action core $AC_s(A)$ of A

$AC_s(A) \leftarrow C_s(A)$;

repeat

```
    foreach action  $a$  in  $AC_s(A)$  applicable in  $s$  do
      foreach action  $b$  in  $O \setminus AC_s(A)$  do
        if  $pre(b) \cap s \neq \emptyset$  and  $pre(b)$  and  $eff(a)$  are not conflict-free then
           $AC_s(A) \leftarrow AC_s(A) \cup \{b\}$  ;
        end
        if  $eff(b)$  and  $eff(a)$  are not conflict-free then
           $AC_s(A) \leftarrow AC_s(A) \cup \{b\}$  ;
        end
      end
    end
```

until $AC_s(A)$ is not changing;

return $AC_s(A)$;

Algorithm 5: The SAC algorithm

input : A SAS+ planning task and state s

output: The expansion set $E(s)$

$L \leftarrow \text{FIND-LANDMARKS}(s)$;

Call Algorithm 4 to find the action core of L as $AC_s(L)$;

return $AC_s(L)$;

There are also other ways to find a landmark action set. For instance, the pre-processor in the LAMA planner [63] can be used to find landmark facts, and all actions that lead to these landmark facts also make up a landmark action set.

Theorem 9 *For a state s , the expansion set $E(s)$ defined by the SAC algorithm is a stubborn set at s .*

Proof: We first prove that our expansion set $E(s)$ satisfies condition A1 in Definition 5, namely, for any action $b \in E(S)$, and actions $b_1, \dots, b_k \notin E(s)$, if (b_1, \dots, b_k, b) is a valid path from s , then (b, b_1, \dots, b_k) is also a valid path, and leads to the same state that (b_1, \dots, b_k, b) does.

To simplify this proof, we can treat action sequence (b_1, \dots, b_k) as a “macro” action B where an assignment $x_t = v_t$ in $pre(B)$ if and only if $x_t = v_t$ is in the precondition of some $b_i \in B$ and $x_t = v_t$ is not in the effects of a previous action $b_j (j < i)$, and an assignment $x_t = v_t$ is in $eff(B)$ if and only if $x_t = v_t$ is in the effect set of some $b_i \in B$, and x_t is not assigned to any value other than v_t in the effects of later action $b_j (j > i)$. In the following proof, we use the macro action B in place of the path (b_1, \dots, b_k) .

To prove A1, we only need to prove that if (B, b) is a valid path, then $s : b \Rightarrow B$. According to Theorem 6, $s : b \Rightarrow B$ if and only if the following four propositions are true.

a) Action b must be applicable in s . We prove this by contradiction. Let $s' = apply(s, B)$, if b is not applicable in s , but applicable in s' , then B supports b . Since all effects of B are from actions in the path (b_1, \dots, b_k) , there exists an action $b_i \in \{b_1, \dots, b_k\}$ such that b_i supports b . However, according to Definition 26, b_i is in the transitive closure of b in $ASG(s)$. According to our algorithm, b_i should be in $E(s)$. This contradicts with our assumption that $b_i \notin E(s)$. Thus, b must be applicable at s .

b) $pre(B)$ and $eff(b)$ are conflict-free. We prove this proposition by contradiction. If $pre(B)$ and $eff(b)$ are not conflict-free, we assume that $pre(B)$ has $x_t = v_t$ that conflicts with an assignment in $eff(b)$. According to the way we define B , there exists

an action $b_i \in (b_1, \dots, b_k)$, such that $x_t = v_t$. Also, since B is applicable in s , we know that x_t takes the value v_t at s also. Therefore, we know that $pre(b_i)$ and $eff(b)$ are not conflict-free. However, according to Definition 27 and Algorithm 4, b_i is in $E(s)$. This contradicts with our assumption that b_i is not in $E(s)$. Thus, $pre(B)$ and $eff(b)$ are conflict-free.

c) $eff(B)$ and $eff(b)$ are conflict-free. The proof of this proposition is very similar to the one above. If they are not conflict-free, we must have action $b_i \in (b_1, \dots, b_k)$, such that $eff(b)$ and $eff(b_i)$ are not conflict-free. However, according to Definition 27 and Algorithm 4, b_i is in $E(s)$. This contradicts with our assumption that b_i is not in $E(s)$. Thus, $eff(B)$ and $eff(b)$ are conflict-free.

d) $pre(b)$ and $eff(B)$ are conflict-free. This proposition is true as we assumed in condition A1 that (B, b) is a valid path from s .

Thus, from Theorem 6, we see that $s : b \Rightarrow B$ and that condition A1 in Definition 5 is true.

Now we verify condition A2 by showing that any solution path p from s contains at least one action in $E(s)$. From the definition of landmark action sets, we know that there exists an action $l \in L$ such that p contains l . From Lemma 5 we know that $AC_s(l)$ contains at least one action, applicable in s , in p . Thus, $E(s)$ indeed contains at least one action in p .

Since $E(s)$ satisfies conditions A1 and A2 in Definition 5, $E(s)$ is a stubborn set in state s . ■

4.3.1 SAC vs. EC

SAC results in stronger reduction than the previous EC algorithm, since it is based on actions, which have a finer granularity than DTGs do. Specifically, SAC causes more reduction than EC for two reasons. First, applicable actions that are not associated with landmark transitions, even if they are in the same DTG, are expanded by EC but not by SAC. Second, applicable actions that do not support any actions in the

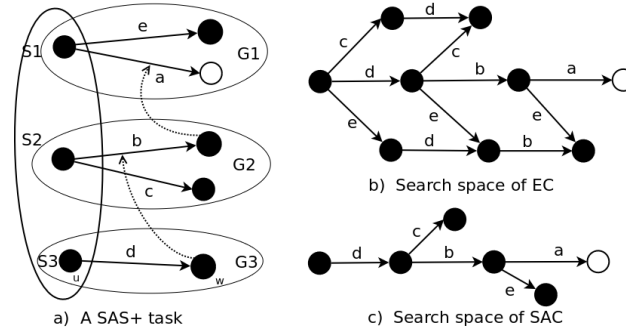


Figure 4.5: Search spaces of EC and SAC

landmark action set, even if they are in the same DTG, are expanded by EC but not by SAC.

To give an example, in Figure 4.5a, G1, G2, G3 are three DTGs. The goal assignment is marked as an unfilled circle in G1. a, b, c, d, e are actions. Dashed arrows denote the preconditions of actions. For instance, the lower dashed arrow means that b requires a precondition $x_3 = w$.

In this example, according to EC, G1 is a goal DTG and G2 and G3 are in the dependency closure of G1. Thus, before executing a , EC expands every applicable action in G1, G2 and G3 at any state. SAC, on the other hand, starts with a singleton set $\{a\}$ as the initial landmark action set and ignores action e . Applicable action c is also not included in the action closure in state s since it does not support a . The search graphs are compared in Figure 4.5 and we see that SAC gives stronger reduction.

4.4 System Implementation

We adopt the Fast Downward (FD) planning system [33] as our code base. The overall architecture of FD is described in Figure 4.6. A complete FD system contains three parts corresponding to three phases in execution: translation, knowledge compilation and search. The translation module will convert planning tasks written in PDDL to a SAS+ planning task. The knowledge compilation module will generate domain transition graphs and causal graph for the SAS+ planning task. The search module

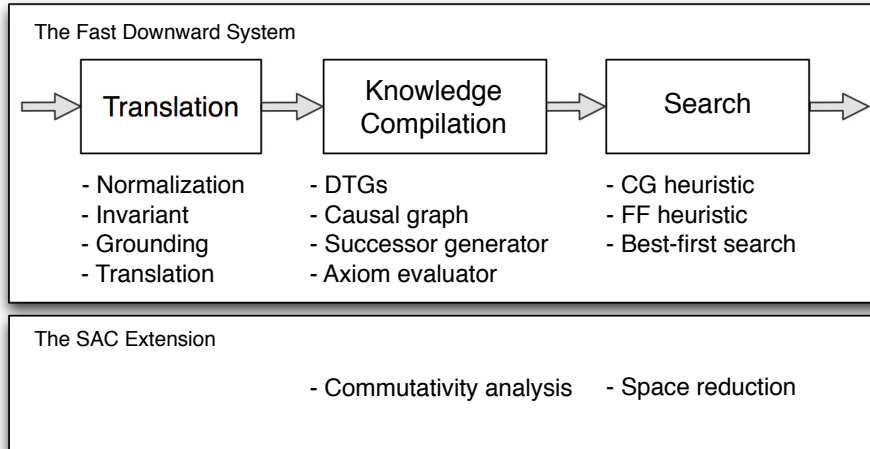


Figure 4.6: System architecture of FD and SAC

implements various state-space-search algorithms as well as heuristic functions. All these three modules communicate by temporary files.

We make two additions to the above system to implement our SAC planning system, as shown in Figure 4.6. First, we add a “commutativity analysis” module into the knowledge compilation step to identify commutativity between actions. Second, we add a “space reduction” module to the search module to conduct state space reduction. The commutativity analysis module is used to build commutativity relations between actions and build the action support graph. It reads action information from the output of knowledge compilation module and determines the commutativity relations between actions according to conditions in Theorem 5. In addition, this module also determines if one action is supported by another and builds the action support graph defined in Definition 25. The reduction module for search is used to generate a stubborn set of a given state. We implement the SAC algorithm in this module. Starting from a landmark action set L as the target action set, we find the action closure $AC_s(L)$ by iteratively adding actions that support actions in the target action set to the target action set until it is not changing. We then use the applicable actions in the action closure as the set of actions to expand at s . In other words, in our SAC system, during the search, for any given state s , instead of using the successor generator provided by FD to generate a set of applicable operators, we use the reduction module to generate a stubborn set in state s and use it as the expansion set.

It is easy to see that the overall time complexity of determining commutativity relationships between actions is $O(|A|^2)$ where $|A|$ is the number of actions. We implement this module in Python. Since the number of actions $|A|$ is usually not large, in most of the cases, the commutativity analysis module takes less than 1 second to finish. This module only runs once for solving a planning problem. Therefore, the commutativity analysis module amounts to an insignificant amount of overhead to the system. Theoretically, the worst case time complexity for finding the action closure is $O(|A|^2)$ where $|A|$ is the number of actions. However, in practice, by choosing the landmark action set L that is associated with transitions in an unarchived goal-related DTG starting from the current state, the procedure of finding action closure terminates quickly after about 4 to 5 iterations. Therefore, adding the reduction module does not increase the overall search overhead significantly either. We implement this module in C++ and incorporate it into the search module of FD.

4.5 Experimental Results

We tested our algorithm on problems in the recent International Planning Competitions (IPCs): IPC 4 and IPC 5. We implemented our algorithm on top of the Fast Downward (FD) planner [33]. We only modified the state expansion part.

We have implemented our SAC algorithm and tested it along with Fast Downward and its combination with the EC extension on a Red Hat Linux server with 2GB memory and one 2.0GHz CPU. The admissible HSP h_{max} heuristic [10] and the inadmissible Fast Forward (FF) heuristic [40] are used in our experiments.

First, we applied our SAC algorithm to A^* search with the HSP h_{max} heuristic [10]. We also turned off the option of preferred operators [33] since it compromises the optimality of A^* search. Table 4.2 shows the detailed results on node expansion and generation during the search. We also compare the solving times for these three algorithms. As we can clearly see from Table 4.2, the numbers of expanded nodes using the SAC-enhanced A^* algorithm are consistently lower than those for the baseline A^* algorithm and the EC-enhanced A^* algorithm. There are some cases where the number of generated nodes for the SAC-enhanced algorithm are slightly larger than

those for the baseline A^* or EC-enhanced A^* algorithm. This is possible due to the tie-breaking of states with equal heuristic values during search.

We can also see that the computational overhead of SAC is low. For instance, in the *feecell* domain, the running time of the SAC-enhanced algorithm is only slightly higher than the baseline and lower than that of the EC-enhanced algorithm, despite their equal number of expanded and generated nodes.

Aside from the A^* algorithm, we also tested SAC on best-first search algorithms. Although POR preserves completeness and optimality, it can also be combined with suboptimal searches such as best-first search to reduce their search space. In this comparison, we turned off the option of preferred operators (also called helpful actions) in our experiment for FD. Preferred operator is another space reduction method that does not preserve completeness, and using it with EC or SAC will lead to worse performance. We will investigate how to find synergy between these two approaches in our future work. We summarize the performance of three algorithms, original Fast Downward (FD), FD with EC, and FD with SAC, in Table 4.3 by presenting the number of problem instances in a planning domain that can be solved within 1800 seconds by each solver. We also ignore small problem instances with solving time less than 0.01 seconds. All three solvers use the inadmissible Fast Forward (FF) heuristic. As we can see from Table 4.3, when combined with a best-first-search algorithm, SAC can still reduce the number of generated and expanded nodes compared to the baseline FD algorithm and the EC-enhanced algorithm. In many problems (e.g. *pipeworld18*, *tpp15*, *truck13*), the reduction in the number of expanded nodes can be of orders of magnitude.

Based on their performances, we can divide the test domains into three groups. The first group of domains exhibits strong commutativity between actions and also some level of interdependency between DTGs. Compared to FD, EC can reduce the number of expanded nodes for these domains, while SAC can reduce the number of expanded nodes even further. Example domains in this group include *pipeworld* and *tpp*. The second group of domains has some commutativity between actions. However, these commutativity relationships cannot be reflected by dependency analysis on the DTG level. On these domains, EC will perform similarly to or sometimes even worse than FD due to high runtime overhead. Our proposed SAC algorithm, on the other hand,

can reduce the number of expanded nodes with less runtime overhead. Thus, on these domains, SAC is clearly better than both FD and EC. Example domains in this group include *driverlog* and *trucks*. The rest are domain groups that have few commutativity between actions. Neither SAC or EC can reduce much of the search space compared to FD. POR techniques are not effective for these domains. Example domains in this group include *airport*, *stroage*, and *rovers*. We can see that the performance of SAC on domains in this group is still comparable to FD, despite the computation overhead of SAC.

Aside from Table 4.3, we also compare performances of SP and SAC in Table 4.4. As we discussed in Section 4.2, SP is not a stubborn set method. Moreover, since SP cannot inherently reduce the number of nodes in optimal searches, our results are based on non-optimal search. We again use the FF heuristics for both SP and SAC with no preferred operators, and compared their performances against their common root, the Fast Downward planner, using the same heuristic function. The set of planning domains used in Table 4.4 is identical to the one we reported in Table 4.3. We pick the first N problems in each domain to conduct the experiment where N is the number indicated in the parentheses after the domain name. For instance, we test the first 25 problems in the *airport* domain. The numbers reported for each domain are average values for each individual problem in that domain.

We see from Table 4.3 that SAC out-performs SP on a set of domains including *driverlog*, *freecell*, *pipesworld*, *tpp*, *trucks*, and *pathway*. It is worth noting that the state-independent causal graphs for problems in *freecell*, *pipesworld*, and *pathway* are all strongly connected, resulting in no stratification of these problems. In these cases, SP rolls back to FD. SAC, on the other hand, can still reduce the search space by finding state-dependent commutative action pairs. SP can also lead to expanding many more states than FD, as illustrated in the *trucks* domain. We investigated this further and found that SP can prune goal paths when the search is very close to a goal, resulting a detour in search. Neither EC nor SAC would prune goal paths during the search.

Table 4.2: Comparison of FD, EC, and SAC using A* with h_{max} heuristic on IPC domains. We show numbers of expanded and generated nodes. “_” means timeout after 300 seconds. For each problem, we also highlight the best values of expanded and generated nodes among three algorithms, if there is any difference.

Domain	FD			EC			SAC		
	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time
airport1	9	9	0	9	9	0	9	9	0
airport2	16	17	0	16	17	0	16	17	0
airport3	38	102	0	35	90	0.01	<u>27</u>	<u>43</u>	0.01
airport4	21	21	0	21	21	0.01	21	<u>20</u>	0.01
airport5	22	28	0	22	28	0.01	22	<u>23</u>	0.01
airport6	138	335	0.01	120	230	0.07	<u>86</u>	<u>202</u>	0.06
airport7	221987	5305641	81.02	221987	5305641	91.07	<u>221901</u>	<u>709575</u>	74.95
airport8	2420	11190	0.73	2364	6621	1.93	<u>699</u>	<u>2860</u>	0.56
airport9	11005	60058	4.66	11005	39027	16.35	<u>4923</u>	<u>24244</u>	5.86
airport10	19	20	0.01	19	20	0.01	19	<u>18</u>	0.01
airport11	22	28	0.01	22	28	0.01	22	<u>23</u>	0.01
airport12	122	328	0.03	104	119	0.07	<u>80</u>	<u>195</u>	0.06
airport13	112	295	0.01	94	182	0.07	<u>76</u>	<u>187</u>	0.08
airport14	2300	11144	0.84	2246	6324	2.14	<u>626</u>	<u>2700</u>	0.61
airport15	1910	9240	0.69	1904	5396	1.94	<u>493</u>	<u>2124</u>	0.46
depot1	159	1000	0.01	159	1000	0.02	159	<u>981</u>	0.02

continued on next page

continued from previous page

Domain	FD			EC			SAC		
	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time
depot2	2294	17803	0.01	2310	17894	0.52	2294	16404	0.34
depot3	2389	21172	0.2	2389	20724	0.3	2389	21168	0.76
depot4	42435	366989	5.08	42435	366989	7.08	42362	364883	4.35
depot5	13096	119388	2.35	13096	119388	4.15	13096	119388	3.07
depot7	9672	91795	0.81	9672	91795	1.89	9658	91460	1.73
depot8	173184	1999709	40.15	173184	1999709	60.8	173157	1993740	51.68
drivelog1	57	373	0	57	355	0	30	183	0
drivelog2	55780	417679	2.35	55387	393334	3.2	52124	325004	4.12
drivelog3	2982	22693	0.12	2858	21247	0.16	2682	19774	0.13
drivelog4	460727	4798803	31.33	446341	4175538	35.38	414148	4004775	28.65
drivelog5	2077987	24224013	167.86	2040088	22120590	202.42	1943657	22156584	168.73
freecell1	63	407	0.01	63	407	0.01	63	407	0.03
freecell2	212	1603	0.06	212	1603	0.18	212	1603	0.23
freecell3	162	1156	0.07	162	1156	0.37	162	1156	0.26
freecell4	792	4765	0.35	792	4765	2.35	792	4765	1.76
freecell5	526	2930	0.49	526	2930	1.81	526	2930	1.81
freecell6	430	2834	0.88	430	2834	3.39	430	2834	2.38
freecell7	1429	8347	2.5	1429	8347	5.5	1429	8347	4.02
freecell8	1682	12015	6.68	1682	12015	16.21	1682	12015	14.66
freecell9	2001	12122	7.44	2001	12122	18.42	2001	12122	16.32

continued on next page

continued from previous page

Domain	FD			EC			SAC		
	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time
freecell10	1953	14383	11.66	1953	14383	31.57	1953	14383	23.54
rover1	323	2043	0	323	2043	0.01	114	558	0
rover2	161	1019	0	161	1019	0	64	239	0
rover3	863	5724	0.01	863	5724	0.04	390	2007	0.01
rover4	291	2399	0	291	2399	0.01	79	397	0
rover5	324204	5714884	6.42	324204	5714884	9.23	95369	1093514	5.1
rover6	-	-	-	-	-	-	251276	2437578	10.48
rover7	156150	2179859	2.64	156150	2179859	5.21	34772	387527	2.01
truck1	390	4306	0.03	390	4306	0.03	390	1145	0.04
truck2	943	13212	0.01	943	13212	0.12	942	2432	0.11
truck3	9162	178481	1.12	9162	178481	1.82	9157	44098	1.03
truck4	37799	258286	1.77	8792	15177	1.77	6841	12570	1.51

Table 4.3: Comparison of FD, EC and SAC with no-preferred operators on IPC’s domains. We show numbers of expanded and generated nodes. “_” means timeout after 1800 seconds. For each problem, we also highlight the best values of expanded and generated nodes among three algorithms if there is any difference.

