Washington University in St. Louis
Washington University Open Scholarship

All Theses and Dissertations (ETDs)

1-1-2009

Partial Order Reduction for Planning

You Xu

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

Recommended Citation
Xu, You, "Partial Order Reduction for Planning" (2009). *All Theses and Dissertations (ETDs)*. 934.
http://openscholarship.wustl.edu/etd/934

This Thesis is brought to you for free and open access by Washington University Open Scholarship. It has been accepted for inclusion in All Theses and Dissertations (ETDs) 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:
Dr. Jeremy Buhler
Dr. Yixin Chen
Dr. Chenyang Lu

PARTIAL ORDER REDUCTION FOR PLANNING

by

You Xu

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

MASTER OF SCIENCE

Dec 2009
Saint Louis, Missouri

copyright by

You Xu

2009

ABSTRACT OF THE THESIS

Partial Order Reduction for Planning
by

You Xu

Master of Science in Computer Science

Washington University in St. Louis, 2009

Research Advisor: Professor Yixin Chen

Partial Order Reduction (POR) is a technique that reduces the search space by recognizing interchangeable orders between actions and expanding only a subset of all possible orders during the search. It has been extensively studied in model checking and has proven to be an enabling technique for reducing the search space and costs.

Several POR algorithms have been proposed in planning, including the Expansion Core (EC) and Stratified Planning (SP) algorithms. Being orthogonal to the development of accurate heuristic functions, these reduction methods show great potential to improve the planning efficiency from a new perspective. However, it is unclear how these POR methods relate to each other and whether there exist stronger reduction methods.

In this thesis, we have proposed a unifying theory that provides a necessary and sufficient condition for two actions to be semi-commutative. We have also revealed that semi-commutativity is the central property that enables POR. We have also interpreted both EC and SP algorithms using this new theory. Further, we have

proposed new, stronger POR algorithms based on the new theory. We have also applied these new algorithms to solve benchmark problems across various planning domains. Experimental results have shown significant search cost reduction.

Acknowledgments

I would like to thank my advisor, Dr. Yixin Chen.

I would like to thank Professors Jeremy Buhler and Chenyang Lu for serving on my Master of Science committee.

I would also like to thank Ruoyun Huang, Minmin Chen, Guohui Yao, Qiang Lu, Jianxia Chen and all other members in our research group for providing insightful comments on the work.

You Xu

Washington University in Saint Louis
Dec 2009

Contents

Abstract . . . . **ii**

Acknowledgments . . . . **iv**

List of Tables . . . . **vii**

List of Figures . . . . **viii**

1 Introduction **1**
1.1 Automated Planning 1
1.2 Observations of Partial Order Structure 3
1.2.1 Example I: A Simplified Truck Problem 3
1.2.2 Example II: A Maze Problem 5
1.3 Contributions 6
1.4 Thesis Outline 7

2 Background and Previous Work **8**
2.1 Classic Planning and Problem Formulations 8
2.1.1 STRIPS Formalism 8
2.1.2 SAS+ Formalism 10
2.2 Existing Methods 11
2.2.1 Model Checking Approaches 11
2.2.2 SAT Approaches 11
2.3 State Space Search for Planning 12
2.3.1 General State Space Search Procedure 12
2.3.2 Heuristic Search 14
2.3.3 Existing Heuristics in Planning 14
2.3.4 Existing Search Methods in Planning 18
2.4 Search Space Reduction in Planning 20
2.4.1 Partial Order Reduction 21
2.5 Conclusion 23

3 General Theory for Partial Order Reduction **24**
3.1 A Unifying Theory 24
3.1.1 Semi-commutative Conditions 24
3.1.2 Semi-Commutative Action and Path Pairs 26
3.1.3 Categories of search and reduction 29

3.2 Interpretations of POR Algorithms 31
3.2.1 Stratified planning (SP) 31
3.2.2 Expansion core (EC) algorithm 35
3.3 New POR Algorithms for Planning 38
3.4 Conclusion 41

4 Experimental results 42

5 Conclusion and Future Work 46

References 47

Vita 49

List of Tables

4.1 Comparison of several algorithms. We give number of generated nodes, number of expanded nodes, and CPU time in seconds. "-" means timeout after 300 seconds. 44
4.2 Comparison of FD and AC^+ on freecell (free) and pipesworld (pipe) domains. We show numbers of expanded and generated nodes. "-" means timeout after 1800s. 45

List of Figures

1.1 Number of generated nodes by the FD planning on the Driverlog domain 2
1.2 A simplified Truck problem 3
1.3 The solution to the Truck problem with only one truck 4
1.4 The solution to the Truck problem with two trucks 4
1.5 The solution to the Truck problem with two trucks 4
1.6 A Maze Problem on 2D plane 5

2.1 The DTG and Causal Graph in the Truck Problem 16

3.1 The causal graph and a stratification of Truck-02. 32
3.2 Stratified Planning Strategy 33

Chapter 1

Introduction

1.1 Automated Planning

Planning is one of our daily activities that involves arranging a series of actions in order to achieve certain goals. There are an abundance of planning problems in our daily life, even though we might not notice. For instance, we can describe the task "filling the gas tank" as a planning problem. This problem involves several actions like "select the gas type", "remove the nozzle", "stop the car", "swipe the credit card" and "open the gas tank lid". Each of these actions will cause the changing of some physical states. For instance, "remove the nozzle" will cause the state of the nozzle to change from in the dock to in your hand. Those state changes caused by actions are usually called effects. We want to arrange actions such that finally the effects of those actions construct the final goal state: a full tank of gas.

Another important concept in planning is called precondition. For instance, the action "remove the nozzle" will not be valid if the nozzle is not on its dock in the first place. Similarly, the action "fill the gas tank" can only happen if the nozzle is in your hand. In both cases, we observe that those actions require "preconditions" that must be satisfied to make those actions executable.

Thus, informally, a planning problem can be defined as to find a sequence of actions such that every action in the sequence gets preconditions satisfied when executed, and the collective effect of these actions will lead us from the initial state to a goal state we want to achieve.

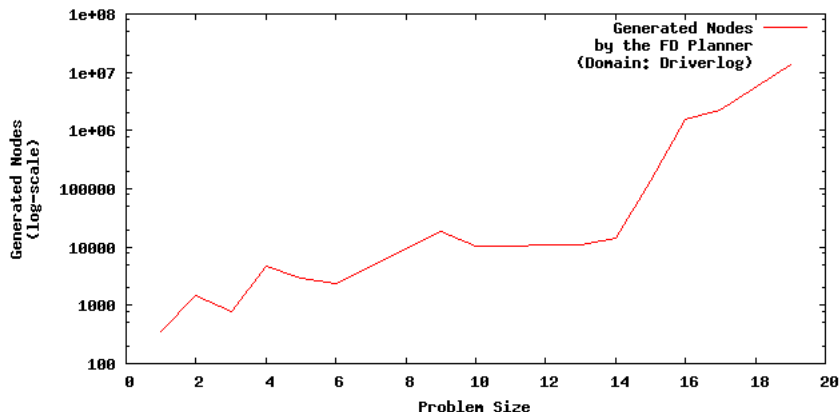


Figure 1.1: Number of generated nodes by the FD planning on the Driverlog domain

While the correct action sequence is easy to get for small instances like "buying gas", other complicated planning tasks are beyond the abilities of human beings because there are too many actions to pick from and usually millions of combinations need to be tested by trial and error. Therefore, it is necessary to explore the method of solving planning problems on modern computers. Since computers usually do not have the domain knowledge of the planning problem, we focus more on planning systems that require no domain specific knowledge. In Artificial Intelligence (AI) research, the subject of using computers to solve planning problems automatically without any domain specific knowledge is called "automated planning".

Not only is automated planning of practical significance, it is also important in theoretical AI research. In fact, many other important AI problems such as the discrete time scheduling problem, the constraint satisfactory problem and the general state space search problem can be formulated as planning problems. Thus, planning problem are considered to be the hardcore of AI research.

The state of the art methods of solving planning problems are to use state space search with heuristic functions acquired from structural analysis. We will explain in

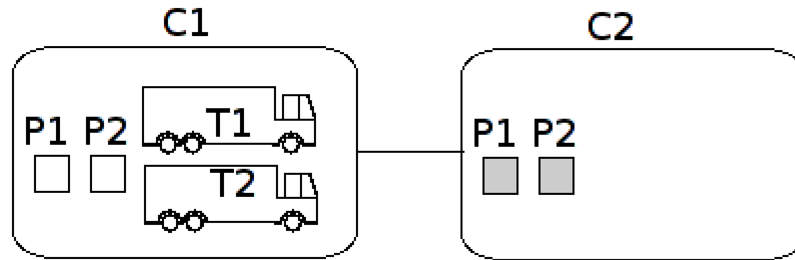


Figure 1.2: A simplified Truck problem

detail the heuristic based state space search methods of solving planning problems in Chapter 2. Here it is sufficient to know that one of the key challenge of using state space search methods is the exponential increase of search space with problem size, even when a high quality heuristic function is present. Figure 1.1 shows the exponential growth of the number of states in the search space when the size of the problem increases in the Driverlog planning domain using Fast Downward planner, which utilizes FF and causal graph heuristics.

1.2 Observations of Partial Order Structure

Both theoretical and experimental results indicate that the search space explosion still exists even when the heuristic estimation is almost perfect [13]. One of the reasons is that equivalent paths and duplicated states are generated during the search. These duplicated paths and duplicated states can be eliminated safely without affecting the correctness of our search. We use the following two examples to illustrate this point.

