T2R2 東京工業大学リサーチリポジトリ
Tokyo Tech Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	古典的STRIPSプランニング問題のためのヒューリスティックアルゴリズム
Title(English)	Heuristic Algorithms for Classical STRIPS Planning Problems
著者(和文)	今井達也
Author(English)	Tatsuya Imai
出典(和文)	学位:博士（理学）, 学位授与機関:東京工業大学, 報告番号:甲第9745号, 授与年月日:2015年3月26日, 学位の種別:課程博士, 審査員:渡辺　治,増原　英彦,田中　圭介,福田　光浩,脇田　建 ,Fukunaga　Alex
Citation(English)	Degree:, Conferring organization: Tokyo Institute of Technology, Report number:甲第9745号, Conferred date:2015/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Tokyo Institute Research Repository)

Heuristic Algorithms for Classical STRIPS Planning Problems

Tatsuya Imai

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Science

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

December 2014

Abstract

Automated planning is used for constructing strategies of intelligent agents. To achieve the given goal a planning agent automatically computes a plan based on knowledge of its abilities and of environments in which it participates. Planning is one of the most important tasks related to artificial intelligence. Research of automated planning has been contributing to the progress of both theory and practice of not only planning research but also other fields of artificial intelligence research.

A class of planning problems corresponds to a model to define the properties of the world and the task of an agent. In this doctoral dissertation, two new algorithms are proposed for a class of planning problems called STRIPS planning.

The STRIPS planning class is one of the most classical and famous classes of planning problems. Although STRIPS planning is the smallest and easiest class, it is however known that even deciding whether an instance of the model of STRIPS planning tasks has a feasible plan to achieve the given goal or not is PSPACE-complete. This is caused by the high expressiveness of the model of the STRIPS planning class, and it is one of the reasons that researchers have still been improving STRIPS planning algorithms.

In recent years algorithms based on the reduction to a pathfinding problem on a weighted directed graph have been one of the central methods to solve STRIPS planning problems. It is known that the sizes of the weighted directed graphs are usually exponentially large compared with the original instances of STRIPS planning problems. Classical brute force pathfinding algorithms cannot solve the problems in permissible time, and hence, researchers have been trying to develop efficient heuristic pathfinding algorithms visiting only a small part of the entire graph. A heuristic pathfinding algorithm is mainly composed of two key elements called a heuristic function and a heuristic search algorithm. A heuristic function estimates the cost to go from a vertex to an end vertex based on information of the instance of the original problem before the reduction to a pathfinding problem. A heuristic search algorithm seeks an end vertex and a path to that end vertex while

speculating costs from vertices to an end vertex by heuristic functions.

In this dissertation a new heuristic search algorithm, for finding a feasible plan for the satisficing STRIPS planning problem, is proposed. This dissertation also proposes a new heuristic function for estimating lower bounds of the optimal costs from vertices to an end vertex, for the cost-optimal STRIPS planning problem.

For the satisficing STRIPS planning problem, whereas some previous heuristic search algorithms blindly trust heuristic functions, the proposed search algorithm stochastically goes towards various directions by probabilities computed from estimations of a heuristic function. The algorithm is designed to avoid misleading by heuristic functions, and as a result, the algorithm tends to find a feasible plan faster than other heuristic search algorithms. Experimental evaluations compared the proposed algorithm with several search algorithms and some practical planning algorithms in terms of the number of solved benchmark instances, running time, the number of visited vertices, and plan quality. The proposed search algorithm outperforms the previous search algorithms and is competitive with the practical planning algorithms.

For the cost-optimal STRIPS planning problem, tightening the lower bound estimations of heuristic functions is one of the central issues. In this dissertation, a new integer linear programming model of a relaxation problem, called the delete relaxation, is proposed. In addition, some enhancements for the model such as variable elimination technique are proposed by incorporating with previous work. The proposed enhancements reduce the size of instances of the proposed model and tighten the optimal cost of the linear programming relaxation of the instances. Experimental results show that the lower bound estimation of the linear programming relaxation of the enhanced model is much tighter than lower bound estimations of heuristic functions in some previous work. Moreover, a heuristic function to compute the optimal cost of the linear programming relaxation of the enhanced model decreases on a large scale the number of visited vertices by heuristic search algorithms. A* search based pathfinding algorithm with the proposed heuristic function is competitive to some state-of-the-art algorithms in terms of the number of solved standard benchmark instances, and outperforms them in terms of number of vertex evaluations.

In the first chapter an overview of automated planning and the contribution are given. The STRIPS planning class and some basic pathfinding algorithms are defined in the second chapter, and in the third chapter recent related work is explained. In the fourth chapter the proposed heuristic search algorithm for the satisficing STRIPS planning problem is defined, and its experimental evaluations are given. The proposed heuristic function

for the cost-optimal STRIPS planning problem and its experimental evaluations are shown in the fifth chapter. In the last chapter the conclusion and future work are discussed.

Acknowledgments

I would like to show my greatest appreciation to professor Osamu Watanabe. Professor Watanabe has led me as my supervisor since I joined his laboratory in 2009. Although professor Watanabe is mainly a researcher of theoretical computer science, he has a great insight and comprehensive knowledge for many other fields. He gave me chances to meet many researchers in other fields, and I learned a variety of fields in computer science in his laboratory. I believe that my study, especially the second contribution of this dissertation, has been inspired by this experience.

Special thanks also go to doctor Akihiro Kishimoto who was an assistant professor of Watanabe laboratory. He held a weekly personal seminar for training me, and it really helped my understanding for artificial intelligence. Kishimoto also gave me a lot of comments and suggestions for my study. A joint work with him is one of the main contributions of this dissertation [61].

I express my deepest gratitude to associate professor Alex Fukunaga from the University of Tokyo. Fukunaga, Kishimoto, and some other people held a weekly seminar about artificial intelligence. He and I started a joint work with this seminar as a start, and this work is the other main contribution of this dissertation [60]. He made an automated selection method of heuristic functions, and he organized and ran the experimental evaluation. I also thank to him for being an examiner of this dissertation.

I am very grateful to the other examiners professor Hidehiko Masuhara, associate professor Ken Wakita, associate professor Mitsuhiro Fukuda, and associate professor Keisuke Tanaka. They gave comments that were not from the viewpoint of someone working with automated planning.

I would like to thank Japan Society for the Promotion of Science for Grant-in-Aid for JSPS Fellows that made it possible to complete this study.

I would like to thank the current chair, professor Naoto Miyoshi, and the staff of the Department of Mathematical and Computing Sciences in Tokyo Institute of Technology. Especially I really thank lecturer Akinori Kawachi currently at the University of Tokushima. Kawachi was also an assistant professor of Watanabe laboratory, and he gave a lot of advice about living

in college and research to me and other students. I would also like to show my appreciation to Tamami Watanabe and Kazuyo Kawaguchi who are the secretaries of the tenth floor of the building W-8 and have cheered me up a lot.

Finally, I would like to express my gratitude to my family and my colleagues for their support and encouragements. Special thanks go to Linus Hermannsson who is a member of Watanabe laboratory. I thank doctor Hidetoki Tanaka, Tomoyuki Hayasaka, Yoshikazu Kobayashi, Kotaro Nakagawa, Hiroki Yamaguchi and all other students in Watanabe laboratory.

Contents

Abstract **i**

Acknowledgments **v**

List of Tables **viii**

List of Figures **ix**

List of Algorithms **xii**

1 Introduction **1**
1.1 Automated Planning 1
1.2 Background of Classical Planning and STRIPS Planning . . . 3
1.3 Problems and Contribution 7
1.4 Outline of This Dissertation 10

2 Preliminaries **13**
2.1 STRIPS-based Planning Problem 13
2.2 Reduction to Pathfinding Problems on Weighted Directed Graphs 16
2.3 Framework of Pathfinding Algorithms and Some Classical Pathfinding Algorithms 18
2.4 The Delete Relaxation of a STRIPS Planning Task 34
2.5 Planning Domain Definition Language 36

3 Related Work **41**
3.1 Non-Pathfinding Reduction 41
3.2 Heuristic Functions Based on the Delete Relaxation 44
3.3 Heuristic Functions Based on the Integer Linear Programming Encodings 46
3.4 Other Heuristic Functions 48
3.5 Satisficing Search Algorithms 50

4 Diverse Best First Search **57**
4.1 Greedy Best First Search with an Inaccurate Heuristic Function 57
4.2 Definition of the Diverse Best First Search Algorithm 60
4.3 Experimental Evaluations 62

5 Integer Programming Model of the Delete Relaxation **75**
5.1 Overview 75
5.2 Basic Model 77
5.3 Enhanced Constraints and Variable Eliminations 79
5.4 Counting Constraints 83
5.5 Automatic Heuristic Selection 87
5.6 Experimental Evaluations 88

6 Conclusion **95**

Bibliography **97**

Appendix **107**

List of Tables

1.1 A summary of the numbers of instances solved by some heuristic search algorithms with the FF heuristic and without enhancements. 9
1.2 The number of instances solved by each enhanced pathfinding algorithm. 9
1.3 The number of solved instances and the number of visited vertices. 10

2.1 An example of a STRIPS planning task. 15
2.2 An example of the delete relaxation of a STRIPS planning task. 35

4.1 The number of instances solved by each algorithm with the FF heuristic and without enhancements 64
4.2 The number of instances solved by each algorithm with the CG/CEA heuristic and without enhancements 66
4.3 Performance of the diverse best first search algorithm with different parameters 67
4.4 Performance of DBFS with resetting one parameter 69
4.5 Performance of the diverse best first search algorithm with different random seeds 69
4.6 The number of instances solved by each algorithm with turning on enhancements 70

5.1 The definition of the variables 77
5.2 Comparison of bounds: $il^{+} = \mathrm{ILP}(T^{+})$, $il^{e+} = \mathrm{ILP}^{\mathrm{e}}(T^{+})$, $il^{e} = \mathrm{ILP}^{\mathrm{e}}(T)$, $il^{e+}_{tr} = \mathrm{ILP}^{\mathrm{e}}_{\mathrm{tr}}(T^{+})$, $il^{e}_{tr} = \mathrm{ILP}^{\mathrm{e}}_{\mathrm{tr}}(T)$. 89
5.3 IPC benchmark problems: # solved with 5 minute time limit. 91
5.4 30 minutes, 2GB RAM: “evals” is the average number of calls to heuristic function, i.e., the average number of visited states in the domain. 93

List of Figures

1.1 A typical example of planning problems. The left side is the initial state of the world, and the right side is the goal. . . . 1

2.1 A domain file of the blocks world domain 37
2.2 A domain file of the blocks world domain 38

4.1 A transition of heuristic values in solving optical-telegraphs #02 in fourth International Planning Competition 58
4.2 Comparison of state expansions between the greedy best first search algorithm and the diverse best first search algorithm with the FF heuristic 66
4.3 Search time for instances solved by the diverse best first search algorithm and the greedy best first search algorithm with the FF heuristic 67
4.4 Comparison of plan lengths for instances solved by the greedy best first search algorithm and the diverse best first search algorithm with the FF heuristic 68
4.5 Comparison of state expansions between FD and DBFS2 . . . 71
4.6 Comparison of plan lengths between FD and DBFS2 72
4.7 Comparison of plan lengths between LAMA and LAMA with DBFS2 73

5.1 Dominance relationships. Edge $X \to Y$ indicates that X is always a lower bound of Y. The four highlighted linear programming variants are used in the A*/autoconf in Tables 5.3-5.4. 87
5.2 Ratio between the optimal costs of the IP's and their LP relaxations, categorized into buckets. [x:y) = "% of instances where the LP/IP ratio is in the range [x:y). 90
5.3 Computation of h^+: Comparison of $\mathrm{IP}^{\mathrm{e}}(T^+)$ and HST on delete-free, relaxed problems 90

5.4 Comparison of the visited vertices between Fast Downward with the landmark cut heuristic and the proposed algorithm. 94
5.5 Comparison of the visited vertices between Fast Downward with the bisimulation merge-and-shrink heuristic and the proposed algorithm. 94

1 Number of visited vertices 108
2 Number of vertex retrievals 108
3 Total time 109

List of Algorithms

1 The pathfinding framework for $(\mathcal{G} = \langle V, E, c \rangle, s, T)$ 20
2 Diverse Best First Search 61
3 Fetching one state 61

Chapter 1

Introduction

1.1 Automated Planning

Automated planning, or planning in short, is reasoning activities of intelligent agents. To achieve the given goal a planning agent automatically computes a plan based on knowledge of its abilities and of environments in which it participates. Planning is one of the most important tasks related to artificial intelligence. Research of automated planning has been contributing to the progress of both theory and practice of not only planning research but also other fields of artificial intelligence research.

It seems that one of the most famous examples of planning problems, or problems related to artificial intelligence, is the *blocks world* problem [89, 40, 42]. In this problem an agent has one hand to grab one block at a time. Given some blocks, the goal of the agent is to stack blocks in a particular order. For example blocks are initially located on the table as the left side of Figure 1.1, and the goal is to stack blocks as described by the right side of the figure. Sometimes an instance of the blocks world

Figure 1.1: A typical example of planning problems. The left side is the initial state of the world, and the right side is the goal.

problem is also an example of tasks of recognizing blocks from the images or of understanding a command described by natural language, and planning

takes charge of reasoning to find an order of moving blocks.

Planning is not only to reason action of an artificial agent like a robot. We can enumerate a number of other examples of automated planning. A lot of combinatorial optimization problems such as logistics and scheduling, puzzles such as sliding puzzle, Rubik's cube, and Sokoban, and even reinforcement learning tasks can also be considered as planning problems. In addition to this, even though we can develop an individual algorithm to solve an individual planning problem, researchers of automated planning aim to develop a generic agent capable of solving a variety of planning problems by only itself. This property of a planning agent is called *domain independence* of the agent. Developing such kind of agents is a challenge to create an intelligent being that is, at least partially, as wise as a human being. It is one of the ultimate goals of artificial intelligence research.

Of course researchers still have not created such intelligent planning agents yet, albeit researchers have studied automated planning since the dawn of artificial intelligence research [79]. However, planning research has brought a variety of progress on practical and fundamental research about artificial intelligence.

On the practical side, a number of automated agents or systems are developed based on planning research. For example, Lipovetzky, Burt, Pearce and Stuckey developed a schedule planner for daily open-pit mining in Australia [74]. Benaskeur, Kabanza, Beaudry, and Beaudoin developed a combat power management planner for military naval operations while cooperating with the Department of National Defence in Canada [8, 7]. Many model checking tools based on planning have been developed recently (e.g., [41], [101], and [102]). A number of teleoperated or autonomous spacecraft reasons activities helped by planning algorithms (e.g., [34], [92], and [62]). These examples are just a small part of the application. Many other systems were developed in planning research.

In addition planning research contributes to developing and improving fundamental algorithms. As shown later, the most basic class of planning problems can be solved by reducing to pathfinding problems, and many heuristic pathfinding algorithms have been proposed in recent years (e.g., [28] and [88]). In connection with research of domain independence, planning researchers generalized some techniques and algorithms in specific domains. For example, Edelkamp imported and generalized a cost estimating method called pattern database heuristic that is originally proposed for puzzles [26]. In contrast to generalizations, Helmert analyzed computational complexities of many combinatorial problems as subclasses of planning problems [50, 51]. A number of multi-agent algorithms have been proposed (e.g., [99], [100], and [93]) that have much application such as transportation, military af-

fairs, and computer games. Probabilistic models such as Markov decision processes have also been energetically studied. They can handle *uncertainty* and *partial observability* of environments. According to Russell and Norvig [89], Koenig first associated Markov decision process with artificial intelligence [69]. After that, a number of research about probabilistic models (e.g., [103]) and partial observing models (e.g., [49]) have been studied.

Of course this doctoral dissertation has not achieved the ultimate goal of planning research yet. However, this dissertation contributes to progress of planning research by proposing two improved fundamental algorithms to solve a certain class of planning problems. In the next section, research of the planning problem class is surveyed.

1.2 Background of Classical Planning and STRIPS Planning

To create domain independent agents, researchers define large classes of planning problems, and then they develop solvers for the classes. A class of planning problems generally corresponds to a model of *planning tasks*. A planning task corresponds to a directive for an agent, and a planning task contains the goal of the agent, the properties of the world, and so forth. Notice that a planning task corresponds to the input string of an instance of a planning problem. Some instances of different planning problems sometimes share the same planning task[1].

This doctoral dissertation focuses on a subclass of *classical planning* called *STRIPS planning*. According to Ghallab, Nau and Traverso [40], the classical planning class has some assumptions on the properties of the agent and the world in which the agent participates, i.e., some assumptions on the model of the planning tasks. In the classical planning class the following properties are assumed:

1. The states of the world are finite and discrete, and the state of the world changes whenever the agent executes an *action*.

2. Changes of states can be fully observable i.e., the agent has complete knowledge about the states of the world and the actions.

3. Changes of states are deterministic.

[1] In the literature, a planning problem, an instance of a planning problem, and a planning task are not distinguished sometimes. This dissertation distinguishes them as precisely as possible.

4. The world is static, i.e., the states of the world do not change at all unless the agent executes an action.

5. A plan[2] is a finite ordered sequence of actions.

6. The purpose of the planning problem is merely to transform the current state into a goal state. In other words, there is no restriction on the trajectory of a plan such as mutual exclusion of actions.

7. All the actions have no duration.

8. The planning phase is an offline process, i.e., the agent assumes that the model of the world does not change at all while it is reasoning.

A planning class without the above assumptions is called a *non-classical planning class* [89].

STRIPS was originally an abbreviation of a planner[3] called *STanford Research Institute Problem Solver* [29]. An instance of planning tasks that can be handled by the STRIPS solver had been called a *STRIPS planning task*. In recent years, instead of the tasks of the original model, an instance of a simplified model is called a STRIPS planning task (e.g., in [13]). A planning problem defined on a STRIPS planning task is called a STRIPS planning problem. A STRIPS planning task is defined based on propositional logic. Each fact of the states of the world corresponds to a propositional variable, and actions change the truth-values of the variables. In the next chapter the formal definition of the STRIPS planning class is presented. In recent years planning researchers have shared the standard benchmark set of STRIPS planning tasks. It contains a variety of domains such as logistics with some constraints (fuel, types of vehicles, or delivery time of packages), puzzles (free-cell, peg-solitaire, or Sokoban), manufacturing planning (assembling products or scheduling machines), controlling robots in discrete worlds, and of course blocks world.

In 1970 's and 1980 's, some relaxations of the above assumptions for classical planning were studied. Especially the relaxation of the fifth constraint on the above list, that is called *partial-ordered planning*, was well studied. In 1975, Sacerdoti proposed NOAH planner [90] and Tate developed NONLIN planner [94] for partial-ordered planning, for example. Because of the rise of non-classical planning, and perhaps because of AI winter [89], it seems that research of the STRIPS planning class decreased during this time.

In the 1990 's, earnest studies of both theoretical analysis and development of planning solvers were started.

[2] The term *plan* of a task describes a feasible solution to achieve the goal.

[3] The term *planner* is used to describe a solver for planning problems.

On theoretical aspects, Bylander showed that computing an optimal plan of a STRIPS planning task is PSPACE-hard and even deciding whether a feasible plan of a STRIPS planning task exists or not is PSPACE-complete [17, 18]. He also showed the computational complexities of planning problems in some restricted subclasses of the STRIPS planning class. In addition to this, Bäckström, Klein, and Nevel analyzed the computational complexities of planning problems in another classical planning class called *SAS+ planning* [5, 4]. It can be easily known that deciding whether a SAS+ planning task has a feasible plan or not is also PSPACE-complete since there exists trivial polynomial time translations between STRIPS planning tasks and SAS+ planning tasks. However, they thoroughly analyzed a hierarchy of some restricted subclasses of the SAS+ planning class. A SAS+ planning task can handle mutual exclusions of facts and properties of the world, and hence SAS+ planning is used to develop algorithms based on such mutual exclusions. In the third chapter a detailed explanation of the SAS+ planning class is given.

The progress of planning solvers in the 90's seems to have been provoked by the improvements in other fields. Kautz and Selman proposed a planner called SATPLAN in 1992 [64]. This planner reduces a STRIPS planning problem to the satisfiability problem (SAT). Blum and Furst developed a planner called GRAPHPLAN in 1995 [10]. GRAPHPLAN handles a STRIPS planning problem without the sequential plan constraint, and this planner reduces the problem to another problem related to network flow problems. Bylander proposed a planner called LPLAN in 1997 [19]. It reduces a STRIPS planning problem to the integer linear programming problem. Many current state-of-the-art planning algorithms seem to have some similarities to these planners. In chapter 3 more detailed explanations of these planners are shown.

In addition to the above planners in the 90's, McDermott proposed a planner called UNPOP in 1996 [76]. This planner reduces a STRIPS planning problem to a pathfinding problem on a weighted directed graph, and then it solves the pathfinding problem with a *heuristic pathfinding algorithm*. It is known that the sizes of the weighted directed graphs generated by this reduction are usually exponentially large compared with the sizes of the original instances. On the one hand, classical brute force pathfinding algorithms tend to visit all the vertices of the graph, and hence they cannot solve huge instances of pathfinding problems in permissible time. On the other hand, heuristic pathfinding algorithms are developed to solve pathfinding problems by visiting only a small part of the entire graphs. A heuristic pathfinding algorithm is mainly composed of two key elements called a *heuristic function* and a *heuristic search algorithm*. A heuristic function estimates the cost to

go from a vertex to an end vertex based on information of the instance of the original problem before the reduction to a pathfinding problem. A heuristic search algorithm seeks an end vertex and a path to that end vertex while speculating costs to go to an end vertex by heuristic functions. Note that, in this dissertation, the term "(heuristic) pathfinding algorithm" is used to denote a complete algorithm for a pathfinding problem, and "(heuristic) search algorithm" to denote a part of pathfinding algorithm to control search directions[4]. Although the original STRIPS planner was also based on the pathfinding reduction, according to Russell and Norvig [89], McDermott invented a heuristic function based on the *delete relaxation* first. The delete relaxation is one of the most popular relaxations in STRIPS planning. Given a STRIPS planning task, all effects of actions that set false-value to variables are removed by the delete relaxation, and this increases feasible plans of the task. Hence, although the delete relaxation is technically a translation of a STRIPS planning task, the delete relaxation makes an instance of a relaxed problem. Instances of relaxed problems can be used for heuristic functions. Note that the term *relaxation* has two meaning in the literature[5]. In this dissertation, a STRIPS planning task made by the delete relaxation for the original task is called a *relaxed task* or a *delete-free task*.

In 1998, Bonet and Geffner proposed another heuristic pathfinding algorithm called Heuristic Search Planner [12], and this started to shed light on the pathfinding reduction and heuristic pathfinding algorithms. The first international competition of planning algorithms called AIPS98 Competition was held in the same period, and Heuristic Search Planner obtained good results in the competition. After that, in 2000, Hoffmann proposed another heuristic pathfinding based planner called Fast Forward planner [57], and it was the best planner of the fully automatic track of AIPS00. Both Heuristic Search Planner and Fast Forward also use the delete relaxation of the original STRIPS planning problem for cost estimation. In the later chapters, the delete relaxation is defined more formally, and some important portions of Heuristic Search Planner and Fast Forward are explained.

The name of the AIPS Competition was changed to International Planning Competition[6], and the eighth International Planning Competition was held in 2014. Techniques used in the distinguished planners of the competitions often became the central topic of planning research. Fast Downward

[4]Some classical algorithms such as breadth first search algorithm are classified into pathfinding algorithms under this definition. However the standard names for these classical pathfinding algorithms are used instead of using names such as "breadth first pathfinding algorithm".

[5]One meaning is a method to relax an instance of a problem, and the other is an instance of a problem made by relaxing.

[6]Now AIPS98 and AIPS00 are called the first and second International Planning Competition.

planner [53] was one of the distinguished planners of the fourth International Planning Competition. Fast Downward uses SAS+ planning and heuristics based on certain graph structures called the *causal graph* and the *domain transition graphs*. LAMA planner [86] was also one of the distinguished planners of the sixth International Planning Competition. LAMA uses another kind of data structure called *landmark*. Many algorithms based on causal graphs, domain transition graphs, or landmarks became important factors of recent planning research.

Note that researchers use the benchmark planning tasks made for International Planning Competitions as a de facto standard benchmark set for research. As mentioned above, it contains a variety of domains such as logistics, puzzles, manufacturing planning, and controlling robots. The benchmark planning tasks are written in a formal language called Planning Domain Definition Language (PDDL) [75]. PDDL has been extended several times, and the current version of PDDL can describe not only STRIPS planning tasks but also many kinds of non-classical planning tasks. One instance of planning tasks written in PDDL is the combination of two data structures called *domain* and *problem*. A domain data structure is used to define kinds of tasks such as logistics and blocks world, and in a domain it is necessary to define concepts of objects and actions and so on (e.g., the concepts of blocks or trucks). A problem data structure is used to define information of a specific instance of tasks[7] such as the initial state of the world. We technically consider a *domain of planning tasks* as a set of planning tasks that share the same domain data structure. In the next chapter a detailed explanation of PDDL is shown.

The sizes of the planning tasks in the benchmark set seem not so large compared to the sizes of typical benchmark instances for domain dependent algorithms. Though some planning problems of the benchmark domains are NP-hard, the number of objects, such as trucks, packages, and blocks, are less than one hundred for the most. However, yet the number of solved instances in the benchmark set, i.e., coverage, of some state-of-the-art planners is not so high. Thus researchers have been improving algorithms, and two new algorithms are proposed in this dissertation.

1.3 Problems and Contribution

In this doctoral dissertation two new algorithms are proposed for the pathfinding reductions of STRIPS planning problems. More precisely, a new heuristic

[7]In planning research, researchers sometimes use the term *problem* to describe an instance of a problem. In addition, researchers sometimes regard a planning task in the same light as an instance of a planning problem. The name of this data structure follows this meaning and view.

search algorithm, for finding a feasible plan of a STRIPS planning task, is proposed in this dissertation. This dissertation also proposes a new heuristic function for estimating lower bounds of the optimal cost from vertices to an end vertex, for finding an optimal plan of a STRIPS planning task. The heuristic search algorithm was originally proposed in a joint work of the author and Kishimoto in 2011 [61]. The heuristic function was originally proposed by the author and Fukunaga in 2014 [60]

One issue of satisficing[8] heuristic search algorithm is that errors of estimations of heuristic functions lead search directions to unpromising areas of the graph. Especially *greedy best first search* blindly decides its search direction by the estimations of a heuristic function. It seems that the greedy best first search algorithm is one of the most classical and sensitive heuristic search algorithm, and yet it is a popular algorithm in the satisficing STRIPS planning problem and is incorporated into high-performance planners. It always greedily selects a next vertex to visit from the set of candidate vertices that have the current best evaluation value. However, if the heuristic functions evaluate vertices inaccurately, greedy best first search may be misled into a useless search direction, thus resulting in performance degradation. Some improved algorithms such as k best first search [28] and alternation method [88] were proposed in the middle of the 2000's, and they still tend to be influenced by errors of estimations.

A simple but effective randomized algorithm is proposed in this dissertation. The algorithm is designed such that it considers diversity of search directions to avoid the errors of heuristic information. The search algorithm stochastically goes towards various directions by probabilities computed from the estimations of a heuristic function. The proposed algorithm is named *diverse best first search*. Experimental evaluations compared the proposed algorithm with the greedy best first search algorithm, the k best first search algorithm, and some state-of-the-art algorithms. Table 1.1 compares three heuristic search algorithms in terms of the number of solved instances in the benchmark set of 1,612 instances of the satisficing STRIPS planning problem. The *FF heuristic* [57] is used for their heuristic functions. 30 minutes time limit and 2 GB memory limit are set for each instance. The greedy best first search algorithm solved 1,209 instances without violating the time and memory limits. The k best first search algorithm requires an integer parameter, and we ran the k best first search algorithm with eight different parameters. We accumulated the results of the different parameters, and k best first search solved 1,288 instances in total. The diverse best

[8]Satisficing means finding a feasible solution. A satisficing planning problem is a problem to find a feasible plan, for example.

first search algorithm solved much larger number of instances compared to these two search algorithms. In addition to this, the benchmark set contains 32 domains, the diverse best first search algorithm solved the largest numbers of instances for all the domain. Similar behaviors can be observed in the experiments with other heuristic functions.

Table 1.1: A summary of the numbers of instances solved by some heuristic search algorithms with the FF heuristic and without enhancements.

Algorithm	**GBFS**	**KBFS**	**DBFS**
# of solved instances (1612)	1,209	1,288	1,451

In addition, this dissertation also proposes a variant of the diverse best first search algorithm with an enhancement technique called (use of) *preferred operators* [53]. Table 1.2 shows a comparison with some practical algorithms. The k best first search algorithms with preferred operators and some different parameters solved 1,382 instances in total, and Fast Downward planner solved 1,458 instances. Fast Downward is based on greedy best first search and several enhancement techniques. The original diverse best first search algorithm with the FF heuristic is already competitive to Fast Downward, and the enhanced version of the proposed algorithm solved 1,481 instances. Other experimental results also show that this approach is successful.

Table 1.2: The number of instances solved by each enhanced pathfinding algorithm.

	EKBFS	**FD**	**DBFS2**
Total (1612)	1,382	1,458	1,481

For cost-optimal STRIPS planning, a new *admissible heuristic function* is proposed. The *A* search algorithm* [43] is known to be a well-established algorithm for the cost-optimal pathfinding problem, and it requires an admissible heuristic function that computes a lower bound of the optimal cost to go to an end vertex for each vertex. It is known that the tighter the admissible heuristic function is, the smaller the number of visited vertices is until the A* search algorithm halts. Hence developing a tight (and fast) admissible heuristic function is one of the central tasks for solving the cost-optimal pathfinding problem.

This dissertation proposes a new integer linear programming model of the cost-optimal STRIPS planning problem for delete-free tasks to develop an

Table 1.3: The number of solved instances and the number of visited vertices.