Domains	FD		EC		SAC				
	Expanded	Generated	Time	Expanded	Generated	Time			
airport13	<u>43</u>	106	0.03	46	102	0.08	45	<u>68</u>	0.05
airport14	78	321	0.07	70	199	0.19	70	<u>161</u>	0.12
airport15	67	224	0.06	66	163	0.2	<u>64</u>	<u>141</u>	0.12
airport16	310	1541	0.35	315	1549	1.05	310	<u>740</u>	0.62
airport17	815	4682	1.15	21819	140496	101.16	<u>809</u>	<u>2315</u>	2.1
airport18	<u>18653</u>	142281	44.3	82007	655661	529.57	18712	<u>76214</u>	78.78
airport19	10564	65299	15.07	10621	65480	48.12	<u>5041</u>	<u>17123</u>	15.5
airport20	<u>25377</u>	182487	52.94	150816	1272996	959.32	25636	<u>116148</u>	53.58
airport21	102	274	0.24	102	256	1.57	102	<u>193</u>	0.56
airport22	149	524	0.51	150	509	3.34	149	<u>370</u>	1.52
airport23	169	620	0.81	169	561	6.03	169	<u>500</u>	1.51
airport24	166	888	1.14	1902	7459	70.58	<u>111</u>	604	1.08
airport25	33751	197109	256.95	-	-	-	<u>18752</u>	<u>66186</u>	161.84
driverlog11	24	284	0.01	24	284	0.02	87	949	0.05
driverlog12	78	1059	0.05	<u>64</u>	<u>224</u>	0.04	150	1875	0.13
driverlog13	583	6396	0.25	75	864	0.06	75	864	0.07

continued on next page

continued from previous page

Domains	FD		EC		SAC		
	Expanded	Generated	Expanded	Generated	Expanded	Generated	Time
driverlog14	248	3378	75	883	75	416	0.05
driverlog16	64971	1331293	50855	1017221	38746	737836	142.11
driverlog17	24860	804116	149938	5012829	15620	537050	138.37
driverlog19	834	26406	727	14443	573	1105	5.19
freecell1	10	59	11	63	10	54	0.08
freecell2	17	114	17	114	17	100	0.01
freecell3	20	144	19	122	20	127	0.13
freecell4	48	268	61	401	35	238	0.17
freecell5	88	510	123	668	63	236	0.35
freecell6	98	510	97	496	58	384	0.7
freecell7	412	1594	233	884	215	814	2.23
freecell8	106	700	71	436	85	501	1.13
freecell9	68	500	112	667	122	719	3.3
freecell10	797	5043	4057	34183	112	597	5.49
freecell11	120	530	119	620	118	527	4.21
freecell12	192	1085	58	420	96	491	3.51
freecell13	412	2655	589	2896	592	2996	13.16
freecell14	287	2274	339	2054	78	560	8.86
freecell15	1913	8642	531	2795	248	1197	18.21
freecell16	3818	31173	409	2381	371	1722	7.14

continued on next page

continued from previous page

Domains	FD		EC		SAC	
	Expanded	Generated	Time	Expanded	Generated	Time
freecell17	246	1402	19.71	244	2849	23.62
freecell18	3480	17263	354.43	6069	32509	685.41
pipesworld7	13	685	1.19	13	685	0.73
pipesworld8	12	550	0.79	12	473	0.74
pipesworld9	319	8297	10.05	165	5002	6.31
pipesworld10	50	1572	2.92	115	3322	5.14
pipesworld11	197	970	0.34	451302	2168501	459.77
pipesworld13	-	-	-	8083	54502	40.44
pipesworld14	3412	16652	6.01	2520	11302	4.33
pipesworld15	568191	2792188	1284.82	561869	2776786	1405.04
pipesworld17	156733	561923	292.99	89829	334833	188.93
pipesworld18	21164	230257	1049.83	491	5057	30.25
pipesworld20	-	-	-	-	-	-
pipesworld21	4486	29504	32.43	-	-	-
tpp8	675	4987	0.12	570	3854	0.14
tpp9	570	3839	0.09	960	7430	0.28
tpp10	585	5861	0.13	3679	37791	1.18
tpp11	6294	68216	3.21	3815	33688	1.93
tpp12	12922	139267	7.42	6888	63805	3.88
tpp13	7757	91630	6.8	5554	70013	5.49
				104	740	10.99
				3036	19476	174.69
				13	685	0.71
				12	531	0.79
				166	4996	5.74
				168	4684	7.4
				1615	7311	4.24
				4272	20342	6.16
				2815	13066	10.66
				93320	802500	316.24
				93169	324196	140.06
				491	2948	28.16
				1417	30469	596.39
				896	4412	8.38
				581	4735	0.16
				1184	10098	0.29
				442	4186	0.14
				2678	28276	1.75
				1947	22579	1.56
				4555	58212	4.68

continued on next page

continued from previous page

Domains	FD		EC		SAC		
	Expanded	Generated	Expanded	Generated	Expanded	Generated	Time
tpp14	12291	1705578	15700	211138	6054	76611	8.02
tpp15	9203	112319	13788	181676	3149	36960	4.82
tpp16	-	-	48549	783502	17808	256920	61.04
tpp20	-	-	-	-	278126	4503504	1309.18
storage13	1353	4606	1354	4607	1634	5645	0.68
storage14	265	2727	212	2137	172	1829	0.26
storage15	57	603	59	622	56	629	0.18
storage16	52	697	31	412	26	357	0.22
storage17	308	4128	337	4480	339	4553	1.15
storage18	361	5628	482	6173	346	5402	2.42
storage19	133171	1426180	133771	1442383	133425	1436330	619.37
trucks1	22	230	22	230	22	74	0.01
trucks2	327	4434	327	4434	113	298	0.03
trucks3	35	672	35	672	34	148	0.05
trucks4	40	931	40	931	40	227	0.02
trucks5	37	1255	37	1255	35	244	0.05
trucks6	47	1822	47	1822	41	255	0.07
trucks7	23913	525066	23913	525066	23914	80585	8.32
trucks8	300	9217	300	9217	257	798	0.17
trucks9	3390	154042	3390	154042	3390	154042	2.95

continued on next page

continued from previous page

Domains	FD			EC			SAC		
	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time
trucks10	139416	8111005	179.09	139416	8111005	186.29	86993	218926	46.46
trucks11	4715	344079	13.89	4715	344079	14.25	4526	12044	8.38
trucks12	320765	30119993	1357.29	320765	30119993	1352.82	320790	1068035	518.89
trucks13	525967	27598625	1181.73	525967	27598625	1182.49	289825	698071	349.43
trucks14	37380	863543	46.23	37380	863543	47.36	20713	496998	36.68
trucks15	59659	1356218	79.9	59659	1356218	84.26	57584	148886	62.78
trucks17	-	-	-	-	-	-	159519	452932	553.03
rovers7	30	464	0.01	30	464	0.01	20	224	0.01
rovers8	36	866	0.01	36	866	0.02	28	554	0.01
rovers9	281	5865	0.06	368	8349	0.11	113	1148	0.02
rovers10	232	7321	0.09	232	7321	0.12	81	1186	0.03
rovers11	1074	24425	0.28	1074	24425	0.35	418	7754	0.13
rovers12	26	531	0.01	26	531	0.01	26	438	0.01
rovers13	175	4523	0.09	175	4523	0.12	191	3685	0.1
rovers14	81	2075	0.03	81	2075	0.04	215	2424	0.06
rovers15	570	16587	0.25	570	16587	0.29	499	15850	0.17
rovers16	545	13945	0.24	545	13945	0.27	103	1771	0.05
rovers17	-	-	-	445	16811	0.54	172	2319	0.11
rovers18	37590	1532969	52.87	37590	1532969	63.4	4801	110878	4.77
openstack5	39	171	0.01	39	171	0.01	39	170	0.01

continued on next page

continued from previous page

Domains	FD			EC			SAC		
	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time
openstack6	109	712	0.05	109	712	0.06	109	712	0.08
openstack7	100	673	0.05	100	673	0.06	100	672	0.09
pathway3	28	344	0.01	28	344	0.01	22	84	0.01
pathway4	76	726	0.01	76	726	0.03	61	624	0.01
pathway5	47	1296	0.02	47	1296	0.03	37	1061	0.02
pathway6	255	6402	0.12	255	6402	0.28	68	626	0.04
pathway7	21147	750778	20.76	21147	750778	47.21	21147	533384	17.28
pathway8	21507	858174	24.93	-	-	-	39088	472722	31.25
pathway9	-	-	-	471	13639	1.39	151	2155	0.19

Table 4.4: Comparison of SP and SAC to FD on IPC domains without preferred operators. We show ratios in the table for SP and SAC compared to FD. We use 2-stratification for SP. Numbers in the parentheses after domain names are the number of problems in that domain that we ran experiments on. Smaller values in the table indicate better performance. Domains marked with a * are not stratifiable, which causes SP to roll back to FD.

Domains	Ratio of SP to FD		Ratio of SAC to FD	
	Expanded	Generated	Expanded	Generated
airport(25)	0.78	0.51	0.65	0.65
depot(10)	0.41	1.3	1.05	1.09
driverlog(18)	1.26	1.99	0.79	0.78
freecell*(20)	1.0	1.0	0.54	0.51
pipesworld*(21)	1.0	1.0	0.87	0.61
tpp(20)	0.41	0.59	0.31	0.34
storage(20)	0.77	0.6	1.06	0.91
trucks(16)	1.52	2.22	0.65	0.46
pathway*(10)	1.00	1.00	0.90	0.78

4.6 Summary

Previous work in both model checking and AI planning has demonstrated that POR is a powerful method for reducing search costs. POR is an enabling technique for model checking, which would not be practical without POR due to its high complexity. Although POR has been extensively studied for model checking, its theory has not been developed for AI planning. In this chapter, we developed a new POR theory for planning that is parallel to the stubborn set theory in model checking.

In addition, by analyzing the structure of actions in planning problems, we derived a practical criterion that defines commutativity between actions. Based on the notion of commutativity, we developed sufficient conditions for finding stubborn sets during search for planning. Furthermore, we applied our theory to explain two previous POR algorithms for planning. The explanation provided useful insights that lead to a stronger and more efficient POR algorithm called SAC. Comparing to previous POR algorithms, SAC finds stubborn sets based on a finer granularity for checking commutativity, leading to stronger reduction. We compared the performance of SAC to the previously proposed EC algorithm on both optimal and non-optimal state space searches. Experimental results showed that the proposed SAC algorithm led to stronger node reduction and less overhead.

Chapter 5

Accelerating Heuristic Search with Random Walks

Partial order reduction techniques proposed in the previous chapters are approaches that leverage the overall structure of search space. In this chapter, we study the local structure of state space as defined by heuristic functions to accelerate heuristic search. In particular, we study the behaviors of heuristic search in local regions where heuristic functions are not informative.

A well-observed phenomenon in heuristic search is that the search algorithm may explore a large number of states without reducing the heuristic function value. This phenomenon, called “plateau exploration”, has been extensively studied in satisfiability (SAT) and constraint satisfaction problems (CSP). In heuristic search, plateau exploration takes up the majority of the search time. Therefore, to accelerate heuristic search, it is important to study ways to accelerate plateau exploration.

In this chapter, we introduce a random walk assisted search algorithm framework. We also establish a theoretical model to analyze the conditions under which random walk is helpful to heuristic search in finding plateau exits. We show the effectiveness of the proposed algorithm by presenting experimental results from recent IPC domains and the Seventh International Planning Competition [5].

5.1 Background

In heuristic search, the number of states explored depends largely on the quality of the heuristic function. In the ideal case, a search with the perfect heuristic function that accurately calculates the distance from any state to goal state would only expand $O(L)$ states, where L is the distance from the initial state to goal. In reality, heuristic search usually explores an exponential number of states as the problem size grows. Heuristic search, even with almost perfect heuristic guidance, may still lead to high search cost for optimal planning [37].

To further understand the impact of heuristics for state space search, we view the search procedure from another perspective— as an optimization procedure that aims to find a state that minimizes the heuristic function value. Here we assume that the heuristic function takes value 0 if and only if it is at some goal state. From this perspective, an immediate insight is that a search is making progress if the best heuristic value found so far is decreasing. To measure this kind of progress, we monitor the *incumbent heuristic value* \hat{h} . Intuitively, during search, for any state s that is removed from the *open* list, the incumbent heuristic value $\hat{h}(s)$ is the smallest heuristic value of all states explored up to s .

For any heuristic search, its incumbent heuristic value decreases monotonically during search and finally reaches 0 when a goal is found. In a typical search where the number of explored states is much larger than the solution length, most consecutive states removed from the *open* list have the same incumbent heuristic value. When heuristic functions are not informative, search can halt on the same incumbent heuristic value for a long time. We call this phenomenon *plateau exploration*.

Inspired by the work of Nakhost *et al.* [56] on using Monte-Carlo Random Walk in solving planning problems, we propose to use random walk procedures to assist heuristic searches. Specifically, we invoke an episode of random walks within a heuristic search when a plateau is encountered (i.e., when the search cannot improve the incumbent heuristic value for an extended period of time).

In this study, we find three advantages of using random walks to assist heuristic search for planning. First, a random walk has the potential to quickly find a state

that reduces the incumbent heuristic value, by jumping out of local minima or jumping over a local maxima with respect to the heuristic function. In contrast, a deterministic heuristic search will have to explore all possible states around the local minima or before the local maxima. Second, compared to heuristic search in which heuristic functions are evaluated at each state, the random walk algorithm can skip heuristic evaluations of most intermediate states during exploration, making space exploration more efficient. Third, random walks require little memory, and therefore do not add space complexity to the original heuristic search. One limitation of using random walks is that the solution found by random walk is no longer guaranteed to be optimal even if it is combined with A^* search. For this reason, we focus on accelerating best-first search, a heuristic search procedure that finds satisfying plans while trying to minimize plan cost.

5.2 Local Properties of Search Space

Unlike local search which always explores neighbor states next (subject to backtracking), heuristic search always fetches the next state from the *open* list ordered by the heuristic function, regardless of whether it is the immediate successor of the current state. To facilitate our study of the local plateau structure of search space, we base our discussion on a local region of the state space.

Definition 28 (Neighbors in a State Space) *Given a state space \mathcal{S} and a state $s \in \mathcal{S}$, an l -neighbor $\mathcal{N}(s, l)$ of s , where l is a positive integer, is a set of all states that can be reached from s within l steps in \mathcal{S} . A special neighbor $\mathcal{N}(s, \infty)$ is the set of all states that are reachable from s .*

For heuristic search in the complete state-space, the objective of heuristic search is to find states that can reduce the current incumbent heuristic value. When discussion is limited on a local region $\mathcal{N}(s, l)$ where s is the starting point, we use $h(s)$ as the incumbent heuristic value in analysis. Namely, $\hat{h}(s) = h(s)$. A heuristic search procedure can be viewed as a multi-phase search where the objective of each phase is to find a state s_e in some l -neighbor $\mathcal{N}(s, l)$ of s such that $h(s_e) < h(s)$. We define state s_e as an *exit* state of $\mathcal{N}(s, l)$ if $h(s_e) < h(s)$. A region $\mathcal{P} \subseteq \mathcal{N}(s, l)$ is a *plateau* if

every state $s_p \in \mathcal{P}$ has $h(s_p) \geq h(s)$. Note that the definition of s_e is dependent on l , the size of the l -neighbor. In the following discussions, we assume l is predetermined.

Similar to the heuristic search conducted on the search space, heuristic search conducted on $\mathcal{N}(s, l)$ exploring states in orders that are determined by both the topological structure of $\mathcal{N}(s, l)$ and the heuristic function. We introduce the following definitions to capture these orders.

Definition 29 (Order of States) *Given a heuristic search procedure \mathcal{A}_h in an l -neighbor $\mathcal{N}(s, l)$ of s , the sequence of states in $\mathcal{N}(s, l)$ explored by \mathcal{A}_h , denoted by \mathcal{L} , is an ordered list. A relation $<$ can be defined between states in $\mathcal{L} = s_1, \dots, s_r$ where for any $s_i, s_j \in \mathcal{L}$, $s_i < s_j$ if and only if $i < j$.*

We would like to point out that the order of states for heuristic search \mathcal{A}_h depends on the local topology of the search space as well as the heuristic function. To understand the interactions between these two, we define the natural order and heuristic order of states.

Definition 30 (Natural Order of States) *Given a state s_0 and two states $s_1, s_2 \in \mathcal{N}(s_0, l)$, s_1 is naturally ordered before s_2 if any path from s_0 to s_2 contains s_1 . In other words, to reach s_2 from s_0 , search has to reach s_1 first.*

Natural order between states is a well-defined partial order relation. We can show this by validating the reflexivity, antisymmetry and transitivity of this relation. We denote this relation by $s_1 \leq_n s_2$ if s_1 is naturally ordered before s_2 , and $s_1 <_n s_2$ if $s_1 \leq_n s_2$ and $s_1 \neq s_2$.

