# Width-Based Backward Search

by

Chao Lei

Student number: 820094

Supervisor: Dr Nir Lipovetzky
The Master Research Project for 75 credit points
Subject Code COMP90068

in the
Department of Computer Science
Melbourne School of Science
**THE UNIVERSITY OF MELBOURNE**

2020-10-31

THE UNIVERSITY OF MELBOURNE

# *Abstract*

Department of Computer Science
Melbourne School of Science

Master of Computer Scince

by Chao Lei
Student number: 820094

The current study on duality mapping proposed by Suda (2013) is a viable strategy to turn progression (forward search) into regression (backward search), and the experiment results suggest that the dual versions of standard IPCs benchmark domains are quite difficult to solve for heuristic-based search. We adopt width-based search (*IW and SIW*) in this thesis to test the performance on dual problems. The experiment results show that dual problems can be solved efficiently when the goal state is restricted to single fluent, but it becomes challenging when the goal state contains conjunctive fluents.

Then, we turn *serialized iterated width*, *SIW*, best-first width search with the evaluation function $f5$, BFWS($f5$), and the polynomial variants of BFWS($f5$), $k$-BFWS($f5$) from progression into regression by modifying the state space. Results show that the backward versions are uncompetitive with the forward versions but still outperform in some IPCs benchmark domains in all *SIW*, BFWS($f5$) and $k$-BFWS($f5$).

Furthermore, we build a bidirectional search, $k$-BFWBS by integrating forward and backward $k$-BFWS($f5$) with six different combinations. Although these six combinations cannot solve more problems than only adopting forward $k$-BFWS($f5$) among tested IPCs benchmark domains, they are distinctive. However, if we run forward $k$-BFWS($f5$) first and then run backward $k$-BFWS($f5$), it outperforms only running forward $k$-BFWS($f5$), which proves that backward search is useful.

All results in this thesis indicate that although backward search is uncompetitive and multiple issues still exist, it is still worthy of having a deep exploration of backward search since it not only finds more plans in some domains but also proposes a different perspective to analyse classical planning tasks. Meanwhile, we point out the weaknesses of backward search and give relevant solutions, and a new method *complete* domain is presented to perfect backward search.

# Declaration of Authorship

I, Chao Lei, declare that this thesis titled, Width-Based Backward Search and the work presented in it are my own. I confirm that:

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School
- the thesis is 23969 words in length (excluding text in images, table, bibliographies and appendices).

Signed: Chao Lei

Date: 2020-10-31

# *Acknowledgements*

I am thankful to my supervisor Dr Nir Lipovetzky for his dedicated assistance, encouragement, guidance, motivation, insightful comments, hard questions, and for sharing their knowledge throughout this project.

My sincere thanks to my family, and in particular, my girlfriend, have always been supportive and understanding. They make this journey worthwhile.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Introduction of Planning

Artificial Intelligence (AI), as the most prevalent topic in science and engineering fields, was emerged soon after World War II, with the name present since 1956. AI currently contains a vast variety of sub-fields, ranging from general functions (learning and perception) to specific programs: such as playing chess, explaining mathematical theorems, writing poetry, autopilot on a crowded street, and diagnosing diseases. Although AI may be understood with different perspectives and goals from different people, a common expectation is to process rational actions for optimal solutions. In other words, an intelligent agent can decide the most suitable action in the expected situation.

A critical part of AI is called Intelligent Planning, which generates a sequence of actions and achieves goals through a plan. Since 1960s, the research in Intelligent Planning was started by computer scientists. Some also use the name as Automated Planning. Planning is the study about how to extract a sequence of actions that can achieve a goal state from the initial state. The environments which are assessed in planning are required to be fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These elements are called classical planning environments. In recent years, the research of Intelligent Planning has been progressed with the development of model design and research investigation, and more attention has been raised in the areas of classical Intelligent Planning.

Starting from the first International Planning Competition(IPC) in 1998, IPCs have established a series of standard model areas in the field of classical planning environments, with the purpose of promoting the birth of advanced planning algorithms and enhancing intelligent planning rapid development. These planning benchmarks include *blocks word, robots, airport scheduling, pipeline transportation, logistics scheduling*, and the benchmarks have been continuously expanding. Meanwhile, with the efforts and participation from many researchers, the intelligent planning has a rapid development in search algorithms, and the number of solved problems have been replaced with the time consumption and plan quality, which is the evaluation criteria for assessing the program performance. At present, the common standard in performance evaluation for planning is to find out the best possible solution advised by planners (systems that solve the planning problems) while keeping the time spent within 30 minutes.

## 1.2 Application of classical planning

### 1.2.1 Aerospace

Aerospace is an important application field in intelligent planning. The Automated Scheduling and Planning ENvironment (ASPEN) developed by NASA is currently acting as an outstanding intelligent planning system in the aerospace industry (Fukunaga et al., 1997). This system is widely used in spacecraft for outer space missions. Its main job is to convert high-level engineering operation instructions into lower levels in order to realise the semi-automatic control of the spacecraft. In addition, it can quickly and easily control the spacecraft to complete some targeted tasks, at which the commander makes orders on specific goals as operation tasks. The action sequence will be automatically extracted by the planning system to achieve the required tasks.

In NASA's Deep Space One mission, an autonomous remote agent system (RA) was adopted for planning, executing, and monitoring the behaviors of the space crafts (Bernard et al., 1999). RA works with intelligent planning methods to obtain plans that can achieve these goals based on the received tasks. Through intelligent planning technology, the ground control personnel can simply set up and use the planning targets to interact with the remote RA, for example, sending an instruction as taking photos of the planet at a certain moment. Then, RA can automatically plan the detailed actions that are required to complete the task, which can simplify the operation and command procedures for the whole event. Therefore, intelligent planning technologies enable and enhance the spacecraft industry to operate with relatively intelligent and autonomous control capabilities.

### 1.2.2 Web Service Composition

At the current stage, web service composition methods are mainly divided into the workflow-based method and the intelligent planning-based method. In the intelligent planning method, the web service composition problem is compiled as an intelligent planning problem. During the concurrent process, the web service is compiled into planning actions, with preconditions, add and delete effects. The sequence of web service is implemented as the sequence of actions in the planning problems.

Currently, there are many popular and effective systems of web service composition planning. For example, SHOP2 is a hierarchical task network planning system (Sirin et al., 2004). In this system, all candidate services are described by OWL-S, and a complete conversion algorithm is used to convert web services into SHOP2 domain. Then, the SHOP2 planner recursively separates the combined tasks into many subtasks until a single web service can solve each subtask.

### 1.2.3 Robot Planning

Robot planning is a classic application field in intelligent planning which is also a very cutting-edge research field. The current main research fields include perception planning, path planning, planning communication, and task planning. For example, a project called Martha is used for studying how to control and manage autonomous robots (Alami et al., 1998), for example, docker robots on existing sites. Docker robots are implemented with multiple technical planning functions, such as path planning, perception planning, planning communication, and task planning. The assigned tasks can be accomplished through autonomous planning and operations by docker robot. Each task includes steps of moving to a loading point, lifting a container, and moving to an unloading point. Meanwhile, the research on human-machine combination planning is becoming more popular. More robots are designed to achieve tasks in human-computer interaction environments. These tasks are often difficult or dangerous for humans, such as city search and rescue mission (Allen and Ferguson, 2002; Malasky et al., 2005; Suchman and Suchman, 2007; Suchman, 1987). In the urban search and rescue scenarios, human operators only need to communicate with the robot remotely and provide high-level instructions and mission goals. The robot then starts to analyse and processes the instructions received, with the goal of how to achieve the missions automatically in the planning system (de Greef et al., 2009; Thornburg and Thomas, 2009).

### 1.2.4 Traffic Control

The idea of urban traffic control UTC is introduced by Wood (1993). It is designed for solving practical traffic problems by controlling traffic lights. Urban traffic control provides irreplaceable benefits for mitigating urban traffic congestion. However, there are two remaining issues when applying intelligent planning technology for traffic management activities. The first question is how to generate a control model which can provide and respond to sound effects in the traffic; the second question is how to plan and manage urban transportation networks. It is well known that the environment of road junctions is often very complex and frequently changeable. Due to the elements of the randomness, dynamics, and diversity of control methods, the model applied in most cases are not able to simulate the actual traffic details comprehensively. Literature from Pozanco et al. (2018) proposed the APTC system, which constantly updates the motion model for applying dynamic traffic scenarios. The method is operated by monitoring the real-time status of each road junction and using a domain-related model update. For large-scale transportation networks, the system divides the network into sub-areas based on the traffic volume which is changing with time. The system then implements distributed planning and integrates the planning results collectively.

## 1.3 Related Works

### 1.3.1 Planning Language

Fikes and Nilsson (1971) introduced the *Propositional* STRIPS language which is the most popular formalism for describing planning tasks. The STRIPS task is described as the initial and goal states which are formed by conjunctions of *Propositional* atoms and are assisted by a set of actions with a precondition, add and delete lists. This formalism can capture complex domains; hence it is adopted for description benchmarks in IPCs. After that, STRIPS developed a profound influence on later languages. Pednault (1989) present a language called Action Description Language (ADL) in particular for robots which is built on the basis of STRIPS with the supplementary universal quantifiers, existential quantifiers, and conditional effects. ADL replaces the *closed world assumption* in STRIPS into the *open world assumption*. McDermott et al. (1998) is the first researcher who proposed the Planning Domain Definition Language (PDDL) in the 1998's IPC, which has been developed as the standard problem description language. By combining the characteristics of STRIPS and ADL, PDDL had been improved to meet the requirements for acting as the official language for later IPCs (Bacchus, 2001; García-Olaya et al., 2011; Gerevini et al., 2009; Hoffmann and Edelkamp, 2005; Long and Fox, 2003; McDermott, 2000). After that, each competition is also evolving with growth and progress for PDDL. By adopting a universal formalism when defining planning domains, PDDL supports faster growth in the field via enabling reuse of research, leading to a more straightforward comparison among systems and approaches. STRIPS, ADL and PDDL, are all based on state variables. Each viable state of the world is an assignment of values to the state variables, and the values of the state variables modified when actions are executed.

### 1.3.2 Forward Search

There are two fundamental search methods in the planning tasks, forward search and backward search (Russell and Norvig, 2010). Forward search proceeds regularly from the initial state to the goal state, by applying actions to generate successors at each state while backward search regresses the goal over actions to create subgoals until the initial state subsume a subgoal.

Researchers have been studying forward classical Intelligent Planning for more than half a century. The earliest research began in 1956 from the studies of Logic Theorist program and then General Problem Solver (GPS) (Newell et al., 1959). These intelligent planning systems, especially GPS systems, have been acting as important roles in artificial intelligence development, although real life planning problems are not solved yet.

Kautz et al. (1992) proposed a method as transforming a planning problem into a satisfiable (SAT) problem. Blum and Furst (1997) raised a new idea of research method, called planning graph. It acts fundamentally different from other common planning methods. Planning graph provided a new perspective on the planning problem in STRIPS-like domains, based on constructing and analysing a compact graph structure. Before start to search, a planning graph will be created to represent subsequent achievable facts in the following level based on the forward application of feasible actions at the present level. Nodes in the plan graph are shown as possible states, and the edges denote reachability through a certain action. The first level acts as the initial state, and the last level includes all the facts included in the goal state, and the mutually exclusive relations between facts and actions are maintained at each level of the plan graph. The planning graph can apply the constraints inherent in the problem in order to reduce the amount of search needed. Later, the *Blackbox* planning system (Kautz and Selman, 1999) which unifies the planning as satisfiability framework with the plan graph approach to STRIPS planning, had an extraordinary performance in the first IPC competition.

From 2000 to 2015, more advanced planning systems had begun to be presented in IPCs. These planning systems operate with heuristic search algorithms such as FF (Hoffmann and Nebel, 2001), FD (Helmert, 2006) and LAMA (Richter and Westphal, 2010) dominating several of the past editions of IPCs. Heuristic search is one of the most successful computation methods for planning (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001). The basic component of this method is the automatic derivation of a heuristic function that uses modelling language (such as STRIPS or PDDL) to inform the search from the declarative representation of the problem (McDermott, 1996). This is usually combined with appropriate search strategies and some search improvements, such as useful operations, continuous evaluation, and multiple search methods (Helmert, 2006; Hoffmann and Nebel, 2001).

Recently, width-based search has been developed different from heuristic search, as the significant progress in the planning field. Because of the support from structural, goal agnostic notion of state novelty, width-based search can mitigate the reliance on heuristics (Lipovetzky and Geffner, 2012). By assigning novelty value to states, width-based search has a powerful exploration mechanism. Meanwhile, Lipovetzky and Geffner (2017) demonstrated state-of-the-art satisficing planning strategies collectively called Best-First Width Search (BFWS), which combines width-based search with traditional heuristics in the greedy best-first search. After adding pruning of search states which are not novel enough, the incomplete but polynomial search $k$-BFWS can solve similar instances compared with IPCs winner planners such as LAMA and FF.

### 1.3.3 Backward Search

The idea of forward search is also called progression. On the opposite, backward search is called regression that belongs to a different branch. Regression for satisficing planning has a long history. Green (1981) conducted the original work of regression on theorem-proving based approaches.

By adopting the planning graph, Blum and Furst (1997) built a regression planner Graphplan. The Graphplan uses planning graph to regress as the heuristic search based on the heuristic function $h_G$ and search algorithm Iterative Deepening A* (IDA*). The $h_G$ is given by the index $j$ of the first level in the graph that contains the atoms in s without a mutex. The notion of (structural) mutex is that pair of atoms which cannot be both true in any reachable states and which can be computed in polynomial time (Blum and Furst, 1997). In each iteration of IDA*, it is a depth-first search and prunes the branch when the sum of the accumulated cost $g(n)$ and the estimated cost $h(n)$ beyond a given threshold.

In satisficing planning, HSPr (Bonet and Geffner, 2001), as one of the best known backward search planners, conducts a weighted A* algorithm (Pohl, 1970) with additive heuristic, $h_{add}$. HSPr is operating as the backward version of HSP2. HSP2 performs a forward search from the initial state to goal state and repetitively computes the heuristic at every state. While HSPr conducts a regression from the goal state to initial sate and HSPr avoids the repetitive computation of the atom costs and heuristic h$_{add}$ at every state. It computes these costs from the initial state and conducts a regression from the goal state. In this case, the estimated costs acquired for all atoms from the initial state could be directly utilised since the estimated distance from the goal state to the initial state is equivalent to the distance from the goal state to the initial state. Due to the faster computation of heuristic in HSPr, it can explore more nodes in the same time and create better plans than HSP2 in some domains. However, HSPr is not perfect since the recomputed heuristic contains indispensable information in many domains, and regression often generates spurious states. However, HSPr and HSP2 show competitive performance when compared with Graphplan. Although HSPr and Graphpan perform with different algorithms and heuristic in regression, they both encountered the same type of problems. Regression usually generates states which cannot achieve to any solutions. To solve this problem, *mutexes* are applied in HSPr and Graphpan to prune unreachable states in regression.

A partial-order planner (POP) is a regression planner even though it only considers the actions during the search, which means that the algorithm does not require special cases for the initial and goal state (Weld, 1994). POP starts the search with the empty plan for a problem and makes uncertain choices until causal links have supported all conjuncts of the precondition of all actions, and the potential interference has protected all threatened links. The ordering restrictions, $O$, of the final plan can specify only a partial order. In this case, the plan is a sequence of actions when any total order is consistent with $O$. The efficiency of the partial-order planner became faster and more adaptable at gaining the quickest path. However, there will be more computational power required for maintaining a queue of partial plans and sorting the queue.

At this stage, although some planners have been improving the search efficiency via applying regression, these planners all recognised the importance of mutex pruning. But, it is still unsatisfying for the result of impact analysis at the invariants on the current planners, especially in regression. In the paper from Alcázar and Torralba (2015), the study focused on evaluating the significance of the invariant adaptation when simplifying search tasks at the preprocessing phase. A state invariant is a logical formula over the atoms of a state that are true in all reachable states that may belong to a solution path. They worked with two types of state invariants: mutual exclusivity (mutex) and *"exactly-1"* invariant groups, and they pointed out the influence of spurious state/action and the weakness of $h_2$ in the detection. Finally, they computed the $h_2$ in two directions, forward and backward alternatively with fixed point procedure, and experiment results suggest that it is very advantageous to exploit invariants in regression.

Suda (2013) raised a new research perspective to conduct backward search. Duality mapping explains that there is no real difference between progression and regression in

STRIPS planning because they are the dual versions of each other. In the dual version of STRIPS problems, the initial state is the complements of goal state with respect to all fluents of the search problem; while the goal state is the complements of initial state with respect to all fluents of the search problems; Actions are reconstructed by swapping its precondition and the delete list to formulate dual actions. The duality mapping essentially transfers forward search into backward search and vice versa; hence this transformation was used to obtain the new theoretical insights to help interpret regression. Although more dual problems were solved via modifications of current heuristic-based planners, which was inspired by regression, the dual IPCs benchmark domains are still quite challenging to solve.

In the past decades, most search algorithms were designed based on the forward. There are many reasons to explain why most current planners abandon backward search. For example, techniques developed for forward search cannot be applied in backward search; duplicate detection is more complex due to generated partial states, and spurious states are hard to find during the search. Spurious states are those states containing a set of facts that are unreachable from the initial state (Alcázar et al., 2013). Early researches on heuristic search in classical planning studied both forward and backward search (Bonet and Geffner, 2001), but these disadvantages cause the worse performance of regression planners compared with progression planners. As a result, the research about backward search in classical planning was interrupted in many cases.

### 1.3.4 Bidirectional Search

Besides backward search, the integration of forward and backward search has been widely adopted and popularly named as the bidirectional search (Alcázar et al., 2013, 2014; Felner et al., 2010; Kuroiwa and Fukunaga, 2020; Politowski and Pohl, 1984). In bidirectional search, the applied forward and backward search start from the initial state and the goal state and the search end when there is an intersection between the frontiers of each search. In bidirectional searches, $Open_f$ and $Close_f$ are used to store generated states and expanded states in the forward direction, and $Open_b$ and $Close_b$ are used to store generated states and expanded states in the backward direction. If heuristic functions in bidirectional search estimate the distance from the current state to the fixed opposite goal state, we call it *front-to-end*, while if heuristic functions estimate the distance from the current state to changing frontiers, we call it *front-to-front*.

Politowski and Pohl (1984) proposed the D-node Retargeting (DNR), *front-to-front* heuristic search. Forward and backward search perform alternatively with $Open_f$ and $Open_b$, expanding $n$ states in one direction and reverses the direction. The $h$-value of the state $s$ is the evaluated distance between $s$ and a $d$-node. $d$ node in forward is a state with the highest $g_b(s)$ in $Open_b$ that $g_b(s)$ is the cost of the found path from $s$ to the goal state, and $d$ node in backward is a state with the highest $g_f(s)$ in $Open_f$ that $g_f(s)$ is the cost of the found path from $s$ to the initial state.

Alcázar et al. (2014) developed a Bidirectional Classical Planner biFD, and experimented series of bidirectional search strategies for detecting forward-backward frontier intersection. In biFD, forward and backward search interleave *front-to-end* and allocate the search effort in the direction with the least time. In addition, they also discussed the *front-to-front* variants with maintaining a set of states in the opposite frontier name as Backward Generated Goals (BGGs). The $h$-value in *front-to-front* version is the estimated cost from $s$ to the state with the lowest $h_{max}$ (Bonet and Geffner, 2001) among BGGs. Furthermore, biFD detected the intersection of the forward frontiers and the backward frontiers by comparing the frontiers through all states in the opposite BGGs. They also proved that biFD achieved better overall coverage than regression planning.