1.2.1 Example I: A Simplified Truck Problem

Truck is a planning domain that appeared in the fifth International Planning Competition. The basic object is to transport several goods from a city to another. Here we

simplify this domain and give a very simple example to illustrate our observations. The simplified problem is shown in Figure 1.2

In this simple problem, we have two trucks $T1$ and $T2$, and two goods $P1$ and $P2$, all initially in City $C1$. The goal is to transport both goods to city $C2$, as shown in gray. We also know that cities $C1$ and $C2$ are connected. For this problem, one possible solution is to use truck $T1$ only to move both goods to $C2$. Figure 1.3 shows the solution plan.

```
load T1 P1     drive T1 C1 C2      unload T1 P1
drive T1 C2 C1
load T1 P2     drive T1 C1 C2      unload T1 P2
```

Figure 1.3: The solution to the Truck problem with only one truck

We can, however, use two trucks to transport two packages, one truck for each package. For instance, if we use T2 for transporting P2, the new solution is listed in Figure 1.4.

```
load T1 P1     drive T1 C1 C2     unload T1 P1
load T2 P2     drive T2 C1 C2     unload T2 P2
```

Figure 1.4: The solution to the Truck problem with two trucks

We observe that the actions related to T1 are independent of the actions for T2. Thus, we can arbitrarily swap the order of some independent actions without affecting the final goal. For example, by swapping the actions “load T2 P2” and “unload T1 P1”, we can get the following solution in Figure 1.5

```
load T1 P1     drive T1 C1 C2     unload T2 P2
load T2 P2     drive T2 C1 C2     unload T1 P1
```

Figure 1.5: The solution to the Truck problem with two trucks

Essentially, by permuting these actions while preserving the required precondition constraints, there are in total $\binom{6}{3} = 20$ equivalent ways to achieve the same goal. Traditional search algorithm will explore all of these paths since it is unwise to ignore

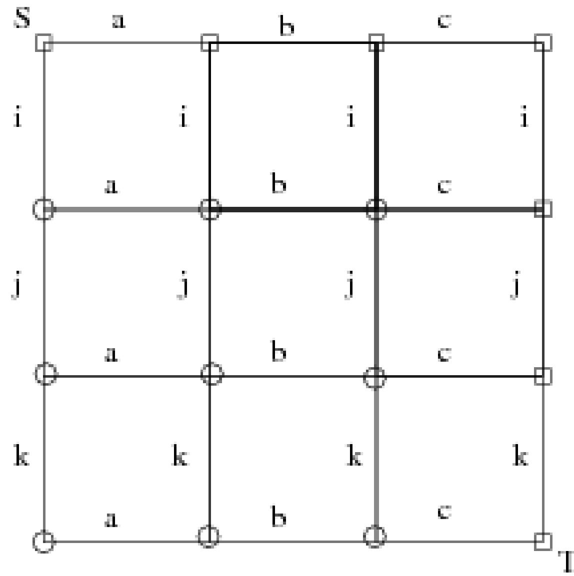


Figure 1.6: A Maze Problem on 2D plane

any path that can lead to the goal. Now if we decide that "load T1 P1" must happen before "load T2 P2" and ignore any paths where this order does not hold, then only $\binom{5}{2} = 10$ possible paths left to explore. We call this order a partial orders since it only involve the orders of a subset of actions. As we can see, imposing a single pair partial order can reduce the path enumeration by half. If we impose more partial orders, for instance, T2 can only work after T1 has finished unloading goods, then there is only one possible solution left – the exemplar solution discussed above.

As we can see from this example, without affecting the correctness of our solution, partial order can be imposed to reduce the number of paths we need to explore during search. We will discuss in detail in the next chapter the relationship between the size of the paths and the size of the search space. Here it is sufficient to know that in this case, partial order can reduce the number of paths by an order of magnitude.

1.2.2 Example II: A Maze Problem

In our first example, we showed that imposing partial order to actions can result in reduced paths. Here we use another example to show that imposing partial order can

result in a reduced number of states. Figure 1.6 shows a simple maze. The explorer of this maze starts from point S and his destination is T. Starting from S, the explorer usually has more than one path to choose from. Thus, the explorer can explore all 14 crossings. However, if a partial order is imposed such that actions a, b, c must be executed before actions i, j, k, the explorer will have only one path to explore, as marked by squares in the above graph. Other nodes marked by circles are previously reachable by an extensive exploration algorithm. They are, however, eliminated after partial order is imposed. In this case, the explored nodes are reduced from 16 to 7, or from N^2 to $2N - 1$ for a general 2-dimensional maze with N edges on each dimension. If we extend this Maze domain to higher dimensions, the explored nodes can be reduced from N^k to $kN - 1$ if the dimension number is k. This example shows that partial orders can also significantly reduce the number of nodes explored in the search space.

In summary, both examples show that by imposing a partial order onto action pairs, the number of paths and number of nodes can be significantly reduced, even when a perfect heuristic is used. We name this family of methods "partial order reduction" as they impose partial orders to the solution to reduce search space.

Although the basic idea behind partial order reduction is intuitive, it is still unclear how to pick these pairs and what partial order should be imposed. It is also unclear what the general theory for doing partial order reduction method for planning is. The answers to these questions comprise the main contributions of this thesis.

1.3 Contributions

The main contributions of this thesis are as follows:

a) We propose a general theory for doing partial order reduction in planning. We prove several necessary conditions for partial order reduction methods.

b) Based on our general theory, we successfully explain two previously proposed space reduction methods in an unified manner.Our explanation also provides more insights on designing other partial order reduction methods.

c) Built on the general partial order reduction theory, we propose a partial order reduction method based on direct analysis of actions. We implement this reduction method and test it extensively on several planning benchmarks. Experimental results show that it performs better than the Stratified Planning, a partial order reduction approach.

1.4 Thesis Outline

This thesis is organized as follows:

In Chapter 2, we introduce the background of automated planning and review previous work. We first introduce several formalisms for automated planning. Second, we introduce the framework of using informed state space search to solve automated planning problems. Finally, we review existing space reduction methods in planning, and discuss their assumptions, major ideas and limitations.

In Chapter 3, we present our theoretical work on the necessary condition of partial order reduction. We first introduce the basic concepts in partial order reduction. Then we present our central theorem based on semi-commutative action pairs. Then, we discuss the basic concept of causal graph analysis and introduce the main ideas behind two existing space reduction methods: Stratified Planning (SP) and Explosion Core (EC). Then, we use our theory to unify these two methods. Finally, we propose a modified version of the Stratified Planning algorithm based on action relationship analysis.

In Chapter 4, we test our proposed algorithms on several planning benchmarks. We compare the performance of our algorithm with the Stratified Planning algorithm on these benchmarks.

Finally, in Chapter 5, we summarize the research work in this thesis and discuss some future directions to extend this research.

Chapter 2

Background and Previous Work

In this chapter, we first introduce several basic concepts of automated planning and three major formulations for classic planning domains.

2.1 Classic Planning and Problem Formulations

Depending on the different categories of constraints on actions, automated planning problems can be divided into two categories: non-temporal planning and temporal planning. Actions in temporal planning problems might have starting time or ending time constraints and durations. In contrast, actions in non-temporal planning have no time constraints and each action can be treated as an atomic operation. In this thesis, we only consider the family of non-temporal planning problems where any possible state in this planning problem can be described using propositional logic. These problems are usually defined as classic planning problems.

Given a planning problem domain, the first step is to formulate it in a language that a computer can understand. We introduce two popular formalisms–STRIPS and SAS+.

2.1.1 STRIPS Formalism

STRIPS (Stanford Research Institute Problem Solver) is an automated planner developed by Richard Fikes and Nils Nilsson. This planner first adopted a propositional

logic system to describe the planning problem. Now, the term "STRIPS" is often used to refer to the propositional logic system for describing planning problems.

Definition 1 *A STRIPS planning task P can be written as a four tuple (F, I, O, G), where F is a set of facts, O is a set of actions. $I \subset F$ and $G \subset F$ are the set of initial facts and goal facts, respectively. For each action $o \in O$, P_o, A_o and D_o denote the set of preconditions, add effects and delete effects, respectively. F are propositional atoms and P_o, A_o and D_o are all logical conjunctions of atoms in the propositional logic system.*

The semantics of STRIPS are as follows: if all preconditions of an action a are satisfied at some state s, a is applicable at s. Add effects will become true after the execution of a at s, and delete effects will no longer be true after the execution.

For instance, in the truck domain, action "load T1 P1 at C1" has preconditions "AT P1 C1, AT P1 C1". Thus, this action is applicable at the initial state. Its add effect is "AT P1 T1" and its delete effect is "AT P1 C1". After the execution of this action, P1 will be in T1 (added) and AT P1 C1 will no longer be true (deleted).

Since everything in the STRIPS formalism is based on the propositional logic, we usually say that STRIPS adopts the propositional logic paradigm for modeling planning problems. Similar to the fact that first order logic is the natural extension to the propositional logic paradigm, Action Description Language (ADL) and its super set Planning Domain Definition Language (PDDL) are two formalisms based on first order logic[1]. Most of the benchmarks used in this thesis are essentially formulated in ADL and PDDL instead of STRIPS because ADL and PDDL descriptions are usually more terse than STRIPS descriptions. This difference is, however, not important for this thesis since we can always transform an ADL/PDDL formulated classic planning problem to a STRIPS representation.

[1] PDDL also contains other extensions for temporal planning problems.

2.1.2 SAS+ Formalism

SAS+ is another way to formulate planning problems. Rather than using propositional atoms, the SAS+ formalism models states by multi-valued state variables.

Definition 2 *In SAS+ formalism, a planning problem is defined as a tuple* $\Pi = (X, O, S, S_I, S_G)$. *Here,* $X = \{x_1, x_2, \cdots x_N)$ *is a set of multiple-valued state variables, each with an associated finite domain* $Dom(x_i)$. O *is a set of actions and each action* $o \in O$ *is a tuple* $(pre(o), eff(o))$, *where both* $pre(o)$ *and* $eff(o)$ *define some partial assignments of variables in the form* $x_i = v_i, v_i \in Dom(x_i)$. S *is the set of states.* S_G *is a partial assignment that defines the goal and* S_I *is a full assignment of all state variables that defines the initial state. Variables involved in* S_G *are defined as goal variables.*

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* or *enabled* in state s. After applying o to s, the state variable assignment will be changed to a new state s' according to $eff(o)$and $eff(o)$. We denote the resulting state of applying an applicable action o to state s as $s' = apply(s, o)$.

There is a natural correspondence between SAS+ and STRIPS formalisms of planning problems. For an action o, P_o, A_o and D_o correspond to the variables assignments in $pre(o)$, $eff(o)$, respectively. We shall also note that there are existing works to translate the STRIPS formalism to a SAS+ formalism. In most of the cases, we do not need to consider the differences between SAS+ and STRIPS. It is, however, important to know that in this thesis, we proved our theory mainly on STRIPS formalism and we applied our theory to Fast Downward, a SAS+ based automated planner. This is because STRIPS is more suitable for theoretical analysis and SAS+ formalism is easier to use in algorithm implementation. In this thesis, we will point out our use of specific formalisms if it is not clear from the context, otherwise we will use whichever is more convenient for description.

2.2 Existing Methods

Many works are proposed for solving classic planning problems. We first briefly review several existing works that are beyond the main scope of this thesis and then dedicate a subsection to discuss the state space search approach for solving automated planning, which is the family of methods this thesis belongs to.

2.2.1 Model Checking Approaches

The first family of method involves using model checking approaches to solve planning problems [7, 11, 8]. The basic idea is to transform the planning problem into a model checking problem and use existing model checkers to solve the planning problem. The binary decision graph (BDD) is used to denote the abstract search space. In every round, if the goal is not in the abstract search space, the model checker will expand one more layer and construct a new BDD using some BDD operations based on the BDD in the last round. Since search space is represented implicitly using BDD, this family of methods is also called symbolic planning. The main problem with this family of approaches is that structural information of the planning domain is usually lost during the translation from planning problem to model checking formulation. For a model checker, even if the search is conducted in an abstract space, uninformed search would still result in huge BDDs.

2.2.2 SAT Approaches

Similar to the first approach, planning problems can be transformed into a sequence of boolean satisfiability problems (SAT) [17, 6]. The basic idea is to treat the solving procedure of a planning task as mapping actions into steps such that goal conditions can be satisfied by executing actions step by step from the initial state. Thus, this method asks a series of questions "can this planning problem be solved in N steps" starting from $N = 1$. These questions are then encoded into a SAT formulation and SAT solvers are employed to decide their satisfiability. We omit the technical details of the encoding here. It is easy to see that if we try from $N = 1$ up to $N = k$

where the SAT problem is first time satisfiable, then k is the minimal number of steps to solve this planning problem. In fact, one of the advantages of using a SAT based method is that it returns the optimal solution in the sense of number of steps. The disadvantages are, however, obvious: SAT problems themselves are in the NPC complexity category; despite the rapid development of SAT solvers, it is still hard to solve large instances of SAT problems. In practice where optimality is not required, state-of-the-art heuristic search based approaches are usually one or more orders of magnitudes faster than SAT based methods.