Note that *reflexivity* holds for any state s because any path from s_0 to s contains state s itself. *Antisymmetry* is also straightforward to verify. For any two states a and b , such that $a \leq_n b$ and $b \leq_n a$, we show $a = b$. Without loss of generality, we assume the shortest path from s_0 to a and b is a path from s_0 to b , of length m . Since we have $a \leq_n b$, by definition a is on the path from s_0 to b . However, since $b <_n a$, b is also on the path from s_0 to a . This can only happen when $a = b$, because if a and b are different, then the subpath from s_0 to a is shorter than m , contradicting

our assumption that s_0 to b is shorter than s_0 to a . *Transitivity* also holds for natural orders. If we have $s_1 \leq_n s_2$ and $s_2 \leq_n s_3$, then any path from s_0 to s_3 would include s_2 , and every path from s_0 to s_2 includes s_1 . This implies any path from s_0 to s_3 includes s_1 , or $s_1 \leq_n s_3$.

Natural order is the topological order of states that any search process must obey. Heuristic function works on top of natural order to order all states according to their heuristic function values.

Definition 31 (Heuristic Order of States) *Given a state s and two states $s_1, s_2 \in \mathcal{N}(s, l)$ that neither $s_1 \leq_n s_2$ nor $s_2 \leq_n s_1$ is true, we have relation $s_1 <_h s_2$ if $h(s_1) < h(s_2)$, or if $h(s_1) = h(s_2)$ and a tie-breaking process puts s_1 before s_2 when they are retrieved from the open list.*

In Definition 31, we assume there is a tie-breaking process that gives orders to states that share the same heuristic value in the *open* list. How tie breaking is conducted is not essential to our discussion here. We can safely assume that there is a deterministic tie-breaking process for heuristic search.

It is intuitive to see how the order of state exploration in a heuristic search is related to natural order and heuristic order. For instance, if we have two different states s_1, s_2 such that $s_1 \leq_n s_2$ holds, a heuristic search procedure would explore s_1 before s_2 . If both s_1 and s_2 are in the *open* list during a heuristic search, and $s_1 <_h s_2$ holds, a heuristic search would explore s_1 before s_2 . We formalized and extend this observation in Theorem 10.

Theorem 10 *Given a state s_0 and two different states s_1, s_2 in $\mathcal{N}(s_0, l)$, and their common ancestor s_a , a heuristic search \mathcal{A}_h on $\mathcal{N}(s_0, l)$ explores state s_1 before s_2 (i.e. $s_1 < s_2$) if and only if one of these two conditions holds:*

C1 $s_1 <_n s_2$,

C2 *there exists a state s_3 such that $s_a <_n s_3 \leq_n s_2$ and $s_1 <_h s_3$.*

Proof: Under the assumption that heuristic search explores s_1 before s_2 , we first examine the natural order between s_1 and s_2 . It is easy to see that we have either $s_1 <_n s_2$ (C1), or the natural order between s_1 and s_2 is not defined. It is not possible to have $s_2 <_n s_1$ because heuristic search would explore s_2 before s_1 .

When the natural order between s_1 and s_2 are not defined, we look at the set of states $S_{a \rightarrow 2}$ on the path from their least common ancestor s_a to s_2 , excluding s_a . This set is well-defined. At least one element, s_2 , is in $S_{a \rightarrow 2}$ because $s_2 \neq s_a$.

For any state $s \in S_{a \rightarrow 2}$, we have either $s_1 < s$ or $s < s_1$. Note that among all states s in $S_{a \rightarrow 2}$ such that $s_1 < s$, we pick the state that has the minimal natural order and denote it by s_3 . State s_3 exists because there is at least one state, $s_2 \in S_{a \rightarrow 2}$, such that $s_1 < s_2$.

Now we consider the immediate predecessor of s_3 , s'_3 . Since s_3 has the minimal natural order among all states that are explored after s_1 , s'_3 must be explored *before* s_1 , whereas s_3 is explored *after* s_1 in heuristic search. In other words, we have $s'_3 < s_1 < s_3$.

Note that when heuristic search explores s'_3 , s_3 , an immediate successor of s'_3 , is inserted into the *open* list by heuristic search. This means $s_3 <_h s_1$ holds. Otherwise, s_3 would be explored before s_1 by heuristic search. Hence, there exists a state $s_3 \in S_{a \rightarrow 2}$, such that $s_a <_n s_3 \leq_n s_2$ and $s_1 <_h s_3$.

On the other hand, if condition C1 is true, by definition $s_1 < s_2$. If C2 holds, we have $s_1 < s_3 \leq_n s_2$. Therefore, $s_1 < s_2$. ■

It is intuitive to understand that inaccurate heuristic values often lead to inefficient heuristic search. Theorem 10 provides a direct interpretation on why heuristic search can be ineffective in a local region. The necessary and sufficient conditions in Theorem 10 directly translate to the following two scenarios.

Scenario 1: Local minima (traps). Local minima (traps) are states where all other states in the neighbor have higher heuristic value.

Let us assume there is a path $s_0, s_1, \dots, s_{i-1}, s_e$ from state s_0 to an exit state $s_e \in \mathcal{N}(s_0, l)$. Ideally, heuristic search from s_0 would advance towards s_e in i steps. According to Theorem 10, when a state s_i is in the *open* list, any state s' in the *open* list that satisfies $s' <_h s_i$ would be explored before s_i . If s' is not part of any exit path, then heuristic search is *trapped* at s' instead of advancing the search along the exit path.

To make this worse, when heuristic function underestimates, it tends to underestimate a set of connected states in a local region, as they may share a similar relaxation. These states would be ordered before s_i according to Theorem 10, and therefore be explored before s_e . These states trap the heuristic search into local minima regions before the search can explore states along the exit path.

It is worth pointing out that heuristic underestimation is not the only cause for trapping. Recall that Theorem 10 only requires $s' <_h s_i$ when s' is explored before s_i . Even if heuristic function is almost perfect in that it does not underestimate the goal distance most of the time, and does make the heuristics of s_i and s' the same, the tie breaking process could still order s' before s_i . These cases can lead to an exponential number of states explored as the search progress for optimal search. This tragic scenario is demonstrated in [37].

Scenario 2. Local maxima (blocks). Local maxima (blocks) are states on an exit path that has heuristic values larger than neighbor states.

This scenario arises from the overestimation of heuristic values on the goal path. Ideally, states along the exit path would have monotonically decreasing heuristic values ($h(s_0) > h(s_1) > \dots > h(s_e)$). If $h(s_i)$ on the exit path is a local minima and s_i is in the *open* list, then any state s' in the *open* list, including the neighbor states of s_i , would get explored before s_i as $s' <_h s_i$. If s_i is naturally ordered before s_e , it becomes a block on the exit path, because the heuristic search would have to explore every state s' such that $s' <_h s_i$. Even when heuristics are accurate for all other states, a block on the exit path still leads to inefficient searches.

Modern heuristic functions such as Fast Forward (h_{ff}), Landmark Count (h_{lm}) and Fast Downward h_{cg} utilize problem relaxation and satisfiable plan extraction to calculate heuristics. More often than not, these inadmissible heuristics create traps and blocks in the search space, leading to plateau exploration during search.

5.2.1 Approaches for Accelerating Plateau Exploration

Several lines of work are available for accelerating plateau exploration in heuristic search.

First, multiple heuristic functions can be used to sort states in the *open* list in different orders [33]. Since different heuristic functions lead to different heuristic orders of states, when one heuristic function encounters traps or blocks on a plateau, other heuristics may give informative guidance and find exits from a plateau. However, extra heuristic function calculations and extra *open* lists can increase the overall time and space complexity of the search algorithm.

Second, Monte-Carlo Random Walk (MCRW) algorithms are capable of escaping from local minima, and have been used to solve planning problems with good performance [56]. However, planners using solely stochastic search strategies are generally slower than deterministic heuristic search planners. Stochastic search also does not perform well on problems with many dead end states.

5.3 Random Walk Assisted Best-First Search

The natural order of states are defined by the state space, while heuristic order is defined by heuristic function. At first glance, there is not much we can do to accelerate heuristic search unless we improve the quality of the heuristic functions themselves.

In the local search region, however, we can devise techniques to accelerate plateau exploration by avoiding traps and blocks. Our proposed search framework is inspired by both the MCRW approach [56] and the multiple heuristic search [33] approach. We use a best-first search procedure to conduct heuristic search most of the time, as

best-first search gives good performance when the heuristic functions are accurate. In addition, when a plateau is detected, a random walk procedure is invoked to assist the best-first search, with an aim of quick escape from the plateau.

5.3.1 Algorithm Framework

We use random walk as the main technique to enhance heuristic search for planning. We start from the definition of random walk.

Definition 32 (Random Walk in State Space) *Given a state space \mathcal{S} and a states $s \in \mathcal{S}$, a random walk is a path in \mathcal{S} such that: each state (other than the first state s) in the path is a randomly chosen successor of the state before it. The length of the path is defined as the length of the random walk, and the first/last state in the path are called the start/end of the random walk. A random walk starting from s with length l is denoted by $w(s, l)$.*

In solving planning problems, examining one random walk is hardly useful in finding a goal state. We introduce *random advancement*, which consists of a group of random walks with the same starting state. Random advancement is a procedure that invokes multiple random walks to find the best possible state in the local neighbor of a state. Formally, we define it as follows.

Definition 33 (Random Advancement) *Given a state space \mathcal{S} , a state $s \in \mathcal{S}$ and a heuristic function h , a `RANDOM ADVANCEMENT(s, l, n)`, where l and n are positive integers, consists of n random walks $w(s, l)$ and returns the end state s^* with the minimum heuristic value.*

Algorithm 6 illustrates the details of conducting a random advancement on state s . Given parameter n , l and the starting state s , a `RANDOM ADVANCEMENT(s, l, n)` invokes n random walks from s with length l and selects the end state with the minimum heuristic function value (Line 10-12). Note that heuristic functions are not evaluated for the intermediate states in random walks. As a result, the heuristic function is evaluated only n times in `RANDOM ADVANCEMENT(s, l, n)`.

Algorithm 6: RANDOM ADVANCEMENT

input : a state s , the parameter l , the parameter n

```
1  $c \leftarrow 0$  ;
2  $s' \leftarrow s$  ;
3  $h_{min} \leftarrow \infty$  ;
4 for  $c \leftarrow 1$  to  $n$  do
5   | for  $j \leftarrow 1$  to  $l$  do /*inner loop for a random walk*/
6   |   |  $o \leftarrow$  a random action applicable to  $s'$  ;
7   |   |  $s' \leftarrow$  APPLY ( $s', o$ ) ;
8   | end
9   |  $h' \leftarrow h(s')$  ;
10  | if  $h' < h_{min}$  then
11  |   |  $s_{min} \leftarrow s'$  ;
12  |   |  $h_{min} \leftarrow h'$  ;
13  | end
14 end
15 return  $s_{min}$  ;
```

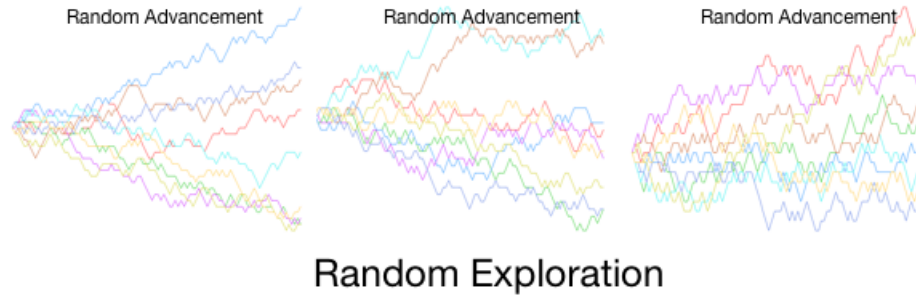


Figure 5.1: Random exploration as a concatenation of random advancements.

Definition 34 Given a state space \mathcal{S} , states $s, s' \in \mathcal{S}$, positive integers l, t, n , a RANDOM EXPLORATION procedure, as described in Algorithm 7, first determines the value of l, n , and t and then uses these values to conduct t consecutive RANDOM ADVANCEMENTS, each using the end state of the previous one as the start state. A random exploration moves the current state from s to the end state of the last random advancement.

A random exploration moves the current state s to a new state s' that is tl steps away, by applying t consecutive random advancements. Figure 5.1 illustrates the

Algorithm 7: RANDOM EXPLORATION

input : a state s
1 $c \leftarrow 0$;
2 $s' \leftarrow s$;
3 determines parameter t , n , and l ;
4 **for** $c \leftarrow 1$ **to** t **do**
5 | $s' \leftarrow \text{RANDOM_ADVANCEMENT}(s', l, n)$;
6 **end**
7 **return** s' ;

Parameters	Mnemonics	Definitions
t	steps	number of advancements in a random exploration
n	number of walks	number of walks in a random advancement
l	length	length of a single random walk

Table 5.1: List of parameters used in RANDOM EXPLORATION

concatenation of random advancements into a random exploration. We use t consecutive random advancements (each with length l) over a single random advancement of length tl such that random walk is biased towards states with smaller heuristic values. This exploitation-exploration tradeoff can be tuned by changing the value of t and l . For now, we assume a procedure is given to determine the values of t , l and n . We list the definitions of these parameters in Table 5.1.

Based on random explorations, we present our random walk assisted best-first search (RW-BFS) algorithm in Algorithm 8. It is a variant of the standard best-first search procedure. In addition to the original best-first search algorithm, RW-BFS adds a DETECT PLATEAU check after expanding a new state (Line 15). If a plateau is detected, a RANDOM EXPLORATION procedure is called to explore the search space in order to find a state that can reduce h^* . Meanwhile, h^* , the incumbent heuristic value, is updated whenever a state with a smaller heuristic value is found (Lines 6-7).

5.3.2 Performance Analysis

In this section, we propose quantitative analysis of the expected cost of a random exploration in finding an exit state on various space topologies. Based on the quantitative measure, we can also tell if random exploration procedures are beneficial in

Algorithm 8: THE RW-BFS FRAMEWORK

input : Initial state s_0

```
1  $open \leftarrow s_0$  ;
2  $h^* \leftarrow h(s_0)$  ;
3 while  $open$  is not empty do
4    $node \leftarrow \text{REMOVE-FIRST}(open)$ ;
5   if  $node$  is GOAL then
6     return  $found$  ;
7   end
8   if  $node$  is not in closed then
9     add  $node$  to closed;
10    insert  $\text{SUCCESSOR}(s)$  to  $open$  ;
11  end
12  if  $h(node) \leq h^*$  then
13     $h^* \leftarrow h(node)$  ;
14  end
15  if DETECT PLATEAU then
16     $open \leftarrow \text{RANDOM EXPLORATION}(s)$ ;
17  end
18 end
19 return no solution
```

certain cases, since using them incurs additional overhead to the best-first search procedure.

Definition 35 (Plateau Graph) Given a planning task \mathcal{T} and a positive integer d , a plateau graph $G_d = (V, E)$ of state s is a simple directed graph (no multi-edges or loops) with $V = \mathcal{N}(s, d)$ satisfying: 1) there is an edge $(s_i, s_j) \in E$ if and only if there is an action that leads s_i to s_j ; 2) for all states s' in V , $h(s') \geq h(s)$, and there are no dead end states in V . An exit state of G_d is a state $s_e \notin G_d$ such that $h(s_e) < h(s)$. Here d is called the radius of the plateau graph G_d .

For a given plateau graph G_d , in the following analysis, we compare the number of heuristic evaluations required to escape from G_d for both best-first search and the random exploration algorithms, as heuristic evaluation takes up most of the time for both algorithms. To simplify our analysis, we assume that the random walk is unbiased, meaning that instead of picking the best state among all n cases shown

in line 11-12 in Algorithm 6, a random state s' is chosen with probability $1/n$ to be s_{min} . We further simplify the structure of G_d as a graph where nodes have the same in- and out-degrees. Without loss of generality, we assume that every node in G_d has p successors and q parents, where $p \geq q \in \mathbb{Z}^+$. We first consider the case where G_d is a tree, in which case $p > q = 1$.

Tree Search

We provide the following result for best-first search when G_d is a tree.

Lemma 6 *Given a plateau graph G_d of s that is a tree, if s_e is an exit state of G_d and $s_e \in \mathcal{N}(s, d+1) \setminus \mathcal{N}(s, d)$, a best-first search procedure, in the worse case where every state in $\mathcal{N}(s, d)$ has to be explored before finding s_e , will evaluate the heuristic function value of $\frac{p^{d+1}-1}{p-1}$ states.*

The proof for Lemma 6 is straightforward. For each i from 1 to d , the number of states in $\mathcal{N}(s, i) \setminus \mathcal{N}(s, i-1)$ is p^i . The total number of heuristic evaluations for best-first search, in the worst case, is $\sum_{i=1}^d p^i$, or $\frac{p^{d+1}-1}{p-1}$.

For an unbiased random walk in $\mathcal{N}(s, tl)$, we have the following result.

Theorem 11 *Given a plateau graph G_d of s that is a tree, let R be an unbiased random walk that explores the search space with length tl , and E be the number of exit states of G_d in $\mathcal{N}(s, tl) \setminus \mathcal{N}(s, tl-1)$, if the heuristic function is evaluated every l steps in R , the expected number of heuristic evaluations before R finds an exit state is $\frac{tp^{tl}}{E}$.*

Proof: Since G_d is a tree, starting from s , R explores states that are exactly tl steps away from s . Since R is unbiased, the probability of hitting an exit state of G_d is therefore $\frac{E}{p^{tl}}$. That is to say, the expected number of random walks to find an exit of G_d is $\frac{p^{tl}}{E}$. Since there are t heuristic evaluations on each path, the total number of expected heuristic evaluations is $\frac{tp^{tl}}{E}$. ■

We see from Lemma 6 and Theorem 11 that an unbiased random walk can assist best-first search finding plateau exit with fewer heuristic evaluations when

$$\frac{p^{d+1} - 1}{p - 1} \geq \frac{tp^{tl}}{E}.$$

One sufficient condition for the above inequality is that $d \leq tl \leq d + \log_p E - \log_p t$. If E is on the magnitude of p^d and t is relatively small, the above condition is approximately:

$$d \leq tl \leq 2d.$$

The above analysis is for unbiased random walks. In practice, random walks can be more helpful since it is not unbiased; it is biased towards good states with lower h . One insight drawn from this is that tl should be neither too small or too large for the random walk exploration to be helpful. However, since the precious d is unknown, it is helpful to try different tl values when doing random exploration in Algorithm 7. For this reason, we use a set of different tl values in the implementation.