Felner et al. (2010) proposed a Single-Frontier Bidirectional Search (SFBS) which belongs to the *front-to-front* heuristic search method. The main character in SFBS is that a single queue ordered according to $h(u, v)$ where $u$ and $v$ are states in opposite frontiers, is adopted to replace the *Open* and *Close* for forward and backward search, and this single queue shift focuses between the forward and backward search expansions according to a jump policy. While, this elegant method poses a problem that a search node corresponds to a pair of states $(x, y)$, and there are multiple search nodes which contain states $x$ and $y$. Thus, SFBS requires the evaluation of $h(u, w)$ for multiple $w$. This is different from other unidirectional searches as well as bidirectional searches that for state $s$, its heuristic value $h(s)$ only needs to be evaluated at most once.

Kuroiwa and Fukunaga (2020) introduced a Top-to-Top Bidirectional Search (TTBS), *front-to-front* bidirectional search strategy. TTBS can be treated as a variant of DNR with some improvements. Firstly, TTBS uses the top (lowest $h$-value) node of the opposite *Open* as $d$-node rather than the state with the highest $g$-value in DNR. Secondly, TTBS limits the frequency of reevaluation heuristic value rather than reevaluates all states in *Open* whenever $d$-node changed in DNR. To detect the intersection of forward and backward search, TTBS checks if a generated state was already generated in the opposite direction. For quick intersection check, TTBS trades off the possibility of missed intersections since it implements as looking up of hash sets where even if an entry $S$ in $gen_b$ subsumes a forward generated state $s$, it is possible that $s \notin gen_b$ when $S$ has undefined values.

## 1.4 Research Gap

Inspired by the duality mapping, it is significant for us to explore the performance of width-based search on dual problems since the notion of width has been adopted to build state-of-the-art planners. Meanwhile, there is a knowledge gap that no current studies have focused on conducting regression in width-based search and integrating forward and backward width-based search. Thus, this thesis aims to test the performance of width-based search when solving dual problems and explore regression in width-based search as well as integrate regression and progression in width-based search. The research questions this thesis will look into are as follows:

### 1.4.1 Research Questions

1) What is the performance of width-based search when solving dual problems?
2) What is the influence of duality mapping on width-based search?
3) How is regression adopted in width-based search?
4) Can backward width-based search solve the current problems?
5) Can backward width-based search solve different problems compared with forward search?
6) Does the integration of forward width-based search and backward width-based search have a better performance in planning?

7) Is backward search competitive with forward search? If not, Why?

To answer the above questions, the research of this thesis will be carried out from the following aspects:

- Following the duality mapping rules, generating the dual version problems, comparing the performance of width-based algorithms on the original problems and dual problems.
- Starting from the design of state space, explaining the current STRIPS state mode, comparing forward search with backward search to give the regression state space.
- Modifying the existing width-based search according to the regression state space, testing the search performance of backward width-based search in different IPCs benchmark domains.
- Integrating forward best first width search and backward best first width search with different combinations, testing their performance and analysing the applicable situations for each combination.
- According to the experiment results in this thesis, answering whether backward search is competitive with forward search and proposing the challenges and further research direction of backward search.

## 1.5 Organization

The content of this thesis is divided into eight chapters. The structure is as follows:

Chapter One: Introduction.

This chapter begins with the introduction of the research background and significance of intelligent planning. Then we present the development history of intelligent planning knowledge and existing application scenarios. The research gap is proposed via extensive literature review, and then the research questions and content of this thesis are elaborated.

Chapter Two: Preliminaries

The second chapter illustrates the basic knowledge of classical planning. There are four aspects which help readers build a comprehensive review of classical planning. Firstly, the concept and basic language of classical planning are introduced. Secondly, the regression state space is formulated by comparing the progression state space. Thirdly, the required properties in regression are explained. Finally, bidirectional search techniques are introduced.

Chapter Three, Duality Width-Based Search.

This chapter compares the performance of Iterated Width ($IW$) and Serialized Iterated Width ($SIW$) in solving the dual problems and original problems in the IPCs benchmark domains. Then we conduct a series of analysis to figure out the weakness when width-based algorithm handles dual problems. Furthermore, we propose the regression state space is necessary for the width-based algorithm to realise backward search.

Chapter Four, Width-Based Backward Search

This chapter explains the principle of width-based search and provides a detailed process of changing forward *SIW* to backward *SIW.* Then, the impact of mutex during the backward search is discussed, and the method to eliminate the negative effect of mutex is applied to improve the regression performance. Then the IPCs benchmark domains are used to compare the performance of forward and backward *SIW.* After that, we use the *fooltille* domain as an example to explain the difference in performance between forward and backward *SIW.*

Chapter Five: Backward Best First Width Based Search

This chapter follows the structure in the fourth chapter. The Best First Width Search (BFWS) is first explained, and how it is processed via modifying BFWS and $k$-BFWS to meet the regression state space, where $k$-BFWS is a polynomial but incomplete variant of BFWS. Then, the mutex eliminating method is adopted to ensure the efficiency of backward search in BFWS and $k$-BFWS. Besides, the available forward heuristic functions in BFWS are transferred into the appropriate backward versions to ensure that they provide not only sufficient information to guide regression, but also maintain high efficiency of regression. Finally, the IPCs benchmark domains are used to compare the performance of forward and backward BFSW and $k$-BFWS.

Chapter Six: Bidirectional Search

This chapter starts with the introduction of bidirectional search. Then we integrate backward $k$-BFWS and forward $k$-BFWS according to different heuristic value updating techniques,*front-to-end* and *front-to-front* and different frontiers intersection methods, *check Head* and *check Close*. Six combinations are considered. We compare these six different combinations and propose characters of each combination method.

Chapter Seven: Discussion and Expectation

This chapter focuses on existing challenges in backward search. We first review the significance of backward search. Then we analyse weaknesses of backward search in three aspects: partial state, mutex detection and forward domain description, and we raise possible solutions. There are limited experiments executed to prove our assumptions. Finally, we propose further implications and research directions for backward search.

Chapter Eight: Conclusion.

This chapter gives a conclusion for this thesis. It reviews the research steps and main topics in each chapter.

# Chapter 2

# Preliminaries

## 2.1 Classical Planning

A classical planning problem can be considered as a path-finding problem in a directed graph. Nodes in this graph represent states, and edges represent actions that expand parent nodes to child nodes under the constrain defined in actions. A plan in the classical planning problem is a sequence of actions that transform the initial state into a goal state, which corresponds with a path from the root node in the graph to leaf nodes whose state is one of the goal states of problem. (Bonet and Geffner, 2001).

The state space provides the basic model for classical planning problems. A state space consists of a set of states $S$, a set of actions $A$, a deterministic transition function $F$ that defines how actions map one state into another state, and the cost function $C(a, s)$ that estimates the cost of applying an action in a specific state. The state space combining with the initial state $s_0$ and a set of goal states $S_G$ is called the state model.

The state model of a classical planning problem can be represented as follows:

Definition 1.1 (Classical Planning Model).

A planning model is a tuple $S = <S, s_0, S_G, A, f, C>$ where:

- $S$ is a finite and discrete set of states $s$,
- $s_0 \in S$ is the initial state,
- $S_G \in S$ is a set of goal states,

- $A$ is a set of actions,
- $A(s) \subseteq A$ are applicable actions in each state $s \in S$,
- $s' = f(s,\ a)$ is the state transition function for $a \in A(s)$ from state $s$ to state $s'$.
- $C(a,\ s)$ denotes the cost of applying action $a$ in state $s$.

the $s(a)$ stands for an action applied in a state $s$ based on the transition function $f(s,\ a)$. A state resulting from a series of actions is always defined as follows:

$$\mathrm{s}[\epsilon] = s$$
$$\mathrm{s}[a_0,\ \ldots,\ a_n] = (s[a_0,\ldots,a_{n-1}])a_n$$

A sequence of actions $a_0,\ a_1,\ \ldots,\ a_n$, is the solution of classical plan problem when generated state trajectory $s_0,\ s_1 = f(s_0,\ a_0),\ \ldots,\ s_{n+1} = f(s_n,\ a_n)$ such that $a_n \in A(s_n)$ is applicable in $s_n$, and the sate $s_{\mathrm{n+1}}$ is a goal state, i.e, the sequence of actions $a_0,\ a_1,\ \ldots,\ a_n$ is the plan of $s_0$ to $s_{n+1}$ if $s[a_0,\ldots,a_n] \in S_G$.

Normally, the cost function assigns the non-negative values to each action under a different state. The cost of a plan can be represented as the total of the cost of its actions. If there is no cost function in the state model, actions are assumed to become the uniform cost and usually be assigned with value 1.

A plan is declared to be optimal when the total cost of actions is minimised among all possible plans achieving a goal state. When the cost is uniform, the optimal plan is the plan with the shortest path.

## 2.2 STRIPS

At the early stage, the declarative language is employed to build the state models. These models are introduced by an explicit description of state space and explicitly coding the transition function and the action applicability conditions to meet the target domain. With new demands for describing more lager and complicated problems, numerating the state space is not reasonable. In this case, one efficient approach is factored representations. A set of variables are assigned to the state when domains are finite and discrete. Boolean variables, known as atoms, facts, or fluents, are the most frequent description in planning. This representation is known as STRIPS (Fikes and Nilsson, 1971).

A planning problem in STRIPS is also described by a tuple P = $<F, O, I, G>$ where:

- $F$ is a set of boolean variables, also known as fluents, facts, or atoms.
- $O$ stands for a set of tuples, which represents a set of operators. Each operator has the precondition, add and delete list under the form $O = <Pre(o), Add(o), Del(o))>$ where $Pre(o)$, $Add(o)$, $Del(o) \subseteq F$
- $I$ denotes the initial state where $I \subseteq F$.
- $G$ denotes the set of goal states where the $G \subseteq F$.

Comparing with the state model, the STRIPS problem P = $<F, O, I, G>$ specifies a progress state-space $<S, s_0, S_G, A, f, C>$ by the STRIPS factored representation. Each state $s \in S$ is a subset $s \subseteq F$ of the set of fluents in which the facts $p \in s$ has the value *true*, while the facts $p' \in F \setminus s$ are assumed to be *false* (Lipovetzky et al., 2013).

The initial state $I$ is $s_0 \subseteq S$. The set of goal states $G$ are those states that meet the goal condition, represented by $S_G = \{ s \mid G \subseteq s\}$. The actions $A(s) \subseteq A$ can be applied in a given state $s$ when the preconditions are true in $s$, denoted as $A(s) = \{ o \mid Pre(o) \subseteq s\}$. The transition function $f$ transfers the state $s$ into $s' = s$ - $Del(o)$ + $Add(o)$ by applying operator $o \in O$ at $s$, where placing the propositions in $Add(o)$ to true and placing the propositions in $Del(o)$ to false.

A planning task has a solution if the state in the path $s_0$, $s_1$,...,$s_n$ contains all goal fluents, and this state is generated by a finite sequence of actions $a_0$, $a_1$,...,$a_n$ with the condition that each action $a_i$ is applicable in $s_i$, and the state $s_0$ is the initial state.

## 2.3 PDDL

Planning Domain Definition Language (PDDL) is a recent attempt method to standardise the planning domain and problem description languages. Along with the characteristics of STRIPS, PDDL also adds a finite number of predicates, variables, and constants for enhancing the description capabilities. PDDL has become a standard problem description language in IPCs.

There are four characters in PDDL to define a search problem: the initial state, the actions that are applicable in a state, the result of applying an action, and the goal test.

Each state is represented as a combination of fluents that are instant and functionless. For example, school $\bigwedge$ major might represent the state of a student. There are two assumptions when describing the state, *closed world assumption* and unique names assumption. The fluents not mentioned are defined as false, and their names are unique individuals. The states are designed carefully in work tasks that each state can be considered as a conjunction of fluents, which can be formed by logical inference McDermott et al. (1998).

PDDL divides the classical planning problem into two parts: domain description and the problem description (Yang et al., 2019). In PDDL, a set of problem descriptions belong to a domain description. The domain description contains the domain name (the name of each domain), requirements (the model elements declaration to the planner that the PDDL model used), the object type definition, predicates, (properties of objects that we are interested in, which can be true or false) and actions (a set of operators and the description consisted of the action name, precondition, and effects with parameters which will be instantiated with objects during execution). Actions are represented by a set of action schemas that implicitly define the actions $(a)$ and result $(s,\ a)$ with functions: $((s \cup Add(a)) \setminus Del(a))$. The schema includes the action name, precondition, effects and a list of variables related to this schema. The precondition and effects of the action represent each conjunction of literals, as either positive or negative. The precondition is used to evaluate the applicable action $a$ at state $s$. The effects are used to determine the result of executing the action $a$ at state $s$ by deleting fluents that are negative literals named as delete list in the effects and adding fluents that are positive literals named as add list in the effects. The problem description contains the problem name, belongs domain name, objects things in the world that interest us, initial state, (conjunction of fluents with instantiation following the closed-world assumption), and goal state (the state where the search should end up).

Although it is well known that PDDL can effectively model many planning problems by its powerful expressivity, most classical PDDL can encounter the STRIPS language in which action preconditions are conjunctions of (positive) literals, and all effects are unconditional. These planners instantiate the actions as defined in PDDL, which include factors such as predicates, variables, and constants transformation into a propositional representation in STRIPS. Thus, the STRIPS representation will be adopted in this thesis, while the planner input files are domains and problems.

It is necessary to explain the evolution process from STRIPS to PDDL. Because by understanding this development and filling the knowledge gaps, the performance of regression in different domains can be assessed and analysed more easily. For the purpose of this dissertation, only STRIPS representation is used, without further discussion on other representations.

## 2.4 Complexity

For a planning problem $P$ in the factored representation, there are two decision problems to consider the theoretical complexity of this problem. One decision problem PlanExt($P$) asks that whether there exists a plan $\pi$ for problem $P$. The other decision problem PlanCost($P$, $k$) is the question of whether there is a plan $\pi$ with cost($\pi$) $< k$ for the problem $P$, where $k$ is a positive constant value. In the worst case, planning problems are unmanageable since both decision problems are *PSPACE*-complete (Bylander, 1994). Even though strict restrictions are adopted, the issues are still challenging. However, sub-optimal planning can sometimes be easy to operate, and hence the planning procedures are generally assessed on a set of benchmarks, without considering the worst case. If the algorithm focuses on finding a plan rather than an optimality guarantee and pursuing the speed rather than quality, the problem is known as *satisficing classical planning*.

## 2.5 Heuristic

The search algorithm is an uninformed search or bind search if the search selects all possible actions from the search space to find all possible solutions for the problem without the help from supplementary information. By contrast, the informed search strategy can find solutions more efficiently than uninformed strategy since it uses problem-specific knowledge such as how far we are from the goal state and the current path cost. Through applying the informed search, agents can spend less time to plan during the search with more reasonable actions. The informed search algorithm uses the idea of heuristic, so it is also named as heuristic search. The heuristic evaluation is a function which is accepted in the informed search, and it helps planners locate the most encouraging path. The current state is treated as the input of a heuristic function, and the output is the estimation of the distance from the current state to the goal state. The well-designed heuristic function $h^*$ maps any state to the optimal solution from that state. Heuristic search is currently the most effective search method in IPCs and has been widely studied by researchers (Bonet and Geffner, 2001; Helmert, 2006; Hoffmann and Nebel, 2001; Richter and Westphal, 2010). Therefore, in the rest of the section, we briefly review heuristics and heuristic search algorithms.

An admissible heuristic function is given as: $h(s) \leq h^*(s)$ for all states $s \subseteq S$. The $h(s)$ provides the lower bound estimated cost of the solution from $s$, while the $h^*(s)$ gives the cost of the optimal plan. Hence, admissible heuristic never overestimates the cost of the optimal solution from any state. The search algorithms explore first to states with the lowest cost when admissible heuristics are applied. As a result, the algorithms are guaranteed to find an optimal solution. Heuristic search, however, is not able to find the best solution when the heuristic function is non-admissible, but it guarantees to find a suitable solution within a reasonable time (Lipovetzky et al., 2013).

### 2.5.1 Delete-Relaxation Based Heuristics

The reasonable method to formulate the admissible or non-admissible heuristics is to simplify the original problems with fewer restrictions on the actions, also known as relaxation (Pearl, 1984a). The most adapted relaxation is delete-relaxation (Bonet and Geffner, 2001) and it has been practiced quite successfully in the planner FF, FD and LAMA. In delete-relaxation, the delete effects of actions are ignored, i.e. $Del(o) = \varnothing$ for all $o \subseteq O$, so the new state $s'$ is generated by only monotonically adding fluents in the add list of applicable actions at current state $s$.

Given a STRIPS problem $P = <F, O, I, G>$, its delate relaxation $P^+$ is represented by the tuple $P^+ = <F, O^+, I, G>$, where:

$$\mathrm{O}^+ = \{ <Pre(o), Add(o), \varnothing > \mid o \subseteq O\}$$

The admissible heuristic $(h^+)$ in delete-relaxation problem $P^+$ is the cost of the optimal plan $(\pi^+)$. In delete-relaxation, each operator needs to be applied only once; then the goal state can be achieved with no more than $|O|$ actions since new states achieved in a relaxed plan increases monotonically. However, the computation of heuristic ($h^+$) is NP-hard (Bylander, 1994).

### 2.5.2 The Max and Additive Heuristic

Since the computation of the $h^+$ is intractable, the max heuristic $h_{\text{max}}$ and additive heuristic $h_{\text{add}}$ are introduced to approximate $h\ ^+$. $h_{\text{add}}$ estimates pessimistically by summing over the cost of a set of fluents. The $h_{\text{add}}$ value at state $s$ is always an upper bound to the optimal cost $h^+$ at $s$ adopting the delete relaxation, and it is not admissible. $h_{\text{max}}$ estimates optimistically by adopting the maximum cost of an fluent in the set. The cost of an fluent depends on the cost of its *best supporter*, and the *best supporter* is the action which makes the fluent true with minimal estimated cost. The estimated cost of an action is determined by the cost of fluents in its precondition (Bonet and Geffner, 2001). The $h_{\text{max}}$ value at state $s$ is always a lower bound to the optimal cost $h^+$ at $s$ (Betz and Helmert, 2009). The max heuristic, unlike the additive heuristic, is admissible because the cost of achieving all subgoals cannot be lower than the cost of achieving the costliest subgoal.

### 2.5.3 The FF Heuristic

$h_{\text{FF}}$ estimates the distance from a given state $s$ to the goal state using a greedy algorithm based on relaxed planning graphs, and the heuristic value is the cost of suboptimal relaxed plan extracted from relaxed planning graphs (Hoffmann and Nebel, 2001). The FF heuristic computation contains two steps: One is building a relaxed planning graph in progression; another is extracting relaxed plan in regression. The relaxed planning graph includes two types of layer, the fluent layer and the action layer. The fluent layer includes all facts that are true in the current state, and the action layer consists of all applicable actions in the former fluent layer. In the relaxed planning graph, the first fluent layer includes all fluents which are true in the initial state and then expands to the action layer which contains all applicable actions at the initial state and then extends to a new fluent layer which includes fluents added by applicable actions and along with fluents held in the former fluent layer, without considering the delete list. The above-repeated expansion processes will stop when there is no new fluent to add to the graph, and then the whole graph is built in polynomial time (Blum and Furst, 1997). For the relaxed plan extraction, all best $h_{\max}$ supporters which add the goal fluents are inserted into the relaxed plan first. Then the best $h_{\max}$ supporters adding the precondition fluents of the best $h_{\max}$ supporters in the relaxed plan will be inserted into the relaxed plan. These steps will be repeated until all precondition flunets of $h_{\max}$ supporters in the relaxed plan are supported by other supporters, or they belong to the first fluent layer. Then, the $h_{\mathrm{FF}}$ value is the number of the best supporters in the relaxed plan (Hoffmann and Nebel, 2001). The relaxed plan also can be defined recursively following the $h_{\mathrm{add}}$ best supporters (Keyder, 2010). The best supporter of a fluent $p \notin s$ based on $h_{\max}$ or $h_{\mathrm{add}}$ is the action $a \in O(p)$ with the smallest $h(a)$, where $O(p)$ is the set of actions in $O$ that add $p$, and $h(a)$ is the sum of heuristic values to achieve fluent $q \in Pre(a)$(Bonet and Geffner, 2001).