Besides these two methods, there are also other approaches such as partial order planning, hierarchical planning [9] and graph planning [2]. Since these works are, however, less relevant to the work in this thesis, we omit the detailed discussions of these methods here.

2.3 State Space Search for Planning

Like SAT approaches and Model checking approaches, state space search methods can be used to solve planning problems. State space search is so far the most successful method in practice, as most of the winning planners of the International Planning Competitions are using heuristic search framework. Since our work in this thesis is also based the state space search method, we explain it thoroughly in this section.

2.3.1 General State Space Search Procedure

In short, the state space search method is the practice of finding a path (or a sequence of actions in planning) that links the initial state to a goal state through exploring states in the "state space".

Figure 1 shows a general state space search algorithm. We usually denote the "insert SUCCESSOR(node)" step as the expansion step.

We also introduce the following definitions that are related to state space search procedure:

Algorithm 1: General State Space Search Procedure

```
Input: problem, fringe
Output: found or failure
closed ← ∅ ;
insert INITIAL STATE to fringe ;
while True do
    if fringe is empty then
        return failure
    end
    node ← REMOVE-FIRST (fringe) ;
    if node is GOAL then
        return found
    end
    if node is not in closed then
        add node to closed ;
        insert SUCCESSOR(node) to fringe ;
    end
end
```

Definition 3 *The state space graph SG is a directed graph where each node is a state, and each edge (u, v) is an edge in SG if and only if v is in* SUCCESSOR*(u). All nodes in this graph comprise the search space for this problem. Here v is the successor of u and v is the precedence of v.*

Definition 4 *A search path is defined as a path in the state space graph. A goal path is a search path starting from the initial state to some goal state.*

Definition 5 *The set of expanded states is the set of all nodes in closed when the search procedure terminates. The explored space is the set of expanded states and the paths between them.*

Later on in this thesis, for simplicity, when there is no ambiguity, we use 'path' for 'search path'.

Solving planning problems using Algorithm 1 is straightforward. We start from the initial state and apply applicable actions in the expansion step to generate successor states until we reach some goal state or get failure. Then, if a goal state can be

found, by tracing back from this goal state, a goal path can be extracted and the corresponding action sequence is a valid solution to the planning problem. The main challenge here is to find the solution without triggering the space explosion problem. Since the state space for a planning problem is fixed once the problem is defined, what we really want to do is to reduce the explored search space. Two major techniques can be used to reduce the explored space: heuristic search and space reduction. We first introduce heuristic search.

2.3.2 Heuristic Search

Recall that in Algorithm 1, $fringe$ is a configurable data structure defined by users. For example, if $fringe$ is implemented as a stack, the above algorithm will be a depth first search procedure. Generally, nodes in $fringe$ can be accessed in any order other than in a FIFO or LIFO manner. The key idea of heuristic search is to use "heuristic function" to help ordering nodes in $fringe$. For a given search space, the heuristic function is defined as follows:

Definition 6 *Heuristic function h is a function that maps nodes in search space into numerical values. For a node n, $h(n)$ typically estimates the distance from n to goal. A heuristic function is called admissible if it always underestimates the distance.*

2.3.3 Existing Heuristics in Planning

Many successful heuristics have been proposed in the last decade. We briefly review several successful heuristics[2] for solving planning problems.

The Fast-Forward Heuristic

The Fast-Forward (FF) heuristic [14] was proposed by Hoffmann in 2000. The Fast-Forward planner that utilizes FF heuristic was the most successful planner in the

[2]Here we refer to heuristics employed in award winning planners or heuristics that have high impact in planning research.

AIPS 2000 planning system competition. To illustrate the FF heuristic, we first introduce a special data structure called the Planning Graph (PG) [20].

PG is a graph that has alternate layers of actions and facts. Starting from the initial state as the initial fact layer, we can apply all applicable actions with respect to this fact layer. All these applicable actions comprise a new action layer. The next fact layer is constructed of the union of the facts in the previous layer and all the add effects of the actions in previous action layer. This procedure will be repeated until the fact layers finally converge to a fixed set. This convergence is guaranteed since there are only a fixed number of facts and fact layers are monotonically increasing. Note that no delete effects of the actions are considered in constructing PG.

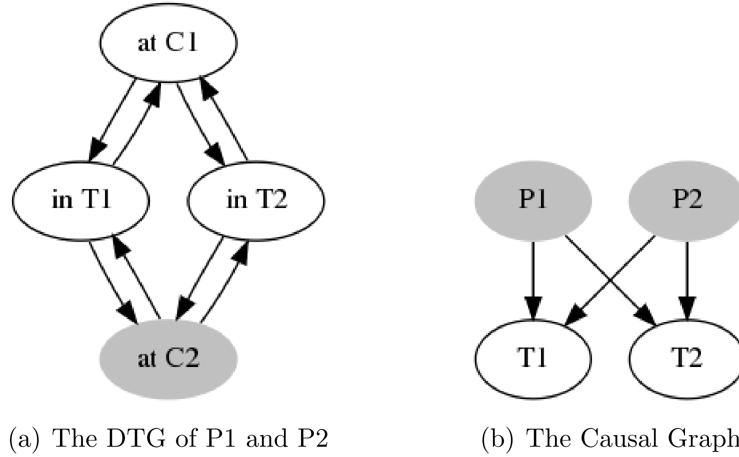
FF evaluates the heuristic value of a given state s by construction PG using s as the initial state. After PG is constructed, FF extracts a goal path from constructed PG and uses the length of this path as the heuristic value. Since the extracted plan is just one of the many possible plans from s to goal and it is not necessarily an optimal plan, FF heuristic is not admissible since it might overestimate the distance from s to goal.

The Fast Downward Heuristic

The Fast Downward (FD) heuristic [12] was proposed by Helmert in 2006. The Fast Downward planner that utilizes FD heuristic was one of the winners of IPC 2006 planning competition. Unlike FF, which uses STRIPS formalism, FD uses SAS+ formalism.

To illustrate the FD heuristic, we first introduce two concepts: Domain Transition Graph (DTG) and Causal Graph (CG).

Definition 7 *For a SAS+ task, for each state variable $X_i, i = 1, \cdots, N$, its Domain Transition Graph (DTG) G_i is a directed graph. the vertex set $V(G_i)$ is $Dom(x_i)$. An edge $e_i = (v_i, v_i')$ belongs to $E(G_i)$, the edge set of G_i, if there is an action o with $v_i \in pre(o)$ and $v_i' \in eff(o)$. We denote the association relationship between action o and edge e_i by $o \vdash e_i$.*



(a) The DTG of P1 and P2 (b) The Causal Graph

Figure 2.1: The DTG and Causal Graph in the Truck Problem

In our truck example, if we treat the position of package P1 as a variable x, it can take four values: in C1, in C2, in T1 and in T2. The corresponding DTG is shown in Figure 2.1(a). All directed edges are associated with a valid action. We mark "at C2" graph since it is our goal. We define a DTG as a goal DTG if its corresponding variable is goal variable.

DTG denotes the transition between variable values. It, however, does not reflect the relationships between variables. Causal Graph (CG) bridges this gap. It is defined as follows:

Definition 8 *Given a SAS+ planning task* Π *with state variable set* X*, its Causal Graph (CG) is a directed graph* $CG(\Pi) = (X, E)$ *with* X *as the vertex set. There is an edge* $(x, x') \in E$ *if and only if* $x \neq x'$ *and there exists an action* o *such that* $x \in trans(o)$ *and* $x' \in dep(o)$*, or,* $x \in eff(o)$ *and* $x' \in trans(o)$*.*

To illustrate the above definitions, we again take the Truck problem as an example.

In the Truck example, we have four variables, namely, position of T1 and T2 and position of P1 and P2. We denote them by variable T_i and P_i where $i \in \{0, 1\}$. Both T_1 and T_2 take two values: at C1 and at C2. We use 0 and 1 to represent them.

To the contrary, both P_1 and P_2 take four values: at C1, at C2, in T1 and in T2. We denote them by integer value $0, 1, 2$ and 3, respectively. The DTGs of P_1 and P_2 are the same, as shown in Figure 2.1(a). Now, for action LOAD P1 to T1 at C1 that changes the value of P_1 from 0 to 2, its precondition contains $T1 = 0$. Therefore, there is a directed edge in the causal graph from P_1 to T_1. Figure 2.1(b) shows the final causal graph of the Truck problem where gray nodes mark goal DTGs.

Intuitively, the nodes in the CG are state variables and the arrows in the 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.

After construction DTGs and the causal graph for a SAS+ planning task, the FD heuristics will prune some directed edges in the causal graph to make this directed graph acyclic, and thus relaxes the original problem. Theoretical results show that if the causal graph of a planning problem is acyclic, it is solvable in polynomial time [12]. FD solves this relaxed problem and uses the length of the goal path from current state s in the relaxed problem as the heuristic value for state s.

Landmark Heuristic

The Landmark (LAMA) Heuristic is proposed by Richter and Helmert in 2008 [18]. The LAMA planner that utilizes the LAMA heuristic was the winner of IPC 2008 planning competition. The LAMA heuristic also uses SAS+ formalism.

To show how LAMA works, we first introduce the definition of landmarks for a planning problem.

Definition 9 *For a SAS+ planning task* Π*, a variable assignment* $x_i = v_i$ *is a landmark if and only if for any goal paths,* $x_i = v_i$ *is true at some states in that path.*

Intuitively, landmarks are the partial assignments of variables that any goal paths must visit.

The LAMA heuristic works in the following way. First, it uses a landmark discovery and verification module to extract n landmarks from Π. Then, partial orders between

these landmarks will be decided based on some deductions and heuristic guesses. Based on these discovered landmarks and their partial orders, a landmark graph is constructed with landmarks as vertices and partial orders between landmarks as directed edges.

For any given state s, LAMA evaluates its heuristic functions as follows. It first decides the number of landmarks that have been visited by the path from the initial state to s as m, and the number of landmarks that are required again from s as k, based on the orders in the landmark graph. Then, the heuristic value $h(s)$ is defined as $n - m + k$. Here the heuristic value $h(s)$ not only depends on state s, but also the path from the initial state to s. Thus, strictly speaking, it is not a heuristic but a pseudo-heuristic. Nevertheless, LAMA planner uses this value as heuristic in best-first search.

In summary, these aforementioned heuristics stand for the state of the art of heuristics in automated planning. They are all very accurate. However, none of them is significantly better than the other two across all planning domains. Thus, in practice, these heuristics are usually combined to achieve a better performance. For instance, the FD planner uses both FF and FD heuristics. The LAMA planner uses both LAMA and FF heuristics.