We can also see from the above analysis that when p and d are relatively small, it is better to use best-first search to explore every state in G_d . This prompts us to be conservative on plateau detection. We discuss the parameter settings for plateau detection in detail in the next section.

Graph Search

Now we extend the above discussion to the case where G_d is a graph. Namely, we consider the case where $q > 1$.

Theorem 12 *Given a plateau graph G_d of s where every node in G_d has an in-degree of $q \geq 1$, if there exists a state $s_e \in N(s, d + 1)$ such that $h(s_e) < h(s)$, a best-first search procedure, in the worst case, will have to evaluate the heuristic value for $\frac{(p/q)^{d+1} - 1}{(p/q - 1)}$ states before finding s_e .*

Proof: We prove this by using mathematical induction. It is easy to see that this proposition is true when $d = 0, 1$, where we have 1 and $1 + \frac{p}{q}$ states, respectively.

Assuming this proposition is true for all $d < k$, we have $|G_{k-1}| = \frac{(p/q)^k - 1}{(p/q) - 1}$ and $|G_{k-2}| = \frac{(p/q)^{k-1} - 1}{(p/q) - 1}$. Thus, there are $(p/q)^{k-1}$ states that are exactly $k - 1$ steps away from x . Since G is a simple graph, according to our assumption, there are $p(p/q)^{k-1}/q = (p/q)^k$ states that are k steps away from a . Thus, we have

$$|G_k| = \frac{(p/q)^k - 1}{(p/q) - 1} + (p/q)^k = \frac{(p/q)^{k+1} - 1}{(p/q) - 1}.$$

Thus this proposition is also true for $d = k$. ■

On the other hand, it is easy to see that the structural change of G does not affect the expected number of heuristic evaluations for an unbiased random walk R , as it does not maintain any information on whether a state has been visited. In this case, Theorem 11 still holds for $q > 1$.

We can see that R can help best-first search in finding an exit of G_d if

$$\frac{(p/q)^{d+1} - 1}{(p/q) - 1} \geq \frac{tp^{tl}}{E}.$$

We can derive a necessary condition for the above inequality by replacing the left side with $(p/q)^{d+1}$. In this case, any tl must satisfy $d \leq tl \leq d + \log_p E - (d + 1)(1 - \log_p q) - \log_p t$. It is easy to see that as q increases from 1 to p , random walk becomes less effective. The insight is that RW-BFS is more effective when there are not many paths (q is smaller compared to q) that can lead to the same state.

Impacts of the dead ends and loops in G_d . So far, we assumed there are no dead ends nor loops in G_d . For best-first search, because of the *closed* list, having dead ends or loops in G_d does not change the number of heuristic evaluations for best-first search. In other words, Theorem 12 still holds in this case.

However, for an unbiased random walk R , loops and dead ends in G_d would decrease the probability of random walk visiting exit states outside G_d . In other words, E , the number of exit states that random walk can reach, becomes smaller in the presence of dead ends and loops. Formally, dead end states and loops are evenly distributed in G_d , the probability that R can find an exit to G_d is lower than $\frac{E}{p^{tl}}$ where E is the number of exit states in $N(s, tl)$. The above analysis shows that random walk is more

helpful on problems where there are few loops and dead ends than problems where dead ends and loops are common.

5.3.3 Parameter Settings

In the previous section, we have shown that under certain conditions the expected number of heuristic evaluations in a random exploration can be smaller than that of a deterministic search in finding plateau exits, and therefore it makes sense to switch to random explorations when deterministic search is not making progress. Now we investigate parameter settings related to random exploration.

Plateau detection. In Algorithm 8, a PLATEAU DETECTION procedure is invoked during search to decide whether random exploration should be invoked. Our analysis in the previous section shows that the plateau detection test cannot be too sensitive nor too unresponsive. If it is too sensitive, random exploration will be invoked frequently and the progress of the best-first search may be hindered by constant interruption. This is especially important in the sequential implementations of Algorithm 8. On the other hand, if this detection is unresponsive, our designed random walks cannot help the best-first search as desired. Therefore, a balanced plateau detection mechanism is needed.

Here we can take into consideration both the topological structure of local search, as well as the progress made on heuristic search. Particularly, we maintain a history of incumbent heuristic values \hat{h} and the size of the *closed* list when \hat{h} is decreased. We run a linear regression and calculate r , the average number of states explored when \hat{h} is reduced by one. A random advancement process is triggered when the value of h^* is not reduced for r consecutive states.

To bypass potential blocks during deterministic search, we also maintain a moving average \bar{h} of heuristic values for m most recent states (excluding dead-end states). Search switches to random advancement if $\bar{h} \geq \hat{h}(1 + \delta)$. In our implementations m is set to $r/2$ and δ is set to 0.2.

Length of a walk. The choice of l affects the efficiency of the random advancement. The computational cost of generating a walk grows linearly with l , so it is desired that

l is short. We also want l to be long enough to walk over blocks in \mathcal{N} to reach an exit state. However, knowing how far away the block and exit states are is as difficult as finding a path to the exit state. As we discussed in the last section, we can estimate tl using d ($d \leq tl \leq 2d$), the radius of the plateau graph. We have found through comprehensive experimentation that d is typically from 1 to 30 for the h_{ff} heuristic we use in random exploration. Given the large range of d , in our implementation, we use four different l values in random advancement: 1, 4, 7 and 10.

Number of walks. Parameter n controls number of random walks conducted in a random advancement and subsequently the number of heuristic evaluations in each advancement. We select n by estimating the state coverage of random walks in $\mathcal{N}(s, l)$. During the walk, the branch factor at each state is recorded by counting the number of successor states. The average branch factor b is then calculated by taking a moving average of the recorded branch factors. We pick n to be the $\min(2000, b^l)$. This way, when small l s such as 1 or 4 are used, n is accordingly smaller such that repetitively random walks in small local regions such as $\mathcal{N}(s, 1)$ and $\mathcal{N}(s, 4)$ are avoided. We also cap the number of walks at 2000 to guarantee that random exploration returns to deterministic search in a finite time.

5.4 Experimental Results

We report results in two parts. In the first part, we report our experimental results on two variants of RW-BFS, namely, the sequential version (RW-BFS_s) and the parallel version (RW-BFS_p). In the second part, we report results from the Seventh International Planning Competition held in 2011.

5.4.1 Part I: Results on IPC 6 (2008) Domains

The baseline in our comparison is LAMA [63], a best-first search planner that is used as the base planner for deterministic search in RW-BFS_s and RW-BFS_p. In our experiment, all three planners use the same settings and same heuristic functions for the best-first search part.

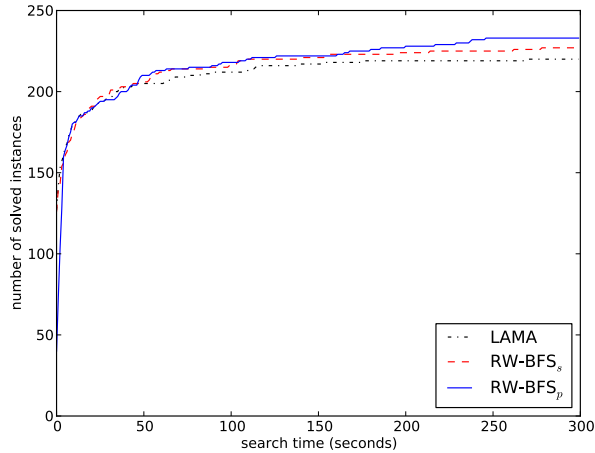


Figure 5.2: Number of problem instances that are solvable by LAMA, RW-BFS_s, and RW-BFS_p in 300 seconds.

Parameters for random explorations are set according to our discussion in the previous section. The number of advancements is set to 4. Both RW-BFS_s and RW-BFS_p use the same set of parameters.

In RW-BFS_p, line 16 in Algorithm 8 are running in parallel using 4 separate threads. RW-BFS_s runs the random exploration procedure inside Algorithm 7 sequentially. For both algorithms, the *open* list is shared by random exploration and best-first search so that possible exit states discovered by random exploration can be inserted into the *open* list directly.

We test all domains in IPC 6 [4], including Elevators (elevator), Woodworking (wood), PARC Printer (parc), Sokoban (sokoban), Openstack (open6a), Peg solitaire (peg), Scanalyzer-3D (scan) and Transport (trans). All experiments are conducted on a quad core workstation with a 2.4GHz CPU and 2GB memory. The running time limit for each instance is set to 300 seconds.

Figure 5.2 shows the number of problem instances solved by the three planners. Both RW-BFS_s and RW-BFS_p solve more problem instances than LAMA, the baseline best-first search planner. In total, they solve 233, 210, and 204 instances, respectively. We would like to point out that problems that LAMA cannot solve within 300s are

P	LAMA			RW-BFS _s						RW-BFS _p			
				Total			RW			Total			RW
	T	L	E	T	L	E	T'	L'	E'	T	L	E	L'
elevator-18	-	-	-	9.9	151	17276	3.8	14	7319	1.94	138	11588	10
elevator-19	5.8	183	8796	6.1	177	9820	1.5	9	2905	4.82	186	7039	4
elevator-20	6.6	162	9739	10.3	163	15799	2.9	13	5027	6.68	147	9422	31
elevator-23	14.3	161	17165	13.4	139	16744	5.8	24	7713	2.47	140	402	10
elevator-24	68.1	188	68344	31.6	231	33167	12.0	17	13385	93.58	191	90455	27
elevator-25	-	-	-	44.8	268	41555	11.4	18	11596	57.63	249	51071	13
elevator-26	-	-	-	59.1	261	51712	21.5	36	20086	30.81	265	17723	25
elevator-27	44.5	310	31249	42.4	353	31434	12.4	24	10220	18.49	187	1280	15
elevator-28	-	-	-	25.0	276	16895	6.2	9	4407	14.23	202	12068	10
elevator-29	21.4	311	12797	11.3	308	7007	0.9	1	666	13.82	213	1641	24
elevator-30	-	-	-	104.3	306	63671	32.3	27	21124	53.75	227	12651	21
wood-3	2.8	35	4728	0.2	35	957	0.1	4	603	1.4	35	106	1
wood-4	11.7	74	16731	18.5	94	64827	17.4	22	63017	14.48	102	60928	38
wood-7	36.2	91	32049	24.1	95	39638	22.3	14	37686	7.98	103	13132	20
wood-8	33.6	96	21623	9.6	94	16082	8.8	10	15443	12.42	181	10423	10
wood-10	44.5	119	23544	87.7	168	85329	83.3	58	82607	28.55	164	21460	15
wood-14	9.3	46	15982	0.5	47	1808	0.3	9	1404	6.6	47	1418	81
wood-16	19.6	94	15592	23.6	179	51161	22.4	77	49425	1.17	178	5511	48
wood-17	34.6	79	37879	7.1	87	16655	6.3	14	15741	22.59	111	29709	20
wood-18	25.6	80	15128	31.7	102	52833	29.7	22	51333	10.21	98	31192	13
wood-19	28.1	86	17742	23.4	106	34498	21.8	22	33182	33.64	95	48797	10
wood-20	-	-	-	50.3	219	50335	47.7	102	48587	22.42	242	21465	110
wood-23	2.3	25	6013	1.9	28	15076	1.8	7	14740	41.11	25	95400	1
wood-26	22.3	93	19221	5.6	99	16312	4.9	13	15269	12.12	97	12305	15
wood-28	-	-	-	28.6	121	32746	27.9	24	32224	11.16	137	14005	91
wood-29	63.7	103	46700	54.9	124	67946	52.0	25	65589	39.60	144	32093	25
parc-17	154.6	81	5205	53.5	75	1765	6.4	1	154	126.14	93	1842	12
parc-24	114.8	29	6657	156.9	28	8546	12.9	1	301	76.40	29	4533	15
sokoban-14	21.5	405	100431	262.1	378	637299	152.1	13	124196	12.88	890	74000	120
sokoban-15	-	-	-	66.3	276	262600	11.9	2	9561	7.33	277	3784	10
sokoban-16	7.7	287	37963	99.7	443	324803	46.0	18	39134	63.27	222	149122	3
sokoban-17	39.7	212	243203	3.8	148	21339	1.0	1	1499	31.34	232	10346	14
sokoban-18	176.6	297	1007158	-	-	-	-	-	-	-	-	-	-
sokoban-21	30.8	288	132957	103.9	411	339820	37.9	10	39425	26.2	410	95404	12
sokoban-22	115.8	336	667481	-	-	-	-	-	-	112.78	713	172216	0
sokoban-23	4.2	249	30485	9.5	230	63234	1.7	2	4040	6.14	258	65120	1
sokoban-24	108.4	279	367882	57.6	312	173380	14.3	4	8915	161.41	313	466868	3
sokoban-25	7.5	177	35097	31.1	391	122051	5.7	5	5159	17.36	245	105773	12
sokoban-26	-	-	-	109.4	417	450086	47.5	10	84417	53.85	583	128279	22
sokoban-27	112.5	81	482420	5.5	158	21916	1.4	4	2441	14.27	157	10719	1
sokoban-28	29.8	450	82203	54.1	436	139494	8.6	0	17230	5.24	465	6197	11
peg-29	8.7	45	37097	20.7	45	161290	11.8	0	123121	9.97	45	13611	0
peg-30	90.9	54	395509	104.3	55	499095	31.2	0	188839	97.71	55	707	0
scan-29	269.6	81	3715	278.8	81	3816	10.0	0	98	246.62	81	3403	0
trans-19	83.1	332	26068	214.6	331	67386	127.8	0	40901	89.35	331	26337	0

Table 5.2: Comparison of the search time (“T”), solution length (“L”), number of heuristic evaluations (“E”) of LAMA, RW-BFS_s and RW-BFS_p. For RW-BFS_p, “E” is the total number of heuristic evaluations of all threads.

problems where there is a large plateau for it to explore. Both RW-BFS_s and RW-BFS_p solve more problems due to the fact that some plateau exploration are avoided during search.

In Table 5.2, we give detailed comparisons of three planners on all IPC-6 problems for which *random exploration* is invoked. Problems in which *random exploration* is never invoked are omitted from our comparison because in this case three algorithms are essentially identical.

To show the contribution and overhead brought by random walk in RW-BFS_s, we also report the time spent in random walk (T'), the length of the sub-path found by random exploration in the final solution path (L'), and the number of heuristic evaluations in random exploration procedure. For RW-BFS_p, we report the length of the sub-path (L') in the final solution path. We omit the time spent in random walk and the number of heuristic evaluations in random exploration procedure for RW-BFS_p as these two metrics are proportional to the runtime of the best-first search in RW-BFS_p, and do not reflect the efficiency of random walks.

We summarize three findings from Table 5.2.

First, these results show that for problems LAMA cannot solve within 300s, e.g., elevator-18, elevator-25, elevator-26, RW-BFS_s and RW-BFS_p can successfully find solutions in which substantial portions of the paths are generated by random walks. Problems in Elevators and Woodworking domains usually have plateaus. Thus, best-first search is frequently stuck on plateaus, which results in costing more search time or even failing to find a solution. Experimental results on these domains clearly show that random walk can assist best-first search to escape from plateaus.

Second, comparing the performance of two sequential planners LAMA and RW-BFS_s to RW-BFS_p, we see that the overhead brought in by alternating between random walk and best-first search can be mitigated by using a parallel implementation, at the cost of using more computing cores. For domains such as Elevator, RW-BFS_p can reduce the solving times by half or more.

Third, according to our performance analysis, if the state space has $q > 1$, the random walk procedure may not be helpful. A closer look at problems in the Pegsol domain

Domain	LAMA	RW-BFS _s	RW-BFS _p
elevator	25	30	29
wood	28	30	30
parc	23	23	29
sokoban	24	24	25
open6a	30	30	30
peg	30	30	30
scan	30	30	30
trans	30	30	30
Total	220	227	233

Table 5.3: Number of solved instances for three planners on IPC 6 domains (Time limit is 300s).

reveals that they indeed have $q > 1$. In the peg solitaire game, there can be multiple moves at each state and there can be multiple action paths arriving at the same state. Results on peg-29 and peg-30 confirmed our analysis that a random walk exploration would not assist the best-first search much when q is close to p . On the other hand, problems in the Sokoban domain are well-known to have many loops and dead ends. In this domain, RW-BFS_s did not outperform LAMA, as random exploration is less effective in finding exit states.

5.4.2 Part II: Results on IPC 7 (2011) Domains

Overview of the Seventh International Planning Competition (IPC 7)

The International Planning Competition is an event organized in the context of the International Conference on Planning and Scheduling (ICAPS). The competition has a set of goals, including, providing an empirical comparison of the state of the art of planning systems, highlighting challenges to the AI Planning community, proposing new directions for research and new links with other fields of AI, and providing new data sets to be used by the research community as benchmarks [5].

The competition is organized into different tracks, including the deterministic track, the learning track and the uncertainty track. We participated in the deterministic

track. This track covers classical planning problems with actions having associated non-negative costs (not necessarily uniform). Apart from solving the problems within reasonable time, the goal of the track is to find low-cost plans, where the cost is defined as the sum of the costs of each plan’s actions.

There is no domain specific knowledge in all participant planners. In fact, benchmark problems were revealed after planner submission was completed. There are 14 domains in the benchmark. Each domain contains 20 problems. Each planner gets a score from 0.00 to 1.00 for each solved task in every domain based on solution quality. All planners are run on the same machine with a 6 GB RAM, 750 GB HD and a 30 minute time limit.

Roamer and Roamer-p Planners in IPC 7

As a team, we participated in the Seventh International Planning Competition (IPC 7) in 2011 with two planners, Roamer and Roamer-p. Both planners are based on Algorithm 8. A detailed comparison of these two planners is presented in Table 5.4.

	Roamer	Roamer-p
algorithm	RW-BFS _s	RW-BFS _p
track	sequential satisficing	multi-core satisficing
rank	5/28	4(2)/8
contributor(s)	Lu and Xu	Xu

Table 5.4: A comparison of Roamer and Roamer-p for IPC 7.

Roamer is a random walk assisted best-first search planner. At the core, Roamer uses the sequential version of Algorithm 8 to accelerate best-first search. It is developed on top of the LAMA planner [63]. Roamer participated in the sequential satisficing sub-track under the deterministic track of IPC 7.