LM-Cut		Merge&Shrink		Integer Linear	
solved	visited	solved	visited	solved	visited
787	27,091,513	727	180,556,416	785	3,121,322

admissible heuristic function. While it is known that an admissible heuristic function to compute the costs of optimal plans of relaxed tasks is empirically tighter than other admissible heuristic functions [9], the cost-optimal delete-free STRIPS planning problem is known to be NP-hard [17, 18]. However, although a naive formulation of the cost-optimal planning problem for a delete-free task as integer linear programming is impractical, the proposed model incorporates landmarks and relevance based constraints, resulting in an integer linear programming that can be used to directly solve the cost-optimal delete-free STRIPS problem. It is shown by experiments that the proposed integer linear programming model outperforms a previous state-of-the-art solver for the cost-optimal delete-free STRIPS problem. In addition, a heuristic function to compute the optimal costs of the linear programming relaxations of the integer linear programming models for relaxed tasks is also proposed in this dissertation. The A* search algorithm with the proposed heuristic is competitive with the state-of-the-art heuristic functions for the cost-optimal STRIPS planning problem. Table 1.3 shows the summary of the results on the set of 1,366 instances of the cost-optimal STRIPS planning problem. For each instance, the time limit is 30 minutes, and the memory limit is 2 GB. The experimental evaluations compared the A* search algorithm with a heuristic function called *landmark cut heuristic* [54, 14] and with another heuristic function called *bisimulation merge-and-shrink heuristic* [80]. Both pathfinding algorithms are implemented on the Fast Downward system. The number of solved instances of the proposed pathfinding algorithm is competitive to the landmark cut heuristic and outperforms the bisimulation merge-and-shrink heuristic. In addition to this, the number of evaluated states of the proposed algorithm is much smaller than the other algorithms.

1.4 Outline of This Dissertation

In the next chapter some formal definitions are shown to explain the proposed algorithms and their relation to related work. The definition of the STRIPS planning class is given in the first section, and then the reduction from a STRIPS planning problem to a pathfinding problem is defined in the second section. In the third section, a framework of pathfinding algorithms that can be used to instantiate pathfinding algorithms is shown,

and some basic pathfinding algorithms that follows the framework are introduced. The delete relaxation of a STRIPS planning task is defined in detail in the fourth section. PDDL is briefly described in the last section of the second chapter. Although PDDL is not directly and essentially related to the proposed algorithms, it was necessary to translate the International Planning Competition benchmarks written in PDDL to STRIPS planning tasks for experimental evaluations.

In the third chapter some recent related algorithms are briefly introduced. Some of them are used in the experimental evaluations. In the first section of this chapter, some non-pathfinding reductions are explained in detail. Although most of them have not been used to directly solve planning problems recently, they are still important to build recent heuristic functions. Some heuristic functions developed from the late 90 's are mainly based on the delete relaxation. Such delete relaxation based heuristic functions are defined in the second section. In recent years some heuristic functions that do not use the delete relaxation were proposed. Especially it seems that most of the previous practical integer linear programming heuristic functions do not use the delete relaxation. In the third section these integer linear programming heuristic functions are introduced. Some other non-delete relaxation based heuristic functions are also surveyed in the fourth section. Finally, some satisficing heuristic search algorithms are introduced in the last section of this chapter. In addition some improvements of heuristic search algorithms used in some state-of-the-art planning algorithms are described in the last section of this chapter.

In the fourth chapter the heuristic search algorithm for the satisficing STRIPS planning problem is proposed. In the first section of this chapter an example of the case that a heuristic function leads greedy best first search to an unpromising area of the problem is shown. The proposed algorithm is defined in the second section, and the results of the experimental evaluations are shown.

The heuristic function for the cost-optimal STRIPS planning problem is proposed in the fifth chapter. In the first section, an overview of the integer linear programming model is briefly given. The integer linear programming model is defined formally in the second section, and some enhancement techniques for the model are proposed from the third section to the fifth section. In the last section of this chapter the experimental evaluations are shown.

Finally the conclusion is discussed in the last chapter.

Chapter 2

Preliminaries

In this chapter some preliminaries for the proposed algorithms are given.

In the section 2.1 the model of STRIPS planning tasks is formally defined, and two STRIPS planning problems are defined. Next the reduction from a STRIPS planning problem to a pathfinding problem is explained in the section 2.2. A framework for pathfinding algorithms is defined, and some basic pathfinding algorithms are introduced in the section 2.3. Especially the greedy best first search algorithm and the A* search algorithm are deeply related to the contribution in this dissertation. In the section 2.4 the delete relaxation of a STRIPS planning problem is defined formally. The delete relaxation is frequently used for cost estimation of heuristic functions. Finally an explanation and an example of PDDL are given in the section 2.5.

2.1 STRIPS-based Planning Problem

In this section, a formal definition of the model of STRIPS planning tasks is given. As already stated, the STRIPS planning class is the class of planning problems that are defined on STRIPS planning tasks. In other words, a STRIPS planning task corresponds to the input string of an instance of a STRIPS planning problem.

As mentioned above, the original model of STRIPS planning tasks was a little complex, and in recent years a task using a model simplified from the original one is called a STRIPS planning task (e.g., in [13]). In this dissertation, a STRIPS planning task $\mathcal{T}$ is defined as a 4-tuple $\mathcal{T} = \langle P, A, I, G \rangle$.

- P is a set of *propositions*. A subset of P is called a *state* of $\mathcal{T}$.
- A is a set of *actions*. Each action $a \in A$ is composed of three subsets of P and a non-negative (rational) number $\langle \mathrm{pre}(a), \mathrm{add}(a), \mathrm{del(a)}, c(a) \rangle$.

These elements are called the *preconditions*, the *add effects*, the *delete effects*, and the *cost* of the action a respectively. An action is often called an *operator* in the literature.

- $I \subseteq P$ is called the *initial state* of $\mathcal{T}$.
- $G \subseteq P$ is called the *goal* of $\mathcal{T}$.

It is assumed that P and A are finite sets (and therefore all the sets are finite).

Each state $S \subseteq P$ corresponds to a certain possible world related to the task $\mathcal{T}$. Although it is mentioned in the last chapter for simplicity that a state is an assignment of truth-values to propositional variables, S is technically the set of propositions that are true in the world. All other propositions that are not in S are false in the world. These two definitions are clearly equivalent.

An action can be applied to a state if the state satisfies a condition, and applying an action to a state changes some facts and properties of the world corresponding to the state. More precisely, an action a is *applicable* to a state S if and only if it satisfies $\mathrm{pre}(a) \subseteq S$. By applying a to S, the set of propositions in the world change from S to $S(a) = ((S \setminus \mathrm{del}(a)) \cup \mathrm{add}(a))$. For a sequence of actions $\pi = (a_0, \cdots, a_n)$, the notation $S(\pi)$ is used to denote $((((S \setminus \mathrm{del}(a_0)) \cup \mathrm{add}(a_0)) \setminus \mathrm{del}(a_1)) \cup \cdots) \cup \mathrm{add}(a_n)$. The sequence π is *applicable* to S if and only if S and π satisfies $\forall i, \mathrm{pre}(a_i) \subseteq S((a_0, \cdots, a_{i-1}))$. The *cost* $c(\pi)$ of an action sequence $\pi = (a_0, \cdots, a_n)$ is defined as $\sum_{i=0}^{n} c(a_i)$.

If a state $S \subseteq P$ satisfies $G \subseteq S$, then S is a *goal state*. The purpose of a STRIPS planning task is to find a sequence of actions to transform I to a goal state. Formally, a feasible solution, i.e., a *plan* is a sequence of actions $\pi = (a_0, \cdots, a_n)$ that satisfies (i) π is applicable to I, and (ii) $G \subseteq I(\pi)$. The notation S_G to denote the set $\{S \subseteq P \mid G \subseteq S\}$.

Given a STRIPS planning task $\mathcal{T}$, the *cost-optimal STRIPS planning problem* on $\mathcal{T}$ is a problem to find a minimum cost plan of $\mathcal{T}$. Given a STRIPS planning task $\mathcal{T}$, the *satisficing STRIPS planning task* on $\mathcal{T}$ is a problem to find any feasible plan (although to find a cheaper plan is preferred practically). If there is no plan in the task, then it is required for these two problems to answer that there is no plan.

Table 2.1 shows an example of a STRIPS planning task. This is an example of blocks world with only two blocks. A tuple such as (A on table) describes a proposition. Although a proposition is an atom with no meaning when a planner runs, propositions in this example are labeled with their roles. For example, (A on table) describes the fact that the block A is located on the table, and (clear B) describes the fact that the roof of the block B is

Table 2.1: An example of a STRIPS planning task.

P	{ (A on B), (A on table), (B on A), (B on table), (clear A), (clear B) }
A	{ move_A_table_B, move_A_B_table, move_B_table_A, move_B_A_table }, where move_A_table_B = $\langle$\{ (A on table), (clear B) \}, \{ (A on B) \}, \{ (A on table), (clear B) \}$, 1\rangle$, move_A_B_table = $\langle$\{ (A on B) \}, \{ (A on table), (clear B) \}, \{ (A on B) \}$, 1\rangle$, move_B_table_A = $\langle$\{ (B on table), (clear A) \}, \{ (B on A) \}, \{ (B on table), (clear A) \}$, 1\rangle$, move_B_A_table = $\langle$\{ (B on A) \}, \{ (B on table), (clear A) \}, \{ (B on A) \}$, 1\rangle$.
I	{ (A on table), (B on table), (clear A), (clear B) }
G	{ (A on B) }

clear. In the initial state of this example the block A and B are located on the table, and the goal of this instance is to put the block A onto the block B. An action move_X_Y_Z moves the block X from Y to Z. For example, the action move_A_table_B moves the block A from the table onto the block B. It can be used only when the block A is located on the table and nothing is on the roof of the block B. The action sequence (move_B_table_A, move_B_A_table, move_A_table_B) is an example of feasible plans of this task not only under semantics, but also under the syntactical and formal definition of a STRIPS planning task. The action sequence (move_A_table_B) with only one action is a feasible and the cost-optimal solution of this task.

As enumerated in the later chapter, there are many kinds of domains that are able to describe as STRIPS planning tasks. Tasks related to logistics, controlling space satellite, running robots, assembling products, and so forth are used in the experiments. The *domain independence* of a model of planning tasks is important not only for practical reasons such as reducing efforts to develop domain dependent algorithms, but also for creating strong intelligent agents in the future.

Due to domain independence, STRIPS planning problems are really hard to solve. Although the model of STRIPS planning tasks seems to be one of the simplest models, it is known that the complexity class of the cost-optimal STRIPS planning problem is PSPACE-hard, and the existence version of the satisficing STRIPS planning problem is PSPACE-complete [18]. Even the cost-optimal blocks world problem is known to be NP-hard in the general case [42]. Hence it seems correct that there is no polynomial time algorithm

to solve STRIPS planning problems, and AI researchers continue to make efforts to develop heuristic algorithms.

2.2 Reduction to Pathfinding Problems on Weighted Directed Graphs

Both the cost-optimal STRIPS planning problem and the satisficing STRIPS planning problem can be reduced to pathfinding problems on weighted directed graphs. More precisely, the states and actions of a STRIPS planning task can be translated to a weighted directed graph that describes transitions of the states, and as a consequence, some STRIPS planning problems can be reduced to pathfinding problems. Many recent planning algorithms and the proposed algorithms in this dissertation run on these reductions. In this section these reductions are formally defined.

A *(weighted directed) graph* is a well-known mathematical structure. It is used to describe a network structure such as road maps and computer networks and so on. Although there are some equivalent definitions, a weighted directed graph $\mathcal{G}$ is defined as a 3-tuple $\mathcal{G} = \langle V, E, c \rangle$ in this dissertation. V is a set of *vertices*. For instance, a vertex could describe a crossing for road maps or a computer node for computer networks. E is a set of *directed edges*. A directed edge, or an edge in short, could for instance describe a traffic lane for road maps or a cable for computer networks. A directed edge e corresponds to an ordered pair of two vertices $(s(e), e(e))$. The vertices $s(e), e(e) \in V$ are the *start vertex* and *end vertex* of the edge e respectively. Note that a directed edge is sometimes called an *arc* or a *directed arc*. To distinguish between the set of actions and the set of arcs symbolically, the word “edge” is used to describe a relation of a pair of vertices in this dissertation. It is assumed that V and E are finite sets. Finally $c : E \to \mathbb{Q}_{\geq 0}$ is a function from a directed edge to a non-negative rational number. The function c is called the *cost function*. For each directed edge $e \in E$, $c(e)$ is called the *cost* of the directed edge e. Note that a *directed graph* is a weighted directed graph without its cost function.

A *directed path*, or a *path* in short, is a sequence of edges $(e_0, \cdots, e_{l-1})$ that satisfies

$$\forall i \in \{1, \cdots, l-1\}, e(e_{i-1}) = s(e_i).$$

If a sequence of edges $(e_0, \cdots, e_{l-1})$ is a directed path, sometimes a sequence of vertices $v_0, \cdots, v_l$ that satisfies

$$\forall i \in \{0, \cdots, l-1\}, \exists e \in E, (v_i, v_{i+1}) = (s(e), e(e))$$

is also called a directed path. The vertices $v_0 = s(e_0)$ and $v_l = e(e_{l-1})$ are called the *start vertex* and *end vertex* of the path respectively, and the path is called a *directed path from v_0 to v_l*. A vertex v is *reachable from* another vertex u if there exists a directed path from u to v. A vertex v that is reachable from another vertex u is sometimes called a *descendant* of u. The number of edges l of a directed path $(e_0, \cdots, e_{l-1})$ is called the *length* of the path. The *cost* of a directed path $(e_0, \cdots, e_{l-1})$ is defined as $\sum_{i=0}^{l-1} c(e_i)$. A minimum cost path from a vertex u to a vertex v is called an *optimal path* from u to v. The cost of an optimal path from u to v is called the *optimal cost* (of a path from u to v). If the length of a directed path is zero, or if the sequence of vertices $(v_0, \cdots, v_l)$ of a directed path with the length $l > 0$ satisfies $\forall i \neq j, v_i \neq v_j$, then the path is called a *simple directed path*.

Given a weighted directed graph $\mathcal{G} = \langle V, E, c\rangle$, a vertex $s \in V$ and a subset of vertices $T \subseteq V$, the problem to find a directed path from s to any vertex in T is called a *(single-source) pathfinding problem*. The vertex s is called the *start vertex* of the problem. The set T is called the set of *end vertices* of the problem. A path from s to an end vertex $t \in T$ is sometimes called *a path from s to T*.

In this dissertation, the problem to find such a path with the minimum cost is called the *cost-optimal pathfinding problem*[1]. Similar to the case of a fixed end vertex, a path from s to a vertex in T with minimum cost is an *optimal path* from s to T, and the cost of an optimal path is the *optimal cost* from s to T. Note that an optimal path from s to an end vertex $t \in T$ is not an optimal path from s to T if the optimal cost from s to t is larger than the optimal cost from s to another end vertex $t' \in T$. If there is no directed path from s to T, then it is necessary to answer that there is no directed path. For clarity, a problem to find any path from s to T is sometimes called a *satisficing pathfinding problem* in this dissertation even though it seems unusual. A 3-tuple $(\mathcal{G}, s, T)$ is used to denote an instance of a satisficing pathfinding problem or a cost-optimal pathfinding problem, where $\mathcal{G}$ is a weighted directed graph, s is the start vertex in $\mathcal{G}$, and T is the set of end vertices in $\mathcal{G}$.

As a lot of papers and books such as [40] stated, given a STRIPS planning task $\mathcal{T} = \langle P, A, I, G\rangle$, we can make a weighted directed graph $\mathcal{G} = \langle V, E, c\rangle$ as follows.

- $V := 2^P$.

[1] A cost-optimal pathfinding problem is sometimes called a *shortest path problem*. An optimal path is a shortest length path when the cost function is a unit cost function. In this dissertation, however, distinction between cost and length is quite important. Thus the term "shortest path problem" is not used in this dissertation.

- For each pair of a state S and an action a that is applicable to S, there exists a corresponding directed edge e that satisfies (i) $s(e) = S$, (ii) $e(e) = S(a)$, and (iii) $c(e) = c(a)$.

A feasible plan of $\mathcal{T}$ clearly corresponds to a directed path from I to S_G. The cost of a plan is equal to the cost of the corresponding directed path. If there is no path, then obviously there is no plan. Therefore the satisficing STRIPS planning problem $\mathcal{T}$ can be reduced to the satisficing pathfinding problem, and the cost-optimal STRIPS planning problem can be reduced to the cost-optimal pathfinding problem. Note that the weighted directed graph made from a STRIPS planning task is called the *search space* [40].

2.3 Framework of Pathfinding Algorithms and Some Classical Pathfinding Algorithms

The satisficing pathfinding problem can be solved by well-known textbook algorithms such as the *breadth first search algorithm* or the *depth first search algorithm*. The cost-optimal pathfinding problem can also be solved by well-known algorithms such as *Dijkstra's algorithm* [23] or the *Bellman-Ford algorithm* [6, 30]. The breadth first search algorithm and the depth first search algorithm run in linear time of the order of the size of the input graph, i.e., the number of vertices plus the number of the directed edges. Dijkstra's algorithm and the Bellman-Ford algorithm run in some low-degree polynomial time of the order of the size of the weighted directed graph. However, the number of vertices of the weighted directed graph made from a STRIPS planning task $\mathcal{T} = \langle P, A, I, G \rangle$ is always $2^{|P|}$ under the reduction above. Although time complexities of these classical pathfinding algorithms can be tightened by the number of vertices reachable from the start vertex, even the number of vertices reachable from I is usually an exponential order of $|P|$ in typical benchmark domains. Hence researchers have been trying to develop heuristic algorithms that are practically more efficient than the classical pathfinding algorithms. Note that pathfinding reduction based planning algorithms and their program implementations do not generate the graph explicitly before their search. Algorithms and programs run while generating a part of the search space dynamically when it becomes necessary. Otherwise generating the entire search space takes an exponential time. However algorithms below are defined as if the graph is explicitly given since there is almost no difference for the pseudo code.

Except the Bellman-Ford algorithm, all other classical pathfinding algorithms introduced above can be formulated on the same framework. In this section, the formal definition of this framework is shown, and strict

definitions of some classical pathfinding algorithms are introduced along the framework. Note that the idea of integrating pathfinding algorithms into one framework is not new. For example, Russell and Norvig defined a similar framework for a restricted class of graphs [89], and the framework for general graphs also seems to be widely used as folklore. However there is no formal definition of the framework for general graphs and proofs for characteristics of the framework as far as the author knows. Hence here the framework is formally organized, and lemma 2.1 and 2.3 are proved.

Let $\langle \mathcal{G} = \langle V, E, c \rangle, s, T \rangle$ be a satisficing/cost-optimal pathfinding problem. By starting from the start vertex, the framework repeats to expand the visited area (i.e., a subgraph of $\mathcal{G}$ whose vertices are already visited) by visiting vertices one by one along the directed edges. When the framework reached an end vertex first, it returns a directed path from the start vertex to the found end vertex, and then it halts. Each framework has two data structures called the *closed list* and the *open list* respectively. Intuitively, the closed list retains the visited area of the graph, and the open list retains start vertices of candidate edges to pass next. Algorithm 1 shows a pseudo code of the framework. Note that visiting a vertex is called *vertex generation* [89]. Retrieving a vertex from the open list and processing something about the vertex is called *vertex expansion.* The end vertex $e(e)$ of a directed edge e on the line 17 is called a *successor* of the vertex u.

The Closed List The closed list retains the visited area as a *directed tree.*

A directed tree is a kind of directed graph. There exists a special vertex r called the *root vertex* for each directed tree. A directed tree $\mathcal{G}_T = (V_T, E_T)$ satisfies that:

(i) there does not exist a directed edge $e \in E_T$ that satisfies $e(e) = r$,

(ii) for each vertex $v \in V_T$ except r, $|\{e \in E_T \mid e(e) = v\}| = 1$ holds, and

(iii) for each vertex $v \in V_T$, there exists just one directed path from r.

For each vertex $v \in V_T$, if $e \in E_T$ satisfies $e(e) = v$, then the vertex $s(e)$ is called the *parent vertex* of v. There is no parent of the root vertex. If $u \in V_T$ is the parent of $v \in V_T$, then v is called a *child* of u. A vertex with no children is called a *leaf* of a directed tree.

In this framework, the directed tree of the visited area is stored by a hash table, i.e., the closed list is a hash table. Each key on the hash table is a vertex, and the set of the keys corresponds to the set of vertices on the directed tree. The value of a key $u \in V_T$ is a 3-tuple (p, e, c), where $p \in V_T$ is the parent vertex of the key u, and $e \in E_T$ is the edge from p to u. Hence, the hash table implicitly forms at least a directed graph with the second

Algorithm 1 The pathfinding framework for $(\mathcal{G} = \langle V, E, c\rangle, s, T)$

1: The closed list $C := \{(s, (\text{null}, \text{null}, 0))\}$; // A hash table.
2: The open list $O := \{s\}$; // A data structure that can keep vertices.
3: **while** O is not empty **do**
4: Retrieve a vertex u from O; // The priority will be defined later.
5: **if** $u \in T$ **then**
6: Let p be an empty sequence of edges;
7: $c := 0$;
8: **while** $u \neq s$ **do**
9: Retrieve the value (v, e, c') of u from C; // c' is not used.
10: Add e to the end of p;
11: $c := c + c(e)$;
12: $u := v$;
13: **end while**
14: Return c and the inverse sequence of p and halt;
15: **end if**
16: Let c be the cost (i.e., the third element of the value) of u in C;
17: **for** $e \in E$ s.t. $s(e) = u$ **do**
18: **if** $e(e)$ is not in C **then**
19: Insert an entry into C with the key $e(e)$ and the value $(u, e, c + c(e))$;
20: Insert $e(e)$ into O;
21: **else**
22: Let (v', e', c') be the value of $e(e)$ in C;
23: **if** $c + c(e) < c'$ **then**
24: Update the value of $e(e)$ in C with $(u, e, c + c(e))$;
25: Re-insert $e(e)$ into O if it is necessary in the instantiation.
26: **end if**
27: **end if**
28: **end for**
29: **end while**
30: return "There is no directed path from r to T." and halt;

condition of a directed tree. Note that the directed tree of the visited area is sometimes modified on the line 24. If we ignore this line, then the graph described by the hash table clearly forms a directed tree. A proof is given later to show that this graph always satisfies all conditions of a directed tree even if we use updating existing entries. If the hash table forms a directed tree, then, for each vertex v on the directed tree, the path from r to v on the directed tree can be easily constructed by going back along directed edges on the tree. The third element c of an entry is an *upper bound* of the cost of the directed path from r to u on the directed tree. This is because the framework does not update the third elements of all the entries when the directed tree is modified. Hence, the third elements can be different from the actual costs. If we ignore updating, then c is the precise cost from r to u on the directed tree. A proof is shown later for the fact that the third entries

are always equal to or larger than the actual cost. Below, the third element of the value of a key u is called *the cost of u in the closed list*. The actual costs of the directed paths on the directed tree also can be calculated easily when the directed path is constructed. At the beginning of the framework, the closed list has only one entry with the key s and the value $(\text{null}, \text{null}, 0)$.

The Open List The open list keeps visited vertices that may reach an unvisited vertex by going along just one directed edge. In some instantiations the open list also keeps visited vertices that may reduce the costs of some directed paths on the directed tree. Note that a retrieved vertex from the open list and a vertex already in the open list are sometimes re-inserted to the open list on the line 25.

To expand the visited area, or, to modify the directed tree, the framework continues to repeat the following process:

(i) At first it retrieves one vertex u from the open list on the line 4. If u is an end vertex, then the framework stops looping and generates a path from s to u. The vertex u must be already in the closed list, and hence the framework can retrieve the cost c of u from the closed list on the line 16.

(ii) Then, for each $e \in E$ that satisfies $s(e) = u$,

- (ii-1) if the vertex $e(e)$ is not visited, then the framework visits $e(e)$. Namely, if a key $e(e)$ is not in the closed list, then the framework saves an entry with the key $e(e)$ and the value $(u, e, c + c(e))$ into the closed list. Then it inserts $e(e)$ into the open list.
- (ii-2) if the vertex $e(e)$ is already visited, and changing the parent of $e(e)$ to u does not seem to decrease the cost to go to $e(e)$, then the framework ignores the directed edge e now. Namely, if a key $e(e)$ is already in the closed list, and if $c + c(e)$ is not smaller than the cost c' of $e(e)$ in the closed list, then the framework ignores $e(e)$ now. Since c' is not always the actual cost of the path to $e(e)$, it depends on the instantiation whether the actual cost on the directed tree decreases or not by changing the parent of $e(e)$.
- (ii-3) if the vertex $e(e)$ is already visited, and if changing the parent of $e(e)$ to u seems to decrease the cost to go to $e(e)$, then the framework changes the parent of $e(e)$ to u. Namely, if a key $e(e)$ is already in the closed list, and $c + c(e)$ is smaller than the cost c' of $e(e)$ in the closed list, then the framework updates the key value of $e(e)$ in the closed list with $(u, e, c + c(e))$. In addition to this, the framework re-inserts $e(e)$ into the open list if necessary. The condition of re-insertion depends on the instantiation.

At the beginning of the framework, the open list has only the start vertex s. When an end vertex $g \in T$ was retrieved from the open list first, then the framework halts and builds the directed path from r to g on the current directed tree. Note that, to find a path from s to T, it suffices to halt when an end vertex was visited (i.e., an end vertex was inserted into the closed list). However it does not suffice to find an optimal path.

The ability to always find a directed path from s to T if it exists is called *completeness* of a pathfinding algorithm [89]. Moreover, the ability to always return an optimal path from s to T when it returns a directed path is called *optimality* of a pathfinding algorithm. In the framework, a new visited vertex is always inserted into the open list on the line 20. Hence the framework always returns a directed path from s to T (1) if there exists a directed path from s to T, (2) if updating entries does not collapse the directed tree, and (3) if it halts. It can be proved that the second condition is always satisfied.

Lemma 2.1. *The closed list forms a directed tree at any time during running the framework.*

Proof: The line 24 cannot violate the first condition of a directed tree since all the costs of edges are nonnegative and the cost of the entry of the start vertex in the closed list is already zero at the beginning of the framework. The second condition is clearly satisfied by the definition of a directed tree in the closed list. Hence it suffices to prove that the third condition of a directed tree is always satisfied. Clearly appending a new entry for an unvisited vertex does not collapse the directed tree if the conditions of a directed tree are satisfied before appending. Thus it is sufficient to focus on updating an entry in the closed list on the line 24.

It can be proved by mathematical induction that (1) the graph formed by the entries in the closed list is a directed tree and (2) the costs of the vertices on the directed tree are always monotonically nondecreasing from the root to leaves. Assume that the framework reached just before the line 23 and that the induction hypothesis is being satisfied now. Then the costs of the vertices on the subtree rooted by the vertex $e(e)$ are also monotonically nondecreasing. The third condition of the definition of a directed tree is violated only when u is a vertex on the current subtree of $e(e)$ and the framework updates the entry of $e(e)$ on the line 24. Otherwise the framework can successfully go back from each vertex to the root vertex along the relations of parents and children, and hence the third condition is satisfied. However it is impossible to satisfy the condition on the line 23 if u is a vertex on the current subtree of $e(e)$ since the costs are monotonically nondecreasing from the root to leaves. Thus, if the framework updated an

entry, then the graph is a directed tree immediately after updating. The cost of the new parent of the vertex $e(e)$ after updating is c, and the new cost of the vertex $e(e)$ is $c + c(e)$. This new cost is smaller than the old cost c', and hence the costs of the children of $e(e)$ are larger than the new cost. Therefore the costs of the vertices on the new directed tree are also monotonically nondecreasing from the root to leaves. □

Here it can be shown that the following lemma holds.

Lemma 2.2. *At any time during running the framework, for each vertex v on the directed tree in the closed list, the cost of v in the closed list is an upper bound of the actual cost of the directed path from s to v on the directed tree.*

Proof: Similar to the previous lemma, this lemma can also be proved by mathematical induction. In addition to this, appending a new entry for an unvisited vertex does not collapse the relations of the costs in the closed list and the actual costs. Thus it is sufficient to focus on updating an entry.

Let $C(v)$ be the actual cost of the directed path from s to a vertex v on the directed tree. Assume that the framework reached just before the line 23 and that the statement of this lemma is satisfied so far. After the entry of the vertex $e(e)$ is modified, the new cost $c + c(e)$ of $e(e)$ in the closed list is an upper bound of the actual cost of the path from s to $e(e)$. This is because c is an upper bound of $C(u)$, and the actual cost $C(e(e))$ from s to $e(e)$ satisfies $C(e(e)) = C(u) + c(e) \leq c + c(e)$. In addition to this, for each child v of the vertex $e(e)$, it can be shown that the cost of v in the closed list is also an upper bound of $C(v)$. Let c_v be the cost of v in the closed list, and e_v be the edge from $e(e)$ to v on the directed edge. The difference of the costs of $e(e)$ and v in the closed list was exactly $c(e)$ when v was connected to $e(e)$. After v was connected to $e(e)$, the cost of v never changed, and the cost of $e(e)$ never increased. Hence $c_v - c' \geq c(e_v)$ is satisfied, and we have $C(v) = C(e(e)) + c(e_v) \leq c + c(e) + c(e_v) \leq c' + c_v - c' = c_v$. Similar inequality is satisfied for each vertex on the subtree of $e(e)$. □

In addition to this, if the input weighted directed graph is finite, then the algorithm always halts (hence it has completeness) even if it re-inserts vertices.

Lemma 2.3. *The framework always halts if the input graph is finite.*

Proof: If the number of re-insertions is finite for each vertex, then the framework clearly halts.

The costs of directed edges are rational numbers. Hence we can create an equivalent pathfinding problem with integer costs. When a vertex v is re-

inserted, the cost of v in the closed list strictly decreases for each re-insertion because of the condition on the line 23. Hence the number of the re-insertion of a vertex v on the integer-cost pathfinding problem is at most the first cost of v. When the original cost function used, the framework clearly works in the same way as the case of the integer cost function. Therefore the number of re-insertion for each vertex is finite. □

Optimality depends on instantiations.

When we ignore the time and space complexity of the implementation of the open list, this framework requires memory in proportion to the maximum size of the open list and the closed list. The number of entries in the closed list is proportional to the number of visited vertices. The number of entries in the open list is bounded by the number of visited vertices. Hence the memory requirement is an order of the number of visited vertices until the framework finds an end vertex, and it is an order of the size of the input graph in the worst case. Estimating time complexity is a little difficult for this framework without an implementation of the open list and re-insertions. However, the number of iterations of the while loop is in proportion to the number of vertex retrievals until the framework halts. Note that the number of vertex retrievals is different from the number of retrieved vertices until the framework halts.

The differences among the instantiated algorithms based on this framework are (1) the priorities of entries in the open list, and (2) the condition of re-insertion. For example, the open list of the breadth first search algorithm is a normal FIFO queue, and it never re-inserts a vertex. The breadth first search algorithm does not have optimality. For another example, the open list of Dijkstra's algorithm is a priority queue, and it always re-inserts a vertex. Dijkstra's algorithm has optimality. Below, some pathfinding algorithms are explained in detail.