### 2.5.4 Heuristic Family $h^{\mathrm{m}}$

Geffner and Haslum (2000) introduced a whole family of admissible polynomial heuristics $h^{\mathrm{m}}$ that trade accuracy for efficiency by regressing. When $m = 1$, $h^m$ is $h_{max}$, and when $m$ is large enough, $h^m$ is the optimal heuristic function and equal to $h^*$. Generally speaking, $h^m$ treats the cost of the most expensive single fluent in the set, as the cost of a set of fluents when $m = 1$; it treats the cost of the most expensive fluents pair in the set, as the cost of a set of fluents when $m = 2$, and so on (Lipovetzky et al., 2013). With the help of $h^2$, the mutex fluents pair $<p, q>$ is that the heuristic value of $h^2(<p, q>)$ is infinite (Geffner and Haslum, 2000). $h^m$ is used to guide an IDA* search and evaluate the performance of optimal planners (HSP, HSPr, and Graphplan) across several domains, and it also gives some advice that the higher-order heuristics are helpful in the search, but this requires sacrificing computing resources (Geffner and Haslum, 2000). However, if we calculate it once at the search beginning, it is a reliable choice to guide regression for optimal planning (Geffner and Haslum, 2000). $h^m$ contributes to a stable foundation for exploring the heuristic search.

## 2.6 Planning as Heuristic Search

Many algorithms have been proposed to perform a forward heuristic search in the state space model effectively, such as Best-First Search (BFS) (Pearl, 1984b), Greedy Best-First Search (GBFS) (Russell and Norvig, 2002), A$^*$ (Hart et al., 1968), and Weighted A$^*$ (Pohl, 1970). All of these algorithms appeared powerful in previous planning tasks. In this thesis, we discuss two of the most successful algorithms Best-First Search, and its variant Greedy Best-First Search to help readers understand the following algorithms. Greedy Best-First Search can combine with the width-based exploration to build the state-of-the-art planning method Best First Width Search (BFWS), which we edit to the regression version.

### 2.6.1 Best-First Search

Best-first search algorithms use two separate lists for saving search states, the *Close* list and the *Open* list. The *Open* list stores the evaluated states $s$ which have not been expanded, while, the *Close* list saves states which have been already expanded. The algorithm repeats the actions of choosing one state from the *Open* list, generating successors of the selected state, inserting the selected state into the *Close* list, and putting successors into the *Open* list based on the evaluated heuristic value. The algorithm stops until the goal $G$ is true in the selected state from the *Open* list for expansion. Then the path is retrieved up to the initial state, and the solution is returned. Best-first search algorithm stores all states in the *Open* list according to the linear combination of accumulated cost up to states and the heuristic value.

### 2.6.2 Greedy Best-First Search

The greedy best-first search (GBFS) focuses on the state that is the closest to the goal state. Thus, it only adopts heuristic functions to select states. At each state selection step, it selects the state with minimum heuristic value to ensure that this state is as close to the goal state as it can without considering the cost of the path. As a result, even in a finite search space, GBFS search is still incomplete. There are seven steps to describe the Greedy-Best-first search logically:

- **Step 1:** Place the initial state into the *Open* list.
- **Step 2:** If the *Open* list is empty, stop and return failure.
- **Step 3:** Select the state $s$ with the lowest value $h(s)$ from the *Open* list.
- **Step 4:** Check selected state $s$, whether it is a goal state. If it is the goal state, then return plan and terminate the search, else move to Step 5.
- **Step 5:** Expand the state $s$, generate the successors $s'$ of state $s$, and insert state $s$ into the *Close* list.
- **Step 6:** For each successor $s'$, the algorithm calculates heuristic value $h(s')$ and then check whether the state $s'$ is in *Open* or *Close* list. If not, add it to the *Open* list based on evaluated heuristic value $h(s')$.
- **Step 7:** Return to Step 2.

GBFS has been successfully adopted by satisficing classical planners because GBFS is likely to lead to a solution quickly.

## 2.7 Width-Based Search

### 2.7.1 Novelty

Lipovetzky and Geffner (2012) introduced a width parameter from a different perspective, which can restrict the complexity of classical planning problems. The width-based algorithms employ a powerful exploration mechanism based on a structural goal-agnostic notion novelty. The specific definition of novelty is described as the new state $s$ is the first state that makes tuples of fluents true, and the size of the smallest tuple of fluents is the novelty of $s$. If the new state does not make any new tuples of fluents true, the novelty of $s$ is $n + 1$ where $n$ is the number of variables.

Following example will serve to explain this, there are four fluents $\{a, b, c, d\}$ in the problem $P = <F, O, I, G>$, and not mentioned fluents are false. The first generated state $s$ in the search problem $P$ is $\{a, b\}$, and it is the first state which makes the tuple of fluent $\{a\}$, tuple of fluent $\{b\}$ and tuple of fluents $\{a, b\}$ true. The definition of novelty is the minimum tuple size in the state, so the novelty of $s$ is 1 where the minimum tuple is $\{a\}$ or $\{b\}$. The novelty of the successor $s' = \{b, c\}$ of state $s$ is 1 since the newest tuple is $\{c\}$. While the novelty of state $s'' = \{a, c\}$, which is generated by $s'$, is 2 since $s''$ is the first state which makes the tuple of fluents $\{a, c\}$ true, not tuple $\{a\}$ which is made true in $s$ and not tuple $\{c\}$ which is made true in $s'$. While if the newly generated state does not make any new tuples of fluents true, such as new states $\{a, b\}$ or $\{b, c\}$ or $\{a, c\}$, the novelty of theses new states will be 5.

### 2.7.2 Iterated Width

Lipovetzky and Geffner (2012) proposed a algorithm, called *Iterated Width* ($IW$), consists of a sequence of $IW(i)$ for $i = 0, 1, 2, \ldots$ and each $IW(i)$ is one $i$-width search. $IW$(i) is a BFS search with state pruning. In each $i$-width search processes, if the bound is set to 2 ($i = 2$), this means the search will only keep new states whose novelty is less or equal to 2, while states will be pruned if the novelty is beyond 2. If we set the bound to 0 ($i = 0$), there is no new state generated during the search. If we set the bound to 1 ($i$ $= 1$), there will generate $n$ new states where $n$ is the number of fluents in this problem. Furthermore, if the bond is set to $n$, the $n$-width search only removes duplicate states. In conclusion, $IW(i)$ is a breadth-first search that prunes newly generated states when their novelty is greater than $i$.

$IW(i)$ algorithm is complete for problems whose widths are bounded by $I$, while it is not necessarily complete since $IW(i)$ always tries to use the minimum width to solve the problem. This means that $IW$ algorithm solves the problem in $IW(i)$ where $i$ is smaller than the problem truly width. This minimum width of problem $P$ is called the effective width $w_e(P)$, and the effective width is always smaller than or equal to the width of the problem $w(P)$. The effective width is not well-present with definition, but it can directly reduce the time complexity during running $IW$ algorithm to search the goal. Since for a solvable problem $P$ with width $w$, $IW(w)$ solves $P$ optimally in time exponential in $w$.

The experiment results provided by Lipovetzky and Geffner (2012), shows that almost 90% of problems in IPCs benchmark domains have a low width (1 or 2) when there is only one fluent in the goal state by splitting the problem with $N$ atomic goals into $N$

problems with a single goal fluent. It is noteworthy that *IW* outperforms other blind-search algorithms such as Iterative Deepening (ID) and Breadth-First Search (BrFS) and is competitive with standard heuristic-based search GBFS + $h_{add}$ at single-goal problems. There is confirmed certainty that by adopting the width notion in breadth-first search for pruning new states whose novelty is above the default value, the performance of blind search procedure can be significantly enhanced. This also suggests that the complexity of problems depends on conjunctive goals. In fact, this thought has already been raised previously, and the goal decomposition is critical and essential as a planning method to handle this situation.

### 2.7.3 Serialization

Lipovetzky and Geffner (2012) raised a concept called "serialization" which is used for applying the width notion in the benchmark problems with conjunctive goals. A serialization $d$ for a problem $P$ = $<F, O, I, G>$ is defined as a sequence of formulas $G^1, \ldots, G^m$, where $m$ is the number of fluents in $G$. $G^1$ only contains one fluent in $G$; $G^2$ is extended by $G^1$ adding one more fluent from $G$; $G^3$ is generated by the same method as mentioned in $G^2$, and so on; $G^m$ contains all fluents in $G$, so $G^m = G$. The serialization decomposes the original problem $P$ into a set of subproblems $P_{\mathrm{d}} = P_1, \ldots, P_m$. $P_1$ represents one subproblem of $P$, but the goal state is $G^1$, and $P_2$ represents one subproblem of $P$, but the goal state is $G^2$ and so on. If each subproblem can be optimally solved, the initial state of subproblem $P_i$ should be the last generated state from the subproblem $P_{i\text{-}1}$, which optimally solves the goal $G^{i\text{-}1}$. The width of a problem in a given serialization $d$ is written as $w(p, d)$, which is the max-width over the subproblems in $P_d$.

The most important contribution of problem serializing is that both width and effective width can be reduced distinctly via decomposing the multigoal problem into a set of subproblems. As a result, this allows the *IW* algorithm to run on actual IPCs benchmarks competitively. The *IW* algorithm can be directly applied in these multigoal problems without applying the serialization. However, both width and effective width are too large, which conflicts with the original intention of the *IW* algorithm that the lower width in classical benchmarks is associated with the affordable time and space complexity.

### 2.7.4 Serialized Iterated Width

A search algorithm called *Serialized Iterated Width* (*SIW*) was proposed by Lipovetzky and Geffner (2012). In order to realize the serialization of problems and find the solution in each subproblem $P_k$, $k = 1,\ldots,|G|$. *SIW* is formed by a sequence of *IW*, and it sequentially calls *IW* procedure over $|G|$ subproblems where the $G$ is the number of fluents in the goal state. Since *IW* acts as a blind-search procedure, *SIW* is able to take such advantage that the goal state of the problem is not required to be informed in advance, and only requires the goal state at the end of the search. The features of *IW* in *SIW* are used both for decomposing problems into a sequence of subproblems and solving them individually. The plan for the problem is the serialization of plans obtained from all subproblems.

*Serialized Iterated Width* at a classic planning problem $P = <F,\ O,\ I,\ G>$ is defined as that *SIW* is a serial of *IW* over the problems $P_k = <F,\ O,\ I_k,\ G_k>$, $k = 1,\ldots,|G|$, (Lipovetzky and Geffner, 2012) where

1. $I_1 = I$,
2. $G_k$ is the first consistent set of fluents achieved from $I_k$ such that $G_{k-1} \subset \mathrm{G_k} \subseteq G$ and $|G_k| = k$; $G_0 = \varnothing$,
3. $I_{k+1}$ is the achieved state of $G_k$, where $1 < k < |G|$.

*SIW* starts with calling 1-th *IW* with the initial state $I$ in $P$. Then *IW* will sequentially call $i$-width search to find the goal $G_1 \subseteq G$. If $G_1$ is achieved in one newly generated state $s_1$, an goal consistency check will be executed for $s_1$ to check whether $G$ can be consistently achieved. The goal consistency check estimates whether the $h_{max}$ value of $s_1$ is infinite when the actions which delete the achieved goal $G_1$ are excluded (Bonet and Geffner, 2001). 1-th *IW* will stop if the $h_{\max}$ value is not infinite. Then *SIW* will run 2-th *IW* with the initial state $s_1$. 2-th *IW* stops when *IW* generates a state $s_2$ that achieves two goals $G_2$ where $G_2 \subseteq G$, and $s_2$ passes the goal consistency check. Then the same processes will be repeated till k-th *IW* generates the state $s_k$ which includes the $G_k = G$. The whole processes continue without the help from heuristics, and the goal will be identified till *IW* generates a set of fluents $G'$ when $G_{k-1} \subset G_k \subseteq G$ and $|G'| = k$. The solution for *SIW* is a concatenating of solutions for each subproblem, $P_1$, ..., $P_k$, where $k = |G|$. This is the procedure of adopting *IW* for *SIW* at both constructing the serialization and solving the subproblems.

Because *IW* is a non-optimality algorithm, *SIW* keeps the same feature which cannot guarantee the optimality of results. Additionally, the original goals of *SIW* are split into a set of subgoals which cannot be guaranteed for reachability; therefore, *SIW* is absolutely not complete. However, *SIW* still acts as a powerful blind-search algorithm since it can reduce the effective width in each subproblem. The effective width of a problem in *IW* is restricted by its actual width $w(P)$, which cannot be guaranteed in *SIW*. Lipovetzky and Geffner (2012) compared *SIW* with Greedy Best-First Search adopting additive heuristic $h_{\text{add}}$ in the heuristic search planner. The results indicate that *SIW* performance becomes powerful if the problem decomposition and the non-goal oriented form of pruning in *IW* are both employed, and it is competitive with the best heuristic estimators $h_{\text{add}}$. The experiment results also show that the effective width of subproblems in the selected domains is ranged from 1 to 4, and the average effective width is between 1 and 2.

### 2.7.5 GBFS-W

The notion of width shows great potentials to assist blind-search algorithms for finding solutions easily. However, there is no doubt that heuristic search still is the mainstream method in classical planning. However, if there are heuristic plateaus occurred during heuristic search such as GBFS, an issue remains as the heuristics function becomes blind. When a large number of iterations in GBFS do not generate states with a lower heuristic value, heuristic plateaus will form, and newly generated states in the *Open* list keep the same heuristic value (Hoffmann and Edelkamp, 2005).

Due to the conspicuous performance of width-based blind search in IPCs benchmark domains, the combination of heuristic and width is expected to overcome the difficulties in classing planning. Lipovetzky and Geffner (2017) introduced the integration of the width-based exploration with the standard greedy best-first search algorithm, named as GBFS-W, and it has been successfully employed by high-performance planners. In order to break the heuristic plateaus, the novelty is adopted. In the GBFS-W algorithm, the novelty only affects the search if the greedy best-first search is obstructed. The experiment results show an obvious improvement after adopting GBFS-W.

### 2.7.6 Best-First Width Search

In the context of the best-first search with lexicographic "preferences", this combination can generate a search algorithm family which is called best-first width search (BFWS) Lipovetzky and Geffner (2017). The "preferences" of a state can be defined as the joint of two or more heuristics and novelty measures. With lexicographic ordering implementation, the evaluation function which is used to guide to BFWS can be represented as $f = <h, w>$, where $h$ are the heuristic functions used in BFWS, and $w$ is the novelty measurement function. $w$ and $h$ perform the lexicographic order, and this order can be simply understood as the importance of each notion.

Even if the definition of novelty in $IW$ algorithm is described clearly, the novelty is measured slightly different by taking multiple functions into account. In BFWS, novelty $w(s)$ is the smallest size of the fluents tuple which is first true in the newly generated state $s$ and false in all states $s'$ generated before $s$ when these states have same heuristic values. For example, if there are $m$ heuristic functions adopted in BFWS, $w(s)$ will be written as $w_{<h_1,...,h_m>}(s)$. The $w(s)$ for the specific $i$-th heuristic, represents as $w_{h_i}$(s), is 1 *iff* $s$ is the first state that makes one fluent true but false in all states $s'$ generated before $s$ where $h_i(s) = h_i(s')$ for all $1 \leq j \leq m$.

BFWS adopting the $f_1$ evaluation function is named as BFWS($f_1$) where $f_1 = <h, w>$. In BFWS($f_1$), the value of $w(s)$ would not be computed exactly for efficiency reasons where novelty is 1 or greater than 1. In BFWS($f_1$) the state $s'$ generated before state $s$ is favored if $s'$ has the smallest heuristic value or with the lowest novelty where $s$ and $s'$ have the same heuristic value. If the role of $h(s)$ and $w(s)$ in BFWS changed, lexicographic ordering will be redesigned with the primary role as novelty for guiding the search, and with the secondary role as heuristic to break plateaus. The evaluation function, in the above description, is represented as $f_2 = <w, h>$. The preferred states in BFWS($f_2$) are those with the smallest novelty and among those, the one that produce the smallest heuristic value. As a result, BFWS($f_2$) is not greedy and may expand states that do not have the minimum heuristic value in *Open* list. The $f_3$ evaluation function which adopts the landmark heuristic as the first role and employs the FF heuristic as the second role can be written as $f_3 = <h_L, h_{FF}>$. The landmark heuristic estimates the distance to the goal state $G$ from a given state $s$ to be the number of landmarks that still need to be achieved, from the landmark graph built once from the initial state (Hoffmann et al., 2004). If $w(s)$ remains as the primary role in the search and combine $h_L$ and $h_{FF}$ to break plateaus, the performance of this integration becomes more competitive than merely adopting $f_3 = <h_L, h_{FF}>$. It forms as BFWS($f_4$) algorithm, which works with the evaluation function as $f_4 = <w, h_L, h_{FF}>$ with $w = w_{h_L}, w_{h_{FF}}$.

The most powerful algorithms among the BFWS algorithm family is BFWS($f_5$), where the evaluation function is $f_5 = <w, \#g>$. The $f_5$ evaluation function uses $\#g$ to break ties where $\#g$ represents the number of unachieved top problem goals in the current state and uses $w$ to guide the search as the primary influence factor where $w$ is computed under counter $\#g$ and counter $\#r$. As a result, $w$ is written as $w = w_{\#g,\,\#r}$. $\#r$ represents the number of fluents that have been true in the latest relax plan, and the relax plan is only computed when the number of unachieved goals decreased in a new state $s'$. Then, all children states generated from $s'$ will keep the same relax plan till a new children state achieves more goals. The fluents $F_\pi$ in a relax plan $\pi$ are fluents in the preconditions or positive effects of actions $a$ in $\pi$ (Lipovetzky and Geffner, 2017). BFWS($f_5$) prefers novelty measures with a 3-value precision (namely, $w(s)$ is 1, 2, or greater than 2) rather than 2-value in BFWS($f_1$) since it takes advantage that $\#g$ and $\#r$ are computationally cheap.

## 2.8 Backward Search

Backward, as an old idea in planning, searches from the goal state rather than forward from the initial state. However, a set of fluents $\{p, q, r\}$ in the forward initial state describes the unique state where fluents $p$, $q$ and $r$ are true, and other fluents are false. Conversely, the same set of fluents only represents fluents $p$, $q$ and $r$ are true if this set of fluents appears in backward states, so the backward states are also named as partial states.

The regression state space can be defined in analogy to the progression state space. The regression space $R_p$ associate with a STRPS problem $(F, O, I, G)$ is given by the tuple $R_p = <S, s_0, S_{\mathrm{G}}, A, f, C>$ where

• The states $s$ are set of flutes from $F$ which are same in progression but they should be viewed as the subgoals to be met, corresponding to a set of world states that satisfy it.

• The initial state $s_0$ is the goal state $G$.

•The goal states $S_{\mathrm{G}}$ are states from which $s \subseteq I$.

• The actions $A(s)$ applied in a given state $s$ are $a \in O$ when the delete fluents are not included in $s$ denoted as $Del(a) \cap s = \phi$ for consistency, and at least one fluent in the add list appears in $s$ denoted as $Add(a) \cap s \neq \phi$ for relevance.

• The transition function $f$ transfers the state $s$ into $s' = s$ - $Add(a)$ + $Pre(a)$ by applying applicable actions $a \in O$.