Plateau detection in Roamer is adaptive. Roamer keeps track of the number of plateaus found during search as n_p (reset to 0 when \hat{h} is changed). If the value of \hat{h} is not decreased after $m = 3000 + (n_p - 1) * 1000$ states, n_p is increased by one, and random explorations are triggered. Roamer would pause best-first search, and

invokes four rounds of random explorations, each with different parameters, to find an exit to the plateau. States found by random exploration that have heuristics smaller than \hat{h} get inserted back to the *open* list for best-first search.

The Roamer planner is a joint work of Lu and I. I implemented the random exploration logic for Roamer. Lu incorporated it with the base planner and tuned the parameters.

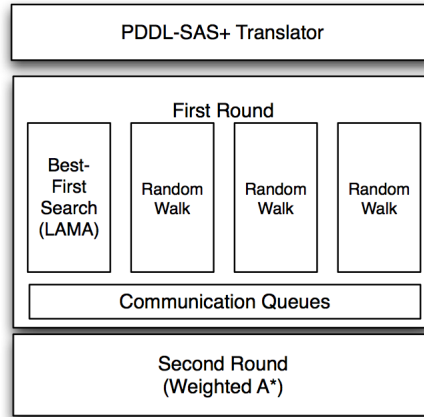


Figure 5.3: Structure of Roamer-p.

The Roamer-p Planner

Roamer-p is a multi-core planner that runs best-first search and random walks in parallel. It is independently developed by me, with helpful suggestions from Lu. Roamer-p participated in the multi-core satisficing sub-track under the deterministic track of IPC 7.

Like Roamer, Roamer-p is also based on the LAMA planner. However, its architecture is significantly different than a traditional sequential planner such as LAMA or Roamer.

Figure 5.3 presents the structure of the Roamer-p planner. It first uses the PDDL-SAS+ translator from LAMA to convert PDDL-encoded planning problems to SAS+ formalisms. Then, it adopts a multi-round search strategy to find solution paths. The first-round search runs one thread of best-first search adapted from the LAMA planner [63] and three threads of random walks. By the rules of IPC 7, all planners in

the multi-core track run on quad-core Linux boxes. Thus, we let Roamer-p use four threads in the first-round search, three of them being random walk threads. Roamer-p can be used with any number of random walk threads.

Random walk and best-first search threads run in parallel in Roamer-p. Roamer-p uses two queues for the communication between random walks and the best-first search. A shared queue Q_{rw} is used for the best-first search thread to send states to three random walk threads. Whenever a plateau is detected, a state on the plateau is pushed to Q_{rw} for random walks to explore.

For a random walk starting from s , when a new state is found with heuristic value less than $h(s)$, all states along the path from s to the new state are inserted into another shared communication queue Q_{bfs} . Best-first search fetches paths from Q_{bfs} , evaluates heuristic values of states in those paths (as random walk would skip heuristic evaluations for intermediate states along the path), and inserts all these states into the *open* list. It is easy to prove that this communication mechanism preserves the completeness of search.

For Roamer-p, the synergy between threads is vital to its performance. During the search, there are cases where random walks are making slow progress while the best-first search is advancing quickly (or vice versa). In these cases, we force threads to check Q_{rw} or Q_{bfs} periodically. When the local best heuristic value is far behind the global one, we force the thread to restart from a better state.

Roamer-p terminates when a goal state is found. States in the goal path may come from different threads. States found by random walk are not necessarily in the *closed* list, so Roamer-p re-assembles path segments together to report the complete goal path.

IPC 7 Results for Roamer

Table 5.5 shows the final score of all planners in the sequential satisficing track. We highlight domains where Roamer performs significantly better than the base planner (lama-2008). These domains are woodworking (wood), elevators (elev) and barman. These three domains all have large plateaus with few dead ends. The IPC 7 results

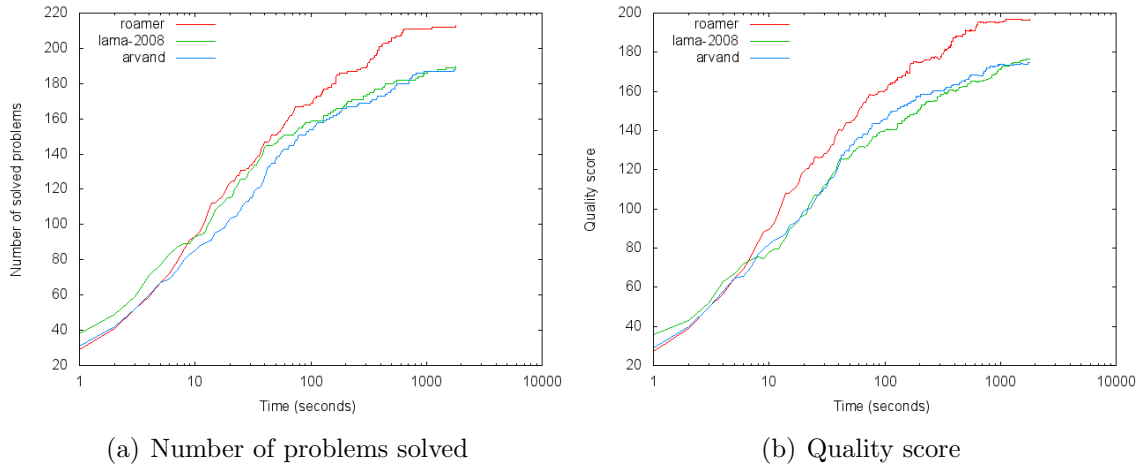


Figure 5.4: Number of problems solved and quality score over time for Roamer, Lama 2008 and Arvand. The x-axis is shown on a logarithmic scale.

have confirmed our analysis that RW-BFS works best on domains with few or no dead ends. Roamer also outperforms Arvand, a stochastic search planner that inspired us to conduct random walks during best-first search. Arvand relies solely on random walks to solve the planning problems. While Arvand performs well on a number of domains, it falls short on domains such as sokoban and parking (park), two domains with many loops in the search space. We highlight the domains in which Roamer outperforms Lama 2008 or Arvand in Table 5.5.

As we can see, by combining the best of two worlds, Roamer outperforms both Lama-2008 and Arvand. In Figure 5.4, we show the number of problems solved and the quality score over time by Roamer, Lama 2008 and Arvand. It is clear that Roamer not only solves more problems than the other two over time, but consistently outperforms Lama 2008 and Arvand in terms of solution quality.

Planner	peg	scan	parc	wood	sokoban	open	tidy	trans	elev	nomys	park	visital	barman	floor	Total
lama-2011	16.26	15.77	17.03	15.44	17.04	17.17	12.07	16.92	15.60	8.65	17.52	15.32	17.94	4.78	207.51
probe	15.00	16.35	8.73	18.48	14.21	9.25	16.82	13.56	13.77	4.35	13.21	17.71	19.40	4.02	184.87
fdss-2	13.77	15.32	15.24	18.40	16.68	13.21	13.40	8.57	17.57	9.26	15.75	2.20	14.22	6.51	180.10
fd-autotune-1	14.21	14.04	16.86	15.30	17.49	13.77	10.99	8.89	15.78	7.18	13.80	0.86	16.87	6.03	172.04
roamer	14.64	15.53	3.87	12.56	14.75	14.54	12.38	16.95	14.73	8.55	15.32	8.75	15.36	1.98	169.91
fdss-1	12.10	13.10	14.64	13.19	17.06	12.84	12.82	9.00	13.11	8.70	15.17	2.25	14.68	4.23	162.89
forkuniform	16.40	10.25	13.92	16.26	16.99	14.05	11.46	6.34	16.98	8.41	15.55	2.96	3.89	4.24	157.70
lama-2008	17.15	14.08	0.47	8.73	13.99	15.28	10.86	17.79	4.57	10.15	15.07	17.67	3.65	2.20	151.64
fd-autotune-2	16.58	10.74	10.48	9.26	14.32	16.71	13.13	6.51	15.32	16.59	6.83	1.98	4.33	8.78	151.55
lamar	15.91	13.86	1.78	13.38	12.63	15.24	14.46	9.87	8.81	10.17	18.03	9.00	5.02	1.91	150.06
randward	18.54	12.69	0.64	13.60	14.57	16.03	13.92	4.18	5.27	6.76	15.03	13.13	2.02	1.53	137.92
arvand	18.73	16.08	13.40	12.37	1.77	11.56	14.64	10.15	12.51	16.02	2.70	5.53	0.00	2.04	137.52
brt	11.74	14.04	3.38	1.50	6.61	2.83	15.10	11.30	16.38	4.15	6.25	3.65	15.67	2.95	115.56
yahsp2-mt	10.75	8.92	8.77	12.21	0.00	0.00	0.00	14.14	0.00	7.89	2.31	18.73	8.75	6.23	98.70
yahsp2	8.70	10.15	14.56	9.35	0.00	0.00	0.00	14.20	0.00	5.33	6.14	17.28	5.57	4.40	95.66
cbp2	12.06	1.63	3.80	1.69	10.39	12.59	12.71	12.38	9.21	3.62	0.00	9.47	0.00	0.00	89.57
dae-yahsp	10.17	9.49	12.15	4.51	0.00	0.00	0.00	12.38	0.00	7.55	0.00	14.45	5.88	5.28	81.85
cbp	11.46	1.63	3.80	0.46	9.93	12.18	12.67	9.36	6.02	3.62	0.00	9.47	0.00	0.00	80.61
lprpgp	10.18	10.23	5.79	0.00	8.32	10.65	0.00	0.00	11.07	4.82	5.77	1.67	1.55	1.12	71.17
madagascar-p	12.48	12.04	17.37	1.00	1.64	0.00	6.99	0.94	0.00	12.15	0.00	0.00	0.00	0.00	64.60
popf2	4.67	11.22	10.32	2.12	4.63	0.00	0.00	0.00	5.64	5.76	3.01	1.67	0.00	0.67	49.70
madagascar	8.84	6.85	17.44	1.00	0.00	0.00	0.71	0.00	0.00	12.86	0.00	0.00	0.00	0.00	47.71
cpt4	0.00	2.03	11.17	15.48	0.00	0.00	0.00	0.00	0.00	13.81	0.00	0.00	0.00	0.00	42.49
satplanlm-c	2.38	4.26	11.98	0.00	0.00	0.00	0.00	0.00	0.00	3.00	0.00	0.00	0.00	0.00	21.62
sharaabi	8.39	0.00	4.22	0.00	0.00	3.64	0.00	0.00	0.43	0.00	0.00	0.00	0.00	0.00	16.68
acoplan	11.45	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.48	0.00	0.00	11.93
acoplan2	11.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.48	0.00	0.00	11.58
total	323.65	260.27	241.81	216.28	213.01	211.55	205.15	203.42	202.76	199.34	187.46	174.72	154.81	68.91	

Table 5.5: Results of IPC 7 for all planners in the sequential satisficing track. Domain names are shortened to fit the table in a page.

Planner	parc	scan	peg	nomys	wood	park	trans	tidy	elev	open	barman	sokoban	visita	all	floor	Total
arvandherd	15.52	16.22	18.67	16.82	15.33	15.04	15.05	15.64	18.35	19.69	17.00	14.83	8.81	2.00	208.95	
ayalsoplan	4.58	10.48	9.07	12.08	11.00	11.35	17.66	0.00	7.72	17.17	15.70	14.37	0.00	1.40	132.59	
phsff	12.35	17.52	7.82	3.83	19.00	18.78	7.13	16.40	18.97	0.00	0.00	7.04	0.80	1.60	131.25	
roamer-p	7.00	8.72	14.73	7.88	8.50	6.23	4.53	14.78	8.75	16.38	6.53	9.05	8.28	2.20	123.55	
yahsp2-mt	7.58	9.73	5.42	9.15	12.83	15.03	16.24	0.00	0.00	0.00	11.50	0.00	20.00	7.00	114.49	
madagascar-p	17.83	10.43	6.90	11.88	1.00	0.00	1.33	7.97	0.00	0.00	0.00	1.58	0.00	0.00	58.94	
madagascar	17.50	4.82	3.00	11.73	1.00	0.00	0.00	0.75	0.00	0.00	0.00	0.00	0.00	0.00	38.80	
acoplan	0.00	0.00	9.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	9.40	
total	82.37	77.92	75.00	73.38	68.67	66.42	61.94	55.54	53.79	53.24	50.74	46.88	37.89	14.20		

Table 5.6: Results of IPC 7 for all planners in the multi-core satisficing track after applying the strict plan validating rule. Domain names are shortened to fit the table in a page.

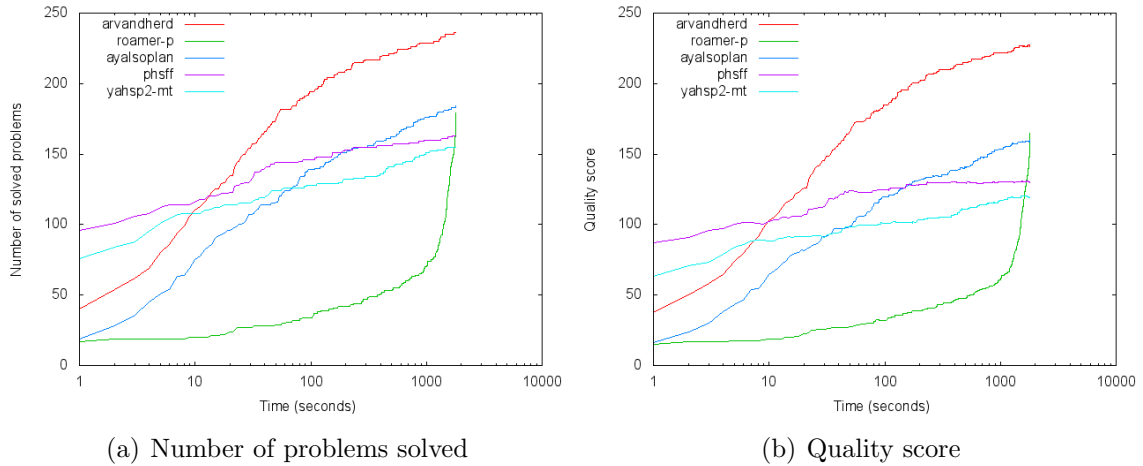


Figure 5.5: Number of problems solved and quality score over time for multi-core planners. The x-axis is shown on a logarithmic scale.

IPC 7 Results for Roamer-p

Roamer-p solved 184 problems and achieved a quality score of 165 in IPC 7, ranking the second among all 8 participating planners¹. We present the final score and ranking of all 8 participants in the multicore satisficing track of IPC 7 in Table 5.6. Among all 8 participants, Roamer-p ranked the 4th place, after Arvandherd, Ayalsoplan and Phsff.

One interesting thing to notice is that Roamer-p has different performance characteristics than other planners. Figure 5.5 shows the number of problems solved and quality score over time for the top 5 participants in the multi-core track. Notice that x-axis is on a logarithmic scale. It is expected that the curve for the number of problems solved versus time is linear for typical heuristic search planners, as computational time grows exponentially with problem size. What is unusual in Figure 5.5(a) is that Roamer-p solves more problems in the [1000, 1800) interval, whereas other planners make slow progress during the same period.

¹Roamer-p did not receive points in 39 different problems in the following domains (the number of problems affected shown between parenthesis): nomystery (1), barman (7), parking (6), scanalyzer (1), sokoban (7), tidybot (3), transport(1), visitall (4) and woodworking (9), even though it successfully solved these problems. A strict IPC 7 rule was enforced that gives zero points to planners that write intermediate invalid plans despite the final plan being valid. If Roamer-p’s intermediate invalid plans had not been counted against it, Roamer-p would have ranked the second [51].

This peculiar behavior is rooted in the inert behavior of random exploration. Recall that random exploration is most effective in accelerating best-first search when search is stuck on plateau exploration. In practice, plateau exploration typically happens in later stages of search, where there are many blocks and traps in the *open* list. Because of that, Roamer-p scales well with problem size.

One would expect Roamer-p to perform even better if it is given more time, since the cutoff time for IPC is at 1800 seconds. We have conducted a different experiment and validated this hypothesis on a multi-core machine with 2.4GHz CPU and 4GB of memory. Given a time limit of 3600s, Roamer-p can solve 246 problems out of 280 problems, whereas the winner of the multi-core tracker, ArvandHerd, can solve 247 problems within an hour. This result, together with the results for IPC 7, show that Roamer-p is a competitive, state-of-the-art planner.

To sum up, we have presented the experimental results on both IPC 6 and IPC 7 domains. The experimental results show that random walk is an efficient scheme in accelerating heuristic search, especially in places where heuristic search are making slow progress. The overhead brought in by random walk is well paid off by the ability of jump out of traps and jump over blocks that occurred in heuristic search. The results from IPC 7 also validated that the RW-BFS algorithm framework we proposed in this chapter can lead to competitive, state-of-the-art planning systems such as Roamer and Roamer-p.

5.5 Summary

Inspired by the Monte-Carlo Random Walk planner and the observation that heuristic search spends most of its time doing plateau exploration, we have developed an algorithm framework that can accelerate heuristic search using random walk.

By analyzing the structure of the local region in the search space, we have identified scenarios where random walk can accelerate heuristic search efficiently, and proposed a random walk assisted best-first search algorithm framework. We have also implemented two planners, Roamer and Roamer-p, and participated in the Seventh International Planning Competition in 2011. Both of our planners performed

well in the competition. The results we gathered from a wide spectrum of testing domains have proven that the proposed framework can accelerate heuristic search for AI planning.

Chapter 6

Accelerating Heuristic Search with Cloud Computing

We have looked at ways to accelerate heuristic search by analyzing the problem structure in Chapter 3 and 4, and ways to escape from traps and blocks in the search space by using random walk in Chapter 5. In this chapter, we focus on improving the efficiency of search by developing new search algorithms and search strategies that can utilize advanced, more powerful computing platforms, such as cloud computing.

Cloud computing is emerging as a prominent computing model. It provides an inexpensive, highly accessible alternative to other traditional high-performance computers. It allows small teams and even individual users to routinely have access to the same large-scale computing facilities used by large companies and organizations, such as Amazon, Google, and Microsoft.

Cloud computing architectures have the potential to make heuristic search much more efficient. However, there are also some key challenges that need to be addressed. For instance, cloud platforms typically provide communication mechanisms that are designed for distributed web applications. These mechanisms have high latency compared with the high-speed communication in computing infrastructures like supercomputers and large-scale clusters.