The Breadth First Search Algorithm Russell and Norvig stated that the breadth first search algorithm was originally developed by Moore in 1959 [77] to solve maze problems [89]. In these days the word "breadth first search" is not just the name of an algorithm. Some researchers used breadth first search as the name of a manner of graph traversal or the names of some graph algorithms with such a traversal manner.

As already mentioned above, the open list of the standard the breadth first search algorithm is a normal FIFO queue. The breadth first search algorithm does not re-insert a vertex. It can be easily proved by mathematical induction that the breadth first search algorithm visits vertices that have paths with shorter length from the start vertex earlier. Hence the directed

path that is the output of the breadth first search algorithm is a directed path from s to T with the shortest length l. In addition, the breadth first search algorithm needs to visit all the vertices that have a directed path from s with shorter length than l before it halts. When the cost function is a unit cost function, the breadth first search algorithm returns an optimal path. Otherwise there is no guarantee of optimality.

Since the breadth first search algorithm never re-inserts a vertex into the open list, the number of iterations on the line 3 is at most $|V|$, and the number of iterations on the line 17 is at most $|E|$ in total of all the outer loop. Hence it runs in linear time of the order of the size of the weighted directed graph.

The Depth First Search Algorithm Cormen, Leiserson, Rivest and Stein [21] mentioned that Hopcroft and Tarjan recognized importance of the depth first search algorithm first. Depth first search is also the name of a kind of graph algorithms, and there mainly exists two variants of the depth first search algorithm. One variant requires the linear time and linear memory of the size of the input graph. It never re-inserts a vertex into the open list. The other variant sometimes re-inserts a vertex since it sometimes throws some entries in the closed list away. Thus it is memory-efficient, however it does not have the guarantee to work in linear time in the general case. Although the latter variant is the important prototype of some advanced memory efficient pathfinding algorithms such as the IDA* search algorithm [70], it is out of the framework above, and hence the latter is not explained anymore.

The open list of (the former of) the depth first search algorithm is a normal FILO stack. The actual running time of the depth first search algorithm heavily depends on tiebreak of successors on the line 17. If the tiebreaks worked extremely well, then the algorithm visits only the vertices on an optimal path from s to T and returns the optimal path. In contrast, the depth first search algorithm visits all the vertices on the input graph and returns a non-optimal path in the worst case. Hence worst-case time complexity of the depth first search algorithm is the order of the size of the weighted directed graph.

Dijkstra's Algorithm As its name suggests, Dijkstra 's algorithm was proposed by Dijkstra in 1959 [23].

The open list of Dijkstra's algorithm is a priority queue on the cost value $c + c(e)$ of the vertex $e(e)$, where c is the cost of $s(e)$ defined on the line 16. As is the case with the cost in the closed list, the priority of each vertex is not dynamically updated when the directed tree is modified. However,

Dijkstra's algorithm always re-inserts a vertex when it reaches the line 25. If the vertex is already in the open list when it re-inserts the vertex, then the cheaper cost c + c(e) is adopted for the priority of the vertex. In other words, the old entry is thrown away, and the new entry is used. Hence the cost of a vertex in the closed list and its priority in the open list are always the same (if it exists in the open list). It can be proved by mathematical induction that, when a vertex is retrieved from the open list on the line 4, its cost in the closed list is the optimal cost from s to it. Thus each vertex is retrieved at most once. In addition to this, the order of retrieved vertices is the order of the optimal cost from s. Finally, it needs to visit all the vertices that have cheaper optimal cost from s than the optimal cost from s to T until it halts. Therefore Dijkstra's algorithm always returns an optimal path from s to T if it exists. Dijkstra 's algorithm is considered as a special case of the A* search algorithm. Proofs of these propositions are given as a corollary of lemmas about some properties of the A* search algorithm, hence here the proof is omitted.

The worst case time complexity of Dijkstra's algorithm depends on the actual implementation of the open list. Dijkstra did not give a detail of the priority queue in his paper. If the time complexity to retrieve a vertex from the priority queue is R, and if the time complexity to insert a vertex to the priority queue is I, then the time complexity of the Dijkstra's algorithm is $O(|V| \times R + |E| \times I)$. Hence, if a brute force search is used to simulate the priority queue, the time complexity of Dijkstra's algorithm is $O(|V|^2 + |E|)$. If a binary heap is used for the priority queue, the time complexity is $O((|V| + |E|) \log |V|)$. Finally, if a Fibonacci heap is used, the time complexity is $O(|V| \log |V| + |E|)$ in amortized time [32].

So far three pathfinding algorithms have been explained in detail. Although there are some other pathfinding algorithms proposed in previous work, all classical algorithms seem to need to visit a large area of the input directed graph as long as the first input is the directed graph. However, if an instance of a pathfinding problem is made by reducing from another kind of a problem as is the case with the reduction from a STRIPS planning problem, then information about the original problem can be used to solve the instance of the pathfinding problem. For example, when we are solving an instance of a pathfinding problem reduced from an instance of a navigation problem to go from Tokyo to Kyoto, then clearly we do not need to consider a path to go to New York. Thus we can prune or give a very low priority for an edge to go to New York. A pathfinding algorithm that uses information of the original problem is called *heuristic pathfinding algorithm* or *informed pathfinding algorithm* [89].

A heuristic pathfinding algorithm is mainly composed of a *heuristic function* and a *heuristic search algorithm*. The word "heuristic" is used in many ways. For pathfinding problems, a heuristic function is a function mapping from a vertex v to an estimation of the optimal cost from v to T. For example, assume that a navigation problem is reduced to a pathfinding problem as similar to the above. A vertex corresponds to a point such as a crossroad. An edge corresponds to a way between two points such as a lane of a road from a crossroad to a crossroad. Finally the cost of a directed edge is the real distance of the way. The destination of the instance of the problem is (a point of) Kyoto denoted by t, and the purpose of the problem is to get a path to t with a distance as small as possible. Then, for a point u, we can define a function that computes the Euclidean distance between u and t as a heuristic function. An estimation of this heuristic function is known to be a lower bound of a distance along actual ways, and even the Euclidean distance between New York and Kyoto is very large. Hence, according to this heuristic function, the estimation of the cost to go from New York to Kyoto is very high. Usually, the higher a heuristic function gives a estimated cost to a vertex, the lower priority a heuristic search algorithm gives to the vertex. Hence, heuristic search algorithms avoid to visit New York and descendants of New York.

However, note that the technical meaning of "estimation" actually depends on heuristic search algorithms. Sometimes there exists strict conditions, and sometimes there is no condition. Anyhow it is empirically known that running time and memory usage depend on accuracy of a heuristic function.

The *greedy best first search algorithm*, the *A* search algorithm* [43], and the *weighted A* search algorithm* [81] are three classical and important heuristic search algorithms.

The Greedy Best First Search Algorithm Let h be a heuristic function. The open list of the greedy best first search algorithm is a priority queue on $h(v)$ for a vertex v. Intuitively the algorithm continues to visit greedily the nearest vertex to T in the open list according to the heuristic function h. Although tiebreak for the same estimations is not mentioned clearly in pseudo codes in most of related work, it seems that FIFO is mostly used in the implementations. Greedy best first search does not re-insert a vertex to the open list.

In the greedy best first search algorithm, there is no strict condition for estimated costs. If $h(v)$ returns the true optimal costs from v to T, then greedy best first search visits only the vertices that are parts on optimal paths from s to T (although the number of such vertices is a linear order of

the number of all vertices in some problems). Otherwise there is no theoretical guarantee for running time and optimality. However, it is empirically known that, if h is almost precise, greedy best first search halts quickly and returns a directed path with the approximately optimal cost.

Of course there is no guarantee that h returns even approximate estimations in the general cases. It is empirically and experimentally known that heuristic functions for the satisficing STRIPS planning problem are quite inaccurate. However some state-of-the-art satisficing STRIPS planning algorithms are developed on top of the greedy best first search algorithm and those heuristic functions. In the later chapter, a case study about a behavior of greedy best first search is given with a heuristic function for the satisficing STRIPS planning problem. The reason why greedy best first search consumes time to explore unpromising area of the graph is also given. To overcome this problem, a new heuristic search algorithm is proposed in chapter 4.

A* Search Algorithm the A* search algorithm was proposed first by Hart Nilsson, and Raphael in 1968 [43]. Colson pointed out that some conditions for using the A* search algorithms were actually not used. Hence Hart et al. revised and extended the detail of the algorithm in 1972 [44].

Let h be a heuristic function. The open list of the A* search algorithm is a priority queue with the priority $h(v) + g(v)$ for a vertex v, where $g(v)$ is the cost of v in the closed list just before the insertion (i.e., $c + c(e)$, where c is defined on the line 16). As is the case with greedy best first search, FIFO is mostly used for tiebreak in implementations. It always needs to re-insert a vertex to the open list for the following important property. The priority of each vertex is not dynamically updated when the directed tree is modified as the same as Dijkstra's algorithm. When a vertex is re-inserted, the old entry in the open list is thrown away, and the new entry is used. Hence the priority of a vertex is always equal to the sum of its estimation and the cost of a vertex in the closed list if it exists in the open list.

Let h^* be the function that returns the true optimal cost from each vertex to T. If a vertex v does not have a directed path to T, then $h^*(v)$ is defined as infinity. If a heuristic function h satisfies $h(v) \leq h^*(v)$ for each vertex v, then h is called an *admissible* heuristic function. In addition to this, the A* search algorithm with an admissible heuristic function has optimality. It is proven by the following two lemmas. Note that lemma 2.4 is a generalization of the lemma 1 in [43], and lemma 2.5 is another proof of theorem 1 in [43] along the definition of the pathfinding framework in this dissertation[2].

[2]The other lemmas for the A* search algorithm in this section are also generalizations

Lemma 2.4. *Given a cost-optimal pathfinding problem $\langle \mathcal{G} = \langle V, E, c \rangle, s, T \rangle$, let h be an admissible heuristic function for the problem. For each vertex v that has a directed path from s, and for each simple directed path P from s to v, one of the following conditions is satisfied at the beginning of every iteration of the while loop of the A* search algorithm (i.e., the line 3 of the framework). The notation $c_P(x)$ is used to denote the cost of the directed path from s to x along P.*

1. *All vertices on the path P are already visited, and are not in the open list. In addition to this the cost of each vertex u in the closed list is equal to or smaller than $c_P(u)$.*
2. *There exists a vertex v on the path P in the open list, and the priority of v in the open list is equal to or smaller than $h(v) + c_P(v)$.*

Proof: According to the definition of the framework, it is obviously impossible that all vertices on the path P are not in the open list and at least one vertex on P is not in the closed list. Hence it is sufficient to consider the following two cases: (1) all vertices on P are in the closed list and are not in the open list, or (2) at least one vertex on P is in the open list.

First assume that, the algorithm is at the beginning of an iteration of the while loop, all the vertices on a directed path P are already visited, and they are not in the open list. In this case, it can be proved that, for each vertex u on the path P, the cost of u in the closed list is equal to or smaller than $c_P(u)$. To show a contradiction, assume that there exists a vertex u on P and the cost of u in the closed list is larger than $c_P(u)$. Let w be the first such vertex on P. The start vertex s cannot be w. This is because the cost of s in the closed list is zero at the beginning of the algorithm, and the algorithm does not increase the costs of entries in the closed list. Hence there exists a vertex x before w along P. In addition, since w is the first vertex with a larger cost than $c_P(w)$ along P according to the definition of w, the cost of x in the closed list is equal to or smaller than $c_P(x)$. Since the vertices x is visited and is not in the open list on the current iteration, x was already retrieved at least one time. Let c_x be the cost of x in the closed list at the last retrieving of x before this iteration. If x was re-inserted after the last retrieving, it contradicts the assumption that all vertices on P are not in the open list. Hence c_x is the current cost of x in the closed list, and thus it is equal to or smaller than $c_P(x)$. Let e be the directed edge between

or modifications of some previous work and folklore for adjusting the above framework. The properties of the A* search algorithm shown by these lemmas are deeply related to the proposed heuristic function. Not only the proofs of these lemmas but also some proofs for other pathfinding algorithms are sometimes folklore and are sometimes wrong (e.g., Akagi, Kishimoto, and Fukunaga showed and corrected an error of the IDA* search algorithm [1]). Hence the proofs of these lemmas are shown in this dissertation.

x and w along P. At the last retrieving of x, the algorithm updated the cost of w in the closed list with the cost $c_x + c(e)$ unless the cost of w was already equal to or smaller than $c_x + c(e)$. This contradicts the assumption since $c_x + c(e) \leq c_P(x) + c(e) = c_P(w)$ holds.

If a vertex on a directed path P is in the open list at the beginning of an iteration of the while loop, let v be the first vertex in the open list along P. Then it can be proved that, for each vertex u on the directed path from s to v along P, the cost of u in the closed list is equal to or smaller than $c_P(u)$ by a similar argument. To show a contradiction, assume that there exists a vertex u between s and v, and assume the cost of u in the closed list is larger than $c_P(u)$. Let w be the first such vertex from s to v along P. The vertex w cannot be s, and there exists a vertex x before w along P. According to the definition of v and w, the vertex x was retrieved at least one time, and the cost c_x of x in the closed list at the last retrieving is equal to or smaller than $c_P(x)$. Hence the algorithm tried to update the cost of w in the closed list with the cost $c_x + c(e)$, where e is the edge between x and w along P. This contradicts that the cost of w in the closed list is larger than $c_P(w)$. Therefore the cost of v in the closed list is equal to or smaller than $c_P(v)$, and the priority of v in the open list is equal to or smaller than $h(v) + c_P(v)$. □

Lemma 2.5. *the A* search algorithm with an admissible heuristic function has optimality.*

Proof: Assume the A* search algorithm with an admissible heuristic function returned a non-optimal path P from s to $t \in T$. Let C^* be the optimal cost from s to T, and let $C > C^*$ be the cost of P. Finally let P^* be an optimal path from s to T.

According to lemma 2.4, P^* satisfies either the conditions of lemma 2.4 at every beginning of an iteration of the while loop. If the first condition is satisfied for P^* at the beginning of the last iteration (i.e., the iteration when the A* search algorithm returned P), the algorithm must have been already stopped. Hence the second condition of lemma 2.4 is satisfied for P^* at the beginning of the last iteration. Let u be a vertex satisfying the second condition of lemma 2.4 for P^*. The priority of u at the beginning of the last iteration is equal to or smaller than $h(u) + c_{P^*}(u)$, and this value is equal to or smaller than C^* since the heuristic function is admissible. On the other hand, the priority $h(t) + g(t)$ of t is equal to or larger than C. This is because $h(t) = 0$ since h is admissible and $g(t)$ is the cost of t in the closed list, i.e., $g(t)$ is an upper bound of P. Therefore it contradicts the fact that t is retrieved from the priority queue at the last iteration. □

By similar arguments with lemma 2.5, the following two lemmas can be proved. These lemmas are generalizations of lemma 3 in [43] and lemma 4 in [44], and these are also already known as folklore.

Lemma 2.6. *Let h be an admissible heuristic function. Let C^* be the optimal cost. For each vertex v, let $g^*(v)$ be the optimal cost from s to v. If a vertex satisfies that $h(v) + g^*(v) > C^*$, then the A* search algorithm with h never retrieves v. In addition to this, if it is satisfied that $h(u) + g^*(u) > C^*$ for each vertex u that has a directed edge to v, then the A* search algorithm with h never visits v.*

Proof: As is the case with the previous lemma, the end vertex of an optimal path is retrieved before a vertex satisfying the condition $h(v) + g^*(v) > C^*$ is retrieved. In addition, if it is satisfied that $h(u) + g^*(u) > C^*$ for each vertex u that has a directed edge to v, then all such vertices are not retrieved before the A* search algorithm halts. Therefore v is not visited. □

Lemma 2.7. *Let h be an admissible heuristic function. Let C^* be the optimal cost. For each vertex v, if there exists a directed path P from s to v, and if $h(u) + c_P(u) < C^*$ is satisfied for each vertex u between s and v along P, then the A* search algorithm visits and retrieves v at least one time respectively.*

Proof: Let v and P be a vertex and a path satisfying the above condition. As is the case with lemma 2.5, before the end vertex of an optimal path is retrieved with the cost C^*, all the vertices on the path P including v are visited and retrieved. □

Whether the A* search algorithm retrieves (or visits) a vertex u satisfying $h(u) + g^*(u) = C^*$ or not depends on tiebreak for vertices of the same priority. The number of vertex retrievals and vertex re-insertions for vertices satisfying the condition of lemma 2.7 depends on an admissible heuristic function in the case of general. However, if a heuristic function is *consistent*, then a vertex satisfying the condition of lemma 2.7 is retrieved at most one time. A heuristic function h is consistent if it satisfies (1) $h(s(e)) \leq c(e) + h(e(e))$ for each directed edge $e \in E$, and (2) $h(t) = 0$ for each end vertex $t \in T$. Although no non-trivial consistent heuristic function is introduced in this dissertation, the following lemmas can be proved.

Lemma 2.8. *A consistent heuristic function is admissible.*

Proof: Let h be the consistent heuristic function. Let v be a vertex that has a directed path to T. Let $P^* = (e_0, \cdots, e_{l-1})$ be an optimal path

from v to T, and let $v = v_0, v_1, \cdots, v_l$ be the vertices on P^*. Then we have $h(v_i) \leq c(e_i) + h(v_{i+1})$ for each $i \in \{0, \cdots, l-1\}$. By substituting $h(v_i) \leq c(e_i) + h(v_{i+1})$ for the right-hand side of $h(v_{i-1}) \leq c(e_{i-1}) + h(v_i)$, we have $h(v) = h(v_0) \leq c(e_0) + h(v_1) \leq \cdots \leq \sum_{i=0}^{l-1} c(e_i)$. The right-most side is equal to the cost of P^*. Hence h is admissible. □

Lemma 2.9. *The A* search algorithm with a consistent heuristic function retrieves each vertex at most one time until the A* search algorithm halts. In addition to this, after a vertex is retrieved, the vertex is never re-inserted into the open list.*

Proof: Let h be a consistent heuristic function used by the A* search algorithm. If the latter statement is true, then the former is clearly true.

Assume the A* search algorithm reached at the beginning of an iteration of the while loop, and vertex v is retrieved from the open list. The priority of v before the retrieving is $h(v) + g(v)$. In this iteration, for each directed edge e which satisfies $s(e) = v$, we have $h(v) + g(v) \leq h(e(e)) + c(e) + g(v) = h(e(e)) + g(e(e))$ since h is consistent. Hence the minimum priority of the open list does not decrease.

This situation continues every iteration. Assume that, after a vertex u is retrieved, another vertex v is retrieved in the current iteration of the while loop, and there exists an edge e satisfying $s(e) = u$ and $e(e) = v$. Let p_u, p_v be the priority of u and v when u and v are retrieved respectively. Let c_u, c_v be the cost of u and v in the closed list when u and v are retrieved respectively. Since $p_u \leq p_v$ holds, we have $c_v + c(e) + h(u) \geq c_v + h(v) = p_v \geq p_u$. Thus $c_v + c(e) \geq p_u - h(u) = c_u$ is satisfied, and therefore u is not inserted into the open list. □

Dijkstra's algorithm is a special case of A^* search algorithm with a zero heuristic function $h \equiv 0$. A zero heuristic function is clearly consistent. Hence the following proposition holds.

Corollary 2.10. *(1) Dijkstra's algorithm halts and returns an optimal path if there exists a directed path from s to T. (2) Dijkstra's algorithm never retrieves a vertex whose optimal cost is larger than the optimal cost of the problem. (3) Dijkstra's algorithm needs to visit every vertex whose optimal cost is smaller than the optimal cost of the problem. In addition to this, the algorithm retrieves those vertices only one time.*

Let h_1 and h_2 be (inconsistent) admissible heuristic functions. If $h_1(v) \geq h_2(v)$ is satisfied for each vertex v, h_1 *dominates* h_2. If h_1 dominates h_2, and if we ignore tiebreak on the optimal cost, the numbers of visited vertices and retrieved vertices of the A* search algorithm with h_1 are always

equal to or smaller than the numbers with h_2 respectively. This is because the number of the vertices satisfying the condition of lemma 2.7 does not increase by using h_1 instead of h_2. The number of vertex retrievals sometimes increases by using h_1 even if h_1 dominates h_2. However, although we do not have specific bounds, it is empirically known that more accurate admissible heuristic function tends to make fewer number of vertex retrievals. Therefore it is necessary to develop a tight and fast admissible heuristic function to reduce the numbers of visited vertices, retrieved vertices, and vertex retrievals of cost-optimal pathfinding algorithm. Note that making an only tight admissible heuristic function is not difficult at all because the optimal cost from a vertex v to T itself is an ultimately tight cost estimation for v. Hence Dijkstra's algorithm itself can be a tight admissible heuristic function. However, solving many instances of the cost-optimal pathfinding problem to solve another instance is clearly too expensive to compute, and the running time may become too long. Therefore it is necessary to develop a well-balanced admissible heuristic function in terms of running time and tightness of estimations.

In chapter 5, a new admissible heuristic function is proposed. It is based on an integer linear programming model for computing the optimal cost of a delete-free task. The cost-optimal delete-free STRIPS planning problem is known to be NP-hard [18]. Hence the heuristic function computes the optimal costs of the linear programming relaxations of the models. Although the proposed heuristic function is more expensive compared to some previous admissible heuristic functions, the heuristic function is much tighter than the competitors. The A* search algorithm with the proposed heuristic function is competitive to some state-of-the-art planning algorithms. The formal definition of the delete relaxation is given in the next section.

The Weighted A* Search Algorithm Pohl proposed the weighted A* search algorithm in 1970 [81].

The weighted A* search algorithm is an extension of the A* search algorithm. The weighted A* search algorithm requires a parameter $w \geq 1$. Instead of the priority $h(v) + g(v)$ for a vertex v, the weighted A* search algorithm uses $wh(v) + g(v)$ as the priority. If $w = 1$ holds, then the weighted A* search algorithm is completely same as the A* search algorithm.

Although the weighted A* search algorithm does not have optimality when $w > 1$, it has an important property about the cost of the output. By proofs similar to the A* search algorithm, the following lemma can be proved.

Lemma 2.11. *Given a feasible instance of the cost-optimal pathfinding problem, the weighted A* search algorithm with an admissible heuristic function*

and a weight w returns a directed path whose cost is equal to or smaller than wC^*, where C^* is the optimal cost of the instance.

Proof Sketch: The proof is similar to the proof of lemma 2.4. For any vertex v and for any directed path P from s to v, it can be proved that either of the followings is satisfied:

(1) for each vertex u on P, u is in the closed list and is not in the open list. In addition, the cost of u in the closed list is equal to or smaller than $c_P(u)$, or

(2) there exists a vertex u on P in the open list and the priority of u in the open list is equal to or smaller than $wh(u) + c_P(u)$.

In addition to this, for each vertex v on an optimal path P, $wh(v)+c_P(u) \leq wC^*$ holds. Hence, contradiction occurs if the cost of the output is larger than wC^*. □

2.4 The Delete Relaxation of a STRIPS Planning Task

Not only for constructing an admissible heuristic function for pathfinding problems, but also for getting a lower bound of the optimal cost of an instance of an optimization problem, the *relaxation* of an instance is commonly used.

This dissertation focuses on the instances of minimization problems. Given an instance O of a minimization problem and another instance R of possibly another minimization problem, let X_O and X_R be the set of feasible solutions of O and R respectively. In addition let c_O and c_R be the cost function of X_O and X_R respectively. If $X_R \supseteq X_O$ holds, and if $c_R(x) \leq c_O(x)$ holds for any $x \in X_O$, then the instance R is a relaxation of the instance O. A relaxation is usually made by modifying the original instance or removing some constraints of the original problem.

For the STRIPS planning class, *the delete relaxation* is well studied in the literature [89]. The delete relaxation is technically a simplification of a STRIPS planning task, and as a consequence of the delete relaxation, instances of the satisficing and cost-optimal STRIPS planning problems are relaxed.

Given a STRIPS planning task $\mathcal{T} = \langle P, A, I, G \rangle$, the delete relaxation makes another STRIPS planning task $\mathcal{T}^+ = \langle P, A^+, I, G \rangle$, where A^+ is a set of delete-free actions defined as $A^+ = \{\langle \mathrm{pre}(a), \mathrm{add}(a), \emptyset, c(a) \rangle \mid a \in A\}$. As mentioned above, the task $\mathcal{T}^+$ is called a *relaxed task* or a *delete-free task*

in this dissertation. The term delete-free task and the notation $T+$ is also used to denote a task that has no delete effect before the delete relaxation. According to the definition of an applicable action sequence, any plan of $\mathcal{T}$ is a feasible plan of $\mathcal{T}^+$. Clearly the cost of a plan is the same. Hence the instance of the satisficing STRIPS planning problem defined by $\mathcal{T}^+$ is a proper relaxation of the instance of the satisficing STRIPS planning problem defined by $\mathcal{T}$. An instance of the cost-optimal STRIPS planning problem can be relaxed similarly. Note that a plan of a relaxed task is sometimes called a *relaxed plan*.

Table 2.2 shows an example of the delete relaxation of the task on Table 2.1. For example, if the action move_B_table_A is applied to a state { (A on table), (B on table), (clear A), (clear B) }, then the successor state is the state { (A on table), (B on table), (clear A), (clear B), (B on A) }.

Table 2.2: An example of the delete relaxation of a STRIPS planning task.

P	{ (A on B), (A on table), (B on A), (B on table), (clear A), (clear B) }
A	{ move_A_table_B, move_A_B_table, move_B_table_A, move_B_A_table }, where move_A_table_B = $\langle$\{ (A on table), (clear B) \}, \{ (A on B) \}, $\emptyset, 1\rangle$, move_A_B_table = $\langle$\{ (A on B) \}, \{ (A on table), (clear B) \}, $\emptyset, 1\rangle$, move_B_table_A = $\langle$\{ (B on table), (clear A) \}, \{ (B on A) \}, $\emptyset, 1\rangle$, move_B_A_table = $\langle$\{ (B on A) \}, \{ (B on table), (clear A) \}, $\emptyset, 1\rangle$.
I	{ (A on table), (B on table), (clear A), (clear B) }
G	{ (A on B) }

Given a STRIPS planning task $\mathcal{T} = \langle P, A, I, G\rangle$, the heuristic function to compute the optimal cost of $\langle P, A^+, S, G\rangle$ for a state $S \subseteq P$ is denoted as h^+ in the literature (e.g., [9]). This is the heuristic function to compute the optimal costs of relaxed tasks. Note that it technically computes the optimal cost of a task made by the delete relaxation of $\langle P, A, S, G\rangle$. Below $\langle P, A^+, S, G\rangle$ is called *the relaxed task of S*. It seems that the first use of h^+ inside a cost-optimal planning algorithm was by Betz and Helmert [9], who implemented domain-specific implementations of h^+ for several domains. They experimentally showed that h^+ is much tighter than some other heuristic functions. In addition to this, they showed that the running time of the A* search algorithm with domain specific h^+ is faster than some other heuristic functions in some domain.

Unfortunately it is known that the complexity class of the cost-optimal delete-free STRIPS planning problem is NP-hard in the general case [18]. We can equate the cost-optimal delete-free STRIPS planning problem with

the cost-optimal pathfinding problem on a *weighted directed hypergraph*. Of course the latter is also known as a NP-hard problem [2]. Hence researchers developed some admissible heuristic functions to compute a lower bound of h^+ in polynomial time in some previous work. The *max heuristic* [12, 13] and the *landmark cut heuristic* [14] are such examples. In addition to this, there are some heuristic functions based on the delete relaxation for satisficing STRIPS planning problems such as the *additive heuristic* [12, 13] and the *FF heuristic* [57, 59]. These heuristic functions compute feasible plans of relaxed tasks and return those costs. These four heuristic functions are explained in detail in the next chapter.

As mentioned above, an integer linear programming model of the cost-optimal delete-free STRIPS planning problem is proposed in the fifth chapter. Although the complexity class of the integer linear programming problem is known to be NP-hard, we can easily make a relaxation in the complexity class P by linear programming relaxation.

To be more precise, note that some heuristic functions compute the cost of a *partial-ordered plan* of a delete-free task. Although a plan is a sequence of actions in classical planning, a plan is the combination of a set of actions and their *total-ordered* in other word. However, for instance, if both (a_1, a_2, a_3) and (a_1, a_3, a_2) are feasible plan of a task, then it is sufficient for us to have a set of actions $\{a_1, a_2, a_3\}$ and a set of orders $\{(a_1 \to a_2), (a_1 \to a_3)\}$. In general, A partial-ordered plan is the combination of a set of actions and their *partial-ordered*. Although a partial-ordered plan can be easily serializable, precisely speaking, the additive heuristic, the FF heuristic, and the proposed integer-linear programming model compute a partial-ordered plan. In the following, if the term "plan" is just used, then it means a total-ordered plan.

2.5 Planning Domain Definition Language

Planning Domain Definition Language, or PDDL in short, is the de facto standard of formal languages to formulate planning tasks. PDDL is used to describe benchmark instances of AIPS98 [75], and whenever a competition is held, PDDL is extended to describe more and more wider classes of planning tasks [3, 31, 27, 38, 39]. The latest version of PDDL is version 3.1 in 2014. Now PDDL can describe not only a STRIPS planning task, but also PDDL can handle fragments of general classical planning and non-classical planning such as *negations, disjunctive clauses, universal and existential quantifiers, numeric propositions, durative actions*, and so on. PDDL is not directly related to the proposed algorithms. However, STRIPS planning tasks made for the International Planning Competitions are used for the experimental

evaluations in this dissertation, and these tasks are written in PDDL. Hence a part of the definition of PDDL is briefly explained in this section.

In PDDL, a planning task is defined by two data structures called *domain* and *problem* respectively. A domain is semantically a subset of planning tasks such as a set of tasks of blocks world. We need to describe the common structures and properties of a domain such as kinds of objects and actions into a domain data structure. Information of an individual task such as the initial state or the goal is written in a problem data structure. In the International Planning Competitions, a *domain file* is used to describe a domain, and a *problem file* is used to describe a problem. We can share and reuse a domain file in planning tasks on one domain.