• $C(a,s)$ is 1 if not specific declaration.

Similar to progression, the solution of regression is a finite sequence of actions $a_0$, $a_1$, ..., $a_n$, applied in a sequence of applicable states $s_0$, $s_1$, ..., $s_n$, using the transition function $s_{i+1} = f(a_i, s_i)$, for $i$=0, 1, ..., $n$, $a_i \in A(s_i)$, and the $s_{n+1} = f(a_n, s_n) \in S_G$.

Different from progression, search states $s$ in regression also represent the subgoals to be met, and it is important to keep backward search manageable. One solution is discarding non-relevant actions. The new states $s'$ should be filtered out if it is a stronger subgoal $s' \supseteq s$ when $s$ regresses over the not-relevant actions, since it is more challenging to regress over $s'$ to meet the goal than regress over $s$. So it is crucial for the performance of regression to only regress over actions that are relevant for the current state.

By comparing progression and regression, there are many interesting points worth noticing. The initial state $I$ in progression plays a similar role to the one played by $G$ in regression and vice verse. Meanwhile, the precondition and delete list of each action exchange their roles in a certain way. Actually, the above similarities are not a coincidence, and progression and regression are strictly related proved by the duality mapping.

## 2.9 Bidirectional Search

Backward search has already been applied earlier in the graph search, but how to conduct effective graph searching is a prominent problem with many attempted practices. The most commonly used search methods are breath-first search (BFS) and depth-first search (DFS). In the general graph, BFS and DFS start the search in one direction from the initial point to the target point. Conversely, we can also start to search from both directions, such as, by using bidirectional search.

Bidirectional search in the graph traversal is used to find the shortest path from the initial state to the goal state in a directed graph. The algorithm runs two searches at the same time. One searches forward from the initial state and the other searches backward from the goal state. When two search routes meet in the middle, the bidirectional search stops. Assuming a tree with a branch factor $b$, the distance between the initial state and the goal state is $d$. The time and space complexity of forward and backward search are both $O(b^{d/2})$ if two searches meet in the middle, and the sum of time or space complexity of these two searches is much smaller than the time or space complexity $O(b^{d})$ of one direction search. There is no doubt that bidirectional search reduces the time and space complexity when two searches meet in the middle.

Standard bidirectional search has $Open_f$ and $Closed_f$ for forward search, and $Open_b$ and $Closed_b$ for backward search. There are mainly two heuristic value updating methods in bidirectional search. The *front-to-end* uses two heuristic functions to guide the search. They are $h_f$ which evaluates the distance from states in $Open_f$ to $h_{goal}$, the goal state of the problem, and $h_b$ which assesses the distance from $h_{start}$, the initial state of problem to states in $Open_b$. The *front-to-front* estimates the heuristic values of states according to how close they are to the opposite search frontiers by computing the heuristic $h(u, v)$ between all pair of states $u$ and $v$ in the frontiers rather than the goal state and initial state. The *front-to-front* would be possible to expand the closest state to the opposite search frontier and possibly leads to meet-in-the-middle behavior.

## 2.10 Duality

Suda (2013) mentioned the notion of duality STRIPS planning task which adopts the duality mapping to generate the dual version of every STRIPS task. The duality mapping includes three steps. Firstly, the precondition and delete list of original action $a$ $=(Pre(a), Add(a), Del(a))$ will be exchanged to generate the dual action $a^d$ $=(Del(a), Add(a), Pre(a))$, and the dual actions set is represented as $A^d = \{a^d | a \in A\}$. Secondly, the initial state $I$ is the complement of the goal state $G$ with respect to all fluents $F$. Thirdly, the goal state $G$ is the complement of the initial state $I$ with respect to all fluents $F$. So, the dual version of a planning task $P = (F, I, G, A)$ is $P^d = (F, (F \backslash G), (F \backslash I), A^d)$.

The most important theorem for the dual task is that for every planning task $P = (F, I, G, A)$ the dual task $P^d$ has a solution if and only if $P$ does. This theorem confirms that there is no substantial difference between progression and regression STRIPS planning, and the described transformation essentially turns progression into regression and vice versa.

# Chapter 3

# Duality Width-Based Search

This chapter will discuss the performance of *IW* and *SIW* algorithms when meeting dual STRIPS problems, and it will present a detailed analysis of the influence of duality mapping on width-based search. In the description of duality mapping, the main idea emphasises three points: There is no difference between forward search and backward search; forward search can be transferred to backward search via duality mapping, and the duality mapping is a way to generate new search problems. In other words, the duality mapping changes the input format of the problems rather than changes the state space to realise the backward effect.

## 3.1 *IW* with Duality Mapping

### 3.1.1 Experiments about Effective Width

To have an overview of width concept in dual problems, we first run *IW* on dual problems. In the experiments, there are 27 domains and 1100 problems selected from the satisficing tracks of previous IPCs. Meanwhile, we generate the dual version of these problems following the duality mapping rules and call the domains as dual domains if all problems are transferred into the dual version and call domains as original domains if all problems do not apply the duality mapping. Then both original problems and dual problems with N goals fluents are split into N problems with one atomic fluent. We run *IW* algorithm on dual problems with the single goal fluent and original problems with

| Domain | P | | $w_e$=1 | | $w_e$=2 | | $w_e$>2 | |
|---|---|---|---|---|---|---|---|---|
| | Original | Dual | Original | Dual | Original | Dual | Original | Dual |
| barman | 281 | 7056 | 10% | 100% | 0% | 0% | 90% | 0% |
| blocks world | 3714 | 398836 | 28% | 100% | 72% | 0% | 0% | 0% |
| driverlog | 259 | 5716 | 45% | 100% | 55% | 0% | 0% | 0% |
| elevators | 752 | 28264 | 0% | 100% | 100% | 0% | 0% | 0% |
| ferry | 210 | 2970 | 8% | 100% | 92% | 0% | 0% | 0% |
| floortile | 506 | 2604 | 97% | 100% | 3% | 0% | 0% | 0% |
| freecell | 320 | 14889 | 11% | 100% | 74% | 0% | 15% | 0% |
| gripper | 460 | 1400 | 0% | 100% | 100% | 0% | 0% | 0% |
| hanoi | 465 | 6325 | 100% | 100% | 0% | 0% | 0% | 0% |
| logistics | 10134 | 774028 | 12% | 100% | 88% | 0% | 0% | 0% |
| mystery | 86 | 24044 | 6% | 100% | 67% | 0% | 16% | 0% |
| nomystery | 210 | 6022 | 0% | 100% | 100% | 0% | 0% | 0% |
| parcprinter | 1128 | 7900 | 84% | 100% | 16% | 0% | 0% | 0% |
| parking | 686 | 36762 | 72% | 100% | 28% | 0% | 0% | 0% |
| pegsol | 660 | 1320 | 90% | 100% | 10% | 0% | 0% | 0% |
| pipesworld | 369 | 23360 | 60% | 100% | 37% | 0% | 3% | 0% |
| rovers | 179 | 2549 | 63% | 100% | 37% | 0% | 0% | 0% |
| scanalyzer | 500 | 3324 | 100% | 100% | 0% | 0% | 0% | 0% |
| snake | 754 | 8798 | 100% | 100% | 0% | 0% | 0% | 0% |
| sokoban | 112 | 5455 | 32% | 100% | 35% | 0% | 33% | 0% |
| storage | 240 | 14132 | 100% | 100% | 0% | 0% | 0% | 0% |
| termes | 410 | 2395 | 56% | 100% | 3% | 0% | 41% | 0% |
| transport | 345 | 83551 | 0% | 100% | 100% | 0% | 0% | 0% |
| tyreworld | 1980 | 41195 | 53% | 100% | 47% | 0% | 0% | 0% |
| visitall | 57194 | 114388 | 100% | 100% | 0% | 0% | 0% | 0% |
| woodworking | 1777 | 11641 | 100% | 100% | 0% | 0% | 0% | 0% |
| zenotravel | 219 | 4239 | 21% | 100% | 79% | 0% | 0% | 0% |
| Summary | 83950 | 1633163 | 50% | 100% | 43% | 0% | 7% | 0% |

TABLE 3.1: Effective width of single goal problems. P is number of resulting problems. Other columns show percentage of problems with effective width 1, 2, or greater

the single goal fluent to compare the average effective width. All experiments discussed below run on the cloud computer with clock speeds of 2.0 GHz Xeon processor, using a 10GB memory limit. We use a time cutoff of 30 minutes for 3.1.

All together 83950 original problems and 1633163 dual problems are presented. For each domain, a total number of problems coming from dual version and original version are respectively counted, and the percentage of problems which can be solved with effective width $k$ equal to 1, 2, or greater than 2 is presented. The last row in 3.1 describes the average percentage of effective width over all domains.

For original problems, there are 50% with $w_e = 1$, 43% with $w_e = 2$, and less than 7% with $w_e > 2$. However, for dual problems, there are 100% with $w_\mathrm{e}$=1. Because the time

| | Original | Dual |
|---|---|---|
| Total Problems | 83950 | 1633163 |
| Solved Problems | 76008 | 1633163 |
| Percentage | 90.54% | 100% |

TABLE 3.2: *IW* solved problems

complexity of $IW(k)$ is exponential in $k$ and $w_e$ is always smaller than or equal to $k$, this means that all dual problems with the single goal fluent are solved in linear, while the majority of original problems with the single goal fluent need linear or quadratic in the number of problem fluents to solve.

### 3.1.2 Experiments about Solved problems

We also compare the number of solved problems by $IW$ among tested original and dual domains with 1633163 dual problems with the single goal fluent and 83950 original problems with then single goal fluent respectively. The results are shown in 3.2 with the same running conditions but no time cutoff. $IW$ solves near 90% original problems while 100% dual problems. The results suggest that $IW$ manages to exploit the low width of dual problems much better than original problems.

### 3.1.3 Results Analysis

There are two reasons to explain that the effective width of dual problems is much lower than original problems, and IW solves more dual problems than original problems.

- Firstly, duality mapping creates a lot of goal fluents in the initial state. It means that many goal fluents are already true in the initial state without searching.
- Secondly, duality mapping makes more fluents true in the initial state. Hence, the goal state can be achieved easily by few actions, and this can reduce the possibility of novelty value beyond the setting.

#### 3.1.3.1 Reason For Solved More Problems

To testify the first reason, we compare the average number of fluents which are true in both initial state and goal state between tested original and dual domains. We also

| | Original | Dual |
|---|---|---|
| F$ig$ | 5 | 818 |
| Percentage | 12.26% | 95.93% |

TABLE 3.3: F$ig$ represents the average number of fluents in both initial and goal states among tested 27 domains. Percentage is calculated by F$ig$ dividing the average number of goal fluents among tested 27 domains.

| | Original | Dual |
|---|---|---|
| ANFIS | 39 | 835 |

TABLE 3.4: ANFIS is the average number of fluents in the initial state among tested 27 domains

compare the average percentage of these fluents among all fluents in the goal state between tested dual and original domains. The results are summarized in 3.3.

The results indicate that nearly 96% goal fluents are already true in the initial state in dual problems while this percentage is only 12 in original problems. As a result, many goal fluents are already true in the initial state without searching.

#### 3.1.3.2 Reason For Lower Effective Width

To justify whether the duality mapping generates more fluents in the initial state, we compare the average number of fluents in the initial state between tested original and dual domains. The results are shown in 3.4. On average, there are 39 initial fluents in original domains while 835 initial fluents in dual domains. It is clear that there are more initial fluents in dual domains.

Then we try to explain why the effective width is lower in dual problems when a lot of fluents are true in the initial state. The effective width, $w_e(P)$ for a problem $P$ with the single fluent in the goal state is the minimum width used to solve $P$. Actually, the algorithm $IW$ does not know this width before the search; thus, it calls $IW(i)$ in order, starting from $i = 0$. The minimum value of $i$ for $IW(i)$ to solve $P$ is the effective width of $P$ (Lipovetzky and Geffner, 2012). In other words, original problems need to call $IW(i)$ where $i > 1$ to solve single goal problems, but dual problems only need to call $IW(i)$ where $i = 1$ to solve single goal problems based on our experiment in 3.1, and $i$ is the minimum novelty value.

Here, we use the *blocks word* domain as an example to explain why dual problems only need novelty 1 to solve single goal problems rather than higher than 1. There are two methods to explain this reason. The first way is more systematic like what Lipovetzky et al. (2013) detailed proof stated that the *blocks World*, *logistics*, *gripper*, and *n-puzzle* domains have a bounded width of 2, as long as the goals are restricted to single fluent. This proof process needs the help of *tuple graph* and *optimal implication*, so it goes beyond the scope of our discussion. The second way is that we propose a simplified problem as an example to prove that the original problem needs $IW(2)$ to solve while the dual problem only needs $IW(1)$. Even if this simplified problem cannot include all situations in the *blocks world* domain, it is easy to understand and can illustrate the overview of the duality mapping.

The *blocks world* domain is one classical untyped STRIPS domain, where stackable blocks need to be re-assembled on a table with unlimited space. There is a robot arm which stacks a block onto a block, unstacks a block from a block, puts down a block, or picks up a block. The initial state defines a complete state while the goal state defines only the required relations between any of two blocks.

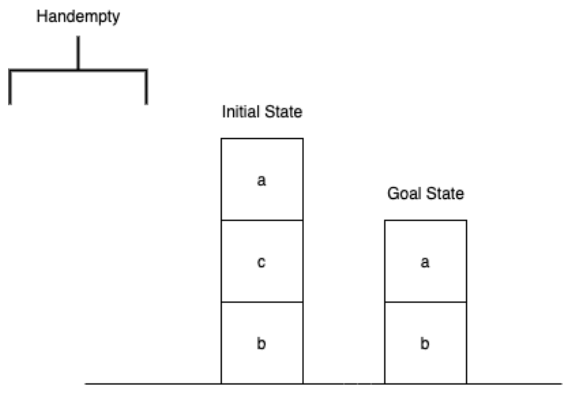

FIGURE 3.1: *Blocks World* domain example

For a problem $P = <F,\ O,\ I,\ G>$, where

$I = \{on(a,\ c),\ on(c,\ b),\ handempty\},\ G = \{on(a,\ b)\}$,

$F = \{ontable(a),\ ontable(b),\ ontable(c),\ clear(a),\ clear(b),\ clear(c),\ on(a,\ b),\ on(b,\ a),\ on(a,\ c),\ on(c,\ a),\ on(b,\ c),\ on(c,\ b),\ handempty\}$,

$O$ = {(:action *pick-up* :*precondition* (*and* (*clear* ?*x*) (*ontable* ?*x*) (*handempty*)) :*Add* ((*holding* ?*x*)) :*Del* ((*ontable* ?*x*) (*clear* ?*x*) (*handempty*)))

(:action *stack* :*precondition* (*and* (*holding*, ?*x*), (*clear* ?*y*)) :*Add* ((*clear* ?*x*), (*handempty*) (*on* ?*x*, ?*y*)) :*Del* ((*holding* ?*x*), (*clear* ?*y*) ))

(:action *put-down* :*precondition* ((*holding* ?x)) :*Add* ((*clear* ?*x*), (*handempty*) (*ontable* ?*x*)) :*Del* ((*holding ?x*)))

(:action *unstack* :*precondition* (*and*, (*on* ?*x*, ?*y*) (*handempty*), (*clear* ?*x*)) :*Add* ((*holding* ?*x*), (*clear* ?*y*)) :*Del* ((*on* ?*x*, ?*y*) (*handempty*), (*clear* ?*x*))

) where (*not* = ?*x* ?*y*)) }

We first set the max novelty value to 1. For problem $P$, the only possible action path, a sequence of actions, to map the initial state to the goal state is *unstack*($a$, $c$), *put-down*($a$), *unstack*($c$, $b$), *putdown*($c$), *pick-up*($a$), and *stack*($a$, $b$). In this action path, the fluent in the add list of action *pick-up*($a$) is already rue before we execute the action *pick-up*($a$) since the fluent *holding*($a$) is true after we execute the action *unstack*($a$, $c$). As a result, if we apply the action *pick-up*($a$) after we applied *unstack*($a$, $c$), the novelty value of newly generated state is higher than 1, and this newly generated state will be pruned. If we set the max novelty value to 2, the action *pick-up*($a$) can generate a new pair of fluents <*holding*($a$),*clear*($b$)> and hence the newly generated state will not be pruned and lead to mapping the goal state finally.

Next, we show the duality mapping can reduce the novelty value to 1. For a $P$ = <$F$, $O$, $I$, $G$>, the dual problem $P_d$ = <$F$, $O_d$, ($F \setminus G$), ($F \setminus I$)>, where

$I$ = {*ontable*($a$), *ontable*($b$), *ontable*($c$), *clear*($a$), *clear*($b$), *clear*($c$), *on*($b$, $a$), *on*($a$, $c$), *on*($c$, $a$), *on*($b$, $c$), *on*($c$, $b$), *handempty*},

$G$ = {*ontable*($a$), *ontable*($b$), *ontable*($c$), *clear*($a$), *clear*($b$), *clear*($c$), *on*($a$, $b$), *on*($b$, $a$), *on*($c$, $a$), *on*($b$, $c$))},

The fluents $F$ and actions will not be repeated shown since they are the same as those in the original problem $P$. Even if dual actions are generated by exchanging the delete list and precondition of original actions, in the *blocks word* domain the delete list and precondition include same fluents in each action. As a result, dual actions are the same as original actions in the *blocks word* domain.

We can find that only the fluent *on*($a$, $b$) is in the goal state but not in the initial state. We first set the max novelty value to 1. For problem $P_d$, the only possible action path to map the dual initial state to the dual goal state is $\textit{stack}(a,\ b)$. Since fluents *holding* $(a)$ and *clear* $(b)$ in the delete list of action $\textit{stack}(a,\ b)$ are both true in the initial state, the action $\textit{stack}(a,\ b)$ can be directly applied in the initial state to achieve the goal sate, and it can generate the new fluent $on(a,\ b)$. So when the novelty value is 1, *IW* can solve the dual problem $P_d$ with single goal fluent $on(a,\ b)$.

According to the above analysis, the main reason for the duality mapping reducing effective width of problems is that the duality mapping makes more fluents true in the initial state. In other words, the majority of actions can be directly applied in the initial state, and this reduces the proposition-dependency to achieve the goal state. Therefore, the goal state can be achieved easily with few actions and lower effective width.

### 3.1.4 Summary

To sum up, the duality mapping changes the structure of initial and goal state in original problems, so the width of original problems has also been changed. The influence of duality mapping should be argued in a more careful way in width-based search.

In fact, the comparison of *IW* algorithm solving dual problems and original problems with the single goal fluent not only describes the effective width of dual benchmarks and original benchmarks but also implies the complexity of benchmarks coming from conjunctive goals. In the field of planning, the goal decomposition is treated as a vital and typical technique. The old intuition also suggests that the planner appears powerful when solving conjunctive goals through some form of decomposition if it can efficiently solve problems with the single goal fluent.

## 3.2 *SIW* with Duality Mapping

*SIW* is a search algorithm that uses the iterated width (*IW*) both for constructing a serialization of the problem and solving the resulting subproblems (Lipovetzky and Geffner, 2012). The following part will compare the performance of *SIW* running on the dual problems with conjunctive goals and original problems with conjunctive goals.

| | Solved Original Problems | Solved Dual Problems |
|---|---|---|
| Summary (1417) | 720 | 22 |

TABLE 3.5: *SIW* solved original problems vs. solved dual problems. The number of solved dual and original problems in each domain by *SIW* is shown in A.2.

| | FF | LAMA | Mp |
|---|---|---|---|
| Original | 1009 | 1192 | 1114 |
| Dual | 136 | 175 | 329 |

TABLE 3.6: the number of original and dual problems solved within 180 seconds by the respective planners.