To address these challenges, we propose a portfolio search algorithm. In particular, we run Monte-Carlo Random Walk (MCRW), a stochastic search algorithm for classical planning [56]. Our key observation is that some stochastic algorithms, such as MCRW, exhibit high variability of their running time. This means when running with different

random seeds, MCRW may have very different running times. Such a variability is attractive for cloud computing because even a simple scheme that launches parallel independent runs can have high (sometimes superlinear) speedup without requiring much inter-processor communication.

In addition to the simple algorithm, we also develop an enhanced portfolio search algorithm with multiple parameter settings. In MCRW, the parameters can greatly impact the search performance. By running a portfolio of searches with varying parameter settings, our scheme greatly enhances the possibility of getting the parameter value to perform well. Our experimental results show this scheme significantly improves efficiency and solution quality.

We further implement our algorithms in Windows Azure, a representative commercial cloud whose potential for scientific research remains largely unexplored. We study the performance characteristics of Windows Azure, and then develop a scalable Azure-based scheme for stochastic search algorithms. We report experimental results to show the advantages of the proposed scheme, including high speedup, scalability, and reduced running time variance. We also show that our scheme is economically sensible.

6.1 Background

6.1.1 Parallel Computing

Parallel computing is a form of computation in which the computations are carried out simultaneously, usually on many computational nodes. A *computational node* is usually a processor (e.g. a CPU) associated with memory and other components. Nodes can carry out computations independently, and can communicate with other nodes. Here are the relevant characteristics of parallel computing. First, for an execution with n computational nodes, the *speedup* S_p is defined as $S_p = T_s/T_p$, where T_s is the sequential runtime and T_p is the execution time of the parallel program. Second, the *efficiency* is defined as $E_p = S_p/n$, which measures how well the computing resources are utilized. Third, the **overhead** is $O_p = W_p/W$ defined as the ratio of

the work performed by parallel formulation (W_p) to that by sequential formulation (W).

Search algorithms that are suitable for parallel and multi-core machines have been extensively studied [48, 22, 64, 7, 77]. However, expensive computing infrastructures, such as supercomputers and large-scale clusters, are traditionally available to only a limited number of projects and researchers. As a result, most AI applications, with access to only commodity computers and clusters, cannot benefit from the efficiency improvements of high-performance parallel search algorithms.

6.1.2 Cloud Computing

A cloud is *a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers* that are dynamically provisioned and presented as one or more unified computing resources [15, 73]. Cloud computing platforms have been used for scientific applications. For instance, AzureBlast [53] studied the applicability of the Windows Azure cloud to the BLAST algorithm using a trivial parallelization with little communication due to the high communication latency. There is a work [55] that compared the utility of a supercomputer to that of public clouds, and analyzed the service times prior to actual execution. The conclusion is that while the supercomputer might be much faster, the turnaround time might actually be much better for the cloud because of the elapsed time from submission to the completion of execution, which gives another reason for our research.

6.1.3 Parallel Search Algorithms

Early work on parallel search is surveyed in [31]. For shared memory systems, synchronized schemes such as *layer synchronization* and *delayed duplicate detection*, do not scale to very large amounts of processors; results with up to eight cores have been reported in [80]. For distributed memory architectures, inter-processor communication is generally needed to ensure efficiency and correctness [64]. Recently, the HDA* algorithm [45], based on asynchronous MPI communication, scales well to up to 128

processing cores. These parallel algorithms are not suitable for the cloud environment which has high latency in inter-process communication and instance failures.

6.1.4 Stochastic Search

As another major class of search algorithms, stochastic search has also been studied for its parallelization. A recent work studied the parallelization of WalkSAT [54], which shows promising results for both a simple scheme that launches multiple independent runs and a scheme with asynchronous sharing of learned clauses. As an important stochastic search method, Monte-Carlo Random Walk, has been studied and applied to several areas of automated planning, such as sampling possible trajectories in automated planning [23, 56], probabilistic planning [12] and robot motion planning [50].

6.1.5 Portfolio Search

A portfolio of algorithms is a collection of different algorithms and/or different copies of the same algorithm running in parallel on different processors or interleaved on one processor [29]. The portfolio idea has been applied to automated planning [66], SAT solver [32] and SMT solver [75]. Theoretical and experimental analyses show that portfolio search can significantly decrease variances of heavy-tailed distributions associated with SAT and constraint satisfaction solvers [30].

6.2 Portfolio Stochastic Search Framework

In this section, we present a portfolio stochastic search (PoSS) algorithm designed to take advantage of the cloud platform while steering away from high-latency communication and node failure problems of cloud platforms.

Algorithm 9: MCRW(Π)

Input: a classical planning problem Π **Output:** a solution plan

```
1  $s \leftarrow s_I$  ;
2  $h_{min} \leftarrow h(s_I)$  ;
3  $counter \leftarrow 0$  ;
4 while  $s$  does not satisfy  $s_G$  do
5   if  $counter > c^m$  or  $dead-end(s)$  then
6      $s \leftarrow s_I$  ;
7      $h_{min} \leftarrow h(s_I)$  ;
8      $counter \leftarrow 0$  ;
9   end
10   $s \leftarrow RandomWalk(s, \Pi)$  ;
11  if  $h(s) < h_{min}$  then
12     $h_{min} \leftarrow h(s)$  ;
13     $counter \leftarrow 0$  ;
14  end
15  else
16     $counter \leftarrow counter + 1$  ;
17  end
18 end
19 return  $plan$  ;
```

6.2.1 Monte-Carlo Random Walk (MCRW)

Our algorithm framework is based on the Monte-Carlo Random Walk (MCRW) method. MCRW is a sequential stochastic search method for planning [56]. It uses a random exploration of the local neighborhood of a search state for selecting a promising action sequence.

MCRW achieves comparable, and sometimes superior, performance to the best deterministic search algorithms in a number of testing domains [2]. It benefits from its exploration strategies. The method is robust in the presence of misleading heuristic estimates, since it obtains information from the local neighborhood. Also, the random exploration can effectively escape from local minima.

Algorithm 9 shows the framework of MCRW in detail. Given a SAS+ planning problem Π , MCRW builds a chain of states $s_I \rightarrow s_1 \rightarrow \dots \rightarrow s_n$ such that s_I is the

	Random Exploration	MCRW
objective	plateau exploration	finding goal state
usage	when search is stuck	replaces heuristic search
restarting policy	no restarting policy	with restarting policy
termination	terminate in finite time	no guarantee on termination

Table 6.1: A comparison between Random Exploration and MCRW.

initial state, s_n is a goal state, and each transition $s_i \rightarrow s_{i+1}$ uses an action sequence found by random walk in the neighborhood of the last state (Line 10). MCRW search fails to find a solution when the minimum obtained h -value does not improve after n trials, or a dead-end state is encountered (Line 6). In this case the MCRW search simply restarts from s_I (Line 7), and resets the counter to 0. The algorithm returns a solution plan which contains a sequence of actions changing state from s_I to a goal state s_G that includes all intermediate paths found by random walks (Line 15).

Note that MCRW and the RANDOM EXPLORATION procedure we used to assist best-first search in the last chapter both use the same inner loop for conducting random walks in the state space. The difference is that MCRW is a full-featured stochastic search method that can find a complete plan from the initial state to a goal state, whereas RANDOM EXPLORATION is used as a subroutine to find plan segments during heuristic search. We list the differences between these two in Table 6.1. RANDOM EXPLORATION has no builtin restarting policy, whereas MCRW restarts the random walk from the initial state when a dead-end state is encountered. The running time of RANDOM EXPLORATION is intentionally capped to guarantee a predictable performance. On the other hand, MCRW may not terminate if it cannot find a better state (in terms of heuristic value) to move to.

6.2.2 Variability in MCRW Searches

MCRW search often exhibits a remarkable variability in the solving time of any particular problem instance, which can be exploited by parallel search algorithms to accomplish short runs. Figure 6.1 shows the run-time distribution of MCRW algorithm on six planning problems from the Fourth International Planning Competition (IPC 4) [2]. Table 6.2 gives the mean, the variance and the standard deviation of the

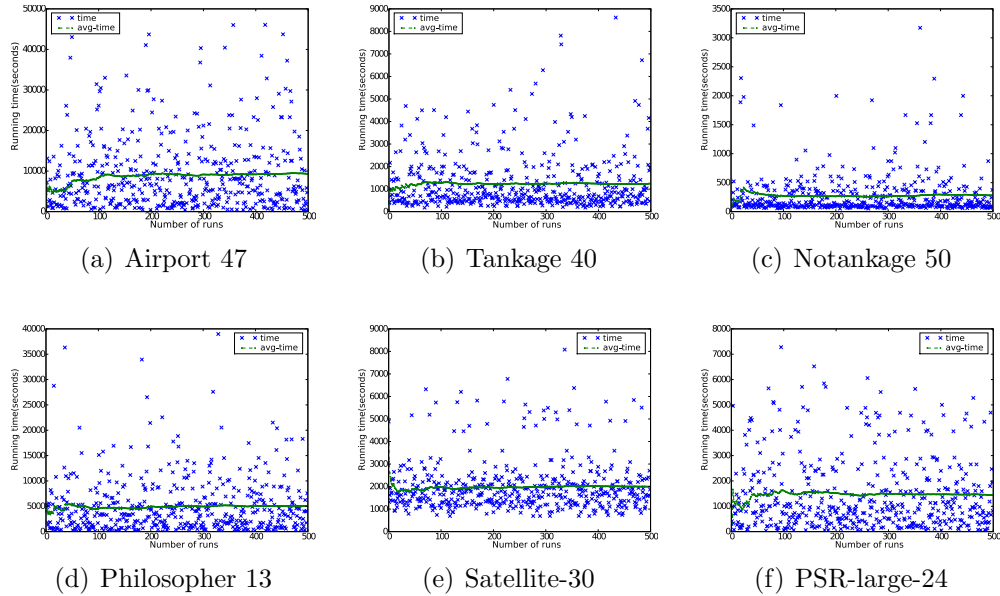


Figure 6.1: The run-time distribution of 500 MCRW runs with different random seeds on six planning problems.

distribution. We can see that many shorter runs take much less time than the average run-time (shown as the green line in Figure 6.1). Such a large variability can actually benefit a parallel scheme that makes multiple independent runs and terminate as soon as one run finds a solution.

	airport-47	tankage-40	notankage-50	philosopher-13	satellite-30	psr-large-24
μ	9288.6	1242.2	279.8	5036.9	1995.7	1445.7
σ^2	$7.8e7$	$1.4e6$	$1.3e5$	$2.9e7$	$1.2e6$	$1.9e6$
σ	8856.7	1196.2	357.4	5355.4	1104.1	1388.3

Table 6.2: The mean (μ), the variance (σ^2) and the standard deviation (σ) of 500 MCRW runs with different random seeds on six planning problems.

6.2.3 Portfolio Stochastic Search (PoSS) Algorithm

We propose a Portfolio Stochastic Search algorithm framework designed to take advantage of the short runs to get substantial speedup.

Algorithm 10 shows the framework of Portfolio Stochastic Search (PoSS). It simply calls N processes to run the MCRW procedure simultaneously and independently

Algorithm 10: PoSS(Π)

Input: a classical planning problem Π

Output: a solution plan

```
1 for each processor  $P_i, 1 \leq i \leq N$  do
2   |  $plan \leftarrow$  MCRW( $\Pi$ );
3   | if  $plan$  is a solution then
4   |   | send solution to controller;
5   |   | // controller would abort all other processors
6   | end
7 end
8 return  $plan$ ;
```

(Line 2). The node that finds the solution would send the solution to a central controller (Line 4). The central controller would then abort all processes. We use asynchronous communication between nodes and the central controller. There is no direct communication between computing nodes, which makes Algorithm 10 perfectly suitable for cloud computing architecture. PoSS also can tolerate the failure of processors since the search of each individual processor is independent where failures on one computational would not affect other processors.

When the communication time between each MCRW and the central controller is minimized, the running time of PoSS is the minimal solving time of N independent runs of MCRW.

We denote the running time of sequential MCRW and PoSS searches by random variable T_m and T^* , and the probability of running time shorter than t for MCRW and PoSS searches by $p_m(x < t)$ and $p^*(x < t)$, respectively. Given N MCRW searches in a portfolio, we have:

$$\begin{aligned} T^* &= \{t | t = \min\{t_1, t_2, \dots, t_N\}, t_i \in T_m\} \\ p^*(x < t) &= p_m(\min\{t_1, t_2, \dots, t_N\} < t) \\ &= 1 - p_m(t_1 \geq t)p_m(t_2 \geq t) \cdots p_m(t_N \geq t) \\ &= 1 - p_m(x \geq t)^N \\ &= 1 - (1 - p_m(x < t))^N. \end{aligned}$$

For instance, suppose $p_m(x < t) = 0.3$ and $N = 8$, we have $p^*(x < t) = 1 - (1 - 0.3)^8 = 0.95$. This means if 30% of the time an MCRW search terminates within t seconds, the corresponding portfolio stochastic search would terminate within t seconds at a probability of 0.95. Thus, even though the probability of short runs is relatively small in sequential MCRW search, the probability of hitting those short runs is large when we have a portfolio of MCRW searches.

Another insight we can draw from the above analysis is that we can drastically increase $p^*(x < t)$ if $p_m(x < t)$ is increased. For the most part, the probabilistic distribution p_m is determined by the efficiency of MCRW search itself. That in turn, is determined by the problem structure and the parameters for MCRW. With the problem structure fixed, we focus on finding proper parameter configurations to accelerate MCRW search.

6.2.4 Enhanced PoSS with Dovetailing

The MCRW algorithm has a few parameters affecting its performance, among which n (number of walks) and l (length of walk) are the most important, since they directly control the process of escaping from local minima and plateaus. If n and l are too small, the local search method is greedy as it tries to immediately exploit their local knowledge instead of exploring the neighborhood of current state. On the other hand, if they are too large, the search may take a long time on exploring the neighborhood of the current state. This exploitation–exploration tradeoff has long been observed in local search [8, 47].

As noted in the last chapter, setting the best parameter values for the random walk procedure can be challenging because we do not have complete information about the state space. MCRW by default sets $n = 2000$ and $l = 10$, which are tuned offline and give good average performance. However, there is no guarantee that this setting will perform well on each individual problem. As an example, we test different parameter values on a randomly selected problem: Airport-17. For each parameter setting, we run MCRW 10 times to get the average running time. The results presented in Figure 6.2 show that the performance with the default setting can be improved by arranging some other parameter settings.

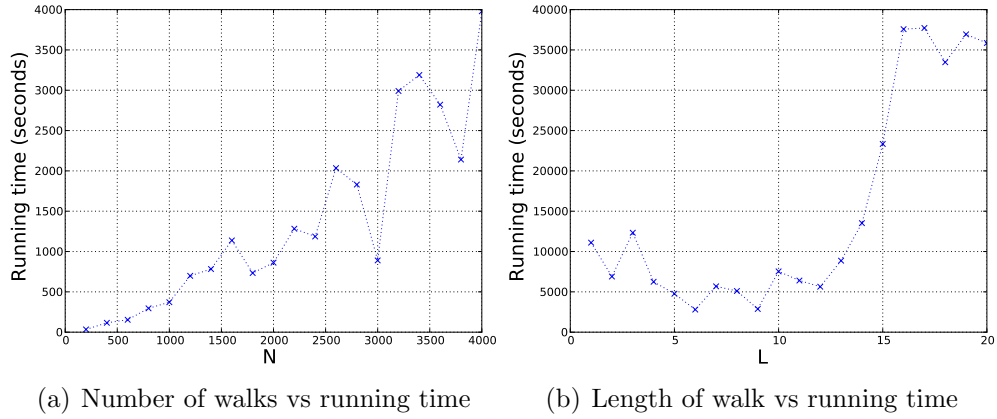


Figure 6.2: The average running time of the MCRW algorithm with different parameter settings on problem Airport-17.

Dovetailing is a procedure that performs search with multiple parameter settings and algorithms simultaneously [66]. It takes as its input a set of pairs of search algorithms and configurations $A = \{(a_0, c_0), (a_1, c_1), \dots, (a_n, c_n)\}$ where c_i is a configuration for algorithm a_i . The set A is also called an algorithm portfolio. Each computational node takes parameter settings from a candidate configuration set $C = \{c_0, c_1, \dots, c_n\}$.

The dovetailing version of PoSS, PoSS^d, adopts the following configurations. We set two general ranges for n and l , where n is from 200 to 3200, and l is from 1 to 15. These two ranges are set based on our empirical studies on planning domains. They are general enough to cover a vast range of planning domains in International Planning Competitions. In PoSS^d, each processor would perform an independent MCRW search with a set of parameters drawn uniformly from the aforementioned range.

6.3 System Implementation

We have implemented PoSS, PoSS^d in both parallel and cloud computing environments. For the parallel computing environment, we used MPI message passing for node communication. The Azure implementation is proven to be technologically challenging. We record our experiences here.

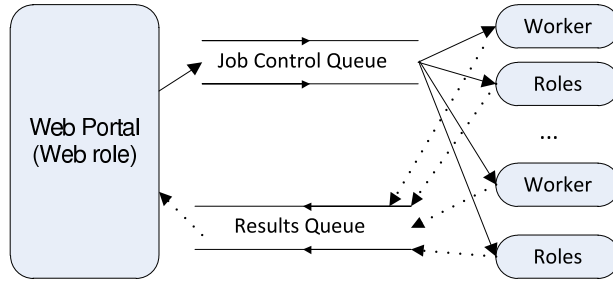


Figure 6.3: System architecture for PoSS in Windows Azure.