PDDL is defined based on list structures like LISP language. Figure 2.1 shows an example of domain files of the blocks world domain. This domain

```
(define (domain blocks-world)
  (:requirements :strips)
  (:predicates (on ?x ?y) (ontable ?x) (clear ?x)
               (handempty) (holding ?x))

  (:action pick-up
   :parameters (?x)
   :precondition (and (clear ?x) (ontable ?x) (handempty))
   :effect (and (not (ontable ?x)) (not (clear ?x))
                (not (handempty)) (holding ?x)))

  (:action put-down
   :parameters (?x)
   :precondition (holding ?x)
   :effect (and (not (holding ?x)) (clear ?x)
                (handempty) (ontable ?x)))

  (:action stack
   :parameters (?x ?y)
   :precondition (and (holding ?x) (clear ?y))
   :effect (and (not (holding ?x)) (not (clear ?y))
                (clear ?x) (handempty) (on ?x ?y)))

  (:action unstack
   :parameters (?x ?y)
   :precondition (and (on ?x ?y) (clear ?x) (handempty))
   :effect (and (holding ?x) (clear ?y) (not (clear ?x))
                (not (handempty)) (not (on ?x ?y)))))
```

Figure 2.1: A domain file of the blocks world domain

data structure is given for the benchmark set of the International Planning Competitions. The tuple `:requirements` describes the requirements of the ability of planners to handle the planning tasks on this domain, and the requirement `:strips` states that the tasks on this domain are based on the model of STRIPS planning tasks. Although the implementations of the proposed algorithms accept the requirements `:typing` and `:action-costs`, explanations about those requirements are omitted here. Essential factors of a domain data structure for STRIPS planning tasks are *predicates* and *action schemata*. A predicate of PDDL is a parameterized proposition. In Figure 2.1, five predicates named `on`, `ontable`, `clear`, `handempty`, and `holding` are defined. We can get actual propositions by substituting *objects* defined in a problem file to arguments of predicates. In this definition, propositions to describe holding a block are used in addition to the example on the Table 2.1. Similarly an action schema is a parameterized action, and four action schemata named `pick-up`, `put-down`, `stack`, and `unstack` are defined in the figure. The tuple `:precondition` describes the precondition of the action schema. The tuple `:effect` describes the combination of the add effect and delete effect of the action schema. The tuple `(not X)` in the effect tuple describes the proposition `X` is an element of the delete effect, and otherwise the proposition is an element of the add effect.

Figure 2.2 shows an example of a problem file of an instance of planning tasks of the blocks world domain. This problem data structure is

```
(define (problem blocks-world-4-0)
  (:domain blocks-world)
  (:objects A B C D)
  (:init (clear A) (clear B) (clear C) (clear D)
         (ontable A) (ontable B) (ontable C) (ontable D)
         (handempty))
  (:goal (and (on D C) (on C B) (on B A))))
```

Figure 2.2: A domain file of the blocks world domain

also quoted from the International Planning Competition benchmarks. In this figure four objects `A`, `B`, `C`, and `D` are defined, and as mentioned above, propositions and actions are implicitly defined by substituting these objects to predicates and action schemata. Note that planners sometimes generate nonsense propositions or actions since they do not understand the meanings of predicates and action schemata. For example, the proposition `(on A A)` or the action `stack(B B)` can be generated. Sometimes we can automatically remove such nonsense elements by using the initial state and the goal.

The tuple `:init` describes the initial state, and the tuple `:goal` describes the goal. In this task four blocks are initially located on the table, and the goal is to stack all blocks by the order `D`, `C`, `B`, and `A` from the top.

Note that, since PDDL is used to describe a planning task, there is no distinction of domain files and problem files between the cost-optimal STRIPS planning problem and the satisficing STRIPS planning problem. Requirement of optimality is decided by users or solvers.

Chapter 3

Related Work

In this chapter, some previous work related to the proposed algorithms is presented. Some algorithms below are used for the experimental evaluations. Related work is described mostly chronologically.

In section 3.1, some kinds of non-pathfinding reductions are introduced. The non-pathfinding reductions in the section 3.1 are proposed in 1990's. Although most of them are not used to directly solve planning problems recently, they are still important to build recent heuristic functions.

After McDermott proposed an algorithm based on heuristic search in 1996, researchers started to study heuristic search algorithms studiously. Some heuristic functions developed from the late 1990 's are mainly based on the delete relaxation. The *max heuristic*, the *additive heuristic*, the *FF heuristic*, and the *landmark cut heuristic* are explained in section 3.2. All of the four heuristic functions are based on the delete relaxation.

In recent years some heuristic functions based on outside of the delete relaxation are proposed. In addition to this, it seems that most of the previous practical integer linear programming heuristic functions are based on non-delete relaxation. In section 3.3, these integer linear programming heuristic functions are introduced. Some other non-delete relaxation based heuristic functions are also surveyed in section 3.4.

Finally, some satisficing heuristic search algorithms are explained in section 3.5. In addition some improvements for heuristic search algorithms are also described in the last section. Those improvements are used in some state-of-the-art planning algorithms.

3.1 Non-Pathfinding Reduction

Affected by the improvements of computers and combinatorial algorithms, reductions from a planning problem to other combinatorial problems blossom in 1990's. Although proposed algorithms are based on the pathfind-

ing reduction, the integer linear programming heuristic function is closely related to non-pathfinding reductions. This section explains some related work with non-pathfinding reductions such as a reduction to the *satisfiability problem*, a reduction to *a problem related to network flow problems*, and a direct reduction to the *integer linear programming problem*.

Kautz and Selman proposed a translation from an instance of a planning problem to a series of instances of the satisfiability problem in 1992 [64]. An encoded instance of the satisfiability problem decides the existence of a feasible plan of a fixed length. Given an integer for the fixed length l, each proposition p on a planning problem is encoded to $(l+1)$ propositional variables $p_0, \cdots, p_l$. A true assignment of the problem corresponds to a feasible plan of length l, and $p_i = true$ holds if and only if the proposition p is true on the i-th step on the plan. Their planner, called *SATPLAN*, searches a plan with the shortest length (i.e., an optimal plan when the cost function is a unit cost function) by solving the encoded problems from $l = 1, 2, \cdots$. Although they used GSAT [91] to solve encoded instances in their approach, of course now we can use any satisfiability problem solver to benefit from advances of research of the satisfiability problem. In recent years satisfiability based planners still have been being improved (e.g., [71, 24]). They still form a portion of the state-of-the-art planning algorithms.

In 1995, Blum and Furst developed a reduction from a partial-ordered STRIPS-like planning problem to a problem to construct a data structure called *planning graph* [10]. They stated that a plan in a planning graph is essentially a flow in the network flow sense. A planning graph has two kinds of alternative layers composed of *proposition nodes* and *action nodes* respectively. Each proposition node corresponds to a proposition of the task. Each action node corresponds to an action of the task. Hence a layer can be considered as a set of propositions or actions. In addition a planning graph has some links between nodes in a same layer. A link describes the mutual exclusion between two propositions or two actions related to nodes. The first layer of a planning graph is a proposition layer with the nodes that correspond to propositions in the initial state. A new action layer is constructed based on the previous proposition layer, and a new proposition layer is constructed based on the previous action layer. If a proposition layer contains the nodes of all goal propositions, and if every pair of the goal nodes does not have a mutual exclusion each other, then the construction of the planning graph finishes. After construction finished, a plan can be extracted by backtracking from the last layer of the planning graph. Although the exact definition of a planning graph and its construction is not explained here, the *relaxed planning graph* of a STRIPS planning task is defined as follows. The relaxed planning graph of a STRIPS planning task is the

planning graph of the relaxed task of the original task made by the delete relaxation. Relax planning graphs are often used for heuristic functions based on the delete relaxation. The first layer of a relaxed planning graph is also a proposition layer composed of the proposition nodes in the initial state. Then i-th action layer is the set of relaxed actions that are applicable to the state with all the propositions appearing on the i-th proposition layer. The $(i+1)$-th proposition layer is the logical disjunction of the i-th proposition layer and all add effects of the actions in the i-th action layer. The construction stops when i-th and $(i+1)$-th proposition layers are the same. A relaxed planning graph does not have any mutual exclusion links. The last proposition layer includes the goal if and only if the delete relaxation has a feasible plan. If the last proposition layer includes the goal, then the combination of the set of all actions appearing in the relaxed planning graph and the partial-order of actions according to the layers is a partial-ordered plan of the delete relaxation.

It seems that the earliest use of integer linear programming in STRIPS planning was by Bylander in 1997, who used a linear programming encoding of unit-cost STRIPS planning problems as heuristic functions [19]. He proposed integer linear programming models for both of the total-ordered satisficing STRIPS planning problem and the partial-ordered satisficing STRIPS planning problem. His encodings also require a positive integer l. One can decide the existence of a total-ordered plan with a fixed length l, and another can decide the existence of a partial-ordered plan with a fixed makespan l. Generally speaking, given an integer for the fixed length l, each proposition p is encoded to $(l+1)$ zero-one integer variables $p_0, \cdots, p_l$. As similar to SATPLAN, a feasible solution of the integer linear programming corresponds to a feasible plan of length l, and $p_i = 1$ holds if and only if the proposition p is true on the i-th step on the plan. Each action a is also encoded to zero-one integer variables $a_0, \cdots, a_{l-1}$. To associate the original instance of STRIPS planning problems and its integer linear programming encoding, some constraints are defined based on the definition of STRIPS planning. For example, an inequality $p_i \geq a_i$ need to be satisfied for each $i \in \{0, \cdots, l-1\}$, a proposition $p \in P$, and an action $a \in A$ that satisfies $p \in \mathrm{pre}(a)$. The objective is to maximize the objective function $\sum_{p \in G} p_l$. There is no feasible plan with length l if there exists no feasible solution on the (integer) linear programming problem. Therefore a heuristic function that searches the minimum length of the feasible encodings is admissible.

In 1999, Vossen, Ball, Lotem and Nau defined another integer linear programming encoding similar to Bylander's. The main difference is that Vossen et al. introduced some auxiliary variables, and the objective of their encoding is to minimize the summation of the costs of used actions [98]. In

2005 Briel and Kambhampati proposed a variable elimination technique for the encoding of Vossen et al. [97]. Their algorithm constructs the relaxed planning graph of the task, and it substitutes zero to useless variables based on the relaxed planning graph.

3.2 Heuristic Functions Based on the Delete Relaxation

According to Russell and Norvig [89], after McDermott proposed the first heuristic search algorithm based on the delete relaxation in 1996 [76], Bonet and Geffner developed the first practical planning algorithm in 1998 [12, 13]. Bonet and Geffner proposed the *additive heuristic* and the *max heuristic*, and they experimentally evaluated the additive heuristic with the *hill-climbing search algorithm* that is explained later. Hoffmann proposed another satisficing STRIPS planning algorithm called *Fast Forward planner* [57, 59]. Fast Forward planner is composed of the *FF heuristic* and the *enforced hill-climbing search algorithm*. It was one of the best planning algorithms of the competition in Artificial Intelligence Planning and Scheduling 2000 [89]. Moreover, the FF heuristic and some other fragments of Fast Forward planner are still used in the current state-of-the-art planning algorithms (e.g., Fast Downward planner [53]). Finally, in contrast to the three heuristics above, the *landmark cut heuristic* is a quite new heuristic function originally proposed in 2009 [54, 14] and also plays an important role in recent planning research.

The Max Heuristic As mentioned above, Bonet and Geffner proposed the max heuristic in 1998. They just expressed that the max heuristic computes an approximation of the optimal cost of the relaxed task of the given state [12]. However, according to Betz and Helmert [9], the max heuristic computes the optimal makespan of partial-ordered plans for the relaxed task of the given state when we regard the cost of an action as the execution time of the action. Hence the cost estimation of the max heuristic is clearly a lower bound of h^+, and the max heuristic is an admissible heuristic function.

The optimal makespan $c(p)$ to achieve a proposition p is naturally defined as $\min\{c(a)+\max_{p'\in \mathrm{pre}(a)} c(p') \mid \exists a \in A, p \in \mathrm{add}(a)\}$ if p is not a member of the initial state. Otherwise $c(p)$ is set to zero. The optimal makespan of the relaxed task is defined as $\max_{p\in G} c(p)$. These makespans can be computed by the algorithm proposed by Knuth [67]. It runs in low-degree polynomial time of the order of the size of the relaxed task of the given state.

It seems that the max heuristic is one of the fastest and loosest nontrivial admissible heuristic functions. It is empirically and experimentally

known that there is a large gap between the max heuristic and h^+ (and hence the optimal cost from the given state to the goal states) [9].

The Additive Heuristic The additive heuristic computes the cost of a feasible plan of the delete relaxation though we can have several views of the additive heuristic as with the max heuristic. As Betz and Helmert have mentioned in their paper [9], the additive heuristic computes the cost of an optimal plan for the relaxed task of the given state under the pessimistic assumption that there are no interactions between the goals and between action preconditions. For example, assume the goal G is defined as $G = \{p_1, p_2\}$. If any plan to achieve p_1 never achieves both of p_2 and preconditions of any plan to achieve p_2, and vice versa, then an optimal plan to achieve G is the concatenation of an optimal plan to achieve p_1 and an optimal plan to achieve p_2.

In general, for each proposition p, the additive heuristic computes the cost $c(p)$ to achieve p under the above assumption. The cost $c(p)$ is defined as $\min\{c(a) + \sum_{p' \in \text{pre}(a)} c(p') \mid \exists a \in A, p \in \text{add(a)}\}$ if $p \notin I$. Otherwise $c(p)$ is set to zero for $p \in I$. Then the additive heuristic returns $\sum_{g \in G} c(g)$ that is equal to the optimal cost to achieve G under the assumption. The additive heuristic can be defined by replacing max operator of the max heuristic by $\sum$. Also as is the case with the max heuristic, these costs can be computed in low-degree polynomial time of the order of the size of the delete relaxation by Knuth's algorithm.

The plan computed by the additive heuristic is a feasible plan for the relaxed task of the given state without the assumption. Hence the cost estimation of the additive heuristic is always equal to or larger than the optimal cost of the relaxed task. In addition to this, it sometimes larger than the optimal cost of the original task of the given state, hence it is not an admissible heuristic function.

The FF Heuristic The FF heuristic also computes the cost of a feasible plan for the relaxed task of the given state. First it constructs the relaxed planning graph of the task of the given state, and after that, it extracts a cheap feasible plan instead of the trivial feasible plan, i.e., all the actions appearing in the relaxed planning graph. Although some variants are proposed for handling non-unit cost [33] or numerical state [58] and so on, the original FF heuristic is defined on the satisficing STRIPS planning problem with a unit cost function. Here only the original version is explained.

Let $l(p)$ be the first layer that the proposition p appears on the relaxed planning graph. Then the algorithm recursively appends actions into the output taking interactions of actions of into account. To be more precise,

the FF heuristic defines the sets $G_i = \{g \in G \mid l(g) = i\}$ before extracting. These sets control interactions of the actions. The extracting algorithm proceeds down from the maximum layer to the layer zero. The goal is to make all G_i empty. If a proposition p is in the current G_i, the algorithm selects an action a that satisfies $p \in \text{add}(a)$. Then the algorithm removes all the propositions in $\text{add}(a)$ from G_i, inserts each proposition $q \in \text{pre}(a)$ into $G_{l(q)}$, and continues to repeat this process until all G_i become empty.

The plan computed by the FF heuristic is obviously a feasible plan for the relaxed task of the given state. However, there is no guarantee to be an optimal plan. Hence the FF heuristic is not an admissible heuristic function.

The Landmark Cut Heuristic In recent years a *landmark* plays an important role in planning research community [63, 54, 86, 65]. The landmark cut heuristic, proposed by Helmert and Domshlak in 2009 [54], is a variant of landmark based heuristic functions.

In planning, a landmark is something that is always required, used, or achieved in every feasible plan. For example, to pick up the block A is an *action landmark* in Figure 1.1. We cannot reach the goal unless moving the block A. A *disjunctive action landmark* is a set of actions in which every feasible plan contains at least one element. It is known that deciding whether an action set L is a disjunctive action landmark or not in general STRIPS planning tasks is PSPACE-hard [84]. However we can easily decide for delete-free STRIPS planning tasks in linear time since deciding plan existence can be solved in linear time. Finding a disjunctive action landmark is also not so difficult for delete-free tasks.

The landmark cut heuristic iteratively finds disjunctive action landmarks by using the max heuristic as a subroutine. It returns the summation of the minimum costs of actions of each disjunctive action landmark. Roughly speaking, the landmark cut heuristic divides and distributes the cost of an action to the disjunctive action landmarks if the action is contained in more than one disjunctive action landmarks. Hence the estimation is always equal to or lower than the optimal cost of the relaxed task of the given state, and therefore the landmark cut heuristic is an admissible heuristic function.

3.3 Heuristic Functions Based on the Integer Linear Programming Encodings

In this section, recent advances in integer linear programming heuristic functions are explained.

Although integer linear programming models in the section 3.1 can also be used to admissible heuristic functions, the first practical admissible heuris-

tic function for optimal planning seems to be proposed by van den Briel, Benton, Kambhampati, and Vossen in 2007 [96]. All of the previous models require the parameter of the plan length. This parameter produces too large instances of the integer linear programming problem to use as a heuristic function. In addition to this, we need to know the length of an optimal plan before computation, or we need to minimize the costs computed from all different parameters. It seems that van den Briel et al. proposed the first compact model. Subsequent researches also followed this kind of compact encodings.

The model proposed by van den Briel et al. is based on a *SAS+ planning task* [5]. Unlike a STRIPS planning task, each state of a SAS+ planning task is *assignments* of *multi-valued variables*, and the effect of an action is defined as new assignments of variables. For example, in a logistic problem described as SAS+ planning, the state is composed of variables for the positions of vehicles, packages, and drivers. An action to move a truck t from a position p_1 to another position p_2 changes the assignment of the variable v_t from $v_t = p_1$ to $v_t = p_2$. The value of the variable v_t cannot be p_1 and p_2 simultaneously. In other words, multi-valued variables naturally describe mutual exclusions of values.

In general, a SAS+ planning task is defined by a 5-tuple $\langle V, d, A, s_I, s_G \rangle$. V is the set of multi-valued variables, and d is the mapping from a variable to the range of the variable. A function s that assigns a value from $d(v) \cup \{\bot\}$ for each variable v is called a *partial assignment*, and a function from v to a value in $d(v)$ is called a *total assignment* or a *state*. The special value $\bot$ means *undefined*. The total assignment s_I is the *initial state*. The partial assignment s_G is the *goal*. Each action a is a 3-tuple $(p(a), e(a), c(a))$. The partial assignments $p(a)$ and $e(a)$ are called the *precondition* and the *effect* of the action a, and $c(a)$ is the cost of a. An action a is *applicable* to a state s if and only if $p(a)(v) = s(v)$ or $p(a)(v) = \bot$ holds for each variable v, and the new state s' is defined as $s'(v) = e(a)(v)$ if $e(a)(v) \neq \bot$ and otherwise $s'(v) = s(v)$ for each variable v. A plan and its cost are defined as same as a STRIPS planning task.

We can also reduce a SAS+ planning problem to a pathfinding problem on a weighted directed graph. There exists trivial polynomial time translations between STRIPS planning tasks and SAS+ planning tasks. Hence the complexity class of the plan existence problem of SAS+ planning tasks is also PSPACE-complete. Searching a compact reduction is a practical and challenging problem. For instance, Helmert proposed an efficient translation from a STRIPS planning task to a SAS+ planning task with taking mutual exclusions of propositions into account [52].

Van den Briel et al. defined an integer variable x_a for the number of occurrence of an action a. The objective is to minimize $\sum_{a \in A} c(a)x_a$. They defined constraints like network flow such as $\sum_{b \in A_i(v,f)} x_b = \sum_{a \in A_o(v,f)} x_a$ for each value f of a variable v, where $A_i(v, f)$ is the set of actions that change the value of the variable v from another value to f, and $A_o(v, f)$ is the opposite[1]. They call this kind of constraints *effect implication constraints*. Van den Briel et al. also proposed some other constraints based on *domain transition graphs* of the variables. The domain transition graph of a variable v is a directed graph $\langle V, E \rangle$, where V is the set of the values of v, i.e., $V = d(v)$. There exists a directed edge e with $s(e) = v_s \in V$ and $e(e) = v_e \in V$ if there exists an action a and it satisfies $p(a)(v) = v_s$ and $e(a)(v) = v_e$. The details of those constraints are omitted here. We can perceive the effect implication constraints as flow constraints on the domain transition graphs.

In 2013 Bonet proposed another integer linear programming based admissible heuristic function called *state equation heuristic* [11]. He proposed a transformation from a SAS+ planning task to a Petri Net, and formalized a linear programming encoding of the Petri net. Bonet's constraints are sometimes called *net change constraints* [83]. The net change constraints have some similarity to the effect implication constraints. Recently Bonet and van den Briel proposed another heuristic function made by combining and improving these two constraints [15].

Pommerening, Röger, Helmert, and Bonet [83] formalized and combined some linear programming constraints and other techniques as a generalized framework called *operator-counting constraints*. They experimentally showed the performances of admissible heuristic functions based on some combinations of the operator-counting constraints with the A* search algorithm. Especially the combination of the net change constraints and the constraints made from disjunctive action landmarks computed from the same algorithm with the landmark cut heuristic [14] is competitive to a state-of-the-art algorithm.

3.4 Other Heuristic Functions

Here heuristic functions related to neither the delete relaxation nor integer linear programming are briefly introduced.

The Causal Graph Heuristic and The Context Enhanced Additive Heuristic Helmert proposed the *causal graph heuristic* in 2004 [52, 53].

[1] This equation is simplified from the original one for a brief explanation.

The causal graph heuristic is defined on a SAS+ planning task, and it computes its estimation by the domain transition graphs of the variables and the *causal graph* [66] of the task.

The causal graph of a SAS+ planning task is a directed graph $\langle V, E\rangle$, where V is the set of the variables of the task. There exists a directed edge e with $s(e) = v_1$ and $e(e) = v_2$ if and only if there exists an action a that satisfies (1) $p(a)(v_1) \neq \perp$ and $e(a)(v_2) \neq \perp$, or (2) $e(a)(v_1) \neq \perp$ and $e(a)(v_2) \neq \perp$. Intuitively the causal graph describes the dependencies of the variables.

The causal graph also plays an important role in planning not only to develop heuristic functions, but also to determine tractable subclasses of planning problems. For example, if the causal graph of a SAS+ planning task is a directed tree and satisfies some other conditions, a feasible plan of the SAS+ planning task can be computed in polynomial time [16]. On the other hand, if the causal graph is a complex cyclic graph, then even determining the existence of a feasible plan is PSPACE-complete. The causal graph heuristic modifies and relaxes the given SAS+ planning task based on its causal graph and domain transition graphs, and then it computes the optimal cost of the modified planning task in polynomial time. It does not have any guarantee to be an admissible heuristic function.

In 2008 Helmert and Geffner reformulated the causal graph heuristic as *the additive heuristic with context*, i.e., interactions between subgoals [55]. In addition they proposed another heuristic function based on additive heuristic with another context called *context enhanced additive heuristic.* The context enhanced additive heuristic experimentally outperformed both additive heuristic and the causal graph heuristic in terms of the coverage of benchmark instances, search time, and the number of vertex expansions, i.e, the number of vertex retrievals.

Pattern Database Heuristic The *pattern database heuristic* is an admissible heuristic function originally proposed for puzzles in 90's [35, 22]. Imagine that some tiles on a sliding puzzle are painted with black. You cannot see the number on a black tile. In the initialization phase before the search, the pattern database heuristic computes and stores the numbers of moves of optimal solutions for all the states of the black-painted puzzle. Since the pattern database heuristic does not distinguish the black tiles, the number of states of the black-painted puzzle is much smaller than the number of states of the original puzzle. Moreover, a brute force algorithm such as the breadth first search algorithm can incrementally compute the optimal costs for all the states. Given a state of the original puzzle for estimation, the pattern database heuristic paints the same tiles of the state by black

as the initialization phase. Then it searches the black-painted state from the database and just returns the optimal cost from the black-painted state to the goal. The black-painted puzzle is clearly a relaxation of the original instance. Hence the pattern database heuristic is an admissible heuristic function.

The pattern database heuristic was introduced to planning community by Edelkamp in 2001 [26]. Although the detailed definition of pattern database for planning is not explained here, there are some variants of the pattern database heuristic for planning such as [47] since there are many possibilities of abstractions. All the pattern database heuristics for planning are admissible as far as the author knows.

Merge-and-Shrink Heuristic The *merge-and-shrink heuristic* computes the optimal cost of another abstracted planning problem [56]. Given a SAS+ planning task, we can generate another equivalent SAS+ planning task by merging some variables of the original task by the Cartesian product method. When the number of variables becomes one by repeating merging, the domain transition graph of the variable is completely isomorphic to the search space of the original SAS+ planning task. On the other hand, a variable shrinks by contracting two values, i.e., vertices, of the domain transition graph of the variable. By repeating merging variables and shrinking variables, we can have a smaller instance of the cost-optimal pathfinding problem with a lower optimal cost. The optimal cost of this instance can be used for an admissible heuristic function for the original instance.

Some variants of merging and shrinking strategies are proposed. The *bisimulation merge-and-shrink heuristic* [80] is one of the state-of-the-art heuristic functions.

3.5 Satisficing Search Algorithms

In this section, some heuristic search algorithms and some enhancements for heuristic search algorithms for the satisficing pathfinding problem are explained.

As shown in detail later, the greedy best first search algorithm has a problem that it consumes large amount of its running time to explore useless area because of inaccuracy of heuristic functions. Some of the following algorithms were proposed to conquer this problem. The common idea of the following algorithms and the proposed search algorithm of this dissertation is *to diversify search directions*. The experimental evaluations compared the proposed search algorithm with such previous algorithms in the next chapter.

Some other algorithms converge search directions to reduce the number of estimations in contrast to the algorithms above. Some of the others were proposed to reduce the number of cost estimations for states (i.e., vertices) since estimations of planning heuristic functions are normally expensive.

Anytime search algorithms are developed to handle time-quality trade off for problems such as real time planning problems. To find a better solution in a given time limit, they repeat non-optimal search after a solution is found.

The k Best First Search Algorithm The *k best first search algorithm* with an integer parameter k is a generalization of the greedy best first search algorithm [28]. This algorithm was proposed by Felner, Kraus, and Korf in 2003. This algorithm uses an open list defined as same as the greedy best first search algorithm. Intuitively the k best first search algorithm simultaneously expands the k best vertices in the open list at a time, and then inserts all the successors of the k vertices into the open list. It repeats this process until finding an end vertex or detecting that there is no solution. We can also implement the k best first search algorithm on the framework in the previous chapter.

The k best first search algorithm and an extended version were applied to STRIPS planning by Linares López and Borrajo in 2010 [73]. The enhanced version incorporates two enhancements called *goal agenda* [68] and *helpful actions* [59]. Helpful actions are explained later. Lopez et. al. [73] concluded that in solving hard problems both the k best first search algorithm and the enhanced version outperform the greedy best first search algorithm and the enforced hill-climbing search algorithm that is also explained later.

The k best first search algorithm partially avoids search plateaus caused by misleading heuristic estimates. However, if the heuristic values of all the k best vertices are erroneously underestimated, it results in searching useless areas. Although increasing k can reduce the possibility of occurring this drawback, the k best first search algorithm tends to behave similarly to the breadth first search algorithm. Thus increasing k loses the benefit from the heuristic information.

Using More Than One Heuristic Function *The alternation method* uses more than one heuristic function for a search algorithm. Röger and Helmert investigated experimental performances of some multiple heuristic methods in 2010 [88] although alternation itself was already used in Fast Downward planner [53].

In the original definition, it manages an open list for each heuristic function, and selects one of the open lists in a round-robin manner in each vertex

retrieval. Successors of the retrieved vertex are inserted to all the open lists with their own estimations. We can also implement alternation method on the previous framework a little forcibly. Alternation diversifies search directions by expecting at least one of heuristic functions evaluates each vertex precisely. However, if all of the heuristic functions inaccurately evaluate unpromising vertices as promising, it suffers from an excessive overhead of expanding valueless vertices.

Dovetailing method is based on a more general idea than alternation. This was proposed also in 2010 by Valenzano, Sturtevant, Schaeffer, Buro and Kishimoto [95]. Dovetailing method independently runs different search algorithms such as the weighted A* search algorithm with various weights. Although it is easily parallelizable, dovetailing runs those algorithms on a single processor. On one step, it selects a search algorithm in a round-robin manner, then the selected algorithm expands one vertex based on its own open list and closed list (and its other data structure). After that, dovetailing selects another search algorithm. It repeats this process until one of the search algorithms finds an end vertex.

Originally Valenzano et al. proposed dovetailing for automatic parameter tuning. They experimentally showed that dovetailing on the weighted A* search algorithm with some parameters outperforms the weighted A* search algorithm with one single best weight on a puzzle domain. Namely, just running simultaneously some algorithms with undeliberated parameters outperforms an algorithm with a tuned parameter. However, it can be used to combine some search algorithms with different heuristics, and it essentially has a similar dilemma to alternation.

Random walk One of other approaches for adding diversity to search algorithms is using random walks. While the following examples of this approach can avoid visiting unpromising area caused by misleading of heuristic estimations, they may suffer from duplicate search effort such as re-expanding the same vertices via different paths many times.

Coles, Fox, and Smith proposed a planner called Identidem in 2007 [20]. They introduced for Identidem a heuristic search algorithm exploring graphs by *stochastic local search*, i.e., random walk. *Local search* is a kind of search manner. In local search, a vertex for expanding next is selected from the successors of the vertex expanded in the current iteration. The search algorithm of Identidem runs some local searches from the start vertex. If a vertex with a strictly lower estimation than the start vertex is found, then the search algorithm runs another set of local searches from the found vertex. The search algorithm repeats this process until it finds an end vertex. In local searches, the search algorithm probabilistically selects one of suc-

cessors of the current vertex. The search algorithm uses a heuristic function for determining probabilities of the successor vertices. This search algorithm does not have completeness.

Nakhost and Müller proposed a planner called ARVAND in 2009 [78]. ARVAND also uses local search for their heuristic search algorithm. Although the search algorithm of ARVAND is similar to the search algorithm of Identidem, one difference is that a new start vertex of local searches is selected after all the current local searches finish. In addition, a new start vertex is selected only from the end vertices of local searches that have the lowest heuristic estimation. The other difference is that the search algorithm of ARVAND does not use heuristic estimations in the middle of local searches. One variant of ARVAND uses Monte Carlo random walk for local searches, i.e., a selection of a next vertex is uniformly at random. Another variant of ARVAND uses for a selection *preferred operators* that are explained in the following subsection. Nakhost *et al.* concluded that "the method is robust in presence of misleading heuristic estimates, it obtains more information from the local neighborhood". However, this search algorithm also does not have completeness.