### 3.2.1 Experiments

The tested benchmark domains come from the satisficing tracks of IPCs of years 1998–2018. The dual problems are generated based on the duality mapping rules. *SIW* is written in C++, and use Metric-FF as an ADL to *Propositional* STRIPS compiler. The experiments are conducted on the cloud computer at 2.0 GHz Xeon processor and 10 GB of RAM. Time and memory outs after 30 minutes or 10GB. The novelty bound is set to 2 to ensure efficiency. The results are summarized in 3.5.

### 3.2.2 Results Analysis

According to the results in 3.5, it is obvious that *SIW* cannot solve the majority of dual problems compared with original problems. *SIW* only solved 22 dual problems, but *SIW* solved 720 original problems among tested 1417 problems. Besides, according to the results in A.2, if there is no problem solved in domains without the duality mapping, these problems still cannot be solved after applying the duality mapping. This result seems to conflict with the intuition that the difficulty of solving problems with the single goal fluent is positively correlative with solving these problems with conjunctive goal fluents. To explain this conflict, we did the following analysis.

It is worth noting that the experiment results represented by (Suda, 2013) also show that dual problems are harder to solve when applying heuristic-based planners such as FF, LAMA and Mp, and the experiment results coming from Suda (2013) are shown in 3.6.

In 3.6, all tested domains are collected from the satisficing tracks of IPCs of years 1998–2011. Together 1564 problems are collective. There is an apparent gap between solved original problems and solved dual problems. However, in the worst case, FF planner still solved 136 dual problems. Comparing solved dual problems with solved original problems, the percentage is nearly 14. This percentage is increased to nearly 30 in Mp planner while it is in only 1.5 in *SIW*. It is extremely lower than other planners. Suda (2013) proposed two reasons for the difficulty of dual problems: One is that adopted planners do not check the usefulness of actions; the other is that invariant information is not recovered by planners. After improvements, heuristic-based planners indeed solved more dual problems but still fewer than original problems. Even if two authentic reasons raised by Suda can explain why *SIW* solved limited dual problems, there is still another unique reason for width-based search, which must be emphasised.

Let us recall the processes of *SIW*. *SIW* uses *IW* both for constructing a serialization of problems and for solving resulting subproblems. The *IW* algorithm, calls $IW(i)$ for $i = 0, 1, 2, \ldots,$ sequentially over each subproblem until all subproblems are solved. $IW(i)$ is a breadth-first search that prunes newly generated states when its novelty values are higher than $i$ (Lipovetzky and Geffner, 2012). The above review shows that the novelty decides the further generated states in *SIW*. The novelty can be treated simply as the minimum size of newly generated tuple of fluents in the search states. The average number of initial fluents in dual problems suggests that the majority of fluents are true in the initial state so that it is harder to find new tuples of fluents during the search, and lots of states are pruned due to the novelty threshold, and it causes search end finally.

### 3.2.3 Summary

To conclude, the duality mapping provides a new perspective to understand the relationship between progression and regression, but the current search strategies, especially width-based search, cannot handle this mapping very well. In practice, the differences between original problems and dual problems follow from asymmetries (with respect to the mapping) of the concrete benchmarks, and they are not inherent to the search models themselves.

# Chapter 4

# Backward Width-Based Search

Even if the duality mapping allows us to transfer the search from progression into regression easily with only editing input problems, *SIW* is unaccommodated with it since the novelty applied in *SIW* needs the precise and succinct initial state in each domain. It is now still hard to make a conclusion that backward search is not suitable for *SIW*. Actually, there are two possible ways of interpreting the duality mapping, which are either new standalone problems or part of the new algorithm. If we treat the duality mapping as new problems, it only proves that *SIW* is not efficient when meeting duality mapping problems. When we treat the duality mapping as a part of the new algorithm, we show next that it is better to generate backward *SIW* by changing the progression state space into the regression state space.

## 4.1 Backward *SIW*

For changing forward *SIW* to backward *SIW*. We base our modifications for the progression state space on the following rules.

- We refer to *SIW* that searches backward from the goal rather than forward from the initial state.
- The goal states are the states for which states are the subset of the initial state rather than the goal states defined in forward search.

- At the beginning of backward *SIW*, the mutex pairs are detected by computing the $h_2$ value of each fluents pair.
- The transition function deletes fluents in the add list of applicable actions and adds fluents in the precondition of applicable actions from the current state to generate the new state.
- The set of applicable actions are relevant, consistent and not mutex. Relevance means the intersection between the add list of applicable actions and current state should not be empty; consistency means the intersection between the delete list applicable actions and current state should be empty; not mutex means newly generated states should not contain any mutex pairs.
- The goal consistency check will be concocted, when the new state $s'$ is generated. If $s'$ achieves a new goal fluent, the *reachable check* is conducted. A *reachable table* is formulated with all fluents in problem $p$ to be false and then set the fluents in $s'$ to be true at beginning. Other fluents turn to be true gradually with fixed point procedure if those fluents are in the precondition of relevant actions. Action $a$ is relevant with *reachable table* when the intersection of add list of $a$ with positive fluents in *reachable table* should not be empty and $a$ cannot delete any achieved goal fluents. If all fluents in the goal state are true in the *reachable table* when the *reachable table* will never change, the new state $s'$ is the new initial state for next *IW*; otherwise, $s'$ will be inserted into the *Open* list and wait to be expanded in the following search after all unachieved goal fluents are checked.
- In backward *SIW*, the effects of add list is the same as the effects of delete list, while the effects of preconditions are the same as the effects of add list, and the effects of the delete list are the same as the effects of precondition with the consistency check.

Backward *SIW* usually finds some states $s$ that are not reachable from the initial state $s_0$. These unreadable states will waste the mass of memory and computing resource and cause search cannot find the plan. To discard these unreadable states in backward search, Blum and Furst (1997) proposed the notion of mutex pair which is the pair of unreachable fluents. For example, in the *blocks word* domain, the following state may be generated during regression.

$s = \{on(b,\ a),\ holding\ (d)\}$

After applying the action $unstack(d,\ a) = \{pre{:}\{on(d,\ a),\ handempty,\ clear(d)\},$ $add{:}\{\ holding\ (d),\ clear(a)\},\ del{:}\{on(d,\ a),\ \ handempty,\ clear(d)\}$, the newly generated state $s'$ would be:

$s' = \{on(b,\ a),\ on(d,\ a),\ \ handempty,\ clear(d)\}$.

The state $s'$ represents a situation in which the block $b$ is on the block $a$, and the block $b$ is also on the block $a$. It is obvious that this state is unreachable in the *blocks word* domain when the initial state is correct. In the planning perspective, the heuristic value would be assigned to infinite to fluent pair $<on(b,\ a),\ on(d,\ a)>$. Here, we adopt the $h_2$ heuristic to calculate the heuristic value of all fluents pairs at the search beginning. If any states contain the fluents pair that the $h_2$ value is infinite, this state will be pruned. The above method is not guaranteed to find all mutex pairs, yet it can be computed fast, and in many domains, it appears to detect all obvious unreachable states.

### 4.1.1 Experiments

In the experiments, all tested domains are selected from the satisficing tracks of IPCs of years 1998–2018. we compare the performance of backward *SIW* with forward *SIW* by using Metric-FF as an ADL to *Propositional* STRIPS compiler. It is conducted on the cloud computer running at 2.0 GHz Xeon processor and 10 GB of RAM; time and memory outs after 30 minutes or 10 GB. The novelty bound is set to 2 to ensure efficiency. The comparisons are summarized in 4.1.

### 4.1.2 Result Analysis

Backward *SIW* indeed solved more problems compared with the duality mapping among tested 1417 problems, 141 vs. 22, but compared with forward *SIW* which solved 710 problems, backward *SIW* is still not competitive. The average plan length in backward *SIW* is shorter than in forward *SIW*, and backward *SIW* needs less average time to solve problems. This does not mean that backward *SIW* has better performance since we find that backward *SIW* usually stops at the beginning of the search and shows no plan found. In addition, if problems cannot be solved by forward *SIW*, neither backward

| Domain | P | S | | T | | Q | |
|---|---|---|---|---|---|---|---|
| | | F-SIW | B-SIW | F-SIW | B-SIW | F-SIW | B-SIW |
| agricola18 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| airport | 50 | 49 | 0 | 62.45 | 0.00 | 177.10 | 0.00 |
| barman14 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| blocks world | 50 | 27 | 10 | 78.64 | **76.01** | 104.88 | **102.40** |
| caldera18 | 20 | 10 | 0 | 58.04 | 0.00 | 20.10 | 0.00 |
| caldera-split-18 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| childsnack14 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| cybersec | 30 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| data-network18 | 20 | 3 | 0 | 6.23 | 0.00 | 36.67 | 0.00 |
| depot | 22 | 18 | 0 | 9.80 | 0.00 | 47.06 | 0.00 |
| driverlog | 20 | 7 | 0 | 0.07 | 0.00 | 25.43 | 0.00 |
| elevators11 | 20 | 18 | 0 | 317.01 | 0.00 | 219.56 | 0.00 |
| ferry | 30 | 30 | 30 | 0.00 | 0.02 | 24.33 | 31.30 |
| floortile14 | 20 | 0 | **2** | 0.00 | 0.07 | 0.00 | 108.50 |
| ged14 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| gripper | 20 | 20 | 20 | 0.07 | 0.29 | 91.00 | 95.00 |
| hanoi | 30 | 3 | 3 | 2.53 | 3.03 | 3.67 | 6.67 |
| hiking14 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| logistics | 50 | 26 | 2 | 199.89 | **10.02** | 152.05 | **35.00** |
| miconic | 50 | 50 | 50 | 0.12 | 0.57 | 58.46 | **25.41** |
| mprime | 70 | 60 | 0 | 33.94 | 0.00 | 7.77 | 0.00 |
| mystery | 60 | 33 | 0 | 0.90 | 0.00 | 7.18 | 0.00 |
| no-mprime | 35 | 30 | 0 | 34.75 | 0.00 | 7.80 | 0.00 |
| no-mystery | 30 | 16 | 0 | 0.91 | 0.00 | 7.13 | 0.00 |
| nomystery11 | 20 | 1 | 0 | 0.01 | 0.00 | 30.00 | 0.00 |
| openstacks | 30 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| openstacks14 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| parcprinter11 | 20 | 20 | 16 | 0.02 | 0.02 | 78.35 | **68.13** |
| parking14 | 20 | 20 | 0 | 129.17 | 0.00 | 65.00 | 0.00 |
| pathways | 30 | 15 | 0 | 28.81 | 0.00 | 94.13 | 0.00 |
| pegsol11 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| pipesworld06 | 50 | 24 | 0 | 45.00 | 0.00 | 33.79 | 0.00 |
| pipesworld-notankage | 50 | 19 | 0 | 1.67 | 0.00 | 32.74 | 0.00 |
| pipesworld-tankage | 50 | 24 | 0 | 45.59 | 0.00 | 33.79 | 0.00 |
| rovers | 20 | 20 | 3 | 1.80 | **0.00** | 35.50 | **12.67** |
| scanalyzer11 | 20 | 17 | 0 | 13.99 | 0.00 | 35.00 | 0.00 |
| settlers18 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| snake18 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| sokoban11 | 20 | 1 | 0 | 0.04 | 0.00 | 429.00 | 0.00 |
| spider18 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| storage | 30 | 23 | 0 | 2.80 | 0.00 | 22.91 | 0.00 |
| termes18 | 20 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 |
| thoughtful14 | 20 | 15 | 0 | 0.28 | 0.00 | 95.47 | 0.00 |
| tpp | 30 | 10 | 4 | 0.09 | **0.02** | 39.60 | **17.50** |
| transport14 | 20 | 14 | 0 | 704.39 | 0.00 | 326.36 | 0.00 |
| trucks | 30 | 3 | 0 | 0.31 | 0.00 | 25.67 | 0.00 |
| tyreworld | 30 | 26 | 0 | 237.78 | 0.00 | 156.50 | 0.00 |
| visitall14 | 20 | 20 | 0 | 145.87 | 0.00 | 2916.85 | 0.00 |
| woodworking11 | 20 | 19 | 0 | 14.26 | 0.00 | 71.16 | 0.00 |
| zenotravel | 20 | 19 | 1 | 0.79 | **0.00** | 36.47 | **6.00** |
| summary | 1417 | 710 | 141 | 64.06 | **10.01** | 163.19 | **56.51** |

TABLE 4.1: Forward *SIW* vs. backward *SIW*. P is the number of problems in each domain. F-SIW is forward *SIW*. B-SIW is backward *SIW*. S is the number of solved problems. *Q* is average plan length and *T* is average time in seconds. The red highlight means backward search outperforms forward search.

*SIW* can, except in the *floortile* domain. Backward *SIW* still leaves much to be desired, so we propose tow improvement methods for it.

## 4.2 Backward *SIW* Improvement

Two issues still exist in backward *SIW*; One is that the goal consistency check does not consider the order between each goal fluent; another is that the backward goal state does not include the negative fluents. To solve these issues in backward *SIW*, the following improvements are adopted.

### 4.2.1 Goal Consistency Check

The order between each goal fluent is extracted from the *goal-order graph* by considering the preceding relationship, and the *goal-order graph* always builds at the beginning of both forward and backward search. In forward search, the goal-order for two-goal fluents $<p\ ,q>$ can be described as $p$ preceding $q$ and $q$ requiring $p$ if all actions add $p$ but *edel* with $q$, and as $q$ preceding $p$ and $p$ requiring $q$ if all actions add $q$ but *edel* with $p$. In backward search, the goal-order for two-goal fluents $<p\ ,q>$ can be described as $p$ preceding $q$ and $q$ requiring $p$, if all actions require $p$ but *edel* with $q$, and as $q$ preceding $p$ and $p$ requiring $q$ if all actions require $q$ but *edel* with $p$. The action $a$ *edel* with fluent $p$ in forward search means that the $h_2$ value of fluents pairs generated by fluent $p$ with any one of fluents in the add list of $a$ is infinite, or pairs generated by fluent $p$ with any one of fluents in the precondition of $a$ is infinite or fluent $p$ appears in the delete list of $a$. The action $a$ *edel* with fluent $p$ in backward search means that the $h_2$ value of fluents pairs generated by fluent $p$ with any one of fluents in the precondition of $a$ is infinite, or fluent $p$ appears in the add list of $a$.

In forward *SIW*, the mutex pairs detection is not important since few unreachable states will be generated. As a result, the mutex pairs detection can be ignored in forward *SIW*; hence the action $a$ *edel* with fluent $p$ in forward *SIW* is simplified to the fluent $p$ belonging to the delete list of $a$. However, the mutex pairs detection is indispensable in backward search, so the *goal-order graph* built in backward *SIW* can describe a more detailed preceding relationship between each goal fluent. Now backward *SIW* is updated with *goal-order graph* built at the beginning of the search, and the preceding relationship of each goal fluent is adopted in the new goal consistency check.

The new goal consistency check will be conducted, when one new state $s'$ is generated. If there is a new *leaf goal* fluent achieved in $s'$, the *reachable check* is adopted. The *leaf goal* fluents are those fluents that no fluents precedes them. In the *reachable check*, a *reachable table* is formulated by making all fluents false, then making fluents in $s'$ true at beginning. Other fluents turn to be true gradually with fixed point procedure if they are in the precondition of relevant actions. In the new goal consistency check, the action $a$ is relevant with *reachable table* when the intersection of add list of $a$ with positive fluent in *reachable table* should not be empty, and $a$ cannot delete any achieved and original goal fluents. If all fluents in the goal state are true in the *reachable table* when the *reachable table* never change, $s'$ is the new initial state for next *IW*, and the *leaf goal* fluents will be updated based on removing consumed fluents (fluents make true) and adding fluents without fluents preceding them; otherwise $s'$ will be inserted into the *Open* list and wait to be expanded in the following search after all *leaf goal* fluents are checked.

To keep the extraction of relevant actions efficiently, we need to exclude all actions which can delete the achieved and origin goal fluents in a reasonable way. We list all necessary symbols for explaining the new goal consistency check.

- $EX$ are all excluded actions;
- $F_G$ are achieved goals fluents;
- $F_{g_i}$ is each achieved goal fluent; i=1, 2, . . . , n and n is the size of $F_G$;
- $F_{UG}$ are unachieved goal fluents;
- $F_{ug_i}$ is each unachieved goal fluent; i=1, 2,. . . , n and n is the size of $F_{UG}$;
- $F_{OG}$ are original goal fluents;
- $F_{og_i}$ is each original goal fluent; i=1, 2, . . . , n and n is the size of $F_{OG}$;
- $F_{LG}$ are *leaf goal* fluents;
- $F_{lg_i}$ is each *leaf goal* fluent; i=1, 2, . . . , n where n is the size of $F_{LG}$;

- $EX_f$ are the excluded actions because of deleting fluent *f*.

In the new goal consistency check, if a new *leaf goal* fluent $F_{lg_1}$ is achieved in the newly generated state $s'$, we first exclude the actions which can delete the $F_{lg_1}$ and $F_G$. These excluded actions should contain $F_{lg_1}$ and $F_G$ in the add list because of the regression state space. The $EX$ is updated by $EX \cup EX_{F_{lg_1}} \cup EX_{F_{g_1}} \cup EX_{F_{g_2}}, \ldots, \cup EX_{F_{g_n}}$. Then, if an original goal fluent $F_{og_1}$ is in $s'$, and $F_{og_1}$ is not equal to $F_{lg_1}$ as well as $F_{og_1}$ is not in $F_G$, we will check whether $F_{og_1}$ is in the add list of all excluded actions $EX$. If true, the actions which delete $F_{og_1}$ also need to be excluded. Then $EX$ is updated by $EX \cup EX_{F_{og_i}}$. This process repeats with fixed point procedure till all $F_L$ and $F_{OG}$ fluents are checked, and $EX$ keeps fixed size. The algorithm of new backward goal consistency check is described in 1, and the original backward goal consistency check is described in 2 for comparing.

**Algorithm 1:** New Backward Goal Consistency Check

```
for $\mathrm{F_{lg_i}}$ in $\mathrm{F_{LG}}$ do
    if $\mathrm{F_{lg_i} \in s}$ then
        $\mathrm{EX = \varnothing}$
        $\mathrm{EX_o = \varnothing}$
        $\mathrm{F_G = F_G \cup F_{lg_i}}$
        $\mathrm{EX = EX \cup EX_{F_{g_1}} \cup EX_{F_{g_2}}, \ldots, \cup EX_{F_{g_n}}}$
        while $\mathrm{EX \setminus EX_o \neq \varnothing}$ do
            $\mathrm{EX_o = EX}$
            for $\mathrm{F_{og_i}}$ in $\mathrm{F_{OG}}$ do
                if $\mathrm{F_{og_i} \neq F_{lg_i} \land F_{og_i} \in s}$ then
                    if EX delete $\mathrm{F_{og_i}}$ then
                        $\mathrm{EX = EX \cup EX_{F_{og_i}}}$
                    end
                end
            end
        end
        if reachable check then
            return True
        else
            $\mathrm{F_G = F_G \setminus F_{lg_i}}$
        end
    end
end
```

**Algorithm 2:** Original Backward Goal Consistency Check

```
for $F_{ug_i}$ in $F_{UG}$ do
    if $F_{ug_i} \in s$ then
        $EX = \varnothing$
        $F_G = F_G \cup F_{ug_i}$
        $F_{UG} = F_{UG} \setminus F_{ug_i}$
        $EX = EX \cup EX_{F_{g_1}} \cup EX_{F_{g_1}}, \ldots, \cup EX_{F_{g_n}}$
        if reachable check then
            return True
        else
            $F_G = F_G \setminus F_{ug_i}$
            $F_{UG} = F_{UG} \cup F_{ug_i}$
        end
    end
end
```

### 4.2.2 Negative Fluents

Another improvement for backward *SIW* is adding negative fluents into the backward goal state. To put it simply, we use $G$ to represent the goal state and use $I$ to represent the initial state in backward search. If the state $s$ contains negative fluents, this means that corresponding positive fluents are false in $s$. We adopt this idea in backward search to ensure that fluents not in $G$ should be false since $G$ meets the *closed world assumption*. We believe that negative fluents in backward search can reduce the number of applicable actions since some of those actions are not consistent with states generated during backward search. In backward *SIW*, negative fluents $Fn^*$ are generated from fluents $F*$ in $I$ but not in $G$, and these negative fluents $Fn^*$ are added into $G$. Meanwhile, if actions contain $F*$ in the add list, the delete list and precondition should add $Fn^*$; if actions contain $F*$ in the delete list, $Fn^*$ should also be added into the add list. One could argue that we can generate negative fluents for all positive fluents, and add these negative fluents into $G$ since all not mentioned fluents in $G$ should be false. It is reasonable, but not suitable for backward *SIW* since the goal-order of each negative fluent is too complex to allow *IW* to serialize subproblems.