2.3.4 Existing Search Methods in Planning

In this subsection, we briefly review several search strategies that appear in the state-of-the-art automated planners.

A^* Search

Recall that in Algorithm 1, $fringe$ is a user-defined data structure. A^* uses a priority queue as $fringe$. For a state s in $fringe$, its priority value f of s is the sum of its heuristic value $h(s)$ and the path cost $g(s)$ measured as the distance from the initial state $\mathcal{I}$ to s. Thus, nodes in $fringe$ are retrieved according to its f value.

The well-known advantage of using A^* is that if the heuristic function is admissible, it is guaranteed to find an optimal solution (if any) in the sense of goal path length.

However, as we discussed in the last subsection, because all state-of-the-art heuristics are not admissible, A^* loses its advantages. Thus, in practice, A^* is only used when the optimal solution is required.

Best-First Search

Similar to A^*, best-first search implements $fringe$ as a priority queue. The priority value of nodes in $fringe$ is their heuristic values. Therefore, REMOVE-FIRST will also pick the node in the $fringe$ with the smallest heuristic value.

Unlike A^* search, the solution found by best-first search is not guaranteed to be optimal even if the heuristic is admissible. However, best-first search is widely used in practice due to the fact that all state-of-the-art heuristics are usually accurate but not admissible. Both FD and LAMA use best-first search as their basic search methods.

Hill Climbing

Hill Climbing can be treated as a special kind of best-first search except that after the REMOVE-FIRST operation, it clears out the entire $fringe$. In other words, at any state s, it only picks the best successor and ignores all other successors.

Hill climbing is generally considered greedy and it is essentially an incomplete algorithm. Possible goal paths might be dropped out during hill climbing. In practice, FF planner uses hill climbing search and ff heuristics, and will restart a best-first search if hill climbing hits a dead end. According to the search space topology analysis [15, 16], for problems where search space has no dead end or local minimal, hill climbing performs surprisingly well.

Deferred Heuristic Evaluation

Deferred heuristic evaluation is a technique to boost the speed of best-first search. Recall that when best-first search is used, states in $fringe$ are ordered by their heuristic values. That is to say, heuristic evaluation for all states must happen before being inserted into $fringe$. Deferred heuristic evaluation delays the evaluation of heuristic value to the expansion step. Instead of evaluating the heuristic value of a state s before inserting it to $fringe$, it copies the heuristic value of s's precedence as the

heuristic value of s. Later on when s is retrieved from $fringe$, its heuristic value will be evaluated before the expansion phase.

The motivation of this method is straightforward. As we discussed in the previous section, state-of-the-art heuristics are generally very accurate. However, evaluating the heuristic value still requires solving a relaxed problem in polynomial time. Thus, heuristic evaluations should happen as little as possible. In practice, deferred heuristic evaluation can reduce the number of evaluations by about one order of magnitude [19]. Because heuristic evaluation takes most of the computation time in the search procedure, reducing the number of heuristic evaluations also significantly reduces the overall search time.

In summary, we discussed the state space search procedure for solving automated planning problems in this section. We also reviewed three state-of-the-art heuristics and four search techniques for solving planning problems. In the next section, we will introduce the search space reduction methods that are independent from the development of heuristics and search techniques and can therefore be combined with techniques discussed in this section to further improve the state space search procedure for solving planning problems.

2.4 Search Space Reduction in Planning

Heuristic search also has some fundamental limitations. Both experimental and theoretical results show that even the almost perfect heuristic function will result in exponential explored space. Also, modern heuristics usually rely on solving relaxed planning problems, thus heuristic evaluation is often computationally expensive. In practice, planners like LAMA and FD running on a modern computer can only expand approximately 1000 nodes per second. Thus, for a problem with more than 90 million nodes in the explored space, no planner can finish exploring it within a day.

Recall that in Chapter 1 we made the observation that not all nodes are worth considering. Thus, finding states in the explored space that can be reduced is vital for solving problems faster. We briefly review three existing space reduction techniques in this section.

Subgoal Partitioning

Subgoal partitioning was proposed by Chen et al. [4]. The planner that uses this technique, SGPlan, won the second place on the suboptimal propositional track in International Planning Competition, 2004.

SGPlan reduces the search space by partitioning a planning task into several subproblems, each of them having initial states and goals. A modifier planner based on FF is then employed to solve these subproblems. Conflicts between subproblems will then be resolved by increase the penalty of these conflicts.

Instead of having a search space consisting of all states in the state space graph, which is the Cartesian product of the search space of each subproblems. Ideally, SGPlan's explored space is propositional to the union of the search spaces of these subproblems, instead of the Cartesian product.

Preferred Operators

As we introduced earlier, state-of-the-art heuristic functions usually solve a relaxed planning problem to get the heuristic evaluation. The main idea of the preferred operator approach is that the solutions to the relaxed problems can also be useful to guide the search. When the relaxed problem is solved, a set of actions that is involved in the solution to the relaxed problem is considered 'preferred' among all applicable actions (operators).

The disadvantage of using preferred operators during the search is that the search space becomes incomplete. It is possible that by selecting preferred operators only, all possible goal paths are eliminated during the search. This motivates us to develop a state space reduction method that can not only reduce the search space but also preserves the completeness of the algorithm.

2.4.1 Partial Order Reduction

Partial order reduction (POR) is a family of methods that reduce the explored space by imposing partial orders to the otherwise unrestricted actions on a search path. It

was proposed first in model checking to combat state explosion by only exploring a representative subset of all possible goal paths. Like automated planning problems, problems in model checking are often solved using state space search.

POR is vital for solving model checking problems. For example, to verify the correctness of a parallel program, it is necessary to verify all possible execution paths. Due to the interleaving of executions in current systems, a set of different execution paths can have exactly the same effect on the system and be only a permutation of the same sequence. Thus, an efficient way is to only pick representative execution paths and ignore all the other permutations that are equivalent to the chosen ones [10].

There are many ways to do partial order reduction. Without introducing too many technical details, we make the following two fundamental statements on POR methods.

POR methods can reduce explored space. Previous works in model checking show that by applying POR methods to several program verification reduces the number of states by one or two orders of magnitude. Our observation on the MAZE problem also reflects the fact that POR methods can significantly reduce the number of nodes in search space.

Partial order reduction will preserve completeness. We define a space reduction method R as "preserving completeness" if and only if when it is combined with any general search procedure P, for any goal path g in the explored space of P, it will be either in $P + R$, or some goal path g' acquired by permuting actions in g, is in $P + R$. Intuitively, a space reduction method is complete if it preserves the goal path with respect to path equivalences.

In this thesis, we focus on the partial order reduction method since it can be combined with any existing search methods in planning and can achieve the equal ability of completeness of original search methods. One challenge is that all of these aforementioned POR methods were proposed initially in the model checking community. Although planning problems and model checking problems are very similar, they have different properties that makes the partial order reduction methods different. So far, there are only two space reduction works in planning, namely, Stratified Planning and Expansion Core. However, neither of these two works mentioned partial order

reduction explicitly, despite the fact that they are all POR methods. In this thesis, we will propose a general theory for doing partial order reduction in planning and therefore unify these two works under the POR framework. We also proved a necessary and sufficient condition for any POR methods based on swappable action pairs. This condition is also be used to guide the design of efficient partial order reduction algorithms.

2.5 Conclusion

In this chapter, we briefly introduced classic planning problems and techniques for solving them. We introduced two ways of reducing explored search space: heuristic search and search space reduction. We compared their advantages and disadvantages. We also discussed the advantages of using partial order reduction to reduce the search space. We develop in the next chapter our general theory for doing partial order reduction in planning.

Chapter 3

General Theory for Partial Order Reduction

In this chapter, we propose a complete theory to characterize the partial order reduction methods in planning. One of the basic element of partial order reduction is to swap actions pairs to reduce action paths. Based on SAS+ formulation, our theory offers a necessary and sufficient condition for swapping action pairs during search. We then use category theory to prove that our semi-commutative path pair ensures a completeness and optimality preserving reduction of the search.

We then apply this theory to two existing space reduction methods and show that those two methods based on planning structure analysis are essentially partial order reduction methods. Finally, we summarize the insights derived from the about two approaches and provide two new POR algorithms for planning.

3.1 A Unifying Theory

3.1.1 Semi-commutative Conditions

In this section, we propose our necessary and sufficient condition for action pairs to be semi-commutative. These conditions are the generalization of action commutativity.

First we define commutative action pairs.

Definition 10 *Actions a and b comprise an commutative action pair if and only if for any given state s where a and b are both applicable,* $apply(b, apply(a, s)) = apply(a, (apply(b, s))$.

If two actions are commutative, then if they are adjacent in any path, we can construct an equivalent path simply by swapping the order to those two actions. It is easy to prove that following lemma for commutative action pairs.

Lemma 1 *In SAS+ formalism, If two action o_1 and o_2, if $pre(o_1) \cap pre(o_2) = \emptyset$ and the effect sets $eff(o_1) \cap eff(o_2) = \emptyset$, then they are commutative.*

This lemma gives us a practical standard to find interchangeable actions. In practice, we found the above condition very strong and lead to less reductions. Thus, a weaker condition is demanded. Note that the interchangeable condition is too strong if we only want to reduce the path enumeration. For instance, if we know that path p_1 can be served as a representative of p_2, we can save the enumeration of p_2. It is, however, unnecessary to let p_2 be also the representative of p_1. That is to say, the relation between two paths can be asymmetric. Since asymmetric conditions are weaker and therefore more general than the symmetric ones, we propose a theory on the asymmetric relationship between action pairs.

The theory we develop is for STRIPS tasks $\Sigma = (F, O, I, G)$. This theory can easily be translated to SAS+ tasks, since there is correspondence between SAS+ and STRIPS formalisms of a planning task. For an action o, P_o, A_o, and D_o correspond to the variables assignments in $pre(o)$, $\mathit{eff}(o) \setminus pre(o)$, and $pre(o) \setminus \mathit{eff}(o)$, respectively.

To state our theory, we first define some notations as follows:

- The union of two sets A and B is written as $A + B$.
- The intersection of A and B is written as AB.
- A state s is a subset of the fact set F, and we define $\overline{s} = F \setminus s$ to be the complementarity.

In our deduction, we also use the following rules.

- $A(B+C) = AB + AC$ (distributive law)
- $\overline{AB} = \overline{A} + \overline{B}$ and $\overline{A+B} = \overline{A}\ \overline{B}$ (De Morgan's laws)

3.1.2 Semi-Commutative Action and Path Pairs

The basic structure used in our theory is the concept of semi-commutative action pairs. Intuitively, if for any path, an action sequence (a, b) can be replaced by (b, a), then a and b are semi-commutative.