The Hill-climbing Search Algorithm The *Hill-climbing search algorithm*, in the broad sense of the word, is another framework of a kind of local search-based heuristic search algorithm. After the hill-climbing search algorithm visits a vertex v, the candidates to expand next are the successors of v that (1) have the smaller estimated cost than $h(v)$ and (2) have the smallest estimated cost among the successors. If there exists no candidate on the step, then some instantiations restart new search from the start vertex with randomized tiebreak, or some others restart from the next most promising vertex in the current successors or in the previous search. Sometimes the algorithm without the first condition of candidates is also called hill-climbing search algorithm.

In planning, Bonet and Geffner used hill-climbing search algorithm with additive heuristic function for Heuristic Search Planner, although they used the greedy best first search algorithm for Heuristic Search Planner 2 [13]. It does not use the first condition of candidates. However it counts the number of the moves without decrease of estimations, and it restarts if the number exceeds a predefined limit. It restarts new search from an unexpanded vertex found during the search. Ties are broken at random.

Hoffmann and Nebel used an algorithm called the *enforced hill-climbing search algorithm* for the Fast Forward planner [59]. The enforced hill-climbing search algorithm repeats breadth first search algorithm with cost estimations by a heuristic function (the FF heuristic for the Fast Forward

planner). It first runs breadth first search algorithm from the start vertex. If a vertex with a lower cost than the start vertex is found, then it selects the vertex as a new start vertex and runs another breadth first search algorithm from the new start vertex. The enforced hill-climbing search algorithm is similar to the search algorithm of Identidem. This search algorithm uses the breadth first search algorithm instead of random walks.

If the enforced hill-climbing search algorithm has visited all the vertices that can be reached from the current start vertex, and an end vertex is not found yet, then the Fast Forward planner throws everything away and starts the greedy best first search algorithm with the FF heuristic.

Preferred Operators A *preferred operator* for a state S is an action (i.e., an operator) that can be applied to the state S and is deemed promising to reach goal states. Although it seems that there is no technical definition *what* a preferred operator is as same as heuristic estimations, Helmert defined *how* to use preferred operators for his Fast Downward planner [53]. Preferred operator technique originates from the use of *helpful actions* that are proposed for the Fast Forward planner [59]. Helmert proposed a *helpful transition*, and he used helpful actions and helpful transitions as preferred operators for Fast Downward. The set of helpful actions of a state S is the intersection of applicable actions to S and the relaxed plan computed by the FF heuristic for the cost evaluation of S. Similarly, helpful transitions of S is the intersection of applicable actions to S and the relaxed plan computed by the causal graph heuristic for the cost evaluation of S.

The use of preferred operators in Fast Downward is similar to the alternation method. The search algorithm of Fast Downward maintains two open lists for a heuristic function. One is the all successor open list, and another is the preferred successor open list. When a state S is expanded, a preferred successor (a successor of S that can be reached by a preferred operator of S) is inserted into only the preferred successor open list. A normal successor is inserted into both open lists. Given a parameter b for state expansions, the search algorithm of Fast Downward deterministically selects the normal open list at a rate of one time per b times of selections of the preferred open list. When use of preferred operators and alternation method for some (say, m) heuristic functions are combined, Fast Downward maintains $2m$ open lists and handles them with the parameter b as similar to the case of one heuristic function. The experiments in the chapter 4 and many other experiments in previous work indicate that the use of preferred operators has a heavy impact to the performances of algorithms even if b is equal to one.

On the one hand, helpful actions and helpful transitions of a state S

are computed when the state S is evaluated. On the other hand, preferred operators of S are necessary when S is retrieved. However it is not necessary to pay extra cost to store preferred operators when we use *deferred evaluation* that is explained in the next subsection.

Deferred Evaluation Deferred evaluation is also proposed as one fragment of Fast Downward planner. Let h be a heuristic function. Instead of estimating the costs of all the successors of a state S for the expansion of S, deferred evaluation uses $h(S)$ as the evaluations of all the (unevaluated) successors of S. This technique reduces the number and the time of cost estimations per one state expansion, however it is empirically known that deferred evaluation increases the number of state expansions. Richter and Helmert investigated effects of deferred evaluation experimentally [85]. They concluded the numbers of solved instances are not so different in algorithms with/out deferred evaluation, however use of deferred evaluation find a little more expensive solution faster compared to disuse of deferred evaluation.

Deferred evaluation has synergy with helpful actions and helpful transitions since computing $h(S)$ for the estimations of the successors of S detects helpful actions and helpful transitions at the same time.

Anytime Search Algorithm *Anytime search algorithms* are developed to handle time-quality trade-off for problems such as real time planning problems.

Although we do not have a clear separation of the complexity classes between general satisficing planning and cost-optimal planning, it is empirically known that the running time of a cost-optimal algorithm is much slower than a satisficing algorithm even for the same planning task. In addition to this, it is also empirically known that parametric algorithms with approximation performance guarantee such as the weighted A* search algorithm have trade-offs between the quality of solutions and running time. Anytime search algorithm is a general kind of algorithm that finds a solution as quickly as possible first and then repeats another search to improve the quality of the solution until end of the time limit.

There are some anytime search algorithms in literature (e.g., [72]). Although the details of these algorithms are not explained here, note that LAMA planner also adopts an anytime search algorithm. LAMA shares some fragments with Fast Downward planner and is implemented on the Fast Downward system [86]. After the first feasible plan is found by an algorithm similar to Fast Downward (the greedy best first search algorithm with alternation of the FF heuristic and landmark count heuristic, preferred operators, and deferred evaluation), LAMA clears vertices in the open list

and closed list, and then starts another search by the weighted A* search algorithm with alternation, preferred operators, and preferred operators. If the search found the second solution, then LAMA runs another search with a smaller weight and repeats this process until the time limit. There is no theoretical guarantee of the approximation performance of the weighted A* based algorithm because of some enhancements. However, the experimental evaluations in [86] showed that LAMA found better solutions in the weighted A* based search phases.

Chapter 4

Diverse Best First Search

The greedy best first search algorithm is a popular and effective algorithm in satisficing planning and is incorporated into high-performance planners. However, if a heuristic function evaluates states inaccurately, greedy best first search may be misled into a valueless search direction, thus resulting in performance degradation. In the first section of this chapter an example of the case that a heuristic function leads greedy best first search to an unpromising area of the search space is shown. In the second section, a new heuristic search algorithm is proposed. It considers diversity of search directions to avoid unpromising area misled by the errors of heuristic information. The experimental results in the third section show that the proposed approach is successful.

4.1 Greedy Best First Search with an Inaccurate Heuristic Function

In satisficing planning, many planning algorithms employ heuristic search strategies including greedy best first search (e.g., [13, 53]). Let h be a heuristic function that estimates the optimal cost from a state to a goal state. As defined in the previous chapters, for each iteration of the loop, the greedy best first search algorithm retrieves the best state S with the smallest $h(S)$ in the open list. It then generates successors of S, and inserts these successors into the open list and into the closed list, unless they have been previously visited. The greedy best first search algorithm continues to repeat this process until finding a goal state or visiting all the states that can be reached from the initial state.

A heuristic function plays an important role in drastically improving performance of the greedy best first search algorithm. While heuristic functions (e.g., [59, 53]) enable state-of-the-art satisficing planners to solve

complicated instances of the satisficing STRIPS planning problem including benchmark instances in the International Planning Competitions, accurate evaluations of states still remain as a challenging task.

Although the greedy best first search algorithm is a fundamental and powerful heuristic search algorithm in STRIPS planning, it has an essential drawback when the heuristic function returns inaccurate cost estimations. Assume that a heuristic function underestimates the difficulties of unpromising states. Then, the greedy best first search algorithm spends much running time in searching only unpromising areas, and it delays moving to promising parts of the search space. This is caused by the greediness of the greedy best first search algorithm, i.e., it has to expand states with small heuristic values first.

Figure 4.1 illustrates a typical transition of the priorities of states selected for expansions (i.e., a transition of the estimations of retrieved states). The FF heuristic is used for the estimations. The STRIPS planning task is an instance of the optical-telegraphs domain of the International Planning Competition. The horizontal axis indicates each expansion of the best state

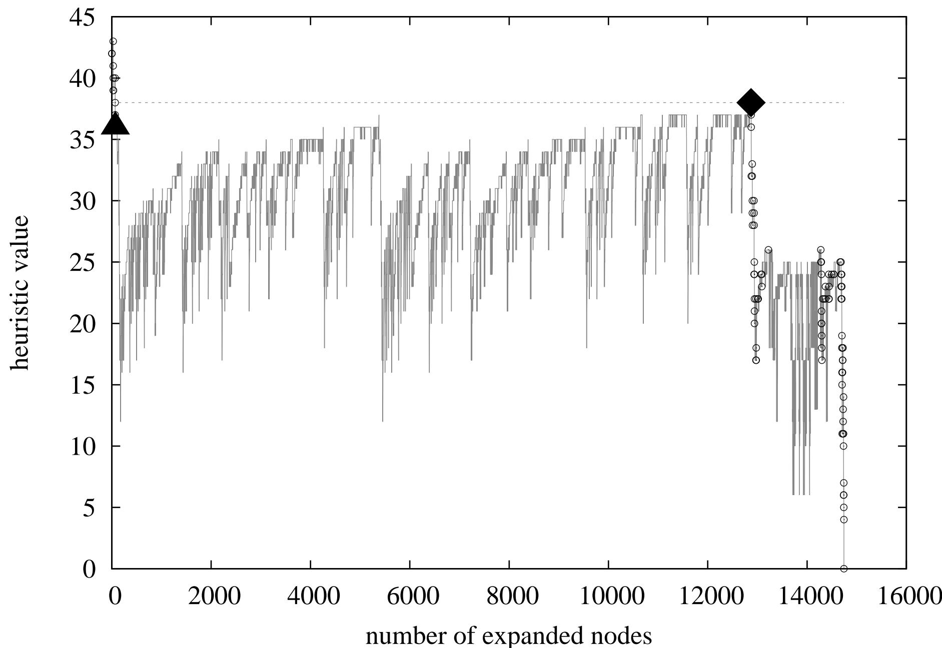

Figure 4.1: A transition of heuristic values in solving optical-telegraphs #02 in fourth International Planning Competition

S in the open list. The vertical axis represents S's corresponding heuristic value for that expansion (i.e., the evaluation of S by the FF heuristic).

Circles, the triangle, and the diamond represent expanded states that are on the feasible plan found by the algorithm. This transition indicates that until retrieving a state D marked by the diamond, the greedy best first search algorithm keeps expanding many useless states erroneously evaluated as more promising than D by the heuristic function, after expanding a state T marked by the triangle. According to the figure, these useless states never contribute to finding the goal state. Noteworthily all these useless states are successors or descendants of the state T since it contradicts greediness of the algorithm if they were members in the open list before D was expanded. In addition D is also a successor of T since D is the successor of T on the output plan. Hence the greedy best first search algorithm did not search the useless area if the FF heuristic evaluated the successors of T other than D as unpromising.

As shown in the third chapter, some previous work tackled this issue by adding *diversity* to search algorithm,which is an ability in exploring different parts of the search space to bypass large unpromising area misled by errors of heuristic functions. For example, several algorithms combined with diversity such as the k best first search algorithm [28, 73] and the alternation method [88] are experimentally shown to be superior to the naive greedy best first search algorithm. However, they still have limited diversity, since they do not immediately expand states mistakenly evaluated as very unpromising ones.

This chapter presents a new heuristic search algorithm that incorporates diversity into search in a different way than previous approaches. The contribution is summarized as:

1. The *diverse best first search* algorithm that is robust to large heuristic evaluation errors. This search algorithm stochastically goes towards various directions by probabilities computed from estimations of a heuristic function. Even if a heuristic function erroneously evaluates promising state S as unpromising, the diverse best first search algorithm can occasionally expand S. The frequency of selecting such S is controlled by the heuristic value of S and the cost of the plan to S kept in the closed list.

2. Experimental results clearly showing that the diverse best first search algorithm is effective in satisficing planning. The diverse best first search algorithm outperforms the greedy best first search algorithm and the k best first search algorithm. Additionally, by combining with several enhancement technique, the diverse best first search algorithm solves more planning instances than the Fast Downward planner [53] and an enhanced version of the k best first search algorithm [73].

4.2 Definition of the Diverse Best First Search Algorithm

The diverse best first search algorithm overcomes issues addressed in the last section. As similar to the search algorithm of Identidem [20], the diverse best first search algorithm diversifies search directions by probabilistically selecting a state that does not have the best heuristic value. As a result, compared to the k best first search algorithm and the alternation method, the diverse best first search algorithm has a higher chance of expanding a state mistakenly estimated to be unpromising. Additionally, unlike the approaches of Identidem or ARVAND [78], the diverse best first search algorithm performs more systematic search by keeping all the expanded states in the closed list. It can therefore effectively reuse search results such as the case of which there are many paths to the same state. Hence, the diverse best first search algorithm is a complete search algorithm.

Algorithms 2 and 3 show the pseudo code of the diverse best first search algorithm. Although the algorithm can be defined based on the framework described in the preliminary chapter a little forcibly, here the proposed algorithm is defined as Algorithm 2 and 3 to understand easily. The main routine of the diverse best first search algorithm described by Algorithm 2 is quite simple. Until finding a goal state or reaching all the states, it repeats the procedures of fetching one state S from the global open list (OL in the pseudo code) and performing greedy best first search rooted at S with the local open list (LocOL in the pseudo code). The diverse best first search algorithm optimistically expects greedy best first search to find a solution for S with the smallest search effort. Thus the number of states expanded per greedy best first search is therefore limited to $h(S)$, which is the minimum number of states that must be expanded to find a goal state with the unit edge cost if $h(S)$ does not overestimate the distance to the goal. Tied heuristic values are broken randomly in greedy best first search. Duplicate search effort is eliminated by the shared global closed list. After greedy best first search expands $h(S)$ states, all the states in the local open list are inserted to the global open list to make these states as candidates for a selection in the next state-fetching phase.

Algorithm 3 presents the procedure of fetching a state for greedy best first search, which is called at line 3 of Algorithm 2. Let $g(S)$ be the cost of a state S in the global closed list. The costs $h(S)$ and $g(S)$ are called h-value and g-value respectively. A state S selected to perform greedy best first search is determined by a probability computed by $h(S)$ and $g(S)$ a little ad hoc. If more than one state have the same pair of h- and g-values, one of them is chosen randomly.

Algorithm 2 Diverse Best First Search

1: insert the initial state into OL;
2: **while** OL is not empty **do**
3: s := fetch a state from OL;
4: LocOL := $\{s\}$;
5: /* Perform greedy best first search rooted at s */
6: **for** i:=1 to $h(s)$ **do**
7: retrieve a state m with the smallest $h(m)$ from LocOL;
8: **if** m is a goal **then**
9: return plan to m from the root;
10: **end if**
11: save m into the global closed list;
12: expand m and save the successors of m into LocOL;
13: **end for**
14: OL: = LocOL $\cup$ OL;
15: **end while**
16: return "no solution";

Algorithm 3 Fetching one state

1: $p_{total} := 0$;
2: (h_{min}, h_{max}) := minimum and maximum h-values in OL;
3: (g_{min}, g_{max}) := minimum and maximum g-values in OL;
4: **if** with probability of P **then**
5: G := select at random from $g_{min}, \cdots, g_{max}$;
6: **else**
7: $G := g_{max}$;
8: **end if**
9: **for all** $h \in \{h_{min}, \cdots, h_{max}\}$ **do**
10: **for all** $g \in \{g_{min}, \cdots, g_{max}\}$ **do**
11: **if** $g > G$ or OL has no state whose h-value and g-value are h and g respectively **then**
12: $p[h][g] := 0$;
13: **else**
14: $p[h][g] := T^{h-h_{min}}$;
15: **end if**
16: $p_{total} := p_{total} + p[h][g]$;
17: **end for**
18: **end for**
19: select a pair of h and g with probability of $p[h][g]/p_{total}$;
20: dequeue a state s with $h(s) = h$ and $g(s) = g$ in OL;
21: return s;

Parameters P and T $(0 \leq P, T \leq 1)$ decide a policy of fetching the next state. When greedy best first search is used for global search, it tends to select states with large g-values due to greediness of repeatedly selecting a successor that appears to be promising. The parameter P enables the diverse best first search algorithm to restart exploring the search space that is closer to the initial state, where the diverse best first search algorithm has not yet exploited enough to find the promising states. The parameter T controls the frequency of selecting a state s based on the gap between the current best h-value and $h(s)$. Lower probabilities are assigned to states with larger h-values to balance exploiting the promising search space and exploring the unpromising part. On the one hand, heuristic estimates are completely ignored if $T = 1$ holds. On the other hand, the diverse best first search algorithm fetches the same state chosen by greedy best first search if $T = 0$ and $P = 0$ hold.

As mentioned above, if greedy best first search selects an unpromising state S, and if a lot of descendants of S have smaller h-values than $h(S)$, the greedy best first search algorithm keeps searching unpromising descendants of S. However, even if the diverse best first search algorithm fetches S, it expands $h(S)$ states and then selects another state that may not be a descendant of S. Only at most $b \cdot h(S)$ states are inserted to OL where b is the largest number of edges of the $h(S)$ states. This number is much smaller than that of the greedy best first search algorithm, since the greedy best first search algorithm must store all the useless descendants.

Combination with some enhancements To compare the proposed search algorithm to some practical planning algorithms, a variant of the diverse best first search algorithm is also defined by combining preferred operators. As similar to the use of preferred operators in the greedy best first search algorithm in Fast Downward, the variant of the diverse best first search algorithm uses an additional global open list that is prepared separately for preferred successors. The variant selects one global open list uniformly at random for each state-fetching. After the algorithm selects one global open list, the state-fetching algorithm selects a state by Algorithm 3, and then the algorithm performs greedy best first search rooted at that state.

4.3 Experimental Evaluations

Setup

The performance of the diverse best first search algorithm was evaluated by running experiments on solving 1,612 tasks of the satisficing STRIPS

planning problem in 32 domains from the first through fifth International Planning Competitions. All the experiments in this chapter were run on a dual quad-core 2.33 GHz Xeon E5410 machine with 6 MB L2 cache. The time and memory limits for solving an instance were set to 30 minutes and 2 GB. If an algorithm solved an instance without violating these limits, then the instance is labeled “solved” by the algorithm. All the implementations are built on top of the Fast Downward planner [53] to use the causal graph heuristic and the context enhanced additive heuristic and to compare with Fast Downward. On Fast Downward, the FF heuristic is also already implemented on the trivial reduction from a SAS+ planning task to a STRIPS planning task. Fast Downward preprocesses the translation from the PDDL representation [27] into the SAS+ representation [5]. The translation times were excluded in the experiments. In the benchmark instances used for these evaluations, the costs of all actions in the all problem are exactly one. Hence the cost of a plan is the same as the length of the plan.

Performance Comparisons without Enhancements

The first evaluation analyzes strengths and weaknesses of the diverse best first search algorithm (DBFS), the greedy best first search algorithm (GBFS), and the k best first search algorithm (KBFS). Enhancements in Fast Downward (e.g., preferred operators and multiple heuristic functions) were disabled, thus measuring the potential of each search algorithm. The FF heuristic [59], the causal graph heuristic (CG) [53], and the context enhanced additive heuristic (CEA) [55] are used for the evaluations. These heuristic functions are already implemented in Fast Downward. The best known random seed for the diverse best first search algorithm was used for each heuristic function. However, the diverse best first search algorithm solved all the instances with the same seed and with $P = 0.1$ and $T = 0.5$, and did not exploit the best seed for each instance.

Table 4.1 shows the number of solved instances when all the algorithms used the FF heuristic. As mentioned above, the k best first search algorithm requires an integer parameter k. In this experiment, the k best first search algorithm was run with the parameter 2^l for all the cases of integer l satisfying $0 \leq l \leq 7$. The table shows the best result of the k best first search algorithm for each domain.

Table 4.1: The number of instances solved by each algorithm with the FF heuristic and without enhancements

Domain	**GBFS**	**KBFS**	**DBFS**
Airport (50)	33	44	**46**
Assembly (30)	18	27	**30**
Blocks (35)	**35**	**35**	**35**
Depot (22)	16	17	**19**
Driverlog (20)	18	**20**	**20**
Freecell (80)	**80**	**80**	**80**
Grid (5)	**5**	**5**	**5**
Gripper (20)	**20**	**20**	**20**
Logistics 1998 (35)	30	31	**33**
Logistics 2000 (28)	**28**	**28**	**28**
Miconic (150)	**150**	**150**	**150**
Miconic Full ADL (150)	135	137	**139**
Miconic Simple ADL (150)	**150**	**150**	**150**
Movie (30)	**30**	**30**	**30**
MPrime (35)	26	27	**33**
Mystery (30)	16	17	**19**
Openstacks (30)	28	28	**30**
Optical Telegraphs (48)	3	3	**5**
Pathways (30)	9	16	**30**
Philosophers (48)	**48**	**48**	**48**
Pipesworld Notankage (50)	31	37	**44**
Pipesworld Tankage (50)	24	25	**35**
PSR Large (50)	31	31	**32**
PSR Middle (50)	**50**	**50**	**50**
PSR Small (50)	**50**	**50**	**50**
Rovers (40)	27	28	**37**
Satellite (36)	25	26	**28**
Schedule (150)	18	46	**129**
Storage (30)	19	21	**25**
TPP (30)	22	23	**29**
Trucks (30)	14	18	**22**
Zenotravel (20)	**20**	**20**	**20**
Total (1612)	1,209	1,288	**1,451**

Table 4.1 clearly indicates the superiority of the diverse best first search algorithm to the k best first search algorithm and the greedy best first search algorithm. The diverse best first search algorithm either solved an equal or larger number of instances than the others in all the domains. In particular, the diverse best first search algorithm performed much better in the Schedule domain. Of 150 instances, the diverse best first search algorithm solved 129 problems while the greedy best first search algorithm and the k best first search algorithm solved only 18 and 46 instances respectively. However,

even if this domain is excluded, the diverse best first search algorithm was still able to solve at least 80 additional instances in total compared with the other approaches. Hence, this results imply the importance of diversifying search directions.

The k best first search algorithm solved additional instances in several domains compared to the greedy best first search algorithm such as Airport, Assembly, Pathways and Schedule. However, the k best first search algorithm usually achieved smaller performance improvements than the diverse best first search algorithm. Additionally, it is observed that selecting the best parameter k played an important role in improving its solving ability, although selecting such k automatically remains an open question. For example, in the Philosophers domain, while the k best first search algorithm with the parameter $k = 1$ (i.e., identical to the greedy best first search algorithm) solved all 48 instances, the k best first search algorithm with $k = 128$ was able to solve only 25 instances.

Figure 4.2 compares the number of states expanded by the greedy best first search algorithm and the diverse best first search algorithm with the FF heuristic for the instances solved by both. The state expansion of greedy best first search was plotted on the horizontal axis against the diverse best first search algorithm on the vertical axis on logarithmic scales. Thus a point below the linear line indicates that diverse best first search expanded fewer states than the greedy best first search algorithm in solving one instance. Figure 4.2 clearly shows that the diverse best first search algorithm outperformed the greedy best first search algorithm especially when solving hard instances. Of 1,208 instances solved by both, it took either the diverse best first search algorithm or the greedy best first search algorithm at least one second to solve each of 279 instances. Of these 279 instances, the diverse best first search algorithm expanded fewer states than the greedy best first search algorithm in solving 209 instances. This resulted in a large difference in search time (see Figure 4.3 comparing the search time with the FF heuristic for the instances solved by both). The diverse best first search algorithm solved 194 instances more quickly and was five times faster than the greedy best first search algorithm in solving the aforementioned 279 instances. The overhead of the diverse best first search algorithm fetching a state did not offset its benefit of achieving drastic reductions of state expansions. In fact, the state expansion rate per second of the diverse best first search algorithm was similar to that of the greedy best first search algorithm. The figures showing performance comparisons between the diverse best first search algorithm and the k best first search algorithm are omitted, since similar results were obtained.

Figure 4.4 compares the quality of plans (i.e., solution lengths) computed

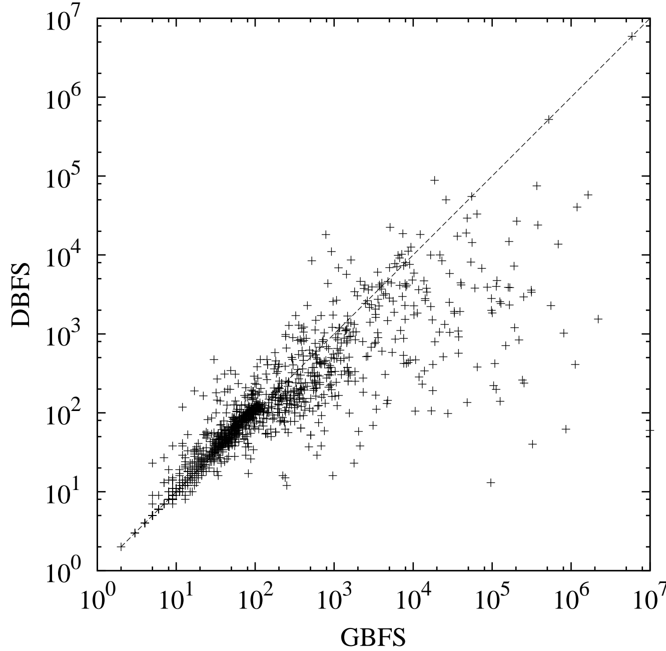

Figure 4.2: Comparison of state expansions between the greedy best first search algorithm and the diverse best first search algorithm with the FF heuristic

by the diverse best first search algorithm and the greedy best first search algorithm with the FF heuristic. The diverse best first search algorithm often returned longer solutions than the greedy best first search algorithm, since the diverse best first search algorithm selected unpromising states that tend to be on a more redundant path to a goal. This phenomenon was similarly observed in ARVAND [78], when their planner was compared against Fast Downward in a few domains. However, since the diverse best first search algorithm still yielded similar plans in many cases, this is a price to pay for achieving performance improvements.

Table 4.2: The number of instances solved by each algorithm with the CG/CEA heuristic and without enhancements

Heuristic	**GBFS**	**KBFS**	**DBFS**
CG	1,170	1,218	**1,358**
CEA	1,202	1,240	**1,388**

Table 4.2 shows the total number of solved instances in all domains with the causal graph heuristic or the context enhanced additive heuristic. The

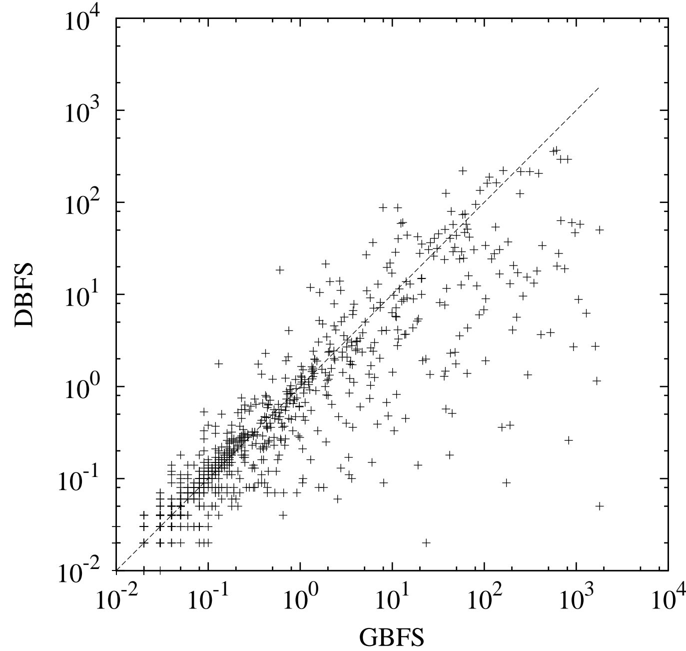

Figure 4.3: Search time for instances solved by the diverse best first search algorithm and the greedy best first search algorithm with the FF heuristic

numbers are calculated in the same way as in Table 4.1. The superiority of the diverse best first search algorithm was confirmed even with different heuristics. The diverse best first search algorithm performed worse than the others only in a few domains.

Performance Comparisons with Various Parameters and Random Seeds

Table 4.3: Performance of the diverse best first search algorithm with different parameters

Heuristic	Average	Minimum	Maximum
FF	1,438	1,432	1,451
CG	1,345	1,335	1,361
CEA	1,370	1,358	1,388

Next, parameters P and T are varied in the range of $0.1-0.3$ and $0.4-0.6$ respectively, in increments of 0.1 (i.e., the nine combinations of parameters are evaluated for each heuristic function). Table 4.3 shows the average, min-

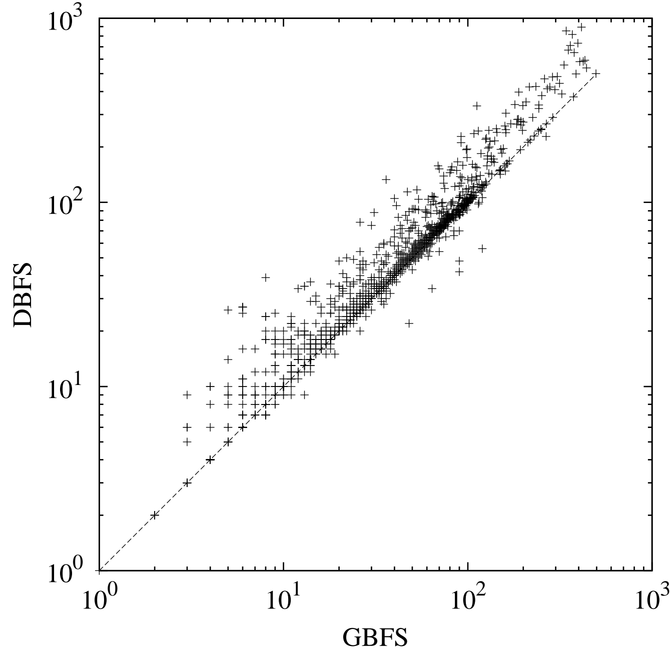

Figure 4.4: Comparison of plan lengths for instances solved by the greedy best first search algorithm and the diverse best first search algorithm with the FF heuristic

imum, and maximum numbers of solved instances. The same random seed was used for each pair of parameters. Results show that the diverse best first search algorithm outperformed both the greedy best first search algorithm and the k best first search algorithm by a large margin with any of the three heuristic functions and even with the worst parameter settings. While the differences between the minimum and maximum numbers of solved instances were 30 with the context enhanced additive heuristic, the diverse best first search algorithm was still able to solve most of the instances. While the best value of T depended on heuristic functions and P, it is observed that performance tended to deteriorate with a larger value of P. All of the worst case scenarios shown in Table 4.3 were obtained with $P = 0.3$.