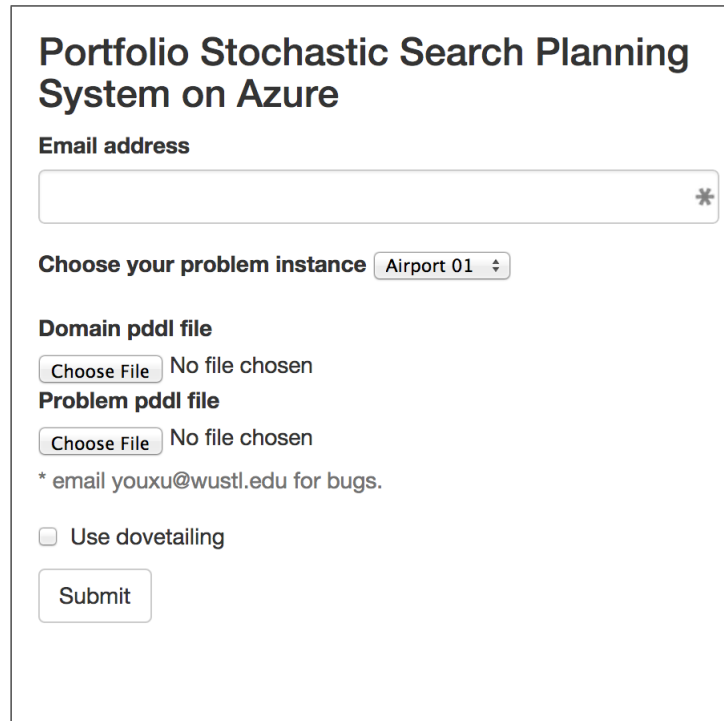
Windows Azure is a representative cloud computing system [16]. It provides a Windows-based cloud computing environment for running applications and storing data on servers in Microsoft data centers. Windows Azure offers *web roles* and *worker roles* that can be used for web hosting and computation respectively. It also provides communication mechanisms like *Queue* which web and worker roles can use to communicate with each other using asynchronized APIs. It is important to note that the latency of communication using Azure Queue is high. On average, adding or retrieving a message of 512 bytes in the queue requires as long as 20 ms, as measured in [38]. We also performed some tests by ourselves and found that Windows Azure Queue can support at most 100 message insertions and retrievals using its RESTful API. Therefore, it is not practical to use parallel algorithms that are designed for high performance clusters with low latency communication infrastructure. Instead, we run MRW algorithms in parallel with very limited communication in Windows Azure .

The system architecture that we use to implement PoSS and PoSS^d in Windows Azure is presented in Figure 6.3. We use an Azure web role to provide a web portal for users to submit jobs. The web role dispatches jobs to worker roles using a *job control queue*. For the PoSS algorithm with N processors, a web role will insert N messages into the job control queue, each containing a job ID, a set of initial parameters and a pointer to the planning problem. We have stored all IPC planning problems in a distributed file system in Azure so workers do not need to retrieve them from the web role.

Worker roles listens to the task queue. If a worker role is not currently working on any planning jobs, and there is a new job in the control queue, the worker role would consume the message and start the MCRM search. Parameters in the message are used to initialize the MCRW search. Once the MCRW search on a worker role finds a solution, it sends a *done* message, along with the job ID, to the job control queue.

Other worker roles that are working on the same planning problem, identified by the job ID, listening to the job control queue asynchronously, will then terminate promptly once the *done* message is posted.

The finished worker role will also send its solution and running time to the *result queue* for result collection. We separate the result queue and the job control queue because we would not want workers to be burdened by result messages, which usually contain a large amount of data and performance statistics. Such a design also leads to minimum communication, while still giving prompt termination of the PoSS algorithm once a solution plan is found.



The image shows a web form titled "Portfolio Stochastic Search Planning System on Azure". The form contains the following elements:

- Email address:** A text input field with a "*" icon on the right.
- Choose your problem instance:** A dropdown menu currently showing "Airport 01".
- Domain pddl file:** A "Choose File" button followed by the text "No file chosen".
- Problem pddl file:** A "Choose File" button followed by the text "No file chosen".
- Footer text:** "* email youxu@wustl.edu for bugs."
- Use dovetailing:** A checkbox that is currently unchecked.
- Submit:** A button at the bottom of the form.

Figure 6.4: A simple Web UI for users to submit planning tasks to PoSS running in Windows Azure.

We have ported the MCRW planner, which is originally developed in C++ on a Linux environment, onto a managed .NET environment in Windows Azure. We have also implemented our own APIs written in C++ for MCRW to access the queues and blob services provided by Windows Azure. Although there are APIs on the .NET platform provided by the Azure SDK to operate the Azure queues and blobs, these existing APIs are not easily accessible from native codes.

Hence, we write our own APIs, which use the RESTful APIs and plain stateless HTTP protocols to access and operate the tables, queues and blobs. Our APIs implement the base64, SHA256 and HMAC algorithm for authentication and HTTP requests using the cUrl library. We plan to open source the PoSS planning system as well as the C++ APIs for Azure cloud after some code cleanup.

Along with the APIs, we also provide a simple Web UI (Figure 6.4) where users who are interested in solving planning problems can use our system as a service. Users can submit planning jobs by submitting standard PDDL files for domains and problems. Alternatively, users can choose from pre-loaded IPC domains. The solutions are emailed to users as an email attachment.

It is worth noting that both the APIs and the web UI we developed can be extended to work with other planners running in Azure. We believe that this work would ease the way for planning researchers to experiment with new research ideas in cloud platforms such as Windows Azure. They would make planning more accessible to users who are not necessarily familiar with or do not have access to state-of-the-art planners.

6.4 Experimental Results

We present our experimental results in two parts by evaluating the performance of MCRW, PoSS and PoSS^d in a parallel computing cluster provided by Washington University and in Windows Azure cloud. Our experiments are conducted for problem domains from the Fourth International Planning competition (IPC 4) [2]. These domains are Airport (air), Pipesworld Tankage (tank), Pipesworld NoTankage (notank), Dining Philosophers (phi), Satellite (sate) and Power Supply Restoration-PSR Large (psr). We pick the most challenging problems (usually with larger problem indices) in each domain for the experiments.

6.4.1 Results for Parallel Computing

Here we run all experiments in a cluster with two computing nodes where each node has 8 Dual Core AMD Opteron Processors (2.15GHz) and 26GB memory. We test the parallelism with 8 and 16 MCRW instances in PoSS and PoSS^d. The time limit, for all the instances, is set to 3600 seconds. Intra-node communication for PoSS and PoSS^d are implemented using the MPI message passing library.

Problems	MCRW		PoSS						PoSS ^d					
	1		8			16			8			16		
	T	#	T	S	#	T	S	#	T	S	#	T	S	#
air-38	246.1	10	128.1	1.9	10	115.9	2.1	10	28.7	8.6	10	29.4	8.4	10
air-39	2228.6	5	1046.8	2.1	10	493.2	4.5	10	319.4	7.0	10	257.3	8.7	10
air-40	1287.0	8	515.6	2.5	10	350.0	3.7	10	252.5	5.1	10	145.6	8.8	10
air-41	1262.2	10	232.7	5.4	10	202.8	6.2	10	49.0	25.7	10	55.0	22.9	10
air-42	708.9	1	704.6	1.0	8	930.8	0.8	10	502.6	1.4	10	285.7	2.5	10
air-43	1643.6	1	980.1	1.7	10	730.6	2.2	10	411.6	4.0	10	305.0	5.4	10
air-44	726.9	10	394.5	1.8	10	313.1	2.3	10	90.0	8.1	10	102.2	7.1	10
air-45	1606.8	9	511.8	3.1	10	451.5	3.6	10	176.5	9.1	10	167.4	9.6	10
air-46	2969.5	1	2163.6	1.4	7	1116.6	2.7	10	718.7	4.1	10	734.7	4.0	10
air-47	1649.2	2	2912.4	0.6	3	2118.7	0.8	7	1543.8	1.1	10	1030.5	1.6	10
Summary	14328.9	57	9590.0	1.5	88	6823.2	2.1	97	4092.9	3.5	100	3112.9	4.6	100

Table 6.3: Comparison of MCRW and PoSS in different number of processors and strategies for the Airport domain. Problems with super linear speedups are highlighted.

For each domain, we choose the 10 hardest problem instances (measured by the average running time of 10 independent MCRW runs). Due to the statistic nature of these algorithms, we run algorithms on each instance 10 times and report the average running time here. Tables 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 give the results of MCRW, PoSS and PoSS^d algorithms in IPC-4 domains. In these tables, “T”, “S” and “#” represent the average running time, the speedup and the number of runs when an algorithm successfully finds a solution within the time limit.

From these tables, we can see that both PoSS and PoSS^d largely reduce the running time and provide substantial speedups. For instance, in problems such as air-41, tank-32, tank-33 and psr-17, they achieve super linear speedups. Super linear speedup means we achieve a speedup of more than n with n processors. These results show

Problems	MCRW		PoSS						PoSS ^d					
	1		8			16			8			16		
	T	#	T	S	#	T	S	#	T	S	#	T	S	#
tank-32	278.5	10	38.1	7.3	10	12.9	21.6	10	10.8	25.8	10	10.9	25.6	10
tank-33	1432.1	10	164.7	8.7	10	35.1	40.8	10	71.8	20.0	10	54.6	26.3	10
tank-34	262.2	10	55.2	4.8	10	29.4	8.9	10	21.4	12.2	10	25.7	10.2	10
tank-35	526.0	10	169.1	3.1	10	82.6	6.4	10	60.2	8.7	10	70.0	7.5	10
tank-36	1231.9	10	296.4	4.2	10	161.1	7.6	10	125.2	9.8	10	137.7	8.9	10
tank-37	871.3	9	155.4	5.6	10	78.4	11.1	10	74.1	11.8	10	88.7	9.8	10
tank-38	1054.3	10	409.3	2.6	10	132.1	8.0	10	107.4	9.8	10	144.2	7.3	10
tank-39	467.1	10	200.6	2.3	10	114.9	4.1	10	123.2	3.8	10	91.7	5.1	10
tank-40	861.4	10	326.4	2.6	10	187.3	4.6	10	281.6	3.1	10	152.1	5.7	10
tank-41	114.6	10	21.8	5.3	10	11.2	10.3	10	17.5	6.6	10	12.2	9.4	10
Summary	7099.6	99	1837.0	3.9	100	844.9	8.4	100	893.2	7.9	100	787.9	9.0	100

Table 6.4: Comparison of MCRW and PoSS in different number of processors and strategies for the Pipesworld Tankage domain. Problems with super linear speedups are highlighted.

Problems	MCRW		PoSS						PoSS ^d					
	1		8			16			8			16		
	T	#	T	S	#	T	S	#	T	S	#	T	S	#
notank-40	24.6	10	13.8	1.8	10	14.9	1.6	10	5.7	4.3	10	8.0	3.1	10
notank-41	8.5	10	4.1	2.1	10	4.4	1.9	10	1.6	5.3	10	2.0	4.3	10
notank-43	616.8	2	652.5	0.9	10	487.2	1.3	10	1144.2	0.5	10	339.5	1.8	10
notank-44	2436.0	1	2091.9	1.2	6	1910.8	1.3	8	1638.5	1.5	7	1010.6	2.4	10
notank-45	410.0	10	90.9	4.5	10	126.1	3.3	10	92.7	4.4	10	55.0	7.5	10
notank-46	1676.8	4	514.7	3.3	10	574.5	2.9	10	1277.3	1.3	10	309.3	5.4	10
notank-47	1097.9	2	1225.6	0.9	8	487.7	2.3	9	1105.0	1.0	9	1365.6	0.8	9
notank-48	1636.1	1	1038.4	1.6	9	571.3	2.9	10	1180.4	1.4	10	529.8	3.1	10
notank-49	81.9	10	42.0	1.9	10	47.0	1.7	10	13.1	6.2	10	22.8	3.6	10
notank-50	165.9	10	77.1	2.2	10	83.6	2.0	10	28.2	5.9	10	48.8	3.4	10
Summary	8154.5	60	5751.1	1.4	93	4307.3	1.9	97	6486.7	1.3	96	3691.3	2.2	99

Table 6.5: Comparison of MCRW and PoSS in different number of processors and strategies for the Pipesworld NoTankage domain.

Problems	MCRW		PoSS						PoSS ^d					
	1		8			16			8			16		
	T	#	T	S	#	T	S	#	T	S	#	T	S	#
phi-6	46.1	10	5.7	8.1	10	1.3	35.0	10	3.3	13.8	10	1.1	43.8	10
phi-7	14.0	10	2.6	5.5	10	1.3	10.5	10	0.5	25.6	10	0.7	20.4	10
phi-8	67.6	10	10.0	6.8	10	4.0	17.0	10	3.5	19.5	10	1.9	35.3	10
phi-9	41.2	10	7.2	5.7	10	1.9	21.4	10	3.3	12.7	10	2.4	16.9	10
phi-10	1006.4	10	181.1	5.6	10	42.7	23.6	10	16.5	61.0	10	8.3	120.9	10
phi-11	130.8	10	11.1	11.8	10	12.4	10.6	10	17.9	7.3	10	10.4	12.5	10
phi-12	929.7	8	94.3	9.9	10	120.5	7.7	10	20.9	44.5	10	30.0	31.0	10
phi-13	1747.5	6	573.1	3.0	10	194.0	9.0	10	85.3	20.5	10	61.9	28.3	10
phi-14	-	0	1768.9	-	7	1215.0	-	5	486.1	-	10	208.5	-	10
phi-15	-	0	939.4	-	4	1305.5	-	8	751.0	-	10	522.9	-	10
Summary	3983.3	74	3593.5	1.1	91	2898.7	1.4	93	1388.4	2.9	100	848.1	4.7	100

Table 6.6: Comparison of MCRW and PoSS in different number of processors and strategies for the Philosophier domain. Problems with super linear speedups are highlighted.

Problems	MCRW		PoSS						PoSS ^d					
	1		8			16			8			16		
	T	#	T	S	#	T	S	#	T	S	#	T	S	#
sate-21	17.1	10	11.5	1.5	10	10.4	1.6	10	6.1	2.8	10	5.8	2.9	10
sate-22	43.6	10	23.2	1.9	10	20.6	2.1	10	11.5	3.8	10	12.6	3.5	10
sate-23	272.1	10	76.2	3.6	10	61.4	4.4	10	54.3	5.0	10	58.2	4.7	10
sate-24	972.3	10	302.5	3.2	10	223.3	4.4	10	375.8	2.6	10	246.6	3.9	10
sate-25	1474.5	10	359.8	4.1	10	364.4	4.0	10	558.6	2.6	10	301.7	4.9	10
sate-26	1298.7	10	474.9	2.7	10	324.7	4.0	10	624.6	2.1	10	318.6	4.1	10
sate-27	1239.2	10	426.5	2.9	10	364.4	3.4	10	248.3	5.0	10	246.3	5.0	10
sate-28	475.5	10	223.1	2.1	10	215.4	2.2	10	153.8	3.1	10	141.7	3.4	10
sate-29	822.1	10	388.9	2.1	10	349.2	2.4	10	211.1	3.9	10	224.5	3.7	10
sate-30	1753.8	10	701.8	2.5	10	677.9	2.6	10	390.3	4.5	10	358.0	4.9	10
Summary	8368.9	100	2988.5	2.8	100	2611.6	3.2	100	2634.6	3.2	100	1913.9	4.4	100

Table 6.7: Comparison of MCRW and PoSS in different number of processors and strategies for the Satellite domain.

Problems	MCRW		PoSS						PoSS ^d					
	1		8			16			8			16		
	T	#	T	S	#	T	S	#	T	S	#	T	S	#
psr-16	0.9	10	0.8	1.0	10	0.3	2.7	10	0.2	4.6	10	0.2	5.4	10
psr-17	64.0	10	6.8	9.4	10	3.3	19.6	10	1.2	53.9	10	1.6	38.9	10
psr-18	117.7	10	33.4	3.5	10	6.3	18.8	10	2.9	41.0	10	1.9	61.5	10
psr-19	17.1	10	7.7	2.2	10	2.2	7.8	10	0.9	19.5	10	0.5	35.0	10
psr-20	1154.5	3	479.4	2.4	8	302.6	3.8	10	283.3	4.1	10	272.9	4.2	10
psr-21	55.3	10	28.9	1.9	10	5.4	10.2	10	3.2	17.5	10	1.2	45.6	10
psr-22	224.2	10	79.1	2.8	10	26.3	8.5	10	19.5	11.5	10	34.3	6.5	10
psr-23	1542.0	2	1027.4	1.5	9	640.4	2.4	10	330.8	4.7	10	167.1	9.2	10
psr-24	1114.0	10	351.7	3.2	9	116.2	9.6	10	267.8	4.2	10	110.1	10.1	10
psr-25	39.6	10	60.2	0.7	10	10.3	3.9	10	4.7	8.4	10	3.2	12.2	10
Summary	4329.2	85	2075.4	2.1	96	1113.1	3.9	100	914.3	4.7	100	593.1	7.3	100

Table 6.8: Comparison of MCRW and PoSS in different number of processors and strategies for the Power Supply Restoration domain. Problems with super linear speedups are highlighted.

that it is advantageous to use PoSS and PoSS^d over MCRW even when we factor in the overhead of running on multiple processors.

We would also like to point out that PoSS and PoSS^d largely reduce the standard deviation of running time, which makes them a much more predictable and hence favorable choice over the original MCRW.

6.4.2 Evaluation in Windows Azure

Table 6.9: Comparison of the running time and cost of PoSS algorithms using different number of nodes in Windows Azure. “T” is the running time in seconds, “S” is the speedup and “C” is the average total cost in US cents.