### 4.2.3 Experiments

We use the same domains and running conditions as in table 4.1 to compare the improved backward *SIW* with the original backward *SIW*. The results are summarized in 4.2.

| Domain | P | S | | | T | | | Q | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | F-SIW | B-SIW | B-SIW+ | F-SIW | B-SIW | B-SIW+ | F-SIW | B-SIW | B-SIW+ |
| agricola18 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| airport | 50 | 49 | 0 | 0 | 62.45 | 0 | 0 | 177.10 | 0 | 0 |
| barman14 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| blocks world | 50 | 27 | 10 | 26 | 78.64 | **79.01** | **44.98** | 104.88 | **102.40** | 113.63 |
| caldera18 | 20 | 10 | 0 | 0 | 58.04 | 0 | 0 | 20.10 | 0 | 0 |
| caldera18 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| childsnack14 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| cybersec | 30 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| data-network18 | 20 | 3 | 0 | 0 | 6.23 | 0 | 0 | 36.67 | 0 | 0 |
| depot | 22 | 18 | 0 | 0 | 9.80 | 0 | 0 | 47.06 | 0 | 0 |
| driverlog | 20 | 7 | 0 | 4 | 0.07 | 0 | **0.02** | 25.43 | 0 | 27.00 |
| elevators11 | 20 | 18 | 0 | 0 | 317.01 | 0 | 0 | 219.56 | 0 | 0 |
| ferry | 30 | 30 | 30 | 30 | 0.00 | 0.02 | 0 | 24.33 | 31.30 | 24.40 |
| floortile14 | 20 | 0 | **2** | **12** | 0.00 | 0.07 | 0.03 | 0.00 | 108.50 | 119.63 |
| ged14 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| gripper | 20 | 20 | 20 | 20 | 0.07 | 0.29 | **0.02** | 91.00 | 95.00 | **89.00** |
| hanoi | 30 | 3 | 3 | 1 | 2.53 | 3.03 | 8.75 | 3.67 | 6.67 | **2.00** |
| hiking14 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| logistics | 50 | 26 | 2 | 18 | 199.89 | **10.02** | 31.05 | 152.05 | **35.00** | 110.17 |
| miconic | 50 | 50 | 50 | 50 | 0.12 | 0.57 | **0.09** | 58.46 | **25.41** | **27.49** |
| mprime | 70 | 60 | 0 | 4 | 33.94 | 0 | **12.89** | 7.77 | 0 | 26.75 |
| mystery | 60 | 33 | 0 | 0 | 0.90 | 0 | 0 | 7.18 | 0 | 0 |
| no-mprime | 35 | 30 | 0 | 4 | 34.75 | 0 | 12.89 | 7.80 | 0 | 26.75 |
| no-mystery | 30 | 16 | 0 | 0 | 0.91 | 0 | 0 | 7.13 | 0 | 0 |
| nomystery11 | 20 | 1 | 0 | **7** | 0.01 | 0 | 0.13 | 30.00 | 0 | 39.57 |
| openstacks | 30 | 0 | 0 | **29** | 0.00 | 0 | 2.60 | 0.00 | 0 | 147.59 |
| openstacks14 | 20 | 0 | 0 | **20** | 0.00 | 0 | 64.27 | 0.00 | 0 | 932.00 |
| parcprinter11 | 20 | 20 | 16 | 16 | 0.02 | 0.02 | 0.03 | 78.35 | **68.13** | **68.13** |
| parking14 | 20 | 20 | 0 | 0 | 129.17 | 0 | 0 | 65.00 | 0 | 0 |
| pathways | 30 | 15 | 0 | 0 | 28.81 | 0 | 0 | 94.13 | 0 | 0 |
| pegsol11 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| pipesworld06 | 50 | 24 | 0 | 0 | 45.00 | 0 | 0 | 33.79 | 0 | 0 |
| pipesworld-notankage | 50 | 19 | 0 | 0 | 1.67 | 0 | 0 | 32.74 | 0 | 0 |
| pipesworld-tankage | 50 | 24 | 0 | 0 | 45.59 | 0 | 0 | 33.79 | 0 | 0 |
| rovers | 20 | 20 | 3 | 11 | 1.80 | **0.00** | **0.07** | 35.50 | **12.67** | 37.64 |
| scanalyzer11 | 20 | 17 | 0 | 14 | 13.99 | 0 | 15.41 | 35.00 | 0 | 39.00 |
| settlers18 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| snake18 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| sokoban11 | 20 | 1 | 1 | 0 | 0.04 | 0 | 0 | 429.00 | 0 | 0 |
| spider18 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| storage | 30 | 23 | 0 | 0 | 2.80 | 0 | 0 | 22.91 | 0 | 0 |
| termes18 | 20 | 0 | 0 | 0 | 0.00 | 0 | 0 | 0.00 | 0 | 0 |
| thoughtful14 | 20 | 15 | 0 | 0 | 0.28 | 0 | 0 | 95.47 | 0 | 0 |
| tpp | 30 | 10 | 4 | 4 | 0.09 | **0.00** | **0.00** | 39.60 | **17.5** | **9.50** |
| transport14 | 20 | 14 | 0 | 0 | 704.39 | 0 | 0 | 326.36 | 0 | 0 |
| trucks | 30 | 3 | 0 | 1 | 0.31 | 0 | 0.02 | 25.67 | 0 | 36.67 |
| tyreworld | 30 | 26 | 0 | **27** | 237.78 | 0 | 24.84 | 156.50 | 0 | 162.04 |
| visitall14 | 20 | 20 | 0 | 8 | 145.87 | 0 | 112.39 | 2916.85 | 0 | **2517.85** |
| woodworking11 | 20 | 19 | 0 | 0 | 14.26 | 0 | 0 | 71.16 | 0 | 0 |
| zenotravel | 20 | 19 | 1 | 1 | 0.79 | **0.00** | **0.00** | 36.47 | **6.00** | **6.00** |
| summary | 1417 | 710 | 141 | 321 | 64.06 | **10.01** | **18.36** | 163.19 | **56.51** | 217.2 |

TABLE 4.2: Backward *SIW* vs. improved backward *SIW*. P is the number of problems in each domain. B-SIW is backward *SIW*; B-SIW+ is the improved backward *SIW*. S is the number of solved problems. *Q* is average plan length and *T* is average time in seconds. The red highlight means backward search outperforms forward search.

### 4.2.4 Result Analysis

According to the results summarised in the 4.2, backward *SIW* solved more problems after improvement compared with without improvement, 321 vs. 141. The improved backward *SIW* also needs less average time to find the plan, but the average plan length is longer than forward *SIW*. Meanwhile, the features of backward search are more clearly showed in the 4.2. For the domains *floortile*, *nomystery*, *openstacks* and *tyreworld* the improved backward *SIW* outperforms forward *SIW*. For the *floortile*, backward search always outperforms forward search, no matter the improvement, so we select this domain to explain why backward search can outperform forward search in some domains.

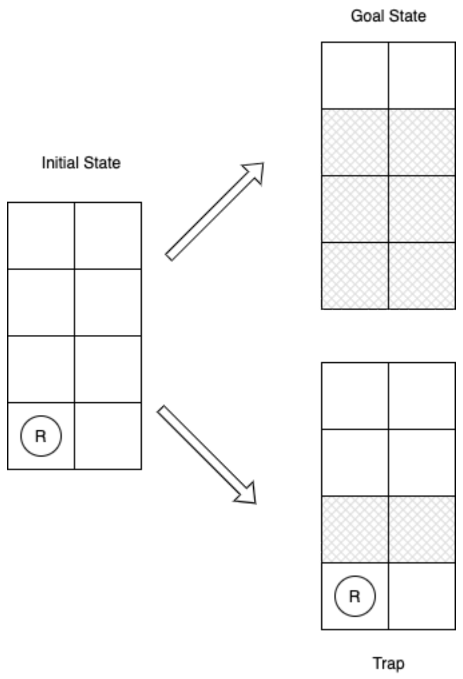

FIGURE 4.1: *Floortile* domain example

In the *floortile* domain A.3, many robots use various colours to paint patterns in floor tiles. The robots can move up, down, left and right around the floor tiles, and they can paint with only one colour at a time, but they can change their spray guns to any available colours. The robots can only paint the tile that is in front (up) or behind (down) them. Robots cannot stand on the tiles if these tiles have been painted. For the IPC set, robots need to move on a large rectangular space made up of tiles, and they need to paint the tiles with black or white colour. All painting tasks follow the same layout: The goal state describes that the first row of tiles is blank, and other rows of tiles are painted with two different colours; the initial state gives the initial positions of robots, the colours of spray guns at the beginning and the clear tiles which can be painted. We need to mention that the goal state only contains the rows of tiles which should be painted rather than the first blank row of tiles since the goal state only contains fluents we concerned in classical planning. To help readers understand the goal state easily, we add the context of blank tiles in the goal state description. The *floortile* domain is hard in forward search because robots could only paint tiles in front of them but not those behind them. If painting tiles are from behind, it may result in a no solution search.

However, backward search changes problems manageable. In backward search, the search starts from the goals state, which contains the information of blank tiles of the first row, and other rows of tiles are painted with two different colours. Backward search tries to clear tiles to achieve the initial state. In backward search, the actions which clear the tiles are those actions which paint the tiles. If we want to clear one white tile with position (1, 1), there are four possible actions:

- $a_1$ where $pre_{a1} = \{$(*clear tile* (1, 1)), (*robot-at robot*1 *tile* (2, 1)), (*robot-has*, *robot1* white)$\}$, $add_{a1} = \{$*painted tile* (1, 1) *white*$\}$, $del_{a1} = \{$*clear tile* (1, 1)$\}$;
- $a_2$ where $pre_{a2} = \{$(*clear tile* (1, 1)), (*robot-at robot*2 *tile* (2, 1)), (*robot-has*, *robot2* white)$\}$, $add_{a2} = \{$*painted tile* (1, 1) *white*$\}$, $del_{a2} = \{$*clear tile* (1, 1)$\}$;
- $a_3$ where $pre_{a3} = \{$(*clear tile* (1, 1)), (*robot-at robot*1 *tile* (0, 1)), (*robot-has*, *robot1* white)$\}$, $add_{a3} = \{$*painted tile* (1, 1) *white*$\}$, $del_{a3} = \{$*clear tile* (1, 1)$\}$;
- $a_4$ where $pre_{a4} = \{$(*clear tile* (1, 1)), (*robot-at robot*2 *tile* (0, 1)), (*robot-has*, *robot2* white)$\}$, $add_{a4} = \{$*painted tile* (1, 1) *white*$\}$, $del_{a4} = \{$*clear tile* (1, 1)$\}$;

In backward search, all these four actions can clear grid (1,1) since the clear fluent $\{$*clear tile* (1, 1)$\}$ appears in the precondition. In backward search, actions $a_1$ and $a_2$ clear the tile (1, 1) when the places of *robot1* and *robot2* are behind of the tile where the robots are located at (2, 1), while actions $a_3$ and $a_4$ clear the tile when the robot places are in front of the tile (1, 1) where the robots are located at (0,1). When we try to apply the actions $a_1$ and $a_2$ to generate the new states, the $h_2$ value of fluents pairs <(*robot-at robot*1 *tile* (2,1))/(*robot-at robot*2 *tile* (2,1)), (*painted tile* (2,1) *white*)/(*painted tile* (2,1) *black*)> in newly generated states would be infinite, since no robots can stand on a painted tile. As a result, only $a_3$ and $a_4$ are accepted since the added fluents (*robot-at* *robot*1 *tile* (0,1)) and (*robot-at robot*2 *tile* (0,1)) would not conflict with the fluent (*tile* (0,1) *blank*) which is not painted.

According to the above analysis, in backward search, only the actions which clear tiles when robots are in front of these tiles are applied to find the plan. These available actions will keep search space manageable since robots only paint tiles in front of them.

## 4.3 Summary

To conclude, backward *SIW* is uncompetitive with forward *SIW*, even though we improved backward *SIW* with the new goal consistency check and negative fluents. However, backward *SIW* proposes a new perspective to analyse the planning tasks. In some domains, backward *SIW* can not solve any problem such as *elevators*, *parking* and *woodworking*, but backward *SIW* can outperform in other domains. Meanwhile, for the *floortile* domain, it is hard to find a plan in progression since the precondition cannot describe the trap. These deliberate designs in progression make regression simpler since progression and regression are asymmetries. The asymmetry character forces us to dig out the reasons why backward search sometimes performs better but sometimes worse. Such uncertainty requires more attempts in different search algorithms. Generally, width-based search also can adopt the regression state space, and its performance is reasonable.

# Chapter 5

# Backward Best First Width Search

A standard width-based method, *SIW* procedure has been proved competitive with greedy best-first search planner using of the additive heuristic $h_{add}$ by (Lipovetzky and Geffner, 2012). Nevertheless, *SIW* is not state-of-the-art. To analyse the performance of the state-of-the-art width-based search after applying backward modifications, we change the best-first width search (BFWS) to backward version.

BFWS is a family of search algorithms. In the BFWS family, the best first search is integrated with the width-based exploration to produce a serial of planning algorithms. In BFWS, the evaluations functions combine lexicographically to break ties and some of which express novelty based preferences. BFWS($f5$) manages to outperform LAMA which wins IPCs twice, without applying many techniques that have been found essential for performance in recent years (Lipovetzky and Geffner, 2017). The evolution function $f5$ weights the state by novelty measures and break the ties by the number of the unachieved goals. The following backward modifications are based on BFWS($f5$) since the state-of-the-art performance is important in classical planning.

## 5.1 BFWS($f5$) Backward Modification

Not only the state space needs to be transferred from progression to regression as mentioned in *SIW* without considering the goal consistent check, but also the computation method of the forward heuristic value needs to be changed when building backward BFWS($f5$). Forward BFWS($f5$) with evaluation function $f_5 = <w, \#g>$ consists of $\#g(s)$, the number of top problem goals that are not true in $s$, and $w(s)$, the novelty of $s$ when given both counter $\#g(s)$ and counter $\#r(s)$. In the backward version, the first change is the computation method of $\#g(s)$. In backward version, $\#g(s)$ is sum of the number of goal fluents that are not true in $s$ and the number of fluents in $s$ that are not true in the goal state $G$. For example, in the *blocks word* domain,

the forward initial state:

$I$={ *ontable* $(a)$, *on*$(b,\ a)$, *clear*$(b)$, *handempty*}

the forward goal state:

$G$={ *ontable* $(a)$, *ontable* $(b)$, *clear* $(a)$, *clear* $(b)$, *handempty* }

the newly generated state $s$ during forward search:

$s$={ *ontable*$(a)$, *holding*$(b)$, *clear*$(a)$}

The value of forward $\#g(s)$ is 3 since the unachieved goals are {*ontable* $(b)$, *clear* $(b)$, *handempty*}. While the value of backward $\#g(s)$ is 5 since the unachieved backward goals are {*on*$(b,\ a)$, *clear*$(b)$, *handempty*}, and fluents in $s$ but not in the backward goal state are {*holding*$(b)$, *clear*$(a)$} when the initial state is treated as the goal state and the goal state is treated as the initial state in backward search. This modification is based on the regression goal check method that the goal state is achieved when the expanded state is the subset of the forward initial state.

The above analysis ignores the order between each goal fluent for the seek of an easy understanding. In fact, the order between each goal fluent is considered in both forward and backward search, and it is extracted by considering the preceding relationship, as mentioned in backward *SIW*.

The other alteration is the computations of $\#r$. Lipovetzky and Geffner (2017) give a specific definition of $\#r(s)$: "If $\pi$ is the set of actions in the last relaxed plan computed in the way to state $s$ and $R$ is the set of fluents associated with such a plan, then $\#r(s)$ is the number of atoms in $R$ that have been made true in some state $s''$ in the way from $s$ to $s'$ including these two states". In progression, the formulation of the relax plan need to recompute best supporters in every new state $s$. It is time-consuming and not efficient. While the recomputation could be avoided by performing the search backward from the goal state rather than forward from the initial state. In forward search, best supporters need to be recomputed many times as long as the new states generate since the initialisation of best supporters is based on the newly generated state to achieve the goal state. However, in backward search, the new states are the sub-goals of the problem. As a result, best supporters just need to compute once at the beginning of backward search and adopt the initial state to initialise and reuse them to extract the relax plan. In addition, for STRIPS problems, the set of fluents $R$ associated with a forward relax plan is given by fluents in the add list of actions in the forward relax plan. Following the rules about applying actions in the regression space state, the set of fluents $R$ associated with a backward relax plan is given by fluents in the precondition of actions in the backward relax plan. For backward $\#r(s)$, we generate the relax plan forward but extract $R$ backward.

Other backward modifications would not be details explained in this thesis. They all obey the rules, replacing the positive effects with precondition effects, replacing the negative effects with positive effects, and replacing the precondition effects with negative effects under the consistency check.