Table 4.4 shows number of solved instances when either P or T is fixed to zero and the other parameter is varied to show the behavior of the diverse best first search algorithm with extremely ineffective parameter settings. No heuristic information is used to diversify search directions with $T = 0$, while only h-values are considered for diversity with $P = 0$. It is not surprising to observe performance degradation compared to the case where P and T are non-zero, since valuable information (i.e., h-values or g-values) is

Table 4.4: Performance of DBFS with resetting one parameter

Heuristic	**FF**	**CG**	**CEA**
$P = 0.1$ (and $T = 0$)	1,366	1,299	1,309
$P = 0.2$	1,362	1,296	1,306
$P = 0.3$	1,358	1,278	1,313
$T = 0.4$ (and $P = 0$)	1,422	1,338	1,381
$T = 0.5$	1,432	1,336	1,377
$T = 0.6$	1,433	1,321	1,364

unused. However, the diverse best first search algorithm still outperformed the greedy best first search algorithm and the k best first search algorithm with all experimented parameter settings, clearly indicating the importance of escaping from search plateaus.

Table 4.5: Performance of the diverse best first search algorithm with different random seeds

Heuristic	Average	Minimum	Maximum
FF	1,447	1,443	1,451
CG	1,353	1,349	1,358
CEA	1,381	1,377	1,388

Table 4.5 shows the performance of the diverse best first search algorithm with different random seeds for each heuristic. Five random seeds with fixed parameters $P = 0.1$ and $T = 0.5$ are examined in the experiment. This table clearly shows that the diverse best first search algorithm was robust to the change of random seeds. Almost all the instances remained solvable even if the seeds are changed. For example, only 11 instances became unsolvable when the best seed was changed to the worst one with the context enhanced additive heuristic.

Performance Comparisons with an Enhancement

Next, the performance of each algorithm was evaluated with turning on enhancements. Table 4.6 shows the number of instances solved by the following algorithms:

EKBFS : The k best first search algorithm with the FF heuristic, enhanced with preferred operators [53] (a.k.a. helpful actions in [59]). As in the paper of López et al. [73], this EKBFS implementation first expands only preferred successors, and then performs the k best first search for the other successors. However, unlike in [73], goal agenda was not incorporated, because it was not implemented in Fast Downward.

Additionally, as it was done in the previous subsection, after running EKBFS with the parameter 2^l for all the cases of integer l satisfying $0 \leq l \leq 7$, the total number is calculated based on the best result in each domain.

FD : The state-of-the-art Fast Downward planner with four enhancements (alternation based on the FF heuristic and the context enhanced additive heuristic, deferred evaluation, preferred operators with the boosting parameter $b = 1000$ [53, 85]). In the preliminary experiments, all combinations of alternation among the FF, context enhanced additive and causal graph heuristics, and the other three enhancements were tried, and the configuration with the best solving ability was chosen.

DBFS2 : As defined above, the diverse best first search algorithm enhanced with preferred operators, and modified as follows: An additional global open list was prepared separately for preferred successors. After the algorithm selects one global open list uniformly at random, it fetches a state by Algorithm 3, and then performs greedy best first search rooted at that state. DBFS2 uses the FF heuristic, and the parameter $P = 0.1$ and $T = 0.5$ were used.

Table 4.6: The number of instances solved by each algorithm with turning on enhancements

	EKBFS	**FD**	**DBFS2**
Total (1612)	1,382	1,458	**1,481**

Despite a smaller number of enhancements currently incorporated into DBFS2 than FD, DBFS2 solved the largest number of instances, showing the superiority of the proposed approach. The performance difference between DBFS2 and EKBFS became smaller than in Table 4.1. This was mainly due to the increased number of instances solved in the Schedule domain. With the help of preferred operators, EKBFS solved 142 instances in this domain, while KBFS did only 46 of 150 instances. However, DBFS2 still outperformed EKBFS by a large margin. It seems to exploit the promising search space that is orthogonal to preferred operators.

Figure 4.5 compares state expansions solved by both FD and DBFS2. DBFS2 drastically reduced state expansions compared to FD. Of 1,445 instances solved by both, DBFS2 expanded fewer states than FD in solving 1,106 instances and was 1.5 times faster in solving the 1,445 instances.

Figure 4.6 shows a comparison of plan lengths between FD and DBFS2. Compared to Figure 4.4, it is observed a smaller difference in the quality of

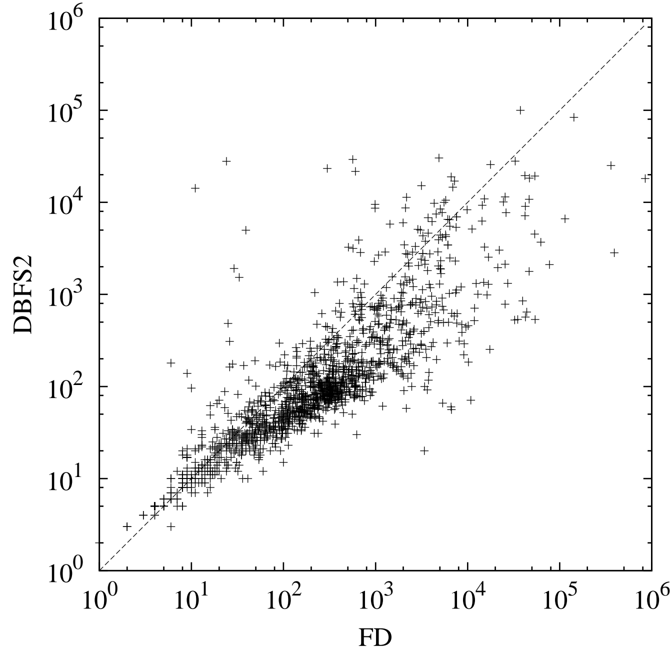

Figure 4.5: Comparison of state expansions between FD and DBFS2

plans, because preferred operators contributed to improving the plan quality of DBFS2 and Fast Downward often returned longer plans than GBFS.

Performance Comparison to LAMA

The LAMA planner is a variant of Fast Downward and returns the better quality of plans by first performing greedy best first based search and then refining plans with a series of weighted A* based search that gradually decreases weight values until it reaches a time limit [86]. The *landmark count heuristic* and the FF heuristic are used in LAMA with various enhancements similar to Fast Downward. One way to combine the proposed approach with LAMA is to replace the first phase of greedy best first based search by DBFS2.

Since the first search phase determines the solving ability, LAMA with DBFS2 solved 1,481 instances as in Table 4.6. On the other hand, LAMA solved 1,445 instances[1].

[1] A main culprit obtaining a smaller number than Fast Downward would be due to a difference in heuristics between LAMA and Fast Downward (landmark count versus CEA). Although this number could be increased to 1,458 by replacing the first search phase by Fast Downward, LAMA with DBFS2 still performed better.

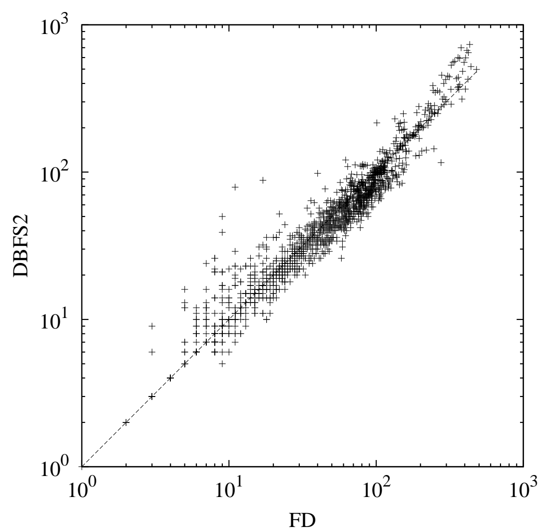

Figure 4.6: Comparison of plan lengths between FD and DBFS2

Figure 4.7 compares plan lengths for the instances solved by the original LAMA and LAMA with DBFS2. The plan quality was mostly similar between these methods. Of 1,438 instances solved by both, LAMA with DBFS2 returned plans with the same lengths as LAMA in 1,310 instances. This indicates that plans can be later refined by LAMA's weighted A* search while DBFS2 can improve its solving ability.

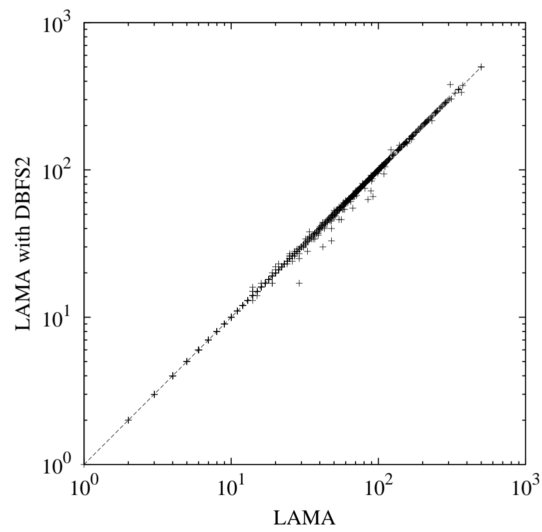

Figure 4.7: Comparison of plan lengths between LAMA and LAMA with DBFS2

Chapter 5

Integer Programming Model of the Delete Relaxation

This chapter proposes a new integer linear programming model for the cost-optimal STRIPS planning problem of delete-free tasks. In the first section, an overview of the proposed integer linear programming model is briefly given. The integer programming model is defined formally in the second section, and some enhancement techniques for the model are proposed from the third section to the fifth section. In the last section of this chapter the experimental evaluations are shown.

5.1 Overview

As described in the previous chapters, the relaxed task by the delete relaxation of a STRIPS planning task is a modification of a STRIPS planning task such that all deletions are eliminated from its operators. It is clear that h^+, the heuristic function to compute the optimal cost of the relaxed task of the given state, is an admissible heuristic function.

In cost-optimal STRIPS planning, h^+ is known to be more accurate than commonly used heuristics such as the merge-and-shrink heuristic [56] or the landmark cut heuristic [54]. It seems that the first use of h^+ inside a cost-optimal STRIPS planner was by Betz and Helmert [9]. They implemented domain-specific implementations of h^+ that run polynomial times for several domains. Betz et al. showed that, in several domain, h^+ is more accurate than the merge-and-shrink heuristic and the max heuristic. In addition, it was also shown that the A* search algorithm with h^+ solved a larger number of instances compared to those two heuristic functions. However, current planners do not directly use a domain independent implementation of h^+. This is because the extra search efficiency gained from using h^+ is

offset or degenerated by the high cost of computing h^+. As mentioned in the preliminary chapter, it is known that the cost-optimal delete-free STRIPS planning problem is NP-hard in fact [18]. Haslum evaluated the use of a domain-independent algorithm for the cost-optimal delete-free STRIPS planning problem [48] as the heuristic function h^+ for the general case of the cost-optimal STRIPS planning problem. He found that the performance was relatively poor [45]. In recent years, there have been several advances in the solvers for the cost-optimal delete-free STRIPS planning problem [37, 82, 48].

In this chapter, a new integer linear programming approach to computing h^+ is proposed.

While a straightforward proposed model, $\mathrm{IP}(T^+)$, for the cost-optimal STRIPS planning problems for delete-free tasks is often intractable and not useful in practice, an enhanced model, $\mathrm{IP}^{\mathrm{e}}(T^+)$, is also developed. $\mathrm{IP}^{\mathrm{e}}(T^+)$ incorporates landmark constraints for the delete relaxation, as well as relevance analysis to significantly decrease the number of variables. It is shown that $\mathrm{IP}^{\mathrm{e}}(T^+)$ allows significantly faster computation compared to the state of the art.

Then, the use of h^+ as a heuristic function for the A* search algorithm is considered. The integer linear programming model $\mathrm{IP}^{\mathrm{e}}(T^+)$ is further augmented with constraints that consider some delete effects, resulting in a new admissible heuristic, $\mathrm{IP}^{\mathrm{e}}(T)$, which sometimes dominates h^+. Since $\mathrm{IP}^{\mathrm{e}}(T^+)$ and $\mathrm{IP}^{\mathrm{e}}(T)$ are integer linear programming models, their linear programming relaxations, $\mathrm{LP}^{\mathrm{e}}(T^+)$ and $\mathrm{LP}^{\mathrm{e}}(T)$, are also admissible heuristic functions for the cost-optimal STRIPS planning problem. Even though $\mathrm{LP}^{\mathrm{e}}(T^+)$ and $\mathrm{LP}^{\mathrm{e}}(T)$ can be quite expensive, the integer linear programming model can be further relaxed by omitting a subset of its constraints, resulting in $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^+)$ and $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$, an linear programming relaxation for a "relaxed" computation of h^+.

The integer linear programming models and their linear programming relaxations are experimentally evaluated by embedding them as heuristics in the A* search algorithm. In addition, a simple method is implemented for automatically selecting which linear programming formulation among $\mathrm{LP}^{\mathrm{e}}(T^+)$, $\mathrm{LP}^{\mathrm{e}}(T)$, $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^+)$, and $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$ to use as the heuristic function, based on a comparison of their values at the initial state. The A* search algorithm with the automated heuristic selection performs comparably to the state-of-the-art cost-optimal planners, Fast Downward with the landmark cut heuristic [54] and Fast Downward using the hybrid bisimulation merge-and-shrink heuristic [80].

5.2 Basic Model

In this section the cost-optimal STRIPS planning problem of a delete free task $T^+ = \langle P, A^+, I, G\rangle$ is formulated as the integer linear program problem. IP(T^+) denotes the integer linear problem derived from T^+, and similarly LP(T^+) denotes the linear programming relaxation of IP(T^+). As shown later, we can derive a feasible solution of IP(T^+) from any feasible and non-redundant (i.e., same actions appear only once) plan of T^+. In addition to this, we can also derive a feasible (and non-redundant) plan of T^+ from any feasible solution of IP(T^+). A feasible and non-redundant plan of T^+ has the same cost as its corresponding feasible solution of IP(T^+).

First, the variables of IP(T^+) are defined as Table 5.1. This table shows a list of the ranges and roles of the variables $\mathcal{U}(p), \mathcal{U}(a), \mathcal{E}(a,p), \mathcal{T}(p), \mathcal{T}(a)$, and $\mathcal{I}(p)$. Their roles are equivalent to the assignments for deriving a feasible solution of IP(T^+) from a feasible and non-redundant plan $\pi = (a_0, \cdots, a_n)$ of T^+. If a proposition $p \in P$ appears more than once in the add effects of

Table 5.1: The definition of the variables

kind	range and assignment
lemma	$\forall p \in P, \mathcal{U}(p) \in \{0,1\}$. $\mathcal{U}(p) = 1$ if and only if $p \in I(\pi)$.
action	$\forall a \in A^+, \mathcal{U}(a) \in \{0,1\}$. $\mathcal{U}(a) = 1$ if and only if $a \in \pi$ holds.
add effect	$\forall a \in A^+, \forall p \in \text{add}(a), \mathcal{E}(a,p) \in \{0,1\}$. $\mathcal{E}(a,p) = 1$ if and only if $a \in \pi$ holds and a achieves p first.
time (proposition)	$\forall p \in P, \mathcal{T}(p) \in \{0, \cdots, \lvert A^+\rvert\}$. $\mathcal{T}(p) = t$ when $p \in I(\pi)$ and p is added by a_{t-1} first. $\mathcal{T}(p) = 0$ otherwise.
time (action)	$\forall a \in A^+, \mathcal{T}(a) \in \{0, \cdots, \lvert A^+\rvert - 1\}$. $\mathcal{T}(a) = t$ when $a = a_t$. $\mathcal{T}(a) = \lvert A^+\rvert - 1$ when $a \notin \pi$.
initial proposition	$\forall p \in P, \mathcal{I}(p) \in \{0,1\}$. $\mathcal{I}(p) = 1$ if and only if $p \in I$.

the actions of a feasible plan, use the index of the first action that achieves p for $\mathcal{T}(p)$ and $\mathcal{E}(a,p)$. Variables $\mathcal{I}(p)$ are auxiliary variables for computing h^+. Although they are redundant when solving an instance of the cost-optimal STRIPS planning problem for a delete-free task only one time, they are useful to avoid reconstructing constraints for each state when IP(T^+) or LP(T^+) are embedded as a heuristic function in a forward-search planner and called for each state.

The objective function seeks to minimize $\sum_{a \in A^+} c(a)\mathcal{U}(a)$. Because of this objective function, the cost of a solution is equal to the cost of the corresponding delete-free plan.

Finally the following six constraints are defined.

1. $\forall p \in G,\ \mathcal{U}(p) = 1$.
2. $\forall a \in A^+,\ \forall p \in \text{pre}(a), \mathcal{U}(p) \geq \mathcal{U}(a)$.
3. $\forall a \in A^+,\ \forall p \in \text{add}(a), \mathcal{U}(a) \geq \mathcal{E}(a,p)$.
4. $\forall p \in P,\ \mathcal{I}(p) + \sum_{a \in A^+ \text{ s.t.} p \in \text{add}(a)} \mathcal{E}(a,p) \geq \mathcal{U}(p)$.
5. $\forall a \in A^+,\ \forall p \in \text{pre}(a),\ \mathcal{T}(p) \leq \mathcal{T}(a)$.
6. $\forall a \in A^+,\ \forall p \in \text{add}(a), \mathcal{T}(a) + 1 \leq \mathcal{T}(p) + (|A^+| + 1)(1 - \mathcal{E}(a,p))$.

The assignments on the table 5.1 clearly satisfy these constraints. Hence there always exists a injective mapping from a feasible non-redundant plan to a feasible solution of IP(T^+).

There exists a feasible plan only if IP(T^+) has a feasible solution. When IP(T^+) is solved optimally, an optimal plan for T^+ is obtained according to the following lemma. For a variable $\mathcal{V}$ of IP(T^+), $\mathcal{V}_F$ describes the assignment of $\mathcal{V}$ on a solution F of IP(T^+).

Lemma 5.1. *Given a feasible solution F for* IP(T^+)*, the action sequence obtained by ordering actions in the set $\{a \mid \mathcal{U}(a)_F = 1\}$ in ascending order of $\mathcal{T}(a)_F$ is a feasible plan for T^+.*

Proof: Let π be the action sequence defined in the statement.

At first it is shown that π satisfies condition (ii) of a feasible plan (i.e., $G \subseteq I(\pi)$) by proof of contradiction. Assume that there exists a proposition $g \in G$ that satisfies $g \notin I(\pi)$. There exists no action achieving g in π according to the assumption. Since F is a solution of IP(T^+), $\mathcal{U}(g)_F = 1$ holds according constraint 1. Since $g \notin I(\pi)$ deduces $g \notin I$, $\mathcal{I}(g)_F = 0$. Therefore, to satisfy condition 4, there must exist an action $a \in A^+$ that satisfies $g \in \text{add}(a)$ and $\mathcal{E}(a,g)_F = 1$. However, to satisfy constraint 3, $\mathcal{U}(a)_F = 1$ has to hold. This means $a \in \pi$, and this contradicts the assumption.

Next it is shown that π satisfies condition (i) (i.e., $\forall i, \text{pre}(a_i) \subseteq I((a_0, \cdots, a_{i-1}))$). For the base case of inductive proof, assume that there exists a proposition $p \in P$ satisfying $p \in \text{pre}(a_0)$ and $p \notin I$. Since $a_0 \in \pi, \mathcal{U}(a_0)_F = 1$ has to hold, and $\mathcal{U}(p)_F = 1$ has to hold according to the constraint $\mathcal{U}(p)_F \geq \mathcal{U}(a_0)_F$. Then, similar to the proof of condition (ii), there must exist an action $a \in A^+$ that satisfies $p \in \text{add}(a)$, $\mathcal{U}(a)_F = 1$, and $\mathcal{E}(a,p)_F = 1$. However, to satisfy constraint 5, $\mathcal{T}(p) \leq \mathcal{T}(a_0)$ has to be true, and $\mathcal{T}(a) + 1 \leq \mathcal{T}(p)$ has to hold to satisfy condition 6. Therefore we have $\mathcal{U}(a)_F = 1$ and $\mathcal{T}(a) < \mathcal{T}(a_0)$, but a_0 is the first action of π, a contradiction.

Similar to the case of $i = 0$, when $i > 0$, if $\mathrm{pre}(a_i) \subseteq I((a_0, \cdots, a_{i-1}))$ is not true, there must exist an action $a \notin (a_0, \cdots, a_{i-1})$ that satisfies $\mathcal{U}(a)_F = 1$ and $\mathcal{T}(a) < \mathcal{T}(a_i)$, contradicting the fact that a_i is the i-th action of the sequence π. □

Corollary 5.2. *Given a feasible delete-free task T^+, the optimal cost of* ILP(T^+) *is equal to the optimal cost of T^+. In addition to this, given an optimal solution F of* ILP(T^+)*, a sequence of actions made by ordering actions in the set $\{a \mid \mathcal{U}(a)_F = 1\}$ by ascending order of $\mathcal{T}(a)_F$ is a optimal plan of T^+.*

5.3 Enhanced Constraints and Variable Eliminations

In this section, some variable elimination techniques and some modifications of constraints are introduced. As shown in the experimental results, these enhancements significantly reduce the time to solve IP(T^+) and LP(T^+). Some of the enhancements are adopted into the proposed model from previous work in planning research. In particular, landmarks, which have been extensively studied in recent years, play very important role.

Note that while some of the enhancements introduce cuts that render some solutions of IP(T^+) mapped from feasible plans infeasible, *at least one optimal plan will always remain.*

Landmark Extraction and Substitution As mentioned in the previous chapters, a *landmark* is an element which needs to be used in every feasible solution. In this dissertation, two kinds of landmarks, that are called *fact landmarks* and *action landmarks* as in [37], are used. Given a STRIPS planning task $\langle P, A, I, G \rangle$, a fact landmark of a set of propositions $P' \subseteq P$ is a proposition that becomes true on some state of every feasible plan that achieves P', and an action landmark of P' is an action that is included in every feasible plan to achieve P'. In the proposed model for a delete-free task T^+, if a proposition p is a fact landmark of G, then the variable $\mathcal{U}(p)$ can be eliminated by substituting $\mathcal{U}(p) = 1$. Similarly, if an action a is an action landmark of G, then we can substitute $\mathcal{U}(a) = 1$. The landmark extraction and substitution clearly do not cut any solutions of IP(T^+) since they are landmarks.

The above definitions are intentional definitions. Hence there could exist a variety of landmark extracting methods. In this dissertation, a set of fact landmarks of $\{p\}$ for each proposition p is computed by an iterative method based on the following Bellman equations of fact landmarks:

- If p is a member of the initial state, then $\{p\}$ is the set of fact landmarks to achieve $\{p\}$.
- If p is not a member of the initial state, then the set of fact landmarks of $\{p\}$ is $\{p\} \cup \bigcap_{p \in \mathrm{add}(a)}(\mathrm{add}(a) \cup \bigcup_{p' \in \mathrm{pre}(a)}(\text{fact landmarks of } p'))$.

In the initialization phase of the iterative method, P is (virtually) set to the first candidates of fact landmarks of $\{p\}$ for each proposition $p \notin I$. In addition, the set of fact landmarks of $\{p\}$ for each member $p \in I$ is set to $\{p\}$, and p is inserted into a FIFO queue. On the main loop of the iterative method, a proposition p is retrieved from the FIFO queue, and the candidate of the set of fact landmarks is updated for each $p' \in \{\exists a \in A^+, p' \in \mathrm{add}(a) \mid p \in \mathrm{pre}(a)\}$ based on the second equation. Moreover, p' is inserted into the FIFO queue if the candidate for p' is changed from the previous one and if p' is not a member of the queue. This process continues until the queue becomes empty. Updating one candidate always reduces the number of elements of the candidate. Hence this method always halts. The correctness of this method is guaranteed by the following lemma.

Lemma 5.3. *Given a delete-free STRIPS planning task $\langle P, A^+, I, G\rangle$, assume any proposition of P can be achieved. Let $L(p)$ be the set of fact landmarks of $\{p\}$ computed by any landmark extracting method. If*

(i) *$L(p) = \{p\}$ for $p \in I$, and*

(ii) *$L(p) = \{p\} \cup \bigcap_{p \in \mathrm{add}(a)}(\mathrm{add}(a) \cup \bigcup_{p' \in \mathrm{pre}(a)} L(p'))$ for $p \notin I$*

are satisfied, then all elements of $L(p)$ are fact landmarks of $\{p\}$.

Proof: Assume that a proposition q satisfies $q \in L(p)$ and q is not a fact landmark of $\{p\}$. From condition (i), (ii) and the definition of a fact landmark, clearly $p \neq q$, $p \notin I$, and $q \notin I$ hold. Then, according to the definition of a fact landmark, there exists a (non-empty) feasible plan of a delete-free task $\langle P, A^+, I, \{p\}\rangle$ that does not achieve q. Let $\pi = (a_0, \cdots, a_n)$ be such a plan, and let a_i be the action in π that achieves p first.

According to condition (ii), the fact that π does not achieve q, and the fact that $p \neq q$ and $q \in L(p)$ hold, we have $q \in \bigcup_{p' \in \mathrm{pre}(a_i)} L(p')$. Let p' be a member of $\mathrm{pre}(a_i)$ that satisfies $q \in L(p')$. Because of condition (i) and the fact that $q \notin I$ holds, we have $q \neq p'$ and $p' \notin I$. Then there exists at least one action in $(a_0, \cdots, a_{i-1})$ that achieves p'. Let a_j be the first action of $(a_0, \cdots, a_{i-1})$ that achieves p'. Again, according to condition (ii), $q \in \bigcup_{p'' \in \mathrm{pre}(a_j)} L(p'')$ is satisfied, and there exists an action a_k in $(a_0, \cdots, a_{j-1})$ that satisfies $q \in \bigcup_{p''' \in \mathrm{pre}(a_k)} L(p''')$. Repetition of this argument continues infinitely. However, the length of π is clearly finite. Hence all members of $L(p)$ are fact landmarks of $\{p\}$ for each proposition $p \in P$. □

In addition to the above fact landmarks, if a proposition p is a fact landmark of G, and if only one action a can achieves p, then a is used as an action landmark of G in the proposed algorithm.

Zhu et al. defined a kind of fact landmark called *causal landmark* [104]. Keyder et al. proposed an AND-OR graph based landmark extracting method for a kind of landmark generalized from causal landmarks [65]. The above algorithm in this dissertation is a generalization of the method of Keyder et al.

Relevance Analysis Backchaining relevance analysis is widely used to eliminate irrelevant propositions and actions of a task. An action a is relevant if (i) $\text{add}(a) \cap G \neq \emptyset$, or (ii) there exists a relevant action a' satisfying $\text{add}(a) \cap \text{pre}(a') \neq \emptyset$. A proposition p is relevant if (i) $p \in G$, or (ii) there exists a relevant action a and $p \in \text{pre}(a)$ holds. In addition to this, as Haslum et al. noted, it is sufficient to consider relevance with respect to only a subset of first achievers of add effect [48]. Although they defined a first achiever by achievability of a proposition, it is completely equivalent to the following definition: an action a is a first achiever of a proposition p if $p \in \text{add}(a)$ and p is not a fact landmark of $\text{pre}(a)$. When the set $\text{fadd}(a)$ is used to denote $\{p \in \text{add}(a) \mid a \text{ is a first achiever of } p\}$, it is sufficient to use fadd instead of add on the above definition of relevance.

If $a \in A^+$ or $p \in P$ is not relevant, we can eliminate a variable as $\mathcal{U}(a) = 0$ or $\mathcal{U}(p) = 0$. In addition to this, if $p \in \text{add}(a)$ but a is not a first achiever of p, we can eliminate a variable as $\mathcal{E}(a, p) = 0$. This variable elimination cuts some feasible solutions. It however does not cut any optimal solutions.

Dominated Action Elimination On a delete-free task, if two actions have same add effect, then it is clearly sufficient to use at most one of two actions. Here a technique that eliminates an useless action (*dominated action*) is introduced by extending this idea.

Lemma 5.4. *Given a feasible delete-free task T^+, there exists an optimal plan that does not contains $a \in A^+$ if there exists an action $a' \in A^+$ satisfying following: (i) $\text{fadd}(a) \subseteq \text{fadd}(a')$, (ii) for any $p \in \text{pre}(a')$, p is a fact landmark of a or $p \in I$, and (iii) $c(a) \geq c(a')$.*

Proof: For any plan $\pi = (a_0, \cdots, a_{i-1}, a, a_{i+1}, \cdots, a_n)$ of T^+, it can be shown that a sequence of actions $\pi' = (a_0, \cdots, a_{i-1}, a', a_{i+1}, \cdots, a_n)$ is also a feasible plan. Each proposition p of $\text{pre}(a')$ is fact landmarks of a, or p is a member of the initial state, hence, if $\text{pre}(a) \subseteq I((a_0, \cdots, a_{i-1}))$, $\text{pre}(a') \subseteq I((a_0, \cdots, a_{i-1}))$ also holds. Since $\text{fadd}(a) \subseteq \text{fadd}(a')$, $\text{add}(a) \setminus \text{fadd}(a) \subseteq I((a_0, \cdots, a_{i-1}, a))$, and $\text{add}(a') \setminus \text{fadd}(a') \subseteq I((a_0, \cdots, a_{i-1}, a'))$ hold ac-

cording to the definition of first achievers, we also have $I((a_0, \cdots, a_{i-1}, a)) \subseteq I((a_0, \cdots, a_{i-1}, a'))$. Therefore $G \subseteq I(\pi')$ holds.

Finally we have $c(\pi) \geq c(\pi')$ since $c(a) \geq c(a')$ holds. Therefore, if a plan contains a, it is not optimal, or there exists another optimal plan which does not contain a. □

If there exists a dominated action a, we can eliminate a variable as $\mathcal{U}(a) = 0$. This variable elimination cuts some feasible solutions of IP(T^+). Moreover, it sometimes cuts some optimal solutions if $c(a) = c(a')$ holds for the condition (iii). However, at least one optimal solution remains.

Robinson proposed similar constraints for a MaxSAT-based planner, but his condition is stricter than condition (ii) [87].