P	MCRW			PoSS																						
	1			4				8				16				120										
	T	C		T	S	C	T	S	C	T	S	C	T	S	C	T	S	C								
air-46	6999.7	23.3	1733.1	4.0	23.1	1240.2	5.6	33.1	675.6	10.4	36.0	121.7	57.5	48.6	962.0	7.3	12.8	320.5	21.8	8.5	253.4	27.6	13.5	61.3	114.2	24.5
air-47	10812.5	36.1	2831.5	3.8	37.8	1066.1	10.1	28.4	769.7	14.0	41.1	434.5	24.9	173.8	1878.4	5.8	25.0	927.3	11.7	24.7	496.3	21.8	26.5	78.0	138.6	31.2
tank-40	1385.4	4.62	512.8	2.7	6.8	241.4	5.7	6.4	122.2	11.3	6.5	106.9	13.0	42.8	480.4	2.9	6.4	220.5	6.3	5.9	283.2	4.9	15.1	109.4	12.7	43.8
tank-41	53.0	0.18	28.8	1.8	0.4	35.6	1.5	0.9	31.7	1.7	1.7	15.5	3.4	6.2	23.5	2.3	0.3	14.9	3.6	0.4	17.7	3.0	0.9	7.8	6.8	3.1
notank-49	96.7	0.3	59.3	1.6	0.8	42.2	2.3	1.1	43.7	2.2	2.3	28.1	3.4	11.25	49.1	2.0	0.7	26.5	3.6	0.7	19.5	5.0	1.0	15.9	6.1	6.4
notank-50	367.6	1.23	110.9	3.3	1.5	76.4	4.8	2.0	57.4	6.4	3.1	55.3	6.6	22.13	98.2	3.7	1.3	74.5	4.9	2.0	49.7	7.4	2.7	17.2	21.4	6.9
sate-29	1234.5	4.12	676.0	1.8	9.0	469.0	2.6	12.5	68.5	18.0	3.7	13.4	92.1	5.3	601.5	2.1	8.0	601.5	2.1	16.0	380.4	3.2	20.3	200.5	6.2	80.2
sate-30	2181.5	7.27	909.8	2.4	12.1	834.3	2.6	22.2	296.9	7.3	15.8	47.1	46.3	18.8	865.8	2.5	11.5	779.6	2.8	20.8	465.6	4.7	24.8	313.8	7.0	125.5
phi-12	1385.0	4.6	506.6	2.7	6.8	274.3	5.0	7.3	216.0	6.4	11.5	152.6	9.1	61.0	199.3	6.9	2.7	66.7	20.8	1.8	47.5	29.2	2.5	19.4	71.4	7.8
phi-13	5156.7	17.2	919	5.6	12.3	669.7	7.7	17.9	497.1	10.4	26.5	138.7	37.2	55.5	767.3	6.7	10.2	202.1	25.5	5.4	130.6	39.5	7.0	6.8	758.3	2.7
psr-24	1779.6	5.9	499.1	3.6	6.7	195.2	9.1	5.2	94.0	18.9	5.0	47.3	37.6	18.9	470.7	3.8	6.3	275.1	6.5	7.3	70.2	25.4	3.7	43.6	40.8	17.4
psr-25	73.7	0.3	28.2	2.6	0.4	20.8	3.5	0.6	10.1	7.3	0.5	10.0	7.4	4.0	20.2	3.6	0.3	28.9	2.6	0.8	25.9	2.8	1.4	6.2	11.9	2.5

We also evaluate both PoSS and PoSS^{ms} using Windows Azure, one of the major commercial cloud computing platforms [16]. For the following evaluation, we request up to 120 processors in Windows Azure. The experiments are conducted on the two hardest problems from each domain. For each instance and setting, we make 10 runs and report the average time and cost.

Financial cost is a major concern for cloud users. Cloud computing adopts a pay-as-you-go model for computational resources. Therefore, although theoretically we can employ a large number of processors for our algorithm portfolios, it is necessary to consider the tradeoff between speedup and cost. We report the running time (T), speedup (S) and monetary cost in US cents (C) in Table 6.9. The running time for PoSS and PoSS^d is measured as the delta between when web role issues messages to the job control queue and when the web role gets the solution from the result queue. In other words, it factors in all the communication overheads of the Azure platform. The cost is calculated based on a unit cost of \$0.12 per hour per CPU core, which is the standard rate for small instances in Windows Azure.

From Table 6.9, we see that the performance of running PoSS and PoSS^d in Windows Azure are similar to the results reported from the local cluster. Despite the overhead brought in by the job control and message passing mechanism we used in the Azure implementation, we have achieved substantial speedups in Windows Azure. Super linear speedups are observed for problems such as air-46, sate-29 and psr-25. Super linear speedup is especially beneficial when we factor in costs. For example, for phi-13 with 120 processors, not only can PoSS^{ms} achieve a great speedup, the financial charge (2.7 US cents) is also the lowest among all reported N s. Hence, for this problem, it is economical to use 120 processors instead of fewer processors.

For other problems, our scheme also achieves good tradeoff. For example, for air-47 and psr-25, to increase the number of processors from 16 to 120 only increases the total charge by less than 50%, but improves the speedup almost 4 times. Therefore, using more processors is still desirable when users are willing to trade a slightly higher cost for a significant speedup. After all, the total charge in a cloud is not very expensive using our algorithm. For the different numbers of processors we tested, it takes less than a US dollar to solve most instances. Both the speedup results and the cost

analysis show that cloud computing is an attractive platform for solving planning problems.

Finally, we point out that, our algorithm is robust under processor failures, which are commonly seen in cloud environments. For PoSS and PoSS^d, a failed run does not affect other parallel runs since they are independent and do not require communication. Previous parallel planning algorithms requiring intricate coordination are much more vulnerable to processor failures in the cloud environment.

6.5 Summary

In summary, we propose a parallel stochastic search (PoSS) algorithm designed to take advantage of the short runs in this distribution which can improve substantially. Our PoSS algorithm uses low frequency communication between computing nodes, which is perfectly suitable for cloud computing architecture. It also can tolerate the failure of processors since the search of each individual processor is totally independent, where one processor's failure won't affect other processors. We also present a parallel dovetailing technique, which is a procedure that performs search with multiple parameter settings simultaneously, to dramatically improve the efficiency of our algorithm. In summary,

- We show that the run-time distribution of Monte Carlo Random Walk (MCRW) algorithm in planning has a remarkable variability and propose a PoSS algorithm which takes advantage of short runs in this distribution.
- We use parallel dovetailing to solve the parameter tuning issue. In MCRW algorithm, the parameters can greatly impact the search performance. In practice, it usually can only find the setting with the best average performance, while there is no guarantee that this setting will perform well on each individual problem. In our parallel algorithm, each processor is assigned a unique parameter configuration chosen randomly from a value range, which is pre-decided through a large number of experiments. Thus, for each certain problem, each processor has the potential to get the parameter value that performs the best. Our

experimental results show that it dramatically reduces the search time of our algorithm.

- We implement two versions of parallel random walk algorithms on Windows Azure. The Windows Azure platform represents a new computing model and its potential for scientific research and applications remains largely unexplored. We study the performance characteristics of Windows Azure, and then develop scalable Azure-based schemes for stochastic search algorithm. We also implement scaffolding for deploying planning algorithms into the cloud environment, as well as a scheme that can launch multiple workers in Windows Azure.

Chapter 7

Conclusion

This dissertation proposed a set of techniques that can accelerate heuristic search for AI planning.

We have developed Stratified Planning, a straightforward and efficient technique for reducing the search space for heuristic search. Based on the insights gained from Stratified Planning and Expansion Core, we have developed a general partial order reduction framework for planning. We have established, for the first time, the connections between existing partial order reduction techniques in planning and the stubborn set theory in model checking. These techniques are orthogonal to heuristics, and therefore can be combined with any heuristics to accelerate search.

In addition to the study of the problem structure, we have studied the local structure of the search space. Based on the observation that heuristic search spends most of its time in plateau exploration, we proposed using random walks to assist heuristic search. The proposed methods have significantly accelerated heuristic search, as shown by results from IPC domains.

Last, we have taken the idea of stochastic search to the cloud computing platform. We have analyzed the advantages and shortcomings of the cloud computing platform, and proposed portfolio stochastic search algorithms that are amicable to the cloud platform. We have applied dovetailing techniques to the portfolio stochastic search scheme and further improved the efficiency of stochastic search. We have implemented the system in Windows Azure and reported super linear speedups in some problems.

7.1 Future Works

Heuristic search is a pervasive and important technique for AI. It occurs in a wide variety of engineering and scientific applications such as planning, scheduling, constraint satisfaction, game playing, VLSI technology, engineering design, power grid design, and computational sustainability. Accelerating heuristic search. therefore, has a broad impact on all of these applications. Here we point out some of the interesting directions for future work.

7.1.1 Symmetry in State Space

Symmetry detection is another way to reduce the search space [25]. It is different than partial order reduction as it explores the isomorphic relations between subgraphs in the search space. For example, consider a domain with three domain transition graphs A_1 , A_2 , and B , where A_1 and A_2 are symmetrical (isomorphic), and actions associated with B have no conflict with any actions associated with A_1 or A_2 . In this case, symmetry removal will expand actions associated with (A_1 and B) or (A_2 and B). This technique is different than partial order reduction proposed in this thesis. Future work here includes exploring the connections between symmetry and partial order reduction to see if the search space can further be pruned.

7.1.2 Helpful Actions

Helpful action (also called preferred operator) is one of the non-complete space reduction approaches [40, 33]. Generally speaking, helpful actions are applicable actions that are deemed to be helpful in solving a planning problem. A typical way of acquiring helpful actions is through heuristic evaluation. Certain heuristic functions such as Fast Forward and Fast Downward rely on solving a relaxed planning problem to calculate the heuristic function value of a state. It is likely that actions appearing in the solution to the relaxed problem are also part of the solution to the original planning problem.

Random exploration algorithms can use these actions as extra information to improve search efficiency. However, there is an exploration-exploiting tradeoff here. Helpful action is local information that can be misleading when the heuristic function is uninformative. It will be interesting to see the interaction of helpful action with random walk assisted heuristic search.

7.1.3 Probabilistic Models for Random Walks Guided by Heuristics

We have used a graph model to analyze when random walk is beneficial in heuristic search. The model we proposed is effective at explaining why random walk works well on some problem domains. There are some existing graph models that predict the hitting time of random walks without the influence of heuristic functions [57]. Similar to our model, these models can be used to analyze the performance of pure random walks on simple problem domains. It would be interesting to see if there is a probabilistic model for random walks guided by heuristics, so we can have better understanding of when to use random walk and how to select parameters for random walk.

7.1.4 Cloud-Based Deterministic Search

Most cloud platforms, including Windows Azure, are not optimized for low latency communication between processes. On the other hand, many parallel search algorithms require extensive inter-process communication. We have proposed to use stochastic search to utilize cloud computing wherein performance is not significantly affected by the high communication latency. There is some existing work to design a hash function that maps states to processors such that communication is localized between computational nodes [78, 79, 80, 14]. It would be interesting to see if techniques like these can be used to parallelize deterministic search in the cloud environment with relatively high communication latency.

References

- [1] IPC 3. The third international planning competition. <http://planning.cis.strath.ac.uk/competition/>, 2002.
- [2] IPC 4. The fourth international planning competition. <http://www.tzi.de/~edelkamp/ipc-4/>, 2004.
- [3] IPC 5. The fifth international planning competition. <http://zeus.ing.unibs.it/ipc-5/>, 2006.
- [4] IPC 6. The sixth international planning competition. <http://ipc.informatik.uni-freiburg.de/>, 2008.
- [5] IPC 7. The seventh international planning competition. <http://ipc.icaps-conference.org/>, 2011.
- [6] E. Amir and B. Engelhardt. Factored planning. In *Proc. IJCAI*, 2003.
- [7] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: a view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [8] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *FOCS*, pages 322–331, 1995.
- [9] A. Benaskeur, F. Kabanza, and E. Beaudry. CORALS:a real-time planner for anti-air defense operations. *ACM Transactions on Intelligent Systems and Technology*, 1(2):1–20, 2010.
- [10] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence, Special issue on Heuristic Search*, 129(1), 2001.
- [11] R. Brafman and C. Domshlak. Factored planning: how, when, and when not. In *Proc. AAAI*, 2006.
- [12] D. Bryce, S. Kambhampati, and D. Smith. Sequential monte carlo in probabilistic planning reachability heuristics. In *ICAPS*, pages 233–242, 2006.

- [13] D. Bryce, M. Verdicchio, and S. Kim. Planning interventions in biological networks. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.
- [14] E. Burns, S. Lemons, R. Zhou, and W. Ruml. Best-first heuristic search for multi-core machines. In *Proc. IJCAI*, pages 449–455, 2009.
- [15] R. Buyya, C.S. Yeo, S. Venugopal, et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
- [16] D. Chappell. Introducing the Windows Azure platform. <http://go.microsoft.com/?linkid=9752185>, 2010.
- [17] Y. Chen, Y. Xu, and G. Yao. Stratified planning. In *Proc. IJCAI*, 2009.
- [18] Y. Chen and G. Yao. Completeness and optimality preserving reduction for planning. In *Proc. IJCAI*, pages 1659–1664, 2009.
- [19] M. Cirillo, L. Karlsson, and A. Saffiotti. Human-aware task planning: an application to mobile robots. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.
- [20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
- [21] A. Coles, M. Fox, D. Long, and A. Smith. Additive-disjunctive heuristics for optimal planning. In *Proc. ICAPS*, 2008.
- [22] S. Dutt and N. Mahapatra. Scalable load balancing strategies for parallel A* algorithms. *Journal of Parallel and Distributed Computing*, 22(3):488–505, 1994.
- [23] A. Fern, S. W. Yoon, and R. Givan. Learning domain-specific control knowledge from random walks. In *ICAPS*, pages 191–199, 2004.
- [24] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proc. IJCAI*, pages 608–620, 1971.
- [25] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proc. IJCAI*, 1999.
- [26] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. In *Proc. of ISTCS*, 1995.
- [27] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. of International Workshop on Computer Aided Verification*, London, UK, 1990.

- [28] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. of Computer Aided Verification*, pages 438–449, 1993.
- [29] C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.
- [30] C. Gomes, B. Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24(1):67–100, 2000.
- [31] A. Grama and V. Kumar. State-of-the-art in parallel search techniques for discrete optimization problems. *TKDE*, 11(1):28–35, 1999.
- [32] Y. Hamadi and L. Sais. ManySAT: a parallel SAT solver. *JSAT*, 6:245–262, 2009.
- [33] M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
- [34] M. Helmert and C. Domshlak. Lm-cut: Optimal planning with the landmark-cut heuristic. In *The Seventh International Planning Competition*, 2009.
- [35] M. Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What’s the difference anyway? In *Proc. ICAPS*, 2009.
- [36] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*, pages 176–183, 2007.
- [37] M. Helmert and G. Röger. How good is almost perfect? In *Proc. AAAI*, 2008.
- [38] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of windows azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 367–376, 2010.
- [39] J. Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22:57–62, 2001.
- [40] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.
- [41] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23:279–295, 1997.
- [42] P. Jonsson and C. Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1-2):125–176, 1998.

- [43] E. Karpas and C. Domshlak. Cost-optimal planning with landmarks. In *Proceedings of the 21st international joint conference on Artificial intelligence*, pages 1728–1733. Morgan Kaufmann Publishers Inc., 2009.
- [44] E. Kelareva, O. Buffet, J. Huang, and S. Thiébaux. Factored planning using decomposition trees. In *Proc. IJCAI*, 2007.
- [45] A. Kishimoto, A. Fukunaga, and Adi Botea. Scalable, parallel best-first search for optimal sequential planning. In *ICAPS*, 2009.
- [46] C. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.
- [47] L. Kocsis and C. Szepesvri. Bandit based monte-carlo planning. In *Proc. of European Conference on Machine Learning*, pages 282–293, 2006.
- [48] V. Kumar, K. Ramesh, and V. Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proc. AAAI*, pages 122–127. Press, 1988.
- [49] A. Lansky and L. Getoor. Scope and abstraction: two criteria for localized planning. In *Proc. AAAI*, 1995.
- [50] S. M. LaValle. Planning algorithm. In *Cambridge University Press, Cambridge, U.K.*, 2006.
- [51] C. López, S. Jiménez, and A. García Olaya. The deterministic part of the seventh international planning competition. Submitted to AI Journal, 2015.
- [52] Q. Lu, Y. Xu, R. Huang, and Y. Chen. Roamer planner random-walk assisted best-first search. In *The Seventh International Planning Competition*, 2011.
- [53] W. Lu, J. Jackson, and R. Barga. AzureBlast: A Case Study of Developing Science Applications on the Cloud. In *Proceedings of the First Workshop on Scientific Cloud Computing, ScienceCloud 2010*. ACM, 2010.
- [54] A. McDonald. Parallel WalkSAT with Clause Learning. *Data Analysis Project Papers, Carnegie-Mellon University*, 2009.
- [55] F. Michael. Slow Moving Clouds Fast Enough for HPC. *HPC Wire*, August 10, 2009.
- [56] H. Nakhost and M. Müller. Monte-carlo exploration for deterministic planning. In *Proc. IJCAI*, pages 1766–1771, 2009.
- [57] H. Nakhost and M. Müller. A theoretical framework to study random walk planning. In *Fifth Annual Symposium on Combinatorial Search*, 2012.

- [58] D. Peled. Partial order reduction: linear and branching temporal logics and process algebras. In *Proceedings of the DIMACS workshop on Partial order methods in verification*, pages 233–257, 1997.
- [59] J. Porteous, M. Cavazza, and F. Charles. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Transactions on Intelligent Systems and Technology*, 1(2):111–130, 2010.
- [60] J. Porteous, L. Sebastia, and J. Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proc. European Conf. on Planning*, pages 37–48, 2001.
- [61] I. Refanidis and N. Yorke-Smith. A constraint based approach to scheduling an individual’s activities. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.
- [62] S. Richter and M. Helmert. Preferred operators and deferred evaluation in satisficing planning. In *ICAPS*, 2009.
- [63] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Proc. AAAI*, pages 975–982, 2008.
- [64] J. Romein, A. Plaat, H. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed search. In *Proc. AAAI*, pages 725–731, 1999.
- [65] K. Talamadupula, J. Benten, S. Kambhampati, P. Schermerhorn, and M. Scheutz. Planning for human-robot teaming in open worlds. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.
- [66] R. Valenzano, N. Sturtevant, J. Schaeffer, K. Buro, and A. Kishimoto. Simultaneously searching with multiple settings: an alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proc. ICAPS*, 2010.
- [67] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, Tampere University of Technology, 1988.
- [68] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, 1989.
- [69] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 1990 International Workshop on Computer Aided Verification*, 1990.
- [70] A. Valmari. Stubborn sets of coloured petri nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 102–121, 1991.

- [71] A. Valmari. On-the-fly verification with stubborn sets. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 397–408, London, UK, 1993. Springer-Verlag.
- [72] A. Valmari. The state explosion problem. *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science, 1491:429–528, 1998.
- [73] L. Vaquero, L. Rodero-Merino, and J. Caceres. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.
- [74] K. Varpaaniemi. On stubborn sets in the verification of linear time temporal properties. *Formal Methods in System Design*, 26(1):45–67, 2005.
- [75] C. Wintersteiger, Y. Hamadi, and L. De Moura. A concurrent portfolio approach to SMT solving. In *CAV*, pages 715–720, 2009.
- [76] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of the 4th International Conference on Concurrency Theory*, pages 233–246, 1993.
- [77] Y. Xu, Q. Lu, R. Huang, and Y. Chen. The roamer-p planner. In *The Seventh International Planning Competition*, 2011.
- [78] R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *Proc. AAAI*, pages 683–689, 2004.
- [79] R. Zhou and E. Hansen. Domain-independent structured duplicate detection. In *Proc. AAAI*, pages 683–688, 2006.
- [80] R. Zhou and E. Hansen. Parallel structured duplicate detection. In *Proc. AAAI*, pages 1217–1223, 2007.