Definition 11 (Valid Path) *For a STRIPS task Σ and a state s_0, a sequence of actions $p = (o_1, \ldots, o_n)$ is a valid path if, let $s_i = apply(s_{i-1}, o_i), i = 1, \ldots, n$, o_i is applicable at s_{i-1} for $i = 1, \ldots, n$. We also say that applying p to s results in the state s_n.*

Definition 12 *An ordered action pair $(a, b), a, b \in O$ is a state-dependent semi-commutative action pair at state s_0 if when (a, b) is a valid path at s_0, (b, a) is also a valid path that results in the same state. We denote such a relationship by $s_0 : b \Rightarrow a$.*

Definition 13 *An ordered action pair $(a, b), a, b \in O$ is a state-independent semi-commutative action pair (or* ***semi-commutative action pair*** *for short) if (a, b) semi-commutative at any state $s \subseteq F$. We denote this relationship by $b \Rightarrow a$.*

Note the following. 1) Semi-commutativity is not a symmetric relationship. $b \Rightarrow a$ does not imply $a \Rightarrow b$. 2) The order in $b \Rightarrow a$ suggests that we should always try (b, a) only during the search instead of trying both (a, b) and (b, a).

Now we state our necessary and sufficient condition for semi-commutative action pair.

Theorem 1 *(Necessary and sufficient conditions for semi-commutative action pair) An ordered action pair $(a, b), a, b \in O$ is a state-dependent semi-commutative action pair at a state s_0 if and only if $P_a D_b = P_b D_a = P_b \overline{s_0} A_a = A_b D_a = A_a D_b = \emptyset$.*

Proof. First we prove the direction from left to right. Suppose $s_0 : a \Rightarrow b$, we have $P_b\overline{s_0} = \emptyset$ since b is applicable at s_0. Hence, $P_b\overline{s_0}A_a = \emptyset$. Since a is applicable at s_0, and (a, b) is a valid path, we have

$$\begin{aligned} \emptyset &= P_b - (s_0 - D_a + A_a) = P_b\overline{(s_0\overline{D_a} + A_a)} \\ &= P_b(\overline{s_0} + D_a)\overline{A_a} = P_b\overline{s_0}\ \overline{A_a} + P_bD_a\overline{A_a}, \end{aligned}$$

which implies $P_bD_a\overline{A_a} = \emptyset$. Note that $D_a\overline{A_a} = D_a$, thus we have $P_bD_a = \emptyset$. Similarly, since (b, a) is also a valid path at s_0, we have $P_a - (s_0 - D_b + A_b) = \emptyset$, from which we can derive $P_aD_b = \emptyset$.

Finally, we consider the two states s_1 and s_2, resulted from applying (a, b) and (b, a) to s_0, respectively:

$$\begin{aligned} s_1 &= (s_0 - D_a + A_a) - D_b + A_b \\ s_2 &= (s_0 - D_b + A_b) - D_a + A_a. \end{aligned}$$

Using $A - B = A\overline{B}$, we get:

$$\begin{aligned} s_1 &= (s_0 + A_a + A_b)(\overline{D_a} + A_a + A_b)(A_b + \overline{D_b}) \\ s_2 &= (s_0 + A_a + A_b)(\overline{D_b} + A_a + A_b)(A_a + \overline{D_a}) \end{aligned}$$

Let $T = (s_0 + A_a + A_b)$, we can simplify s_1 to:

$$\begin{aligned} s_1 &= T(\overline{D_a}A_b + \overline{D_a}\ \overline{D_b} + A_aA_b + A_a\overline{D_b} + A_b + A_b\overline{D_b}) \\ &= T\overline{D_a}\ \overline{D_b} + TA_a\overline{D_b} + TA_b \\ &= s_0\overline{D_a}\ \overline{D_b} + A_a\overline{D_b} + A_b \end{aligned}$$

Similarly, $s_2 = s_0\overline{D_b}\ \overline{D_a} + A_b\overline{D_a} + A_a$. We know that s_1 is identical to s_2 if and only if $s_1 - s_2 = \emptyset$ and $s_2 - s_1 = \emptyset$. We denote $s_0\overline{D_b}\ \overline{D_a}$ by K and have:

$$\begin{aligned} s_1 - s_2 &= (K + A_a\overline{D_b} + A_b)\overline{(K + A_b\overline{D_a} + A_a)} \\ &= A_b\overline{K}D_a = (\overline{s}_0 + D_a + D_b)A_bD_a \\ &= \overline{s}_0A_bD_a + A_bD_a + D_bA_bD_a = A_bD_a \end{aligned}$$

Therefore, we can see that the necessary and sufficient condition for $s_1 - s_2 = \emptyset$ is that $A_b D_a = \emptyset$. Symmetrically, $s_2 - s_1 = \emptyset$ if and only if $A_a D_b = \emptyset$.

Now we prove the second part. Suppose we have $P_a D_b = P_b D_a = P_b \overline{s_0} A_a = A_b D_a = A_a D_b = \emptyset$, assume (a, b) is a valid path at s_0, we prove that (b, a) is a valid path and leads to the same state.

Since (a, b) is a valid path at s_0, we have $P_b \overline{s_0}\ \overline{A_a} = \emptyset$. Also, we have assumed that $P_b \overline{s_0} A_a = \emptyset$. Hence we have $P_b \overline{s_0} = P_b \overline{s_0}\,\overline{A_a} + P_b \overline{s_0} A_a = \emptyset$ and b is applicable at s_0. Further, we know that a is applicable after the execution of b since we have $P_a - (s_0 - D_b + A_b) = P_a \overline{s_0}\ \overline{A_b} + P_a D_b = \emptyset + \emptyset = \emptyset$. Thus, (b, a) is a valid path at s_0.

Last, since $s_1 - s_2 = A_b D_a = \emptyset$ and $s_2 - s_1 = A_a D_b = \emptyset$, (a, b) and (b, a) lead to the same state. ■

Corollary 1 *An ordered action pair $(a, b), a, b \in O$ is a semi-commutative action pair if and only if $P_a D_b = P_b D_a = P_b A_a = A_b D_a = A_a D_b = \emptyset$.*

Definition 14 *For a path $p = (a_1, \cdots, a_n)$, another path $q = (b_1, \cdots, b_n)$ is* ***1-swap away*** *from p if there exists $i, 2 \leq i \leq n$ such that $b_i = a_{i-1}$, $b_{i-1} = a_i$, and $a_j = b_j$ for any $j, j \notin \{i-1, i\}, 1 \leq j \leq n$.*

Definition 15 (1-Swap Semi-Commutative Path Pair) *For a STRIPS planning task $\Sigma = (F, O, I, G)$, for two paths p and q that are 1-swap away, consider the action pair (a, b) that is swapped, p and q are 1-swap semi-commutative if $b \Rightarrow a$.*

Definition 16 (Semi-Commutative Path Pair) *Two paths p and q are semi-commutative if $p = q$ or if there exists a sequence of paths $p_1 = p, p_2, \cdots, p_k = q$ such that p_{j-1} is 1-swap semi-commutative with p_j for $j = 2, \cdots, k$. We denote the relation as $q \Rightarrow p$.*

Definition 17 (Commutative Path Pair) *Two paths p and q are commutative if $p \Rightarrow q$ and $q \Rightarrow p$. We denote this relation by $p \Leftrightarrow q$.*

A path p is **swappable** from a path q if p can be converted to q through a number of 1-swaps.

Intuitively, if $q \Rightarrow p$, then q leads to the same state as p does and contains the same set of actions. Hence, a search can explore q only instead of both q and p without sacrificing completeness or optimality. Here, we assume the optimality metric is to minimize the total action cost, where each action has a positive cost.

3.1.3 Categories of search and reduction

In the following, we use category theory to describe partial order based reduction. See for example [1] for introduction to category theory. Category is a relatively new alternative to set as the foundational notion of mathematics, a representation upon which logical constructions can be codified.

A category is a directed graph whose vertices (called objects) and arrows (called morphisms) satisfy certain additional requirement. In a category, each object A has an identity morphism $1_A : A \to A$, and each pair of morphisms $f : A \to B$ and $g : B \to C$ is assigned another morphism $g \circ f : A \to C$ as the composition of morphisms. All the morphisms should satisfy the identity laws (for $g : A \to B$, $g \circ 1_A = g$, $1_B \circ g = g$) and associative laws ($f \circ (g \circ h) = (f \circ g) \circ h$).

Definition 18 (Category of Paths) For a STRIPS planning task $\Sigma = (F, O, I, G)$, the category of paths $\mathbf{pth}_\Sigma$ defines the following data:

- The objects include all finite-length paths whose elements are in O;
- There is a morphism from a path p to a path q if and only if $q \Rightarrow p$.

We see that $\mathbf{pth}_\Sigma$ is indeed a category since it satisfies the identity laws and associative laws.

A subcategory of a category C is a category S whose objects are objects in C and whose morphisms are morphisms in C with the same identities and composition of morphisms.

Definition 19 (Representative Subcategory) *For a category C, a subcategory S of C is representative if for each C-object B there exists a S-object A_B and a C-morphism $r_B : B \rightarrow A_B$.*

Definition 20 (Category of Searches) For a STRIPS planning task $\Sigma = (F, O, I, G)$, the category of searches $\mathbf{sch}_\Sigma$ defines the following data:

- The objects include all subcategories of $\mathbf{pth}_\Sigma$;
- There is a morphism between $A \rightarrow B$ if and only if B is a representative subcategory of A.

We can verify that $\mathbf{sch}_\Sigma$ is a category since each object has an identity morphism and the morphisms satisfy the identity and associative laws.

Definition 21 (Search) For a STRIPS task Σ, a search is an object in $\mathbf{sch}_\Sigma$. A search A is a **goal search** if every path in A leads to a goal state from I. The optimal paths in a goal search form an **optimal search**.

The above abstract definition defines a search on a planning task as a set of paths with structures within the set, represented by the morphisms between paths.

Definition 22 (Action-Preserving Reduction) For a search A of a STRIPS planning task $\Sigma = (F, O, I, G)$, an action preserving reduction is another search B such that there exists a morphism $A \rightarrow B$ in $\mathbf{sch}_\Sigma$.

Intuitively, a search B is a category of paths. If there is a morphism $B \rightarrow S$ in $\mathbf{sch}_\Sigma$, then S is a representative subcategory of B. That is, for each object (path) p in B, there exists a path q in S, such that there is a B-morphism $p \rightarrow q$. However, a B-morphism $p \rightarrow q$ implies that p and q form a semi-commutative path pair, which means that q is a valid path if p is and they lead to the same state. Hence, a representative subcategory of a search B represents a completeness and optimality preserving reduction of the search.

In summary, we have proposed our central result in Theorem 1 and Corollary 1. They give necessary and sufficient conditions for both action dependent and independent semi-commutative action pairs. Using category theory, we have also proved that our condition ensures a completeness and optimality preserving reduction of the search.

3.2 Interpretations of POR Algorithms

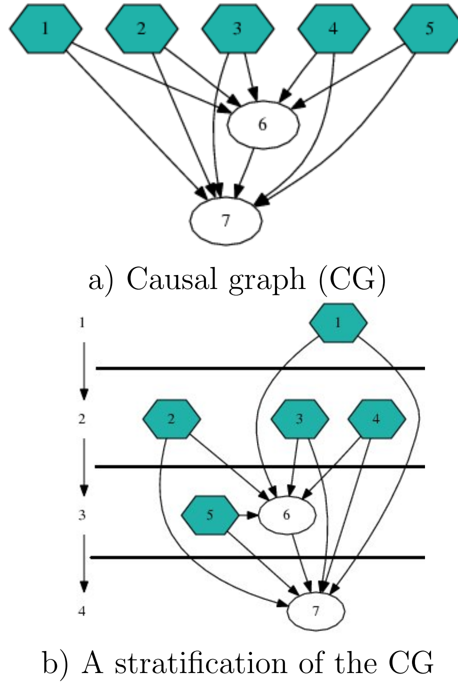
In this section, we interpreted two previous POR algorithms, stratified planning (SP) and expansion core in the above theoretical framework. Even these two algorithms look different from the surface, we reveal that semi-commutativity of action pair is the central property in both algorithms.