Immediate Action Application On a delete-free task T^+, applying some types of actions to the initial state do not hurt optimality. In this work, an action with cost zero as [36] and an action landmark as [37] are adopted to use to this enhancement. For a delete-free task T^+, if an action $a \in A$ satisfies $c(a) = 0$ and $\text{pre}(a) \subseteq I$, then a sequence made by connecting a before an optimal plan of $\langle P, A^+ \setminus \{a\}, I \cup \text{add}(a), G\rangle$ is an optimal plan of T^+. Similarly, if an action a is an action landmark of T^+ and a is applicable to I, you can apply a to I immediately.

For IP(T^+), variables $\mathcal{T}(p)$ for $p \in I$ can be eliminated by substituting zero. Given a sequence of immediate applicable actions $(a_0, \cdots, a_k)$ (it must be a correct applicable sequence), we can eliminate some variables as follows: (i) $\mathcal{U}(a_i) = 1$, (ii) $\mathcal{T}(a_i) = i$, (iii) $\forall p \in \text{pre}(a_i), \mathcal{U}(p) = 1$, and (iv) $\forall p \in \text{add}(a_i) \setminus I((a_0, \cdots, a_{i-1})), \mathcal{U}(p) = 1, \mathcal{T}(p) = i$ and $\mathcal{E}(a_i, p) = 1$.

Iterative Application of Variable Eliminations The variable elimination techniques described above can interact synergistically with each other resulting in a cascade of eliminations. For example, landmarks increase non relevant add effects, which increases dominated actions, which can result in new landmarks. Therefore, a iterative variable eliminating algorithm which applies eliminations until quiescence is used in this work.

Inverse action constraints It can be defined the following inverse relationship between a pair of actions for a delete-free task T^+. For two actions $a_1, a_2 \in A^+$, a_1 is an *inverse action* of a_2 if it satisfies following: (i) $\text{add}(a_1) \subseteq \text{pre}(a_2)$, and (ii) $\text{add}(a_2) \subseteq \text{pre}(a_1)$. By the definition, it is clear that if a_1 is an inverse action of a_2, then a_2 is an inverse action of a_1. Inverse actions satisfy following fact.

Lemma 5.5. *Given a delete-free task T^+, let $\pi = (a_0, \cdots, a_n)$ be a feasible plan. If $a_i \in \pi$ is an inverse action of $a_j \in \pi$, and if $i < j$ holds, then*

$\pi' = (a_0, \cdots, a_{j-1}, a_{j+1}, \cdots, a_n)$ *is also a feasible plan.*

Proof: Since π is a feasible plan of T^+, we have $\text{pre}(a_i) \subseteq I((a_0, \cdots, a_{i-1})) \subseteq I((a_0, \cdots, a_{j-1}))$. According to the definition of inverse actions, $\text{add}(a_j) \subseteq \text{pre}(a_i)$ holds, and we have $\text{add}(a_j) \subseteq \text{pre}(a_i) \subseteq I((a_0, \cdots, a_{j-1})) = I((a_0, \cdots, a_j))$. Hence $(a_{j+1}, \cdots, a_n)$ is applicable to $I((a_0, \cdots, a_{j-1}))$, and $G \subseteq I(\pi') = I(\pi)$. □

Corollary 5.6. *For a delete-free task T^+, a feasible solution $\pi = (a_0, \cdots, a_n)$ is not optimal if $a_i \in \pi$ is an inverse action of $a_j \in \pi$ and both of a_i and a_j have non-zero cost.*

Let $\text{inv}(a, p)$ denote the set of inverse actions of an action a which have p as add effect. There are several possible ways to use above proposition (e.g., $\mathcal{U}(a) + \mathcal{U}(a') \leq 1$, for all $a' \in \text{inv}(a)$). On $\text{IP}(T^+)$, due to avoid adding a huge number of constraints, constraint 2 is modified as follows:

2. $\forall a \in A^+,\ \ \forall p \in \text{pre}(a), \mathcal{U}(p) - \sum_{a' \in \text{inv}(a,p)} \mathcal{E}(a', p) \geq \mathcal{U}(a)$.

The mark e (e.g. $\text{LP}^{\text{e}}(T^+)$) is used to denote the integer linear programming model after all of the reductions that explained so far have been applied to the original model.

Constraint Relaxation Enhancements for eliminating variables and new constraints to speed up the computation and to tighten the gap of the linear programming relaxation have been presented. As shown experimentally in Section 5.6, computing $\text{IP}^{\text{e}}(T^+)$ or $\text{LP}^{\text{e}}(T^+)$ remains relatively expensive, even if we use all of the enhancements described above.

Thus, a relaxation for $\text{IP}(T^+)$ is introduced. $\text{IP}(T^+)$ without constraints 5 and 6 is called*time-relaxed* $\text{IP}(T^+)$, denoted $\text{IP}_{\text{tr}}(T^+)$. Similarly $\text{LP}(T^+)$ without same constraints is called *time-relaxed* $\text{LP}(T^+)$, denoted $\text{LP}_{\text{tr}}(T^+)$. The mark e is also used to denote the enhanced models for $\text{IP}_{\text{tr}}(T^+)$ and $\text{LP}_{\text{tr}}(T^+)$. It can be seen that if the relevance of propositions and actions has an ordering (i.e. it does not have a cycle) on T^+, then the optimal costs of $\text{IP}(T^+)$ and $\text{LP}(T^+)$ are the same as the optimal costs of $\text{IP}_{\text{tr}}(T^+)$ and $\text{LP}_{\text{tr}}(T^+)$ respectively. The experiments in Section 5.6 show that the relaxation is quite tight (i.e., $\text{IP}(T^+)$ and $\text{IP}_{\text{tr}}(T^+)$ often have the same cost), and that $\text{IP}_{\text{tr}}(T^+)$ can be computed significantly faster than $\text{IP}(T^+)$. $\text{LP}(T^+)$, $\text{LP}^{\text{e}}(T^+)$, and $\text{IP}^{\text{e}}(T^+)$ have same behavior.

5.4 Counting Constraints

So far, we have concentrated on efficient computation of the cost-optimal delete-free STRIPS planning problem, and all of the (integer) linear pro-

gramming models are bounded by the optimal cost of the relaxed task. However, the proposed integer linear programming model can be extended with constraints regarding delete effects. By adding variables and constraints related to delete effects of actions, the model can also calculate lower bounds on the number of times each action must be applied for the original task in contrast to the fact that each action appears only once in an optimal plan of a delete-free task.

New variables are defined as follows:

- $\forall a \in A, \mathcal{N}(a) \in \{0, 1, \cdots\} : \mathcal{N}(a) = n$ if and only if a is used n times.
- $\forall p \in P, \mathcal{G}(p) \in \{0, 1\} : \mathcal{G}(p) = 1$ if and only if $p \in G$.

$\mathcal{G}(p)$ is also an auxiliary variable as $\mathcal{I}(p)$. New constraints are defined as follows:

7. $\forall a \in A, \mathcal{N}(a) \geq \mathcal{U}(a)$.
8. $\forall p \in P, \mathcal{G}(p) + \sum_{p \in \text{predel}(a)} \mathcal{N}(a) \leq \mathcal{I}(p) + \sum_{p \in \text{add}(a)} \mathcal{N}(a)$,

where $\text{predel}(a) = \text{pre}(a) \cap \text{del}(a)$. Finally, the objective function is modified so as to minimize $\sum_{a \in A} c(a)\mathcal{N}(a)$. Given a STRIPS planning task T, $\text{IP}(T)$ is used to denote an integer linear program composed of $\text{IP}(T^+)$ with the above modifications. New constraints correspond to the *net change* constraints that were recently proposed in [83], as well as the effect implication constraints in [96] (both are defined on SAS^+ formulations).

Intuitively, the final constraint states that the number of uses of actions adding p must be equal to or larger than the number of uses of actions requiring and deleting p at the same time in a feasible plan of T. Any feasible plan of a STRIPS planning task always satisfies this condition. Hence, for any task T and any feasible plan π for T, we can clearly derive a feasible solution of $\text{IP}(T)$ with same cost as π. In addition to this, here a stronger lemma can be proved for modifications of models by the enhancements in the previous section.

Lemma 5.7. *Given a task T, let $\text{IP}(T^+)'$ be an integer linear program composed of $\text{IP}(T^+)$ with some variable eliminations and some new constraints. Let $\text{IP}(T)'$ be an integer linear program made from $\text{IP}(T^+)'$ with new variables, new objective function, and counting constraints as the above definition. For any action sequence π of T, let π^+ be the relaxed action sequence corresponding to π. If (1) there exists a subsequence $\pi^{+'}$ of π^+ that satisfies $\text{IP}(T^+)'$ by the derivation on Table 5.1, and (2) π satisfies constraint 8 when $\mathcal{N}(a)$ is set to the number of occurrences of a in π for each action $a \in A$, then there exists a feasible solution of $\text{IP}(T)'$ that has same cost as π.*

Proof: Let π be an action sequence of T that satisfies condition (1) and (2). Let $\pi^{+'}$ be a subsequence of π that satisfies $\mathrm{IP}(T^+)'$, and $F^{+'}$ be the assignment on a feasible solution of $\mathrm{IP}(T^+)'$ corresponding to $\pi^{+'}$.

Define an assignment F for $\mathrm{IP}(T)'$ as:

- $\mathcal{V}_F := \mathcal{V}_{F^{+'}}$ for each variable $\mathcal{V}$ that is defined in Table 5.1, and
- $\mathcal{N}(a)_F :=$ (the number of occurrences of a in π) for each $a \in A$.

Then clearly the variables $\mathcal{U}(p)_F, \mathcal{U}(a)_F, \mathcal{E}(a,p)_F, \mathcal{T}(p)_F, \mathcal{T}(a)_F$, and $\mathcal{I}(p)_F$ satisfy all constraints of $\mathrm{IP}(T^+)'$. Since $\pi^{+'}$ is a subsequence of π^+, and since $\mathcal{U}(a)_F = 1$ holds only if $a \in A^+$ is a member of $\pi^{+'}$ according to the definition of $\mathcal{V}_F$, constraint 7 is satisfied for each action $a \in A$. The variables $\mathcal{N}(a)_F$ also satisfy constraint 8 according to condition (2).

From the definition of $\mathcal{N}(a)_F$, the cost of the objective function is equal to the cost of π. □

By using lemma 5.7, the following can be proved.

Lemma 5.8. *Given a task T, let π be an action sequence of T. If π satisfies condition (1) and (2) of lemma 5.7 for* $\mathrm{IP}(T)$ *itself, then there exists a feasible solution with the same cost as π for* $\mathrm{IP}(T)$ *with any combination of landmark extraction and substitution, relevance analysis, and inverse action constraints.*

Proof: Let $\pi^{+'}$ be a subsequence of π^+ that satisfies condition (1) of lemma 5.7 for $\mathrm{IP}(T)$. Note that $\pi^{+'}$ is a feasible plan of T^+. Then let $\pi^{+''}$ be the subsequence of $\pi^{+'}$ that is made by removing irrelevant actions and inverse actions. The sequence $\pi^{+''}$ is also a feasible plan of T^+, and this satisfies $\mathrm{IP}(T^+)$ with variable eliminations computed by landmark extraction and substitution and relevance analysis, and inverse action constraints. □

Corollary 5.9. *Given a task T, for any feasible plan π of T, there exists a feasible solution of* $\mathrm{IP}(T)$ *that has the same cost as the cost of π. In addition to this, there exists a feasible solution of* $\mathrm{IP}(T)$ *with any combination of landmark extraction and substitution, relevance analysis, and inverse action constraints that has the same cost as the cost of π.*

Proof: The subsequence of the relaxed plan π^+ that is made by removing repetitions of actions has its corresponding assignment of $\mathrm{IP}(T^+)$. In addition, any feasible plan of a STRIPS planning task always satisfies condition 8 when $\mathcal{N}(a)$ is set to the number of occurrences of a in π. □

Unfortunately the counting constraints conflict with dominated action elimination and zero cost immediate action application. When counting constraint is used, it is necessary to disable zero cost immediate action application and to modify the condition of dominated actions as follows:

Definition 5.10. *Given a feasible task T, an action a is a dominated action of action a' if (i)* $\mathrm{add}(a) \subseteq \mathrm{add}(a')$*, (ii) for any $p \in \mathrm{pre}(a')$, p is a fact landmark of a or $p \in I$, (iii)* $\mathrm{pre}(a') \cap \mathrm{del}(a') \subseteq \mathrm{pre}(a) \cap \mathrm{del}(a)$.

We cannot use the new dominated actions to make a feasible plan of T anymore since fact landmarks are sometimes deleted after they are achieved. However the following lemma can be proved.

Lemma 5.11. *Given a feasible task T, let $\pi = (a_0, \cdots, a_n)$ be an action sequence that satisfies condition (1) and (2) of lemma 5.7 for* $\mathrm{IP}(T)$*. If an action $a_i \in A$ is dominated by another action $a' \in A$, and if π contains a dominated action a_i, $(a_0, \cdots, a_{i-1}, a', a_{i+1}, \cdots, a_n)$ also satisfies condition (1) and (2) of lemma 5.7 for* $\mathrm{IP}(T)$*.*

Proof: Replacing a_i to a' does not hurt condition (2) of lemma 5.7 according to condition (i) and (iii) of the new dominated actions. Hence, if there exists a subsequence of π^+ that does not contain a_i^+ and satisfies condition (1) of lemma 5.7, then the statement is clearly proved.

Assume that only subsequences of π^+ that contain a_i^+ satisfy condition (1) of lemma 5.7. Let $\pi^{+'} = (a_{i_0}^+, \cdots, a_{i_m}^+)$ be such a subsequence of π^+, and $a_{i_j}^+$ be a_i^+. Note that $\pi^{+'}$ is a feasible solution of T^+ since it satisfies condition (1) of lemma 5.7. As similar to the case of relaxed tasks, $\mathrm{pre}(a_{i_j}^+) \subseteq I((a_{i_0}^+, \cdots, a_{i_{j-1}}^+))$ holds because of condition (ii). In addition, $I((a_{i_0}^+, \cdots, a_{i_j}^+)) \subseteq I((a_{i_0}^+, \cdots, a_{i_{j-1}}^+, a'^+))$ holds since $\mathrm{add}(a_i^+) \subseteq \mathrm{add}(a'^+)$ holds. Hence an action sequence made by replacing a_i^+ in $\pi^{+'}$ to a'^+ is also a feasible solution of T^+, namely, it satisfies condition (1) of lemma 5.7. □

Corollary 5.12. *Given a task T, let π be a feasible solution of T. There exists a feasible solution of* $\mathrm{IP}(T)$ *with any combination of landmark extraction and substitution, relevance analysis, inverse action constraints, and the new dominated action elimination that has cost equal to or less than the cost of π.*

LP and tr are also used as same as corresponding relaxations for $\mathrm{IP}(T^+)$. $\mathrm{IP^e}(T)$ and $\mathrm{LP^e}(T)$ denote the models constructed by applying all of the valid reductions to $\mathrm{IP}(T)$ and $\mathrm{LP}(T)$ respectively.

Relationship among the ILP bounds Based on the definitions, it can be trivially shown that: $\mathrm{IP_{tr}}(T^+) \leq \mathrm{IP^e_{tr}}(T^+) \leq \mathrm{IP}(T^+) = \mathrm{IP^e}(T^+) \leq \mathrm{IP}(T) = \mathrm{IP^e}(T)$. As for the linear programming relaxations, it is satisfied that $\mathrm{LP_{tr}}(T^+) \leq lpT^+ \leq \mathrm{LP^e}(T^+)$, $\mathrm{LP_{tr}}(T^+) \leq \mathrm{LP^e_{tr}}(T^+) \leq \mathrm{LP^e}(T^+)$, $\mathrm{LP_{tr}}(T) \leq lpT \leq \mathrm{LP^e}(T)$, and $\mathrm{LP_{tr}}(T) \leq \mathrm{LP^e_{tr}}(T) \leq \mathrm{LP^e}(T)$. However,

$\mathrm{LP}^{\mathrm{e}}(T)$ does not always dominate $\mathrm{LP}^{\mathrm{e}}(T^{+})$ since sets of eliminated variables are different because of dominated action elimination and zero-cost immediate action application. Figure 5.1 illustrates the dominance relationships among the bounds.

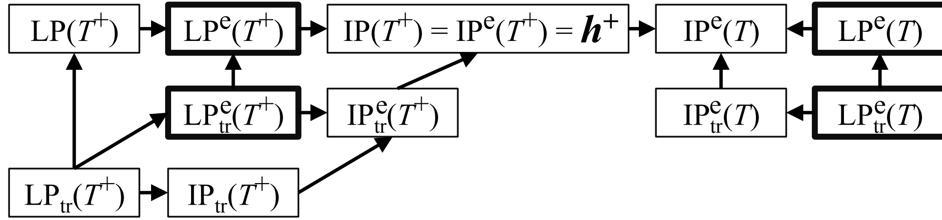

Figure 5.1: Dominance relationships. Edge $X \rightarrow Y$ indicates that X is always a lower bound of Y. The four highlighted linear programming variants are used in the A*/autoconf in Tables 5.3-5.4.

5.5 Automatic Heuristic Selection

While $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^{+})$ and $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$ are dominated by $\mathrm{LP}^{\mathrm{e}}(T^{+})$ and $\mathrm{LP}^{\mathrm{e}}(T)$, respectively, the time-relaxed linear programming models are significantly cheaper to compute than their non-relaxed counterparts. In addition, although $\mathrm{IP}^{\mathrm{e}}(T)$ dominates $\mathrm{IP}^{\mathrm{e}}(T^{+})$, it is possible for $\mathrm{LP}^{\mathrm{e}}(T^{+})$ to be larger than $\mathrm{LP}^{\mathrm{e}}(T)$. Thus, we have a set of 4 viable linear programming heuristics, none of which dominate the others when considering both accuracy and time. The "best" choice to optimize this tradeoff between heuristic accuracy and node expansion rate depends on the problem instance.

In this work, a simple mechanism implemented for automatically selecting one of the linear programming heuristics to be used for each planning task. First, it computes $\mathrm{LP}^{\mathrm{e}}(T^{+})$, $\mathrm{LP}^{\mathrm{e}}(T)$, $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^{+})$, and $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$ for the initial state of the given planning tasks. It then selects one based on the following rule: Choose the heuristic with the highest value. Break ties by choosing the heuristic that is cheapest to compute. Although the "cheapest" heuristic could be identified according to the CPU time to compute each heuristic, for many instances, the computations are too fast for robust timing measurements, so it simply breaks ties in order of $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^{+})$, $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$, $\mathrm{LP}^{\mathrm{e}}(T^{+})$, $\mathrm{LP}^{\mathrm{e}}(T)$ (because this ordering usually accurately reflects the timing order). A more sophisticated method for heuristic selection may result in better performance (c.f. [25]), and is an avenue for future work.

5.6 Experimental Evaluations

Below, all experiments used the CPLEX 12.6 solver to solve integer linear programs. All experiments were single-threaded and executed on a Xeon E5-2650, 2.6GHz. A set of 1,366 IPC benchmark problems (from 1998 to 2011) distributed with Fast Downward is used. The implementation of the A* search algorithm with the proposed heuristic functions can currently handle the subset of PDDL which includes STRIPS, types, and action-costs. The full list of domains and the numbers of instances per domain is shown in Table 5.4.

Comparison of ILP Bounds The quality of the integer/linear programming bounds is assessed by evaluating the optimal costs computed for these bounds.

The first experiment computes the ratio between the optimal cost of the integer linear program and its linear programming relaxation of the relaxed tasks of the benchmark set (Figure 5.2). The ceiling of the costs of the linear programming relaxations are taken because the benchmarks have integer action costs. As shown in Table 5.2, the gap between the integer programs and their linear programming relaxations are quite small. In fact, for the majority of instances, the gap between the rounded-up value of linear programming relaxation and the value of the original integer linear program is zero for $\mathrm{IP}^{\mathrm{e}}(T^{+})$, $\mathrm{IP}^{\mathrm{e}}(T)$, $\mathrm{IP}^{\mathrm{e}}_{\mathrm{tr}}(T^{+})$, $\mathrm{IP}^{\mathrm{e}}_{\mathrm{tr}}(T)$, so the linear programming relaxation is frequently a perfect approximation of h^{+}.

Next, to understand the impact of various sets of constraints in the integer linear programming formulations, Table 5.2 compares the optimal costs of pairs of integer linear programs and linear programs. The integer program ratio for $\mathrm{IP}(T^{+})$ vs $\mathrm{IP}^{\mathrm{e}}(T^{+})$ is always 1 because they both compute h^{+}. However, on almost every single domain, the value of the linear programming relaxation of the extended formulation $\mathrm{LP}^{\mathrm{e}}(T^{+})$ is significantly better (higher) than the basic formulation $\mathrm{LP}(T^{+})$, indicating that variable elimination and the additional constraints serve to tighten the linear programming bound. Thus, the enhancements to the basic model described in Section 5.3 provide a significant benefit. $\mathrm{LP}^{\mathrm{e}}(T)$ tends to be higher than $\mathrm{LP}^{\mathrm{e}}(T^{+})$, indicating that that counting constraints enhances accuracy; note that in some cases $\mathrm{LP}^{\mathrm{e}}(T^{+})$ is higher than $\mathrm{LP}^{\mathrm{e}}(T)$. The time-relaxations $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^{+})$ and $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$ are usually very close to $\mathrm{LP}^{\mathrm{e}}(T^{+})$ and $\mathrm{LP}^{\mathrm{e}}(T)$, indicating that the time relaxation achieves a good tradeoff between computation cost and accuracy.

Evaluating ILP for Delete-free planning To evaluate the speed of solving the proposed models, $\mathrm{IP}^{\mathrm{e}}(T^{+})$ is compared with Haslum et al.'s

Table 5.2: Comparison of bounds: $il^{+} = \text{ILP}(T^{+})$, $il^{e+} = \text{ILP}^{\text{e}}(T^{+})$, $il^{e} = \text{ILP}^{\text{e}}(T)$, $il^{e+}_{tr} = \text{ILP}^{\text{e}}_{\text{tr}}(T^{+})$, $il^{e}_{tr} = \text{ILP}^{\text{e}}_{\text{tr}}(T)$.

	il^{+} / il^{e+}		il^{e} / il^{e+}		il^{e+}_{tr} / il^{e+}		il^{e}_{tr} / il^{e}	
	LP	IP	LP	IP	LP	IP	LP	IP
airport	.53	1.00	.99	1.00	.99	.99	1.00	.99
blocks	.92	1.00	.92	.92	1.00	1.00	1.00	1.00
depot	.54	1.00	.93	.99	.99	.92	1.00	.99
driverlog	.97	1.00	.91	.95	.96	.84	1.00	.96
elevators-opt08	.39	1.00	1.16	.96	.97	.64	1.00	.70
elevators-opt11	.36	1.00	1.17	.96	.96	.62	1.00	.73
floortile-opt11	.99	1.00	.93	.94	1.00	.97	1.00	.98
freecell	.48	1.00	1.01	1.00	.97	.92	1.00	.98
grid	-	-	.79	.85	.98	.79	1.00	.88
gripper	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
logistics98	.54	1.00	.89	1.00	.98	.88	1.00	1.00
logistics00	.47	1.00	.99	1.00	.99	.99	1.00	1.00
miconic	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
movie	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
no-mprime	.58	1.00	1.10	.97	.88	.66	1.00	.94
no-mystery	.58	1.00	1.03	.98	.92	.72	1.00	.96
nomystery-opt11	.97	1.00	.97	.97	1.00	1.00	1.00	1.00
openstacks	.38	1.00	1.00	1.00	1.00	1.00	1.00	1.00
openstacks-opt08	0	1.00	1.00	1.00	1.00	1.00	1.00	1.00
openstacks-opt11	-	-	1.00	1.00	1.00	1.00	1.00	1.00
parcprinter-08	.99	1.00	.92	.92	1.00	1.00	1.00	1.00
parcprinter-opt11	.99	1.00	.94	.94	1.00	1.00	1.00	1.00
parking-opt11	.90	1.00	.97	.97	.94	.87	.94	.86
pegsol-08	0	1.00	.81	.72	1.00	.68	1.00	.86
pegsol-opt11	0	1.00	.88	.73	1.00	.67	1.00	.86
pipes-notankage	.62	1.00	.94	.95	.92	.83	.97	.90
pipes-tankage	.62	1.00	.95	.96	.98	.87	1.00	.96
psr-small	.87	1.00	.38	.38	1.00	1.00	1.00	1.00
rovers	.63	1.00	.86	.77	1.00	1.00	1.00	1.00
satellite	.99	1.00	.99	.99	1.00	1.00	1.00	1.00
scanalyzer-08	1.00	1.00	1.00	1.00	1.00	.96	1.00	1.00
scanalyzer-opt11	1.00	1.00	1.00	1.00	1.00	.96	1.00	1.00
sokoban-opt08	.37	1.00	.88	.87	.99	.95	.99	.94
sokoban-opt11	.34	1.00	.90	.88	.99	.97	1.00	.96
storage	.55	1.00	.95	.91	1.00	1.00	1.00	1.00
transport-opt08	.26	1.00	3.42	1.00	.99	.36	1.00	.58
transport-opt11	-	-	-	-	.99	.43	-	-
visitall-opt11	1.00	1.00	.95	.93	.99	.97	.99	.95
woodworking08	.81	1.00	.94	.94	1.00	1.00	1.00	1.00
woodworking11	.80	1.00	.94	.94	1.00	1.00	1.00	1.00
zenotravel	.99	1.00	.92	.98	.96	.90	1.00	.99

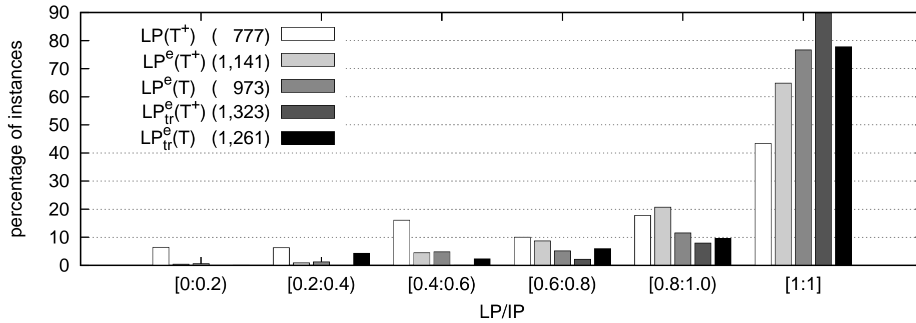

Figure 5.2: Ratio between the optimal costs of the IP's and their LP relaxations, categorized into buckets. [x:y) = "% of instances where the LP/IP ratio is in the range [x:y).

algorithm [48] ("HST"), which is one of the state-of-the art solvers for the cost-optimal delete-free STRIPS planning problem, of a set of 1,346 IPC benchmarks from the Fast Downward benchmark suite. Both solvers were run with a 15-minute time limit on each instance. The most recent version of HST was configured to use CPLEX to solve the hitting set subproblem, as suggested by Haslum [46].

The number of delete-free, relaxed instances that are solved by both planner is 905. HST solved 1,117 instances, and $\mathrm{IP}^{\mathrm{e}}(T^+)$ solved 1,186 instances. $\mathrm{IP}^{\mathrm{e}}(T^+)$ was faster than HST on 575 instances, and HST was faster than $\mathrm{IP}^{\mathrm{e}}(T^+)$ on 330 instances. Figure 5.3 shows the ratio of runtimes of HST to the ILP solver, sorted in increasing order of the ratio, time(HST)/time($\mathrm{IP}^{\mathrm{e}}(T^+)$). The horizontal axis is the cumulative number of instances. Overall, $\mathrm{IP}^{\mathrm{e}}(T^+)$ outperform the state-of-the-art delete-free solver and indicates that direct computation of the optimal cost of a delete-free task using integer linear programming is a viable approach.

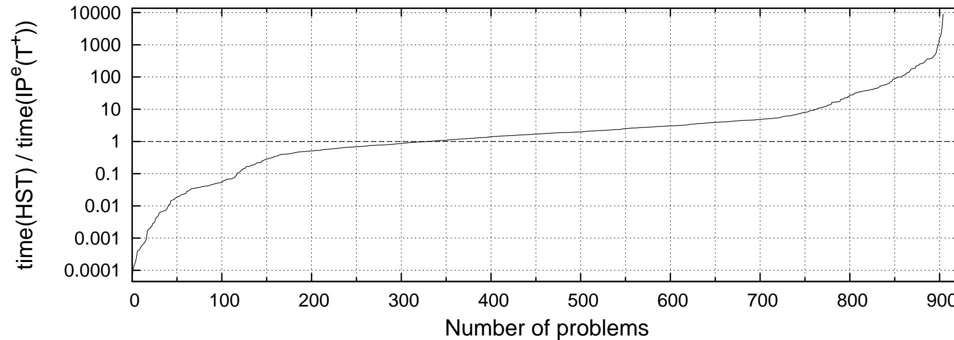

Figure 5.3: Computation of h^+: Comparison of $\mathrm{IP}^{\mathrm{e}}(T^+)$ and HST on delete-free, relaxed problems

Table 5.3: IPC benchmark problems: # solved with 5 minute time limit.

Configuration	# solved	Description
FD/LM-cut	718	the landmark cut heuristic (`seq-opt-lmcut`)
FD/M&S IPC2011	687	IPC 2011 merge-and-shrink heuristic [80]
FD/$h^{\max}$	551	the max heuristic
A*/h^+	342	*hsp_f* planner using A* and h^+ heuristic [48, 45]
A*/IP(T^+)	358	basic IP formulation for h^+
A*/LP(T^+)	477	LP relaxation of IP(T^+)
A*/IP(T^+)+land	425	IP(T^+) + Landmarks
A*/LP(T^+)+land	564	LP relaxation of IP(T^+)
A*/IP$^{\text{e}}$(T^+)	582	IP(T^+) with all enhancements in Sections 5.3-5.3
A*/LP$^{\text{e}}$(T^+)	652	LP relaxation of IP$^{\text{e}}$(T^+)
A*/IP$^{\text{e}}$(T)	463	IP$^{\text{e}}$(T^+) with counting constraints (Section 5.4)
A*/LP$^{\text{e}}$(T)	608	LP relaxation of IP$^{\text{e}}$(T)
A*/IP$^{\text{e}}_{\text{tr}}$(T^+)	606	time-relaxation (Section 5.3) of IP$^{\text{e}}$(T^+)
A*/LP$^{\text{e}}_{\text{tr}}$(T^+)	674	LP relaxation of IP$^{\text{e}}_{\text{tr}}$(T^+)
A*/IP$^{\text{e}}_{\text{tr}}$(T)	554	time-relaxation of IP$^{\text{e}}$(T)
A*/LP$^{\text{e}}_{\text{tr}}$(T)	661	LP relaxation of IP$^{\text{e}}_{\text{tr}}$(T)
A*/autoconf	722	Automated selection of LP at root node(Section 5.5)

Evaluating h^+-based heuristics in a cost-optimal planner In the last experiment, the proposed models are embedded into the A* search algorithm. Various configurations of the proposed model based heuristics are compared first, as well as several configurations of Fast Downward (FD), given 5 minutes per an instance and a 2GB memory limit. For Fast Downward with the bisimulation merge-and-shrink heuristic, the IPC2011 hybrid bisimulation m&s configuration (`seq-opt-merge-and-shrink`) is used.[1] The # of problems solved by each configuration is shown in Table 5.3.