## 5.2 Experiments

We selected backward BFWS($f5$) and backward k-BFWS($f5$) in backward BFWS family as examples to compare with their forward versions. BFWS($f5$) is a complete search algorithm, and k-BFWS($f5$) as a variant of BFWS($f5$) is polynomial but incomplete through pruning the states $s$ whose novelty $w$(s) exceeds the bound *k*. The value of $k$ is equal to 2 in the following experiments for the purpose of keeping efficiency. Algorithms implemented are based on LAPKT. Experiments coverage over benchmarks from the satisficing tracks of IPCs of years 1998–2018 and are performed on the cloud computer with 2.0GHz Xeon processor; time and memory outs after 30 min or 10GB. 5.1 shows the results of the comparison between forward and backward two different BFWS algorithms.

| Domain | F-$k$-BFWS | B-$k$-BFWS | F-BFWS($f5$) | B-BFWS($f5$) |
|---|---|---|---|---|
| agricola18 (20) | 0 | 0 | 0 | 0 |
| airport (50) | 37 | 25 | 37 | 25 |
| barman14 (20) | 20 | 0 | 20 | 0 |
| blocks world (50) | 30 | 30 | 30 | 30 |
| caldera18 (20) | 0 | 0 | 0 | 0 |
| caldera-split-18 (20) | 0 | 0 | 0 | 0 |
| childsnack14 (20) | 0 | **6** | 2 | **4** |
| cybersec (30) | 15 | 10 | 0 | 10 |
| data-network18 (20) | 8 | 0 | 8 | 0 |
| depot (22) | 22 | 3 | 22 | 3 |
| driverlog (20) | 20 | 20 | 20 | 20 |
| elevators11 (20) | 20 | 13 | 20 | 13 |
| ferry (30) | 30 | 30 | 30 | 30 |
| floortile14 (20) | 2 | **20** | 2 | **20** |
| ged14 (20) | 19 | 0 | 19 | 0 |
| gripper (20) | 20 | 20 | 20 | 20 |
| hanoi (30) | 4 | **6** | 11 | **13** |
| hiking14 (20) | 3 | 2 | 5 | 5 |
| logistics (50) | 44 | 21 | 47 | 22 |
| miconic (50) | 50 | 50 | 50 | 50 |
| mprime (70) | 65 | 2 | 65 | 4 |
| mystery (60) | 38 | 14 | 38 | 15 |
| no-mprime (35) | 32 | 1 | 32 | 1 |
| no-mystery (30) | 19 | 7 | 19 | 8 |
| nomystery11 (20) | 13 | 10 | 14 | 11 |
| openstacks (30) | 27 | 26 | 27 | 26 |
| openstacks14 (20) | 15 | 0 | 15 | 0 |
| parcprinter11 (20) | 13 | **18** | 13 | **18** |
| parking14 (20) | 20 | 0 | 20 | 0 |
| pathways(30) | 24 | 5 | 24 | 6 |
| pegsol11(20) | 9 | 1 | 20 | 5 |
| pipesworld06(50) | 30 | 6 | 34 | 6 |
| pipesworld-notankage (50) | 50 | 15 | 50 | 15 |
| pipesworld-tankage (50) | 30 | 6 | 34 | 6 |
| rovers (20) | 20 | 19 | 20 | 19 |
| scanalyzer11 (20) | 18 | **20** | 18 | **20** |
| settlers18 (20) | 0 | 0 | 0 | 0 |
| snake18 (20) | 11 | 0 | 10 | 0 |
| sokoban11 (20) | 5 | 1 | 15 | 1 |
| spider18 (20) | 13 | 0 | 13 | 0 |
| storage (30) | 29 | 14 | 29 | 15 |
| termes18 (20) | 1 | **2** | 10 | 8 |
| thoughtful14 (20) | 20 | 5 | 20 | 5 |
| tpp (30) | 30 | 10 | 30 | 12 |
| transport14 (20) | 7 | 0 | 6 | 0 |
| trucks (30) | 7 | **14** | 9 | **14** |
| tyreworld (30) | 30 | 12 | 30 | 12 |
| visitall14 (20) | 19 | 17 | 19 | 17 |
| woodworking11 (20) | 20 | 11 | 20 | 11 |
| zenotravel (20) | 20 | 20 | 20 | 20 |
| total coverage (1417) | 979 | 512 | 1017 | 540 |
| average time | 101.70 | **66.78** | 76.96 | **66.79** |
| average quality | 169.29 | **163.10** | 185.62 | 199.07 |

TABLE 5.1: Backward BFWS($f5$) vs. forward BFWS($f5$); backward $k$-BFWS vs. forward $k$-BFWS where k=2. F means forward search while B means backward search. The red highlight means backward search outperforms forward search. The average time and average quality for each domain is shown in A.1

## 5.3 Result Analysis

The experiment results show that forward $k$-BFWS($f5$) solved 979 problems among total 1417 problems while backward $k$-BFWS($f5$) solved 512 problems. Although backward $k$-BFWS($f5$) solved fewer problems than forward search, it outperforms forward search in the domains like *childsnack, floortile, parcprinter and scanalyzer*; however, in domains like *barman*, *ged* and *parking*, forward $k$-BFWS($f5$) solved 20, 16 and 14 problems respectively among 20 problems where backward search cannot do anything about problems. It is worth noting that backward $k$-BFWS($f5$) needs less time to find the plan compared with forward $k$-BFWS($f5$), 66.78 sec vs. 101.70 sec, and the same is true in BFWS($f5$), 66.79 sec (backward) vs. 76.96 sec (forward). For the average plan length, backward $k$-BFWS($f5$) can find the plan with the shorter length while this advantage is gone in BFWS($f5$).

Then as 5.1 indicates that forward BFWS($f5$) solved 1017 problems among total 1417 problems and an extra of 38 problems compared with forward $k$-BFWS($f5$). Backward BFWS($f5$) solved 540 problems and an extra of 28 problems compared with backward $k$-BFWS($f5$). This is reasonsable since BFWS($f5$) is complete search. The feature of complete search does improve the incomplete forward and backward search slightly. Moreover, if all problems in some domains like *barman*, *ged*, *parking*, cannot be solved by backward $k$-BFWS($f5$), neither can backward BFWS($f5$) solve these problems.

## 5.4 Summary

To conclude, backward $k$-BFWS($f5$) is not competitive with forward $k$-BFWS($f5$), and the same is true when backward BFWS($f5$) is compared with forward BFWS($f5$). Meanwhile, the complete forward and backward search really solved more problems, but this improvement is limited. Especially in some domains, complete backward search has no positive effect since the average plan length is increased. This result suggests that the complete search cannot offset the weakness of backward search and that some benchmark domains are still quite challenging for backward search. Then we propose another method to offset the weaknesses of forward and backward search in different domains by integrating forward search with backward search.

# Chapter 6

# Bidirectional Search

Besides the independent forward and backward search, the possibility to perform the combined forward and backward search already exists. This bidirectional method concurrently searches in both forward and backward directions, which has been successfully applied in optimal classical planning, and the use now still keeps expanding since it saves time and space complexity when problems can be solved meet-in-the-middle as proved in 2.9.

Bidirectional search provides a platform to perform backward BFWS as a competitive search strategy. Standard bidirectional search has two *Open* and *Close* lists, $Open_f$ and $Close_f$ for forward search and the $Open_b$ and $Close_b$ for backward search. The plan is found when the forward and backward frontiers intersect. Besides, according to different heuristic value updating methods, bidirectional search can be divided into *front-to-front* version and *front-to-end* version.

## 6.1 *k*-BFWBS

Bidirectional search used in this thesis can be treated as a simplification of TTBS (Felner et al., 2010) with some modifications. Firstly, TTBS reevaluates the heuristic value of states in *Open* if $d$ is not *similar* to $D(s)$, where $d$ is the top (lowest $h$-value) state in the opposite *Open*; $D(s)$ is the state which is the $d$-node when $s$ is evaluated; *similar* means $d$ is $D(s)$ or a successor of $D(s)$. While in this thesis, the top node $d$ is only used to evaluate the heuristic value of successors of state $s$ in the opposite *Open*, and the reevaluation is removed since we optimistically believe that bidirectional search always keeps progression under the direction of the top state (with the lowest heuristic value) in frontiers and the directions of its successors. Secondly, TTBS deems the goal achieved under two conditions: The forward or backward expanded state $s$ intersects with the original goal state or with $d$, for forward $S_G \subseteq s \vee \mathrm{d} \subseteq \mathrm{s}$, for backward $S \subseteq s_0 \vee S \subseteq d$; the forward or backward state $s'$ generated by state $s$ intersect with states generated by $d$, for forward $s' \in gen_b$, for backward $s' \in gen_f$. Furthermore, the intersection detection of $s' \in gen_b$ or $s' \in gen_f$ is implemented as looking up of hash sets. This is efficient, but it possibly misses intersections. In contrast, we take two different goal check methods in our bidirectional search.

1. We detect the intersection of the state $s$ which has the lowest $h$-value in *Open* with all states in the opposite *Close* (named as *check Close*).
2. We detect the intersection of the state $s$ which has lowest $h$-value in *Open* with $d$ (named as *check Head*).

Besides different goal check methods, different $h$-value updating methods *front-to-front* and *front-to-end* are also adopted.

### 6.1.1 Six Different Combinations

Bidirectional search in this thesis integrates forward $k$-BFWS($f5$) and backward $k$-BFWS($f5$) with six different combinations.

1. The $\#g$ and $\#r$ are evaluated by *front-to-end* and goal check with *check Head*, named as FB_H*;*
2. The $\#g$ and $\#r$ are evaluated by *front-to-end* and goal check with *check Close*, named as FB_C;
3. The $\#g$ is evaluated by *front-to-end* while $\#r$ is reevaluated by *front-to-front* and goal check with *check Head*, named as FB_R_H;
4. Both $\#g$ and $\#r$ are reevaluated by *front-to-front* and goal check with *check Head*, named as FB_R_G_H;

5. Both $\#g$ and $\#r$ are reevaluated by *front-to-front* and goal check with *check Head*. Meanwhile, if forward or backward search stops, the other search will try to achieve the top state (with lowest $h$-value) in the opposite *Close*, named as FB_R_G_C_H;

6. Both $\#g$ and $\#r$ are reevaluated by *front-to-front* and goal check with *check Close*, named as FB_R_G_C.

This $k$-BFWS($f5$) based bidirectional search is named as $k$-BFWBS. In this thesis, forward $k$-BFWS($f5$) and backward $k$-BFWS($f5$) alternate in 1-step each. $k$-BFWBS keeps searching when forward $k$-BFWS($f5$) or backward $k$-BFWS($f5$) stops. Forward search stops when $Open_f = \varnothing$, and backward search stops when $Open_b = \varnothing$, and $k$-BFWBS changes to single forward or backward search when one search stops. It returns no solution until forward $k$-BFWS($f5$) and backward $k$-BFWS($f5$) both stop.

## 6.2 Experiments

To better analyse the performance of six different combinations, three baselines are selected.

1. Only run forward $k$-BFWS($f5$);

2. Only run backward $k$-BFWS($f5$);

3. Forward $k$-BFWS($f5$) is executed first, and then backward $k$-BFWS($f5$) is executed if the plan is not found by forward $k$-BFWS($f5$)), named as FB.

Selected benchmark domains come from the satisficing tracks of IPCs of years 1998–2018 in our experiments, and the results about total solved problems are summarised in 6.1. Three main aspects, problems solved by forward search in 6.2, problems solved by backward search in 6.3, and problems solved when the search meets in the middle in 6.4, are used to evaluate $k$-BFWBS. The meet-in-the-middle is an important aspect as it can reduce the time and space complexity. $k$-BFWBS is implemented by using LAPKT toolkit. All experiments are performed on the cloud computer with 2.0GHz Intel Xeon processors with time outing after 30 min and memory outing after 10GB.

| Domain | P | S | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | F | B | FB | FB_H | FB_C | FB_R_H | FB_R_G_H | FB_R_G_C | FB_R_G_C_H |
| agricola18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| airport | 50 | 37 | 25 | **46** | 27 | 24 | 27 | 14 | 18 | 14 |
| barman14 | 20 | 20 | 0 | 20 | 20 | 1 | 20 | 0 | 0 | 0 |
| blocks world | 50 | 30 | 30 | **45** | 24 | 25 | 25 | 12 | 17 | 19 |
| caldera18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| caldera-split-18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| childsnack14 | 20 | 0 | **6** | **2** | **6** | **2** | **3** | **1** | 0 | **1** |
| cybersec | 30 | 15 | 10 | **19** | 12 | 4 | 12 | 6 | 6 | 6 |
| data-network18 | 20 | 8 | 0 | 8 | 7 | 4 | 5 | 0 | 1 | 0 |
| depot | 22 | 22 | 3 | 22 | 22 | 21 | 22 | 3 | 4 | 3 |
| driverlog | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 14 | 17 | 14 |
| elevators11 | 20 | 20 | 13 | 20 | 20 | 10 | 14 | 0 | 0 | 0 |
| ferry | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 23 |
| floortile14 | 20 | 2 | **20** | **20** | **20** | **20** | **20** | **19** | **18** | **19** |
| ged14 | 20 | 19 | 0 | 19 | 18 | 15 | 17 | 0 | 0 | 0 |
| gripper | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 19 | 20 | 19 |
| hanoi | 30 | 4 | **6** | **6** | **6** | **6** | **6** | 4 | 4 | 4 |
| hiking14 | 20 | 3 | 2 | 3 | **5** | 1 | 1 | 0 | 0 | 0 |
| logistics | 50 | 44 | 21 | 44 | 37 | 17 | 35 | 10 | 3 | 15 |
| miconic | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 39 | 20 | 45 |
| mprime | 70 | 65 | 2 | 65 | 58 | 55 | 53 | 49 | 54 | 49 |
| mystery | 60 | 38 | 14 | 38 | 35 | 32 | 30 | 29 | 25 | 29 |
| no-mprime | 35 | 32 | 1 | 32 | 27 | 26 | 26 | 24 | 27 | 24 |
| no-mystery | 30 | 19 | 7 | 19 | 17 | 16 | 15 | 15 | 11 | 15 |
| nomystery11 | 20 | 13 | 10 | 13 | 11 | 9 | 11 | 2 | 2 | 2 |
| openstacks | 30 | 27 | 26 | 27 | 27 | 25 | 27 | 10 | 18 | 10 |
| openstacks14 | 20 | 15 | 0 | 15 | 0 | 0 | 0 | 0 | 0 | 0 |
| parcprinter11 | 20 | 13 | **18** | **18** | **18** | **16** | **18** | 1 | 2 | 1 |
| parking14 | 20 | 20 | 0 | 20 | 20 | 20 | 20 | 0 | 0 | 0 |
| pathways | 30 | 24 | 5 | 24 | 24 | 22 | 24 | 5 | 4 | 5 |
| pegsol11 | 20 | 9 | 1 | 9 | 9 | **11** | 9 | 0 | 7 | 2 |
| pipesworld06 | 50 | 30 | 6 | 30 | 30 | 30 | 29 | 12 | 10 | 12 |
| pipesworld-notankage | 50 | 50 | 15 | 50 | 49 | 45 | 47 | 23 | 16 | 22 |
| pipesworld-tankage | 50 | 30 | 6 | 30 | 30 | 30 | 29 | 12 | 10 | 12 |
| rovers | 20 | 20 | 19 | 20 | 20 | 20 | 20 | 15 | 11 | 15 |
| scanalyzer11 | 20 | 18 | **20** | **20** | **20** | **20** | **20** | 6 | 13 | 6 |
| settlers18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| snake18 | 20 | 11 | 0 | 11 | 6 | 3 | 5 | 1 | 2 | 1 |
| sokoban11 | 20 | 5 | 1 | 5 | 5 | 2 | 5 | 1 | 1 | 2 |
| spider18 | 20 | 13 | 0 | 13 | 9 | 7 | 8 | 0 | 0 | 0 |
| storage | 30 | 29 | 14 | 29 | 29 | 27 | 27 | 16 | 17 | 16 |
| termes18 | 20 | 1 | **2** | **2** | **2** | 0 | **2** | 1 | 1 | 0 |
| thoughtful14 | 20 | 20 | 5 | 20 | 20 | 16 | 17 | 5 | 5 | 5 |
| tpp | 30 | 30 | 10 | 30 | 30 | 20 | 24 | 9 | 7 | 9 |
| transport14 | 20 | 7 | 0 | 7 | 6 | 2 | 2 | 0 | 0 | 0 |
| trucks | 30 | 7 | **14** | **10** | **13** | **10** | **13** | 6 | 4 | 6 |
| tyreworld | 30 | 30 | 12 | 30 | 30 | 30 | 30 | 3 | 3 | 3 |
| visitall14 | 20 | 19 | 17 | **20** | 18 | 18 | 17 | 0 | 0 | 0 |
| woodworking11 | 20 | 20 | 11 | 20 | 20 | 20 | 20 | 1 | 1 | 1 |
| zenotravel | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 18 | 18 | 18 |
| summary | 1417 | 979 | 512 | **1041** | 947 | 822 | 895 | 435 | 427 | 447 |

TABLE 6.1: Total solved problems with six different $k$-BFWBS combinations and three baselines. The red highlight means better than forward search. P is the number of problems in each domain. S is the number of solved problems.

| Domain | P | FS | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | F | FB | FB_H | FB_C | FB_R_H | FB_R_G_H | FB_R_G_C | FB_R_G_C_H |
| agricola18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| airport | 50 | 37 | 37 | 7 | 0 | 7 | 3 | 3 | 3 |
| barman14 | 20 | 20 | 20 | 20 | 0 | 20 | 0 | 0 | 0 |
| blocks world | 50 | 30 | 30 | 14 | 0 | 10 | 0 | 0 | 0 |
| caldera18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| caldera-split-18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| childsnack14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cybersec | 30 | 15 | 15 | 2 | 0 | 2 | 0 | 0 | 0 |
| data-network18 | 20 | 8 | 8 | 7 | 2 | 5 | 0 | 0 | 0 |
| depot | 22 | 22 | 22 | 22 | 16 | 22 | 2 | 0 | 2 |
| driverlog | 20 | 20 | 20 | 20 | 15 | 20 | 0 | 0 | 0 |
| elevators11 | 20 | 20 | 20 | 20 | 9 | 14 | 0 | 0 | 0 |
| ferry | 30 | 30 | 30 | 28 | 3 | 28 | 0 | 0 | 0 |
| floortile14 | 20 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| ged14 | 20 | 19 | 19 | 18 | 15 | 17 | 0 | 0 | 0 |
| gripper | 20 | 20 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| hanoi | 30 | 4 | 4 | 2 | 2 | 2 | 3 | 1 | 3 |
| hiking14 | 20 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| logistics | 50 | 44 | 44 | 35 | 1 | 34 | 1 | 0 | 1 |
| miconic | 50 | 50 | 50 | 2 | 0 | 22 | 5 | 0 | 4 |
| mprime | 70 | 65 | 65 | 58 | 51 | 53 | 44 | 8 | 44 |
| mystery | 60 | 38 | 38 | 34 | 23 | 29 | 28 | 1 | 28 |
| no-mprime | 35 | 32 | 32 | 27 | 25 | 26 | 22 | 4 | 22 |
| no-mystery | 30 | 19 | 19 | 16 | 12 | 14 | 14 | 1 | 14 |
| nomystery11 | 20 | 13 | 13 | 6 | 0 | 6 | 0 | 0 | 1 |
| openstacks | 30 | 27 | 27 | 14 | 9 | 14 | 0 | 0 | 0 |
| openstacks14 | 20 | 15 | 15 | 0 | 0 | 0 | 0 | 0 | 0 |
| parcprinter11 | 20 | 13 | 13 | 3 | 0 | 3 | 0 | 0 | 0 |
| parking14 | 20 | 20 | 20 | 20 | 19 | 20 | 0 | 0 | 0 |
| pathways | 30 | 24 | 24 | 21 | 0 | 21 | 2 | 0 | 2 |
| pegsol11 | 20 | 9 | 9 | 9 | 2 | 9 | 0 | 0 | 0 |
| pipesworld06 | 50 | 30 | 30 | 29 | 25 | 28 | 7 | 0 | 7 |
| pipesworld-notankage | 50 | 50 | 50 | 48 | 38 | 46 | 17 | 0 | 17 |
| pipesworld-tankage | 50 | 30 | 30 | 29 | 25 | 28 | 7 | 0 | 7 |
| rovers | 20 | 20 | 20 | 13 | 12 | 13 | 4 | 0 | 4 |
| scanalyzer11 | 20 | 18 | 18 | 5 | 4 | 5 | 0 | 0 | 0 |
| settlers18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| snake18 | 20 | 11 | 11 | 6 | 2 | 5 | 1 | 0 | 1 |
| sokoban11 | 20 | 5 | 5 | 4 | 0 | 4 | 0 | 0 | 1 |
| spider18 | 20 | 13 | 13 | 9 | 6 | 8 | 0 | 0 | 0 |
| storage | 30 | 29 | 29 | 27 | 24 | 25 | 12 | 2 | 12 |
| termes18 | 20 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| thoughtful14 | 20 | 20 | 20 | 20 | 5 | 17 | 4 | 2 | 4 |
| tpp | 30 | 30 | 30 | 29 | 14 | 23 | 4 | 0 | 4 |
| transport14 | 20 | 7 | 7 | 6 | 2 | 2 | 0 | 0 | 0 |
| trucks | 30 | 7 | 7 | 2 | 1 | 2 | 6 | 0 | 6 |
| tyreworld | 30 | 30 | 30 | 30 | 0 | 30 | 0 | 0 | 0 |
| visitall14 | 20 | 19 | 19 | 8 | 0 | 7 | 0 | 0 | 0 |
| woodworking11 | 20 | 20 | 20 | 19 | 19 | 19 | 0 | 0 | 0 |
| zenotravel | 20 | 20 | 20 | 8 | 6 | 8 | 2 | 1 | 2 |
| summary | 1417 | 979 | 979 | 700 | 387 | 668 | 188 | 23 | 189 |
| percentage | - | 100% | 92% | 74% | 47% | 74% | 43% | 5% | 42% |