3.2.1 Stratified planning (SP)

We summarize the key idea of the SP algorithm. A more formal treatment of SP can be found in [5]. SP is based on the SAS+ formalism. For a SAS+ planning task, for an action $o \in O$, define:

- 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 $eff(o)$.

Definition 23 *Given a SAS+ planning task* Π *with state variable set* X*, its* ***causal graph (CG)*** *is a directed graph* $CG(\Pi) = (X, E)$ *with* X *as the vertex set. There is an edge* $(x, x') \in E$ *if and only if* $x \neq x'$ *and there exists an action* o *such that* $x \in trans(o)$ *and* $x' \in dep(o)$*, or,* $x \in aff(o)$ *and* $x' \in trans(o)$*.*


a) Causal graph (CG)

b) A stratification of the CG

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

SP uses a **stratification** of the CG. A stratification of $CG(\Pi) = (X, E)$ is a partition of the set X: $X = (X_1, \cdots, X_k)$ 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 $trans(o)$. Note that all state variables in a same $trans(o)$ must be in the same level. We show an example of stratification in Figure 3.1 where the right part is the causal graph and the left part is the stratification.

Definition 24 (Follow-up Action) *For a SAS+ task* Π*, an action* b *is a follow-up action of* a *(denoted as* $a \rhd b$*) if* $aff(a) \cap dep(b) \neq \emptyset$ *or* $aff(a) \cap aff(b) \neq \emptyset$*.*

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

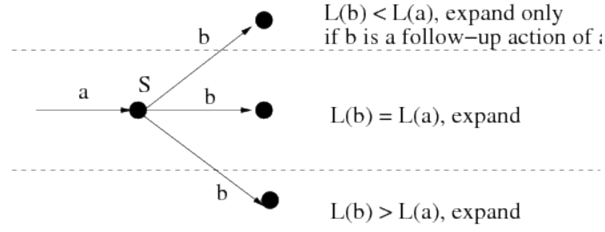


Figure 3.2: Stratified Planning Strategy

that leads to s. Then, for each applicable action b at state S, SP makes the following decision:

- If $L(b) < L(a)$ and b is not a follow-up action of a, then do not expand b (we say that b is not **SP expandable** after a). Otherwise, expand b.

Figure 3.2 illustrates the strategy of stratified planning algorithm.

Now we interpret SP in our framework. For a SAS+ task Π, consider its equivalent STRIPS task Σ. Each search algorithm corresponds to a set of paths it explores, which corresponds to an object A in $\mathbf{sch}_\Sigma$. Consider the set of paths that will be examined when the search is combined with SP. Let the SP-reduced path set correspond to an object A_{SP} in $\mathbf{sch}_\Sigma$.

Lemma 2 *If an action b is not SP-expandable after a, then $b \Rightarrow a$.*

Proof. If b is not SP-expandable after a, then $L(a) > L(b)$ and b is not a follow-up action of a. Since b is not a follow-up action of a, we know that $\mathit{aff}(a) \cap \mathit{dep}(b) = \mathit{aff}(a) \cap \mathit{aff}(b) = \emptyset$. Therefore, $P_b A_a = \emptyset$ and $D_b A_a = \emptyset$. Also, we see that $P_b D_a = \emptyset$ because b is applicable immediately after a is executed. Moreover, since $L(a) > L(b)$ implies that there is no edge from a state variable associated with a to a state variable associated with b, from Definition 23, we can show that $P_a D_b = D_a A_b = \emptyset$. Thus, we proved that $P_a D_b = P_b D_a = P_b A_a = A_b D_a = A_a D_b = \emptyset$. According to Corollary 1, we have $b \Rightarrow a$. ■

Theorem 2 *For any search A in **sch**$_\Sigma$, its SP-reduced search A_{SP} is a representative subcategory of A.*

Proof. We need to show that A_{SP} is a representative subcategory of A. That is, for each path p in A, we show that there is a path p_{SP} in A_{SP} such that there is a morphism $p \rightarrow p_{SP}$ in $\mathbf{pth}_\Sigma$. We prove this by induction on n, the length of p.

The case is true when $n = 1$ since any action is a follow-up action of no-op. Now we assume for any path p with length no more than k in p, the proposition is true. We prove the case where $n = k + 1$.

For a path $p^0 = (a_1, \ldots, a_{k+1})$, consider the prefix $p_1 = (a_1, \ldots, a_k)$. By the induction hypothesis, there is a path $p_2 = (a_1^1, \ldots, a_k^1)$ such that $p_1 \rightarrow p_2$ is a morphism in $\mathbf{pth}_\Sigma$.

Now we consider a new path $p^1 = (a_1^1, \ldots, a_k^1, a_{k+1})$. If a_{k+1} is SP-expandable after a_k^1, then $p^0 \rightarrow p^1$ is a morphism in $\mathbf{pth}_\Sigma$.

If a_{k+1} is not SP-expandable after a_k^1, consider a new path $p^2 = (a_1^1, \ldots, a_{k-1}^1, a_{k+1}, a_k^1)$. From Lemma 1, we know that $a_{k+1} \Rightarrow a_k^1$, which implies that p^2 is a valid path leading to the same state as p^1 does.

Let $p_3 = (a_1^1, \ldots, a_{k-1}^1, a_{k+1})$, we know that there is a path $p_4 = (a_1^2, \ldots, a_k^2)$ such that $p_3 \rightarrow p_4$ is a morphism in $\mathbf{pth}_\Sigma$. Define $p^3 = (a_1^2, \ldots, a_k^2, a_k^1)$.

Comparing p^2 and p^3, we know that $L(a_{k+1}) > L(a_k^1)$, namely, the level of the last action in p^2 is strictly larger than that in p^3. We can repeat the above process to generate p^4, $p^5, \cdots$, as long as $p^0 \rightarrow p^j$ is not a morphism in $\mathbf{pth}_\Sigma$.

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 and yield a path p^j such that $p^0 \rightarrow p^j$ is a morphism in $\mathbf{pth}_\Sigma$. ■

The above proof explains SP as a reduction that reduces each search in $\mathbf{sch}_\Sigma$ to a representative subcategory. Hence, SP preserves completeness and optimality since any path explored by a search can be mapped to an equivalent path explored under SP-reduction.

3.2.2 Expansion core (EC) algorithm

We give a short outline of EC first. For detailed description of the EC algorithm, refer to [3].

For a SAS+ task, each state variable $X_i, i = 1, \cdots, N$ is associated with a **domain transition graph (DTG)** G_i, a directed graph with vertex set $V(G_i) = Dom(x_i)$ and edge set $E(G_i)$. An edge (v_i, v_i') belongs to $E(G_i)$ if there is an action o with $v_i \in pre(o)$ and $v_i' \in eff(o)$ in which case we say that o is associated with the edge $e_i = (v_i, v_i')$ (denoted as $o \vdash e_i$).

Definition 25 *An action o is* ***associated*** *with a DTG G_i (denoted as $o \vdash G_i$) if o is associated with any edge in G_i.*

Definition 26 *For a SAS+ task, for each DTG $G_i, i = 1, \ldots, N$, for a vertex $v \in V(G_i)$, an edge $e \in E(G_i)$ is a* ***potential descendant edge*** *of v (denoted as $v \lhd e$) if 1) G_i is goal-related and there exists a path from v to the goal state in G_i that contains e; or 2) G_i is not goal-related and e is reachable from v. A vertex $w \in V(G_i)$ is a* ***potential descendant vertex*** *of v (denoted as $v \lhd w$) if 1) G_i is goal-related and there exists a path from v to the goal state in G_i that contains w; or 2) G_i is not goal-related and w is reachable from v.*

Definition 27 *For a SAS+ task, given a state $s = (s_1, \cdots, s_N)$, for any $1 \leq i, j \leq N, i \neq j$, s_i is a* ***potential precondition*** *of the DTG G_j if there exist $o \in O$ and $e_j \in E(G_j)$ such that*

$$s_j \lhd e_j, \;\; o \vdash e_j, \quad \textit{and } s_i \in pre(o) \tag{3.1}$$

Definition 28 *Given a SAS+ state $s = (s_1, \ldots, s_N)$, for any $1 \leq i \neq j \leq N$, s_i is a* ***potential dependent*** *of the DTG G_j if there exist $o \in O$, $e_i = (s_i, s_i') \in E(G_i)$ and $w_j \in V(G_j)$ such that*

$$s_j \lhd w_j, o \vdash e_j, w_i \in pre(o)$$

Definition 29 *For a state s, the potential dependency graph PDG(s) is the directed graph with DTGs as vertices and there is an edge from G_i to G_j if and only if s_i is a potential precondition or potential dependent of G_j.*

Definition 30 *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 $(v, w) \in E(H)$.*

At a state s, EC method only expands actions in those DTGs within such a dependency closure of the PDG(s) that contains at least one DTG with an unarchived goal.

For any state s, an action a is goal-relevant if there exists a path from s to a goal state that contains a.

Lemma 3 *For a state s and a dependency closure C of $PDG(s)$, for any goal-relevant action a associated with a DTG in $PDG(s) \setminus C$, and any action b associated with a DTG in C that is applicable at s, we have $b \Rightarrow a$.*

Proof. Since b is applicable at s, we know $P_b \subseteq s$. Since b is associated with a DTG within C, no fact in P_b is a potential precondition of a and we have $P_bP_a = \emptyset$, which leads to $P_bD_a = \emptyset$ since $D_a \subseteq P_a$. On the other hand, since DTGs in C are not a potential dependent of those not in C, a precondition of b is not affected by a and we have $P_bD_a = \emptyset$ and $P_bA_a = \emptyset$. Finally, since a and b do not associate with a same DTG, we have $A_bD_a = A_aD_b = \emptyset$. All the five conditions in Corollary 1 are met. ■

To ensure action-preserving reduction, we give a list of conditions that is similar to the idea in stubborn set [21], a well-known technique for search space reduction in model checking.

Definition 31 (Stubborn Set) *For a planning task, a set of actions $T(s)$ is a stubborn set at a state s if*

- *A1 For any action $b \in T(s)$ and actions $b_1, \cdots, b_k \notin T(s)$, if $(b_1, \cdots, b_k, b)$ is a prefix of a path from s to a goal state, then $(b, b_1, \cdots, b_k)$ is a valid path from s and leads to the same state as $(b_1, \cdots, b_k, b)$ does.*

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

A valid path $(a_1, \cdots, a_n)$ is **stubborn-set conforming** at a state s_1 if $a_i \in T(s_i)$ for $i = 1, \cdots n$ where $s_{i+1} = apply(s_i, a_i)$. For any search A in $\mathbf{sch}_\Sigma$, the stubborn-set reduced search of A, A_{SS}, is the subset of A that includes all stubborn-set conforming paths.