As shown in Table 5.3, the basic integer linear programming model performs the worst in the proposed models, and is comparable to A*/h^+. As noted in [45], straightforward use of h^+ as a heuristic function is unsuccessful (significantly worse than Fast Downward using $h^{\max}$). However, the addition of landmark constraints is sufficient to significantly increase the number of solved problems compared to A*/h^+, and A*/IP$^{\text{e}}$(T^+) is better than $h^{\max}$ and can be considered a somewhat useful heuristic function. The time-relaxation results in significantly increases performance compared to A*/IP$^{\text{e}}$(T^+) and A*/IP$^{\text{e}}$(T). In addition, for all integer linear programming models, the A* search algorithm using their corresponding linear programming relaxations as the heuristic functions performs significantly better than directly using the integer linear programming models as heuristic functions. A*/LP$^{\text{e}}$(T^+), A*/LP$^{\text{e}}_{\text{tr}}$(T^+), and A*/LP$^{\text{e}}_{\text{tr}}$(T), are all competitive with the bisimulation merge-and-shrink heuristic. While A*/LP$^{\text{e}}$(T), does not per-

[1]While this is tuned for 30 minutes and suboptimal for 5 minutes, we wanted to use the same configuration as in the 30-minute experiments below.

form quite as well, there are some instances where A*/$\mathrm{LP^e}(T)$ performs best. Finally, A*/autoconf, which uses automated heuristic selection (Section 5.5) performs quite well. A*/autoconfis significantly better than its 4 components ($\mathrm{LP^e}(T^+)$, $\mathrm{LP^e_{tr}}(T^+)$, $\mathrm{LP^e_{tr}}(T)$, $\mathrm{LP^e}(T)$), and is competitive with Fast Downward with the landmark cut heuristic.

Table 5.4 compares the coverage of the following algorithms on the IPC benchmark suite with 30 minute CPU time limit and 2GB memory limit: (1) A*/autoconf, which uses the linear programming heuristic selection mechanism described in Section 5.5 to choose among $\mathrm{LP^e}(T^+)$, $\mathrm{LP^e}(T)$, $\mathrm{LP^e_{tr}}(T^+)$, $\mathrm{LP^e_{tr}}(T)$, (2) FD using the landmark cut heuristic [54], and (3) FD using the IPC2011 bisimulation merge-and-shrink configuration (`seq-opt-merge-and-shrink`)[80].

These results indicate that A*/autoconf is competitive with both Fast Downward using the landmark cut heuristic, as well as the IPC2011 merge-and-shrink portfolio configuration. None of these planners dominate the others, and each planner performs the best on some subset of domains. Compared to the two other methods, A*/autoconf seems to perform particularly well on the freecell, parcprinter, rovers, trucks, and woodworking domains. A*/h^+[48] solved 443 problems with a 30-minute time limit, which is significantly less coverage than the proposed linear programming based planners with a 5-minute time limit (Table 5.3).

As described in Section 5.5, A*/autoconf selects the linear programming heuristic function to use for each instance based on a comparison of the values at the initial state. $\mathrm{LP^e_{tr}}(T^+)$ was selected on 755 instances, $\mathrm{LP^e_{tr}}(T)$ on 447 instances, $\mathrm{LP^e}(T^+)$ on 119 instances, and $\mathrm{LP^e}(T)$ on 25 instances. On the remaining 20 instances, A*/autoconf timed out during computations of the linear programs for the bound selection process at the initial state, indicating that for some difficult instances, the computation for the linear programs can be prohibitively expensive.

The right column of each algorithm on Table 5.4 shows the average number of visited vertices in solved instances of each domain. Obviously the proposed algorithm solves the benchmark instances with much smaller numbers of visited vertices compared with the landmark cut heuristic and the bisimulation merge-and-shrink heuristic. In 699 instances solved by both Fast Downward with the landmark cut heuristic and the proposed algorithm, Fast Downward solved 135 instances with smaller numbers of visited vertices than the proposed algorithm, and the proposed algorithm solved 458 instances with smaller numbers. Similarly, of 625 instances, the proposed algorithm visited fewer numbers of vertices than Fast Downward with the bisimulation merge-and-shrink heuristic the in solving 517 instances. In addition, Figure 5.4 and 5.5 show the scatter plots of the numbers of visited vertices.

Table 5.4: 30 minutes, 2GB RAM: "evals" is the average number of calls to heuristic function, i.e., the average number of visited states in the domain.

	Fast Downward LM-Cut		Fast Downward M&S		A*/autoconf	
Domain (# problems)	solved	evals	solved	evals	solved	evals
airport(50)	**28**	13403	23	461855	25	4640
barman-opt11(20)	**4**	1614605	**4**	5944586	3	473561
blocks(35)	28	95630	28	880799	**29**	51523
depot(22)	**7**	261573	**7**	1746549	**7**	34046
driverlog(20)	**14**	245920	13	4355507	13	56933
elevators-opt08(30)	**22**	1189951	14	10132421	13	66011
elevators-opt11(20)	**18**	1196979	12	11811143	10	65695
floortile-opt11(20)	**7**	2354266	**7**	10771362	**7**	152836
freecell(80)	15	180560	19	6291413	**45**	2177
grid(5)	2	94701	**3**	11667600	**3**	14197
gripper(20)	7	1788827	**20**	3131130	6	404857
logistics98(35)	6	169645	5	6825245	**7**	143897
logistics00(28)	**20**	212998	**20**	3007288	**20**	212985
miconic(150)	**141**	16635	77	3872365	**141**	15087
movie(30)	**30**	29	**30**	29	**30**	31
no-mprime(35)	**24**	55549	22	1490714	18	7260
no-mystery(30)	16	880031	**17**	3725239	12	1105
nomystery-opt11(20)	14	20744	**19**	9951860	14	754
openstacks(30)	**7**	157100	**7**	202732	**7**	4973
openstacks-opt08(30)	19	3254361	**21**	6347048	11	165070
openstacks-opt11(20)	14	4412937	**16**	8326670	6	294006
parcprinter-08(30)	19	699592	17	3129238	**29**	668
parcprinter-opt11(20)	14	949416	13	4091925	**20**	854
parking-opt11(20)	3	435359	**7**	8044843	1	2991
pegsol-08(30)	27	224149	**29**	705639	26	85760
pegsol-opt11(20)	17	370401	**19**	1092529	16	151110
pipes-notankage(50)	**17**	234717	**17**	1777823	13	6021
pipes-tankage(50)	12	361767	**16**	2447552	7	1926
psr-small(50)	49	178328	**50**	221152	**50**	4056
rovers(40)	7	77783	8	3395947	**11**	209551
satellite(36)	7	155990	7	1890912	**10**	26897
scanalyzer-08(30)	**15**	259961	14	6785907	8	4374
scanalyzer-opt11(20)	**12**	324943	11	8636568	5	6975
sokoban-opt08(30)	**30**	669669	24	3938226	23	75743
sokoban-opt11(20)	**20**	173004	19	3338708	19	77681
storage(20)	**15**	86439	**15**	1006600	**15**	21598
transport-opt08(30)	**11**	16807	**11**	1158282	10	58616
transport-opt11(20)	6	30550	**7**	4473292	5	116375
trucks(30)	10	462320	8	8478357	**15**	61067
visitall-opt11(20)	11	1255455	16	129229	**17**	20378
woodworking08(30)	17	759825	14	876479	**28**	767
woodworking11(20)	12	1076372	9	1357935	**18**	699
zenotravel(20)	**13**	318142	12	6727643	12	16571
Total (1366)	787		727		785	

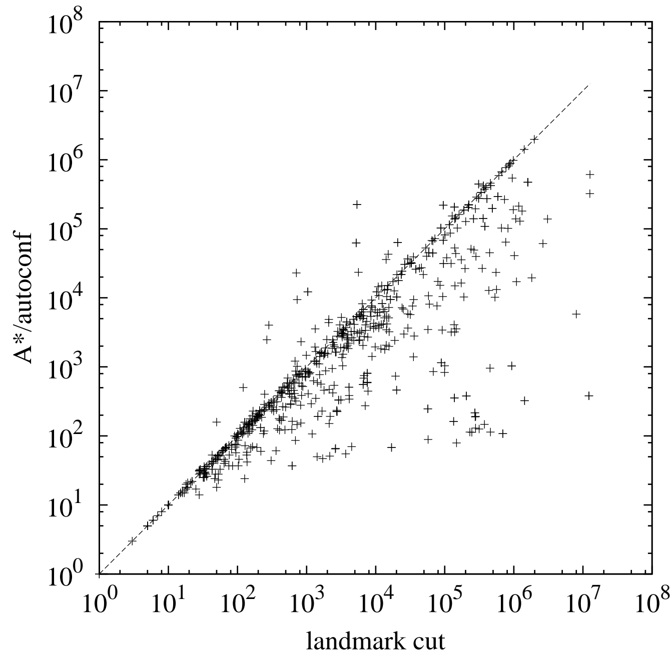

Figure 5.4: Comparison of the visited vertices between Fast Downward with the landmark cut heuristic and the proposed algorithm.

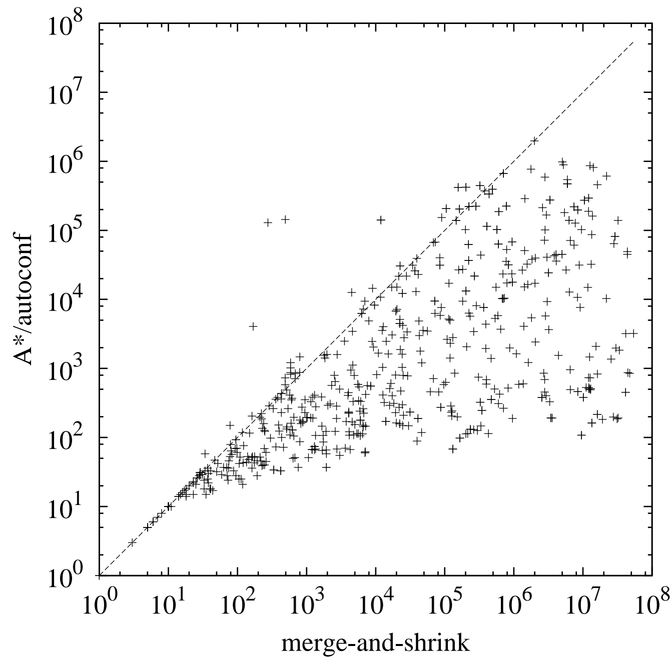

Figure 5.5: Comparison of the visited vertices between Fast Downward with the bisimulation merge-and-shrink heuristic and the proposed algorithm.

Chapter 6

Conclusion

In the last chapter, the results of this dissertation are briefly summarized, and open problems and future work are discussed.

The fourth chapter described the diverse best first search algorithm. This heuristic search algorithm is designed for avoiding plateaus of search caused by misled heuristic estimates. Experimental results showed that the diverse best first search algorithm outperformed the greedy best first search algorithm and the k best first search algorithm in satisficing planning. By incorporating preferred operators, diverse best first search performed better than the Fast Downward planner and the enhanced k best first search algorithm. Additionally, by combining diverse best first search with the LAMA planner, the proposed approach not only improved the solving ability of LAMA but also returned plans with reasonable quality. It is therefore concluded that the diverse best first search algorithm can be a strong candidate as a new basement of heuristic search algorithms.

There are several ideas to the strengthen diverse best first search algorithm. One is to develop a better state-fetching method by incorporating state-of-the-art techniques elegantly restarting search such as ARVAND [78]. Another is to combine diverse best first search with some enhancements specific to satisficing planning. In the current implementation, the diverse best first search algorithm is combined with only preferred operators. Synthesizing diverse best first search with other enhancements such as goal agenda [68], boosting [85], and multiple heuristic functions would exploit the more promising search space in an orthogonal way to the diverse best first search algorithm. Additionally, since the diverse best first search algorithm is a general heuristic search algorithm, it should not be limited to planning in principle. Applying diverse best first search to other domains is therefore of interest and value as future work.

In the fifth chapter a new integer linear programming formulation of h^+ is proposed for cost-optimal STRIPS planning. The major contribution of this work is: (1) an enhanced model for h^+ using landmarks, relevance analysis, and action elimination is proposed. It outperforms one of the previous state-of-the-art techniques for computing the optimal costs of delete-free tasks [48]; (2) It is shown that the linear programming relaxations of the proposed models are quite tight; and (3) A A*-based forward search planner, A*/autoconf, is proposed by embedding the relaxed linear programs as heuristic functions. It is shown that A* search using $\mathrm{LP}^{\mathrm{e}}(T^+)$, $\mathrm{LP}^{\mathrm{e}}(T)$, $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^+)$, or $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$ as its heuristic function is competitive with some state-of-the-art heuristic functions. Using a simple rule to select from among $\mathrm{LP}^{\mathrm{e}}(T^+)$, $\mathrm{LP}^{\mathrm{e}}(T)$, $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T^+)$, and $\mathrm{LP}^{\mathrm{e}}_{\mathrm{tr}}(T)$, A*/autoconf is competitive with the landmark cut heuristic. A*/autoconf performs well in some domains where other planners perform poorly, so the proposed methods are complementary to previous heuristics.

While it has long been believed that h^+ is too expensive to be useful as a heuristic for forward-search based planning, this work demonstrates that a linear programming relaxation of h^+ can achieve the right tradeoff of speed and accuracy to be the basis of a new class of heuristics for domain-independent planning. Integrating additional constraints to derive heuristics more accurate than h^+ (e.g., the inclusion of net change constraints [83] in Section 5.4) offers many directions for future work.

Bibliography

[1] Y. Akagi, A. Kishimoto, and A. Fukunaga. On transposition tables for single-agent search and planning: Summary of results. In *Proceedings of the Third Symposium on Combinatorial Search*, pages 1–8, 2010.

[2] Giorgio Ausiello, Paolo G. Franciosa, and Daniele Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. *Theoretical Computer Science*, 2202:312–328, 2001.

[3] Fahiem Bacchus. Subset of PDDL for the AIPS2000 planning competition. *The AIPS-00 Planning Competition Comitee*, 2000.

[4] Christer Bäckström and Inger Klein. Planning in polynomial time. In *Expert Systems in Engineering Principles and Applications*, pages 103–118. Springer, 1990.

[5] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[6] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1956.

[7] Abder Rezak Benaskeur, Froduald Kabanza, and Eric Beaudry. CORALS: A real-time planner for anti-air defense operations. *ACM Transactions on Intelligent Systems and Technology*, 1(2):13, 2010.

[8] Abder Rezak Benaskeur, Froduald Kabanza, Eric Beaudry, and Mathieu Beaudoin. A probabilistic planner for the combat power management problem. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 12–19, 2008.

[9] Christoph Betz and Malte Helmert. Planning with h^+ in theory and practice. In *Proceedings of the 32nd Annual German Conference on Artificial Intelligence*, pages 9–16. Springer, 2009.

[10] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

[11] Blai Bonet. An admissible heuristic for SAS+ planning obtained from the state equation. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 2268–2274, 2013.

[12] Blai Bonet and Héctor Geffner. Planning as heuristic search: New results. In *Proceedings of the Fifth European Conference on Planning*, pages 360–372, 1999.

[13] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[14] Blai Bonet and Malte Helmert. Strengthening landmark heuristics via hitting sets. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence*, pages 329–334, 2010.

[15] Blai Bonet and Menkes van den Briel. Flow-based heuristics for optimal planning: Landmarks and merges. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 47–55, 2014.

[16] Ronen I. Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *Journal of Artificial Intelligence Research*, 18:315–349, 2003.

[17] Tom Bylander. Complexity results for planning. In *Proceedings of the Twelfeth International Joint Conference on Artificial Intelligence*, volume 10, pages 274–279, 1991.

[18] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1):165–204, 1994.

[19] Tom Bylander. A linear programming heuristic for optimal planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 694–699, 1997.

[20] Andrew Coles, Maria Fox, and Amanda Smith. A new local-search algorithm for forward-chaining planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling 2007*, pages 89–96, 2007.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[22] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[23] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[24] Yannis Dimopoulos, Muhammad Adnan Hashmi, and Pavlos Moraitis. μ-satplan: Multi-agent planning as satisfiability. *Knowledge-Based Systems*, 29:54–62, 2012.

[25] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. Online speedup learning for optimal planning. *Journal of Artificial Intelligence Research*, 44:709–755, 2012.

[26] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the Sixth European Conference on Planning*, pages 13–34, 2001.

[27] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th International Planning Competition. Technical report, Albert-Ludwigs-Universität Freiburg, Institute für Informatik, 2004.

[28] Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39:19–39, 2003.

[29] Richard. E. Fikes and Nils. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 5(2):189–208, 1971.

[30] Lester Randolph Ford. Network flow theory. 1956.

[31] Maria Fox and Derek Long. PDDL2.1: An Extension of PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[32] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[33] Raquel Fuentetaja, Daniel Borrajo, and Carlos Linares López. A unified view of cost-based heuristics. In *ICAPS 2009 Workshop on Heuristics for Domain-Independent Planning*, pages 70–77, 2009.

[34] Alex Fukunaga, Gregg Rabideau, Steve Chien, and David Yan. Towards an application framework for automated planning and scheduling. In *Proceedings of 1997 the IEEE Aerospace Conference*, volume 1, pages 375–386, 1997.

[35] Ralph Udo Gasser. *Harnessing computational resources for efficient exhaustive search*. Ph.D. Dissertation, Swiss Federal Institute of Technology in Zurich, 1995.

[36] Avitan Gefen and Ronen Brafman. The minimal seed set problem. In *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*, pages 319–322, 2011.

[37] Avitan Gefen and Ronen Brafman. Pruning methods for optimal delete-free planning. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 56–64, 2012.

[38] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy, 2005.

[39] Alfonso E. Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5):619–668, 2009.

[40] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning Theory and Practice*. Elsevier, 2004.

[41] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *Recent Advances in AI Planning*, pages 1–20. 2000.

[42] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2):223–254, 1992.

[43] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics*, SSC-4(2):100–107, 1968.

[44] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Bulletin*, (37):28–29, 1972.

[45] Patrik Haslum. Incremental lower bounds for additive cost planning problems. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 74–82, 2012.

[46] Patrik Haslum. Personal communication, 2014.

[47] Patrik Haslum, Blai Bonet, and Héctor Geffner. New admissible heuristics for domain-independent planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1163–1168, 2005.

[48] Patrik Haslum, John Slaney, and Sylvie Thiébaux. Minimal landmarks for optimal delete-free planning. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 353–357, 2012.

[49] Milos Hauskrecht. Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 13(1):33–94, 2000.

[50] Malte Helmert. On the complexity of planning in transportation domains. In *Proceedings of the Sixth European Conference on Planning*, pages 120–126, 2001.

[51] Malte Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[52] Malte Helmert. A planning heuristic based on causal graph analysis. In *Proceedings of International Conference on Automated Planning and Scheduling*, volume 16, pages 161–170, 2004.

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

[54] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 162–169, 2009.

[55] Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 140–147, 2008.

[56] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 176–183, 2007.

[57] Jörg Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Foundations of Intelligent Systems*, pages 216–227. Springer, 2000.

[58] Jörg Hoffmann. The metric-FF planning system: Translating ”ignoring delete lists” to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[59] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[60] Tatsuya Imai and Alex Fukunaga. A practical, integer-linear programming model for the delete-relaxation in cost-optimal planning. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, pages 459–464, 2014.

[61] Tatsuya Imai and Akihiro Kishimoto. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*, pages 985–991, 2011.

[62] Froduald Kabanza, Khaled Belghith, Philipe Bellefeuille, Benjamin Auder, and Leo Hartman. Planning 3d task demonstrations of a teleoperated space robot arm. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 164–173, 2008.

[63] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, pages 1728–1733, 2009.

[64] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–363, 1992.

[65] Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for AND/OR graphs. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence*, pages 335–340, 2010.

[66] Craig A. Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[67] Donald E. Knuth. A generalization of dijkstra’s algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[68] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. 12:339–386, 2000.

[69] Sven Koenig. Optimal probabilistic and decision-theoretic planning using Markovian. Technical report, 1992.

[70] Richard Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 97:97–109, 1985.

[71] Walter S. Lasecki and Henry Kautz. Planning with tests, branches, and non-deterministic actions as satisfiability. Technical report, University of Rochester, 2012.

[72] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, 2003.

[73] Carlos Linares López and Daniel Borrajo. Adding diversity to classical heuristic planning. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, pages 73–80, 2010.

[74] Nir Lipovetzky, Christina N. Burt, Adrian R. Pearce, and Peter J. Stuckey. Planning for mining operations with time and resource constraints. In *Proceedings of the Twenty Fourth International Conference on Automated Planning and Scheduling*, pages 404–412, 2014.

[75] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - the planning domain definition language. 1998.

[76] Drew V. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the Third International Conference on AI Planning Systems*, volume 96, pages 142–149, 1996.

[77] Edward F. Moore. *The shortest path through a maze*. Bell Telephone System, 1959.

[78] Hootan Nakhost and Martin Müller. Monte-Carlo exploration for deterministic planning. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence 2009*, pages 1766–1771, 2009.

[79] Allen Newell, John C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *Proceedings of the Internatinoal Conference on Information Processing*, pages 256–264, 1959.

[80] Raz Nissim, Jörg Hoffmann, and Malte Helmert. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *Proceedings of the Twenty-Second*

International Joint Conference on Artificial Intelligence, pages 1983–1990, 2011.

[81] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, (1):193–204, 1970.

[82] Florian Pommerening and Malte Helmert. Optimal planning for delete-free tasks with incremental LM-cut. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 363–367, 2012.

[83] Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. LP-based heuristics for cost-optimal planning. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 226–234, 2014.

[84] Julie Porteous, Laura Sebastia, and Jörg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of the Sixth European Conference on Planning*, pages 37–48, 2001.

[85] Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 273–280, 2009.

[86] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[87] Nathan Robinson. *Advancing Planning-as-Satisfiability.* Ph.D. Dissertation, Griffith University, 2012.

[88] Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pages 246–249, 2010.

[89] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 2nd edition, 2003.

[90] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, volume 1, pages 206–214, 1975.

[91] Bart Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial intelligence*, pages 440–446, 1992.

[92] Rob Sherwood, Anita Govindjee, David Yan, Gregg Rabideau, Steve Chien, and Alex Fukunaga. Using ASPEN to automate EO-1 activity planning. In *Proceedings of the 1998 IEEE Aerospace Conference*, volume 3, pages 145–152, 1998.

[93] Michal Štolba and Antonín Komenda. Relaxation heuristics for multiagent planning. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 298–306, 2014.

[94] Austin Tate. *Using goal structure to direct search in a problem solver.* dissertation, Ph.D. Dissertation, The University of Edinburgh, 1976.

[95] Richard Valenzano, Nathan Sturtevant, Jonathan Schaeffer, Karen Buro, and Akihiro Kishimoto. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pages 177–184, 2010.

[96] Menkes H. L. van den Briel, J. Benton, Subbarao Kambhampati, and Thomas Vossen. An LP-based heuristic for optimal planning. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 651–665, 2007.

[97] Menkes H. L. van den Briel and Subbarao Kambhampati. Optiplan: A planner based on integer programming. *Journal of Artificial Intelligence Research*, 24:919–931, 2005.

[98] Thomas Vossen, Michael O. Ball, Amnon Lotem, and Dana Nau. On the use of integer programming models in ai planning. In *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 304–309, 1999.

[99] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 380–387, 2008.

[100] Ko-Hsin Cindy Wang and Adi Botea. MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, 42(1):55–90, 2011.

[101] Martin Wehrle and Malte Helmert. The causal graph revisited for directed model checking. In *Static Analysis*, pages 86–101. Springer, 2009.

[102] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Transition-based directed model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 186–200. Springer, 2009.

[103] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-replan: A baseline for probabilistic planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, volume 7, pages 352–359, 2007.

[104] Lin Zhu and Robert Givan. Landmark extraction via planning graph propagation. *the Thirteenth International Conference on Automated Planning and Scheduling, Doctoral Consortium*, pages 156–160, 2003.

Appendix

Here several figures, showing results of the experiments, are attached. Including these figures in this dissertation was recommended by the examiners during the examination.

Figure 1 shows the cumulative numbers of solved instances of the A* search algorithm with the landmark cut heuristic, the A* search algorithm with the merge-and-shrink heuristic, and the A* search algorithm with the automatic configuration of the proposed integer linear programming heuristics. The horizontal axis describes the number of vertex evaluations, and the vertical axis shows the number of solved instances, when the maximum number of vertex evaluations is limited by the horizontal axis. As shown by the figure, the proposed algorithm solved the largest number of instances when the maximum number of vertex evaluations were less than ten million. The A* search algorithm with the landmark cut heuristic solved the second largest number of instances. We can see a similar result for the number of vertex retrievals in Figure 2.

Figure 3 shows the cumulative numbers of solved instances limited by the total running time. For each total running time, the A* search algorithm with the landmark cut heuristic solved the largest number of instances. The proposed heuristic solved the second largest number of instances whereas the running time of the proposed heuristic per one vertex expansion is the slowest among the three heuristic functions.

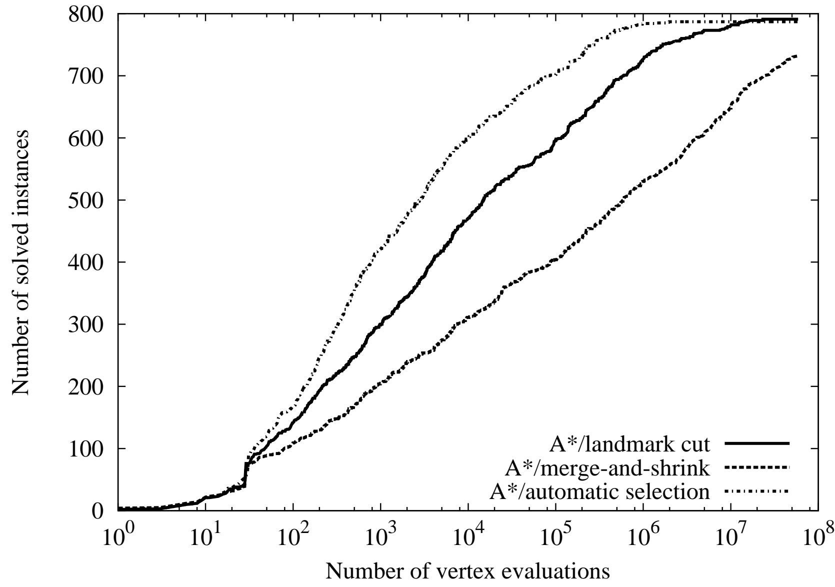

Figure 1: Number of visited vertices

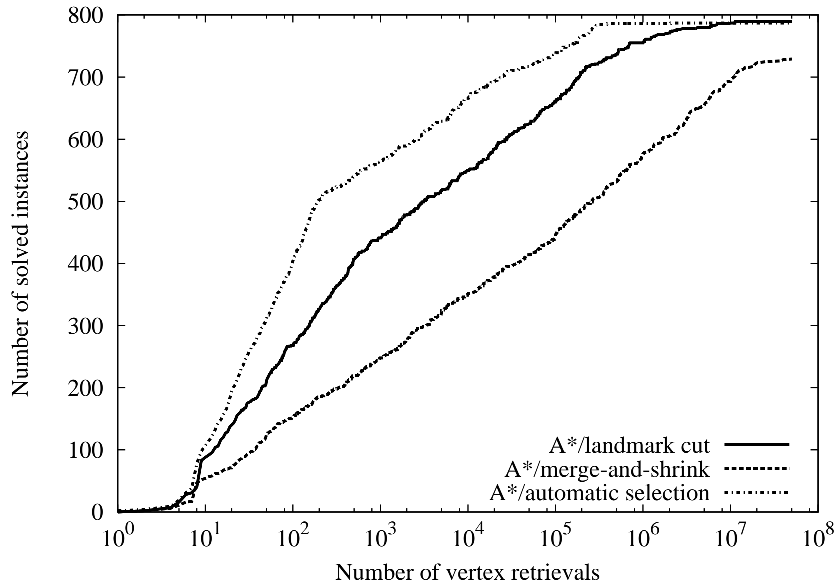

Figure 2: Number of vertex retrievals

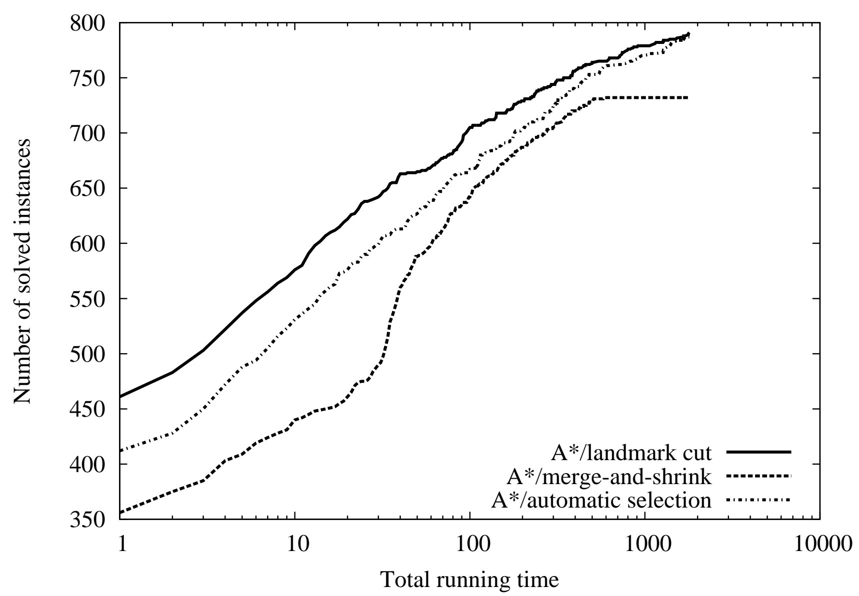

Figure 3: Total time