Theorem 3 *For any goal search A in $\mathbf{sch}_\Sigma$, there is a morphism $A \rightarrow A_{SS}$ in $\mathbf{sch}_\Sigma$.*

Proof. We sketch the main idea. The proof is essentially the same as the proof to the stubborn set method in model checking [21], which is based on an induction on the length of paths. For any state s, for each path $p = (a_1, \cdots, a_n)$ from s to goal, according to A2 in Definition 31, we know that there must exist an action $a_i, 1 \leq i \leq n$ such that $a_i \in T(s)$. Then, according to A1, we can permute p into a path $q = (a_i, a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$ that also reaches the goal. Using induction, we can prove that any path p can be permuted to a path q such that $q \Rightarrow p$ and A_{SS} is a representative subcategory of A. ■

Lemma 4 *The actions that the EC algorithm expands at any state s form a stubborn set $T(s)$.*

Proof. For a state s, let the dependency closure chosen by EC be $C \in PDG(s)$. For any action b expanded by EC, and actions $b_1, \cdots, b_k$ that do not associate with a DTG in C, if $(b_1, \cdots, b_k, b)$ is a prefix of a path to goal, then we know $b \Rightarrow b_i$ for $i = 1, \cdots, k$ from Lemma 3. Therefore, $(b, b_1, \cdots, b_k)$ is also a valid path and A1 in Definition 31 is proved. Moreover, since C includes at least one DTG G that with an unarchived goal, some action in G must be used in any path to goal. Since G is in the closure, all actions in G are expanded and A2 in Definition 31 is shown. ■

From Theorem 3 and Lemma 4, we can prove that EC is an action-preserving reduction.

Theorem 4 *For any goal search A in $\mathbf{sch}_\Sigma$, its EC-reduced search A_{EC} is a representative subcategory of A.*

We see that EC is stronger than SP in the sense that it is an action preserving reduction for goal searches only, not any search. By restricting the reduction to goal searches only, EC does not guarantee that a non-goal path will be mapped to a EC-conforming path, which is a good thing that helps avoid exploring useless paths.

3.3 New POR Algorithms for Planning

From the last section, we see that, in essence, both SP and EC detect and exploit the semi-commutativity of actions. We have the following insights.

- Both SP and EC are based on the SAS+ formalism, and utilize the notion of DTGs (state variables). However, the use of DTGs is not essential for POR reduction. They are used only to ensure certain conditions in Theorem 1. Further, both SP and EC give *sufficient but not necessary* conditions for finding semi-commutative action pairs. For example, for domains such as pipesworld where the CG has only one strongly-connected component, SP and EC can give no reduction, although semi-commutative action pairs still exist in these domains.
- Both SP and EC have certain advantages. For a state s and applicable actions $a_1, \cdots, a_n$, SP will expand each of $a_i, i = 1, \cdots, n$ and for each a_i may prune some actions b_i if $b_i \Rightarrow a_i$. EC has the advantage of not having to expand all a_i. Instead, it divides actions into those in $T(s)$ and those not in $T(s)$. As a cost, it needs to ensure that *any action in $T(s)$ must be semi-commutative with any action not in $T(s)$* (i.e. a closure), which may miss certain semi-commutativity and chance of reduction.

Based on the above two observations, we propose our new algorithm, **action closure (AC) reduction.** Unlike SP and EC, the AC algorithm does not analyze the CG and use DTG as the basic unit of decision (whether to be expanded or not). Instead, it treats each action as the basic unit of decision.

Definition 32 (Action Dependency Graph) *For a STRIPS planning task, its action dependency graph (ADG) is defined as a directed graph in which each vertex is an action, and there is an edge from action a to b if and only if* $P_aD_b \neq \emptyset$ *or* $P_bD_a \neq \emptyset$ *or* $P_bA_a \neq \emptyset$ *or* $A_bD_a \neq \emptyset$ *or* $A_aD_b \neq \emptyset$.

Definition 33 (Contracted ADG) *Given an ADG, its contracted ADG (CADG) is a graph where each vertex is a maximum strongly connected component (SCC) of the ADG and there is an edge between two SCCs if there is an edge in the ADG from a vertex in one SCC to a vertex in another SCC.*

A topological sort on the CADG generates an ordered sequence of its vertices: (SCC_1, $\cdots$, SCC_N), where SCC_1 is the SCC with zero in-degree in the CADG. The topological sort is not unique and we currently choose one randomly. Given a topological sort of the CADG, each action a is assigned a **layer** $l(a)$, which is the index of the SCC the action belongs to, i.e. $a \in \mathrm{SCC}_{l(a)}$.

Definition 34 *An action b is supported by an action a if and only if* $P_bA_a \neq \emptyset$.

The **AC algorithm** works as follows. For each state s, let the action that leads to s be a,

- B1 If A_a includes a goal fact, it expands all applicable actions;
- B2 Otherwise, it finds the minimum index $M, M \leq N$, such that $\mathrm{SCC}_1 \cup \cdots \cup \mathrm{SCC}_M$ include all the applicable actions that are supported by a.

Lemma 5 *For any two actions a and b such that* $l(b) < l(a)$, *we have* $b \Rightarrow a$.

Proof. If $l(b) < l(a)$, there is no edge from a to b in the ADG. Thus, $P_aD_b = P_bD_a = P_bA_a = A_bD_a = A_aD_b = \emptyset$. The conditions in Corollary 1 are met. ■

A path is **AC-conforming** if it can be possibly generated by a search with the AC algorithm. For any search A, the AC-reduced search is the subcategory of A that includes all AC-conforming paths as objects.

Definition 35 (Optimality Stubborn Set) *For a planning task, a set of actions $T(s)$ is an optimality stubborn set at a state s if*

O1 For any action $b \in T(s)$, and actions $b_1, \cdots, b_k \notin T(s)$, if $(b_1, \cdots, b_k, b)$ is a prefix of a path to goal, then $(b, b_1, \cdots, b_k)$ is a valid path from s that leads to the same state as $(b_1, \cdots, b_k, b)$.

O2 Any optimal *path from s to a goal contains at least one action in $T(s)$.*

Theorem 5 *For any optimal search A in $\boldsymbol{sch}_\Sigma$, there is a morphism $A \to A_{OSS}$ in $\boldsymbol{sch}_\Sigma$, where A_{OSS} is the subcategory of A that conforms to optimality stubborn sets.*

Theorem 5 can be proved using a proof parallel to that of Theorem 3.

Lemma 6 *The actions that the AC algorithm expands at any state s form an optimality stubborn set $T(s)$.*

Proof. For a state s, assume AC expands applicable actions in $T = \{\text{SCC}_1 \cup \cdots \cup \text{SCC}_M\}$. Consider any actions $b_1, \cdots, b_k$ that are not in T. If $(b_1, \cdots, b_k, b)$ is the prefix of an optimal path, then we know $b \Rightarrow b_i$ for $i = 1, \cdots, k$ from Lemma 5. Therefore, $(b, b_1, \cdots, b_k)$ is also a valid path and O1 is proved. Moreover, if no action is used in T, since T includes all the actions supported by a, we can delete a to obtain a better plan (unless a adds a goal which is covered by condition B1 in the AC algorithm). Hence, any optimal path must include at least one action in T and O2 is satisfied. ■

From Theorem 5 and Lemma 6, we have shown that AC is an action preserving reduction.

Theorem 6 *For any optimal search A in $\boldsymbol{sch}_\Sigma$, its AC-reduced search A_{AC} is a representative subcategory of A.*

The AC algorithm is a stubborn set method. Last, we propose an enhanced version of the AC algorithm that adds the idea of stratified planning. For each state s with

a leading action a, the **AC$^+$ algorithm** applies the same conditions B1 and B2 as used by AC, while imposing one more restriction.

B3 For each applicable action b, if $b \Rightarrow a$, then it does not expand b.

This condition B3 can be viewed as a SP style reduction, added to the stubborn set reduction in the AC algorithm. The correctness of AC$^+$ is obvious as it simply checks the semi-commutativity of two adjacent actions. For any path p there is a path q that conforms to B3. Hence, any optimal search A can be reduced to a search A_{AC} and then to A_{AC^+}, all action-preserving.

Theorem 7 *For any optimal search A in $\boldsymbol{sch}_\Sigma$, its AC$^+$-reduced search A_{AC^+} is a representative subcategory of A.*

3.4 Conclusion

In this chapter, we have proposed our necessary and sufficient conditions for semi-commutative action pairs. The asymmetric relationship between actions is essential for a better partial order reduction. Based on our theory, we have also successfully explained the Stratified Planning and the Expansion Core algorithm under this unified framework. Finally, inspired by Stratified Planning and based on our theory, we have also proposed two new partial order reduction algorithms. We will test the performance of both algorithms in the next section.

Chapter 4

Experimental results

We test on STRIPS problems in the recent International Planning Competitions (IPCs): IPC3, IPC4, and IPC5. We implemented our algorithm in the Fast Downward (FD) planner [12]. We only modified the state expansion part.

Table 4.1 shows the results of FD, SP, AC, and AC^+ on the testing domains except for pipesworld and freecell domains, whose results are shown in Table 4.2. All algorithms give the same solution quality. We see that the performance of the original SP is consistently better than the original FD. AC^+ can significantly improve SP in driverlog and tpp domains in terms of the numbers of generated and expanded nodes. AC is generally better than FD in terms of both generated and expanded nodes.

Comparing AC against SP, we see that typically AC generates more states but expands less, since AC is a stubborn set style reduction which tends to expand less nodes. Due to a deferred heuristic evaluation scheme in FD, the number of heuristic evaluations is determined by the number of expanded nodes. As a result, the CPU time of AC often is less than that of SP, even if AC generates more nodes. The trucks, storage, and tpp domains best illustrate this point. AC^+ has a similar comparison against SP and is faster than SP in most instances except for the trucks domain.

Comparing AC^+ against AC, we see that AC^+ is better in driverslog, depot, storage, and tpp=. AC^+ is orders of magnitude better than AC for some instances from driverslog and depot. AC^+ is faster than the original FD in most instances except for the trucks domain.

Now we turn to the surprising results on the pipesworld and freecell domains in Table 4.2. These two domains are deemed very difficult since their CG is densely connected and cannot be decomposed into multiple strongly connected components. Therefore, SP, EC and AC all fail to give any reduction. Surprisingly, AC^+ can give significant reduction. In Table 4.1, we compare FD and AC^+. We did not report SP and AC since they cannot give any reduction and their state expansions are the same as FD. We see that AC^+ can reduce the number of expanded and generated nodes by orders of magnitude for many instances such as freecell-15 and pipesworld-12. It is encouraging that POR algorithms can work not only for those largely decomposable domains but also those domains whose state variables are highly inter-depended.

Domains	Fast Downward			Stratified Planning			AC			AC$^+$		
	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time	Expanded	Generated	Time
driverlog11	280	2858	0.07	215	998	0.04	254	4240	0.07	173	1842	0.06
driverlog12	1810	21582	0.11	2380	8719	0.26	1150	18808	0.22	326	3484	0.13
driverlog13	599	7155	0.18	402	2126	0.09	635	11634	0.2	324	3984	0.15
driverlog14	527	6173	0.18	370	1723	0.1	555	12568	0.21	271	3136	0.11
driverlog15	1288	18823	0.45	972	6202	0.35	2393	74680	1.23	383	5928	0.19
driverlog16	439226	8831575	105.71	192324	1400388	70.93	379769	12644130	72.91	-	-	-
driverlog17	9211	303992	5.44	5438	63710	4.48	3765	190376	2.57	-	-	-
driverlog18	13524	353873	17.2	21620	163436	24.94	38682	1704738	37.2	2867	41254	2.96
truck6	339	5071	0.07	339	2506	0.09	256	6414	0.08	296	5454	0.23
truck7	38532	165934	2.23	38532	81317	2.43	39782	180080	1.13	220179	1194652	7.28
truck8	1966	11558	0.37	1970	5749	0.2	537	7578	0.25	123793	770216	6.01
truck9	236058	2023106	17.29	236058	1002496	28.34	19809	102328	1.9	2584060	15899640	241.84
truck10	325002	3064955	29.82	325002	1519147	47.03	215737	1039650	16.58	468971	2091052	33.98
truck11	99902	1542311	10.45	99902	766989	16.85	77034	449580	6.5	422792	2518740	35.2
depot1	23	91	0.01	18	28	0	44	392	0	-	-	-
depot2	65	485	0.02	84	173	0.01	85	1022	0.05	-	-	-
depot3	6121	48336	1.19	819	2097	0.17	7277	75322	1.62	1762	11454	0.19
depot4	9291	78046	2.64	10366	25840	2.92	8004	87924	1.94	4786	30894	0.76
depot5	343364	2884118	103.36	30538	72874	13.98	22871	241254	6.29	14401	93618	4.07
depot7	28204	261112	3.83	40997	109669	8.45	18740	217698	2.83	8000	55848	0.85
depot8	162784	1674483	55.54	108157	349662	45.67	569532	8150860	137.4	33672	290238	7.9
depot9	-	-	-	165286	572968	226.26	192483	2877660	138.3	28454	216286	19.7
depot10	51542	726134	14.06	35314	121058	13.78	31735	552866	6.79	482	5162	0.18
depot11	215106	3316774	198.47	119489	456139	135.1	205746	3677638	122.57	1577	18828	1.23
depot13	257	4045	0.14	179	730	0.17	265	5480	0.46	5943	62406	2.23
depot14	-	-	-	-	-	-	274978	2832700	217.26	-	-	-
depot15	-	-	-	-	-	-	46357	478958	106.83	-	-	-
depot16	-	-	-	393419	2176390	257.24	653345	15021200	249.81	65774	667106	21.28
storage1	4	7	0	-	-	-	4	14	0	4	14	0
storage2	4	9	0	4	3	0	4	18	0	4	18	0
storage3	4	11	0	4	5	0	4	22	0.01	4	22	0.02
storage4	32	85	0.02	35	34	0	32	170	0.02	32	170	0.02
storage5	20	94	0.01	21	43	0.02	20	188	0.03	19	130	0.02
storage6	31	176	0.04	30	75	0.02	31	352	0.04	31	260	0.05
storage7	235	634	0.02	240	248	0.04	233	1250	0.05	227	1216	0.05
storage8	95	480	0.07	90	200	0.04	109	1064	0.08	189	1108	0.09
storage9	93	744	0.04	91	355	0.04	93	1488	0.1	159	1342	0.12
storage10	1521	4364	0.23	1372	1479	0.2	1494	8620	0.18	1516	8700	0.18
storage11	297	1774	0.18	326	893	0.1	293	3566	0.1	2066	11398	0.68
storage12	1496	11550	0.77	357	1511	0.15	1323	20094	0.55	1752	10583	0.31
storage13	4930	17046	1.31	5697	7565	1.01	5414	37662	0.71	8160	55656	1.33
storage14	2668	18730	1.11	2459	8029	0.71	2837	39492	0.86	1263	8164	0.55
storage15	325	2673	0.36	355	1267	0.2	308	5048	0.38	2583	20806	0.52
storage16	276	3385	0.64	273	1591	0.27	289	1542	0.23	259	2974	0.34
tpp1	6	8	0	6	3	0	6	16	0	6	16	0
tpp2	9	17	0	9	6	0	11	34	0	11	34	0
tpp3	12	29	0	12	12	0	16	54	0	16	54	0
tpp4	15	44	0	15	18	0	19	72	0	22	78	0
tpp5	22	92	0	22	33	0	88	452	0	122	454	0.01
tpp6	664	3641	0.06	617	1229	0.04	261	1882	0.03	94	444	0.02
tpp7	1591	9403	0.17	2199	5840	0.14	1250	8394	0.03	431	2410	0.04
tpp8	4685	37683	0.29	4181	13904	0.31	932	6670	0.07	486	2642	0.05
tpp9	3630	25924	0.59	4044	9734	0.39	1675	13262	0.18	1177	7460	0.04
tpp10	12242	110251	1.66	9634	32313	1.07	6685	63868	0.22	2339	14382	0.27
tpp11	13148	126912	2.08	24193	95847	3.82	5973	51660	1.11	1621	9614	0.26
tpp12	36690	364082	4.48	23754	71293	3.51	18366	154012	1.41	5195	47802	0.87
tpp13	24066	295068	4.35	32150	156888	8.34	26175	412436	3.44	16375	292526	2.07
tpp14	68494	894865	14.74	52963	209880	19.1	42991	643584	6.61	16982	175226	2.23
tpp15	37145	477808	8.71	54522	252323	20.85	28275	403406	4.54	16506	171188	2.23

Table 4.1: Comparison of several algorithms. We give number of generated nodes, number of expanded nodes, and CPU time in seconds. ”-” means timeout after 300 seconds.

Domains	Fast Downward			AC^+		
	Expanded	Generated	Time	Expanded	Generated	Time
free1	32	190	0.07	30	322	0.16
free2	42	262	0.06	56	108	0.58
free3	53	397	0.28	72	693	0.36
free4	116	533	0.24	108	542	0.16
free5	796	4191	1.67	1808	14858	3.44
free6	390	2825	1.68	475	4608	3.1
free7	535	3281	2.42	534	4246	3.34
free8	2379	10110	10.72	532	6818	2.79
free9	5754	53638	23.83	428	4218	2.23
free10	2052	14510	15.69	902	11410	12.88
free11	2406	9001	16.7	721	6961	11.83
free12	1362	8013	9.52	634	6396	4.21
free13	12083	77311	138.41	12849	125532	111.94
free14	4431	40529	46.83	605	7558	6.13
free15	35329	307397	463.72	2841	32298	48.39
free16	-	-	-	11757	146286	171.75
free17	657	4870	12.3	330	3104	7.06
pipe1	23	115	0.01	18	138	0.04
pipe2	158	709	0.02	139	870	0.06
pipe3	184	2666	0.31	70	1026	0.3
pipe4	202	2712	0.1	139	2420	0.34
pipe5	47	701	0.23	40	906	0.36
pipe6	64	930	0.13	68	1290	0.14
pipe7	358	15371	1.69	522	24219	1.38
pipe8	1781	70706	3.33	760	25576	1.64
pipe9	1373	43171	4.23	1478	64516	3.51
pipe10	476729	13794006	1499.7	646	25811	2.95
pipe11	303622	1493750	235.9	1706	9042	1.13
pipe12	228593	1783627	350.94	680	22778	3.01
pipe13	62797	555162	117.6	474	6265	2.19
pipe14	177670	972962	193.52	93385	452848	80.24
pipe15	260672	1306367	254.71	27651	130042	20.17
pipe18	7807	98160	46.31	6547	96915	31.16
pipe19	218224	2290321	396.64	5344	44078	34.35

Table 4.2: Comparison of FD and AC^+ on freecell (free) and pipesworld (pipe) domains. We show numbers of expanded and generated nodes. ”-” means timeout after 1800s.

Chapter 5

Conclusion and Future Work

In this thesis, we have proposed a theory to unify various POR algorithms and explained their completeness and optimality preserving properties.

Based on the new theory, we have proposed two new reduction algorithms and evaluated their performance.

There are still many open problems in this direction. For example, the use of categorical notions can be further studied. The category theory provides a foundation for describing abstract algebraic structures. For example, an important goal of POR reduction is to find the minimum action preserving set of paths in a search, which can be represented by the notion of the *terminal object* in $\mathbf{sch}_{\Sigma}$.

References

[1] M. Barr and C. Wells. *Category Theory for Computer Science.* Prentice-Hall, 1995.

[2] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:1636–1642, 1995.

[3] Y. Chen and Y. Guo. Completeness and optimality preserving reduction for planning. In *Proc. IJCAI*, 2009.

[4] Y. Chen, B. Wah, and C-W. Hsu. Temporal planning using subgoal partitioning and resolution in sgplan. *J. Artif. Int. Res.*, 26(1):323–369, 2006.

[5] Y. Chen, Y. Xu, and Y. Guo. Stratified planning. In *Proc. IJCAI*, 2009.

[6] Yixin Chen, Ruoyun Huang, Zhao Xing, and Weixiong Zhang. Long-distance mutual exclusion for planning. *Artif. Intell.*, 173(2):365–391, 2009.

[7] A. Cimatti, E. Giunchiglia, F. Giunchiglia1, and P. Traverso. Planning via model checking: A decision procedure for ar. In *Recent Advances in AI Planning*, 1997.

[8] S. Edelkamp and M. Helmert. On the implementation of mips. In *In Proceedings of AIPS-00 Workshop on Model Theoretic Approaches to Planning*, pages 18–25. AAAI press, 2000.

[9] Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94*, pages 1123–1128. AAAI Press.

[10] Sami Evangelista and Christophe Pajault. Some solutions to the ignoring problem. In *SPIN*, pages 76–94, 2007.

[11] F. Giunchiglia and P. Traverso. Planning as model checking. In *ECP*, pages 1–20, 1999.

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

[13] Malte Helmert and Gabriele Röger. How good is almost perfect? In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 944–949. AAAI Press, 2008.

[14] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22:57–62, 2001.

[15] Jörg Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *IJCAI*, pages 453–458, 2001.

[16] Jörg Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In *AIPS*, pages 92–100, 2002.

[17] Henry A. Kautz, Bart Selman, and Jörg Hoffmann. SatPlan: Planning as satisfiability. In *Abstracts of the 5th International Planning Competition*, 2006.

[18] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *AAAI*, pages 975–982, 2008.

[19] Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *ICAPS*, 2009.

[20] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[21] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, 1989.

Vita

You Xu

Date of Birth	January 8, 1984
Place of Birth	Yangzhou, China
Degrees	B.S. Mathematics, May 2006
Professional Societies	Association for Computing Machines The Free Software Foundation

Dec 2009

Partial Order Reduction for Planning, Xu, M.S. 2009