Table 6.2: Forward search solved problems with six different $k$-BFWBS combinations and two baselines. P is the number of problems in each domain. FS is the number of the forward search solved problems. The last row is the percentage of forward search solved problems among total solved problems.

| Domain | P | BS | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | B | FB | FB_H | FB_C | FB_R_H | FB_R_G_H | FB_R_G_C | FB_R_G_C_H |
| agricola18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| airport | 50 | 25 | 9 | 15 | 0 | 15 | 1 | 0 | 1 |
| barman14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| blocks world | 50 | 30 | 15 | 5 | 0 | 5 | 10 | 0 | 2 |
| caldera18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| caldera-split-18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| childsnack14 | 20 | 6 | 2 | 6 | 2 | 3 | 1 | 0 | 1 |
| cybersec | 30 | 10 | 4 | 10 | 2 | 10 | 0 | 0 | 0 |
| data-network18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| depot | 22 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| driverlog | 20 | 20 | 0 | 0 | 0 | 0 | 11 | 0 | 11 |
| elevators11 | 20 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ferry | 30 | 30 | 0 | 0 | 0 | 0 | 25 | 0 | 18 |
| floortile14 | 20 | 20 | 18 | 20 | 19 | 20 | 13 | 0 | 13 |
| ged14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| gripper | 20 | 20 | 0 | 20 | 0 | 20 | 18 | 0 | 18 |
| hanoi | 30 | 6 | 2 | 2 | 0 | 2 | 0 | 0 | 0 |
| hiking14 | 20 | 2 | 0 | 2 | 0 | 1 | 0 | 0 | 0 |
| logistics | 50 | 21 | 0 | 2 | 0 | 1 | 7 | 0 | 2 |
| miconic | 50 | 50 | 0 | 3 | 0 | 8 | 0 | 0 | 1 |
| mprime | 70 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 2 |
| mystery | 60 | 14 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| no-mprime | 35 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| no-mystery | 30 | 7 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| nomystery11 | 20 | 10 | 0 | 5 | 0 | 5 | 2 | 0 | 1 |
| openstacks | 30 | 26 | 0 | 13 | 0 | 13 | 1 | 0 | 1 |
| openstacks14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| parcprinter11 | 20 | 18 | 5 | 15 | 0 | 15 | 0 | 0 | 0 |
| parking14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pathways | 30 | 5 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| pegsol11 | 20 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| pipesworld06 | 50 | 6 | 0 | 1 | 0 | 1 | 2 | 0 | 2 |
| pipesworld-notankage | 50 | 15 | 0 | 1 | 0 | 1 | 3 | 0 | 2 |
| pipesworld-tankage | 50 | 6 | 0 | 1 | 0 | 1 | 2 | 0 | 2 |
| rovers | 20 | 19 | 0 | 6 | 0 | 6 | 4 | 0 | 4 |
| scanalyzer11 | 20 | 20 | 2 | 9 | 2 | 9 | 1 | 0 | 1 |
| settlers18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| snake18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sokoban11 | 20 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| spider18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| storage | 30 | 14 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| termes18 | 20 | 2 | 1 | 2 | 0 | 2 | 1 | 0 | 0 |
| thoughtful14 | 20 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| tpp | 30 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| transport14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| trucks | 30 | 14 | 3 | 11 | 0 | 11 | 0 | 0 | 0 |
| tyreworld | 30 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| visitall14 | 20 | 17 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| woodworking11 | 20 | 11 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| zenotravel | 20 | 20 | 0 | 12 | 5 | 12 | 15 | 1 | 15 |
| summary | 1417 | 512 | 62 | 166 | 31 | 166 | 120 | 1 | 100 |
| percentage | - | 100% | 8% | 18% | 4% | 18% | 27% | 0.20% | 22% |

TABLE 6.3: Backward search solved problems with six different $k$-BFWBS combinations and two baselines. P is the number of problems in each domain. BS is the number of backward search solved problems. The last row is the percentage of backward search solved problems among total solved problems.

| Domain | P | M | | | | | |
|---|---|---|---|---|---|---|---|
| | | FB_H | FB_C | FB_R_H | FB_R_G_H | FB_R_G_C | FB_R_G_C_H |
| agricola18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| airport | 50 | 5 | 24 | 5 | 10 | 15 | 10 |
| barman14 | 20 | 0 | 1 | 0 | 0 | 0 | 0 |
| blocks world | 50 | 5 | 25 | 10 | 2 | 17 | 17 |
| caldera18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| caldera-split-18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| childsnack14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| cybersec | 30 | 0 | 2 | 0 | 6 | 6 | 6 |
| data-network18 | 20 | 0 | 2 | 0 | 0 | 1 | 0 |
| depot | 22 | 0 | 5 | 0 | 1 | 4 | 1 |
| driverlog | 20 | 0 | 5 | 0 | 3 | 17 | 3 |
| elevators11 | 20 | 0 | 1 | 0 | 0 | 0 | 0 |
| ferry | 30 | 2 | 27 | 2 | 5 | 30 | 5 |
| floortile14 | 20 | 0 | 1 | 0 | 6 | 18 | 6 |
| ged14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| gripper | 20 | 0 | 20 | 0 | 1 | 20 | 1 |
| hanoi | 30 | 2 | 4 | 2 | 1 | 3 | 1 |
| hiking14 | 20 | 0 | 1 | 0 | 0 | 0 | 0 |
| logistics | 50 | 0 | 16 | 0 | 2 | 3 | 13 |
| miconic | 50 | 45 | 50 | 20 | 34 | 20 | 40 |
| mprime | 70 | 0 | 4 | 0 | 3 | 46 | 3 |
| mystery | 60 | 0 | 9 | 0 | 1 | 24 | 1 |
| no-mprime | 35 | 0 | 1 | 0 | 1 | 23 | 1 |
| no-mystery | 30 | 0 | 4 | 0 | 1 | 10 | 1 |
| nomystery11 | 20 | 0 | 9 | 0 | 0 | 2 | 0 |
| openstacks | 30 | 0 | 16 | 0 | 9 | 18 | 9 |
| openstacks14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| parcprinter11 | 20 | 0 | 16 | 0 | 1 | 2 | 1 |
| parking14 | 20 | 0 | 1 | 0 | 0 | 0 | 0 |
| pathways | 30 | 2 | 22 | 2 | 3 | 4 | 3 |
| pegsol11 | 20 | 0 | 9 | 0 | 0 | 7 | 0 |
| pipesworld06 | 50 | 0 | 5 | 0 | 3 | 10 | 3 |
| pipesworld-notankage | 50 | 0 | 7 | 0 | 3 | 16 | 3 |
| pipesworld-tankage | 50 | 0 | 5 | 0 | 3 | 10 | 3 |
| rovers | 20 | 1 | 8 | 1 | 7 | 11 | 7 |
| scanalyzer11 | 20 | 6 | 14 | 6 | 5 | 13 | 5 |
| settlers18 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| snake18 | 20 | 0 | 1 | 0 | 0 | 2 | 0 |
| sokoban11 | 20 | 1 | 2 | 1 | 1 | 1 | 1 |
| spider18 | 20 | 0 | 1 | 0 | 0 | 0 | 0 |
| storage | 30 | 1 | 3 | 1 | 4 | 15 | 4 |
| termes18 | 20 | 0 | 0 | 0 | 0 | 1 | 0 |
| thoughtful14 | 20 | 0 | 11 | 0 | 1 | 3 | 1 |
| tpp | 30 | 1 | 6 | 1 | 5 | 7 | 5 |
| transport14 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| trucks | 30 | 0 | 9 | 0 | 0 | 4 | 0 |
| tyreworld | 30 | 0 | 30 | 0 | 3 | 3 | 3 |
| visitall14 | 20 | 10 | 18 | 10 | 0 | 0 | 0 |
| woodworking11 | 20 | 0 | 0 | 0 | 1 | 1 | 1 |
| zenotravel | 20 | 0 | 9 | 0 | 1 | 16 | 1 |
| summary | 1417 | 81 | 404 | 61 | 127 | 403 | 159 |
| percentage | - | 9% | 49% | 7% | 30% | 94% | 36% |

Table 6.4: Problems solved when search meets in the middle with six different $k$-BFWBS combinations. P is the number of problems in each domain. M is the number of solved problems when the search meets in the middle. The last row is the percentage of the meet-in-the-middle solved problems among total solved problems

.

## 6.3 Result Analysis

FB solved the most problems compared with other search strategies. Among 1417 problems, FB solved 1041 problems which is much higher than problems solved by only running forward $k$-BFWS($f5$) or backward $k$-BFWS($f5$); hence, backward search solved different problems compared with forward search. Especially, in the *floortile* domain, the majority of problems are solved by backward search, and the same is true in other six combinations. As a result, even though backward search is uncompetitive with forward search, it is still helpful to solve more problems.

$k$-BFWBS with *front-to-end* and *check Head* (FB_H) solved 947 problems, which is similar to the number of 979 problems solved by only running forward $k$-BFWS($f5$). Among 947 problems, 81 problems are solved by meet-in-the-middle, which is nearly 9%. This result shows that FB_H is competitive with forward $k$-BFWS($f5$), but it still does not outperform forward $k$-BFWS($f5$). Two reasons can explain this. For one thing, FB_H needs to switch between forward search and backward search, while forward $k$-BFWS($f5$) avoids this process. For the other thing, problems solved by forward and backward search meet-in-the-middle are limited. It means that with the same searching time, only running forward search can execute the search more deeply while forward or backward search in FB_H cannot reach the same depth.

Besides the difference of total solved problems, the FB_H solved 700 problems by forward search among total solved 947 problems, nearly 74%; however, the FB_H only solved 166 problems by backward search among total solved 947 problems, nearly 18%. Although 18 percent of solved problems by backward search is higher than what other two combinations FB_C and FB_R_G_C have done, the gap between problems solved by forward and by backward is obvious. To be honest, forward search still plays a more important role in $k$-BFWBS among all tested combinations.

$k$-BFWBS with *front-to-end* and *check Close* (FB_C) solved 822 problems which are fewer than what forward $k$-BFWS($f5$) and FB_H did. It is true that the *check Close* is time-consuming since it checks the expanded state with all states in the opposite *Close* so that it can avoid the lost intersection. Among solved 822 problems, 404 problems are solved in the middle, and the percentage is nearly 49, which is quite higher than FB_H.

In FB_H and FB_C, the heuristic evolution methods are based on the *front-to-end*. In other words, FB_H and FB_C can be regarded as running forward $k$-BFWS($f5$) and backward $k$-BFWS($f5$) concurrently with the *check Head* and *check Close* added during the search. This modification is slight and easily achieved. With these slight modifications, both FB_H and FB_C outperforms backward $k$-BFWS($f5$) obviously which only solved 512 problems among total 1417 problems. For the domains *floortile* and *parcprinter*, backward $k$-BFWS($f5$) solved more problems than forward $k$-BFWS($f5$). Similarly, both FB_H and FB_C solved more problems than forward $k$-BFWS($f5$) in these domains. To conclude, the $k$-BFWBS with modifications on goal check methods does offset the weakness of each other, forward search and backward search.

The following four combinations of FB_R_H, FB_R_G_H, FB_R_G_C and FB_R_G_C_H are discussed together since they all update the heuristic value based on the *front-to-front*. Different from FB_H and FB_C, which only add new goal check method, the modification of these four searches are more complex since the computation of heuristic value also needs to be changed, but it can help the search meet in the middle.

$k$-BFWBS with *front-to-front* $\#r$ updating and *check Head* (FB_R_H) solved 895 problems. This number is fewer than forward $k$-BFWS($f5$) and FB_H, but is bigger than FB_C. Among solved 895 problems, 668 problems are solved by forward search, which is nearly 74%; 166 problems are solved by backward search, which is nearly 18%; 61 problems are solved in the middle, which is nearly 7%. Compared with the FB_H, the percentage of problems solved in the middle is nearly the same, 9% for FB_H and 7% for FB_R_H. However the *fort-to-front* $h$-value updating causes fewer problems solved in FB_R_H compared with FB_H since the updated $h$-value cannot ensure the goal state is reachable. This influence is limited in FB_R_H since $\#r$ just assists the computation of novelty in $k$-BFWS($f5$).

$k$-BFWBS with *front-to-front* $\#r$ and $\#g$ updating and *check Head* (FB_R_G_H) solved 435 problems. Among 435 problems, 188 problems are solved by forward search, which is nearly 43%; 120 problems are solved by backward search, which is nearly 27%, and 127 problems are solved in the middle, which is nearly 30%.

Compared with FB_R_H, the number of solved problems decrease from 915 to 435 while the percentage of problems solved in the middle increases from 7 in FB_R_H to 30 in FB_R_G_H. This result suggests that the *fort-to-front* $\#g$ $h$-value updating can increase the possibility of meet-in-the-middle, but it also causes fewer problems solved. The influence of fort-to-front $\#g$ updating is significant since $\#g$ plays an important role in $k$-BFWS($f5$) to break ties. Interestingly, in FB_R_G_H, the percentage of problems solved by backward search is nearly 27, which is the highest percentage among tested combinations.

FB_R_G_C_H solved 447 problems among total. FB_R_G_C_H updates $\#r$ and $\#g$ with *front-to-front* and check goal state with *check Head*. Meanwhile, if one search stops, the other search will keep trying to meet the top state in opposite *Close* in FB_R_G_C_H. Among solved 447 problems, forward search solved 189 problems, which is nearly 42%; backward search solved 100 problems, which is nearly 22%; and 159 problems are solved in the middle, which is nearly 36%. Compared with FB_R_G_H, the number of total solved problems and the percentage of forward, backward and meet-in-the-middle solved problems are slightly different. This is reasonable since the modification of trying to meet the top state in opposite *Close* aims to force FB_R_G_C_H to have more chances to meet in the middle, and the number of solved problems when the search meets in the middle indeed increases from 127 in FB_R_G_H to 159 in FB_R_G_C_H.

$k$-BFWBS with *front-to-front* $\#r$ and $\#g$ updating and *check Close* (FB_R_G_C) solved 427 problems which is the least among all combinations. Among 427 problems, forward search solved 23 problems; backward search only solved 1 problem; 403 problems are solved in the middle, and the percentage is 94, which is the highest. In FB_R_G_C, all modifications work for meet-in-the-middle, and these attempts are resultful.

## 6.4 Summary

The FB solved the most problems compared with other tested search strategies, and it means that backward search is helpful to solve more problems when combining with forward search. $k$-BFWBS can outperform backward $k$-BFWS($f5$) search when $\#g$ is updated with *front-to-end*, but it becomes worse when $\#g$ is updated with *front-to-front*. The updating methods about $\#r$ also can influence the performance of $k$-BFWBS, but it is not as powerful as $\#g$ such as FB_R_H. However, after adopting the *front-to-front* $\#g$ updating in FB_R_H, more problems can be solved meet-in-the-middle. Meanwhile, the *front-to-front* $\#g$ updating with *check Head* is more in favor of backward search, since the percentage of solved problems by backward is higher than other combinations with *front-to-end* $\#g$ updating. For example, in FB_H and FB_R_H, the percentage of solved problems by backward search are both 18 while in FB_R_G_H and FB_R_G_C_H, the percentage is 27 and 22 respectively. Other features are *check Close* and *check Head*. It is clear that *check Close* is powerful to help the search meet in the middle while it causes fewer solved problems. In contrast, *check Head* can ensure the number of solved problems but cannot help the search meet in the middle.

Chapter 7

# Discussion and Further Work

## 7.1 Importance of Backward Search

The above analysis demonstrates backward search is uncompetitive with forward search in both SIW and BFWS. Even if we try to integrate forward and backward $k$-BFWS to build $k$-BFWBS, it still can not outperform forward $k$-BFWS. It is hard to conclude that backward search is useless. Actually, there are two possible ways to interpret the results. We may either view backward search as a different perspective to understand the classical planning or treat backward search as a form of the search algorithm to compete with other search algorithms. We prefer the first view in this thesis. Even though current researches on regression suggest that regression is harder than progression and that planners solved fewer problems after converting to the backward version as showed in the above experiments, regression is still worth analysing since it not only explains the search process from goal state but also has outperformed progression in some domains. Meanwhile, if we run forward search first and then run backward search such as FB, more problems are solved compared with only running forward search.

## 7.2 Challenges of Backward Search

### 7.2.1 Challenge of Partial State

For backward search, we adopt a series of techniques to keep the search manageable, but some issues have not been solved yet. In the following part, we move to the problem definition to review backward search. In the regression state space, partial states commonly exist. In partial states, the included fluents are true, but other excluded fluents are possibly true. In other words, in regression, there are multiple uncertain fluents that they should be true but not be included in the generated states. We raise the following question that whether backward search can perform better if we reduce this uncertainty.

We propose a goal state complement solution in this thesis. The goal state as the initial state for backward search determines the number of positive fluents at search beginning, while the goal state keeps the same characters as the partial states in regression. The fluents in the goal state are certainly true, but other fluents are uncertain. As a result, we specify the goal state by adding certainly positive fluents into it. Even though we do not achieve the complement of all partial states in regression, the goal state complement method also removes some useless search paths and speeds up the search. We add the fluent $Fn$ into the goal state with the simple rule that the fluent $Fn$ can be added into the goal state if there is an action including $Fn$ and goal fluents in the add list, and $Fn$ is not mutex with any one of goal fluents. The goal state complement method has not been formalised efficiently, so limited domains are tested. The results show that the goal state complement can reduce the number of generated and expanded states in domains *blocks world*, *parking* and *elevators*.

### 7.2.2 Challenge of Mutex Detection

The mutex check is another reason which causes backward search harder. As mentioned, all newly generated states should check that all pairs of fluents in that states are mutex or not. If one pair of fluents is mutex in state $s$, the newly generated state $s$ should be pruned. The mutex check can ensure the generated states are reachable from the initial state to reduce the search paths. On the other hand, if the mutex check cannot detect all unreachable states, the search paths will increase and waste the computing resource. In this thesis, the mutex means the $h2$ value of the pair of fluents is infinite while the $h2$ cannot capture all mutexes. For example, in one IPC domain *elevator*, there are three passengers who want to use the elevators. But during backward search, the $h2$ regards the pair of fluents <*passenger* $p0$ *at floor n0*, *elevator slow*1 *hold* 3 *passengers*> is reachable since the $h2$ value is not infinite after computing. However, it should be infinite since there are only three passengers, but four passengers are described in this fluents pair. A lot of efforts have been paid to improve the detection of mutex, but it is an NP-hard problem if all mutexes need to be detected. Even if other mutex computation methods are better than $h2$ such as $h3$, it is still uncompleted and hampers backward search efficiency.

### 7.2.3 Challenge of Forward Domain Description

Besides the mutex detection, the *complete* domain description also can remove the unreachable states. Compared with the mutex detection, *complete* domain description is more easily achieved, and it does not require complex computation. In the previous researches, the word *complete* has never been used to describe the domains since researchers assume that all benchmark domains in IPCs are defined with enough information to conduct the search. However, the IPCs benchmark domains are described for forward search rather than backward search. The forward domains only contain all information needed to conduct forward search while it is not enough for backward search. Previous researches all avoided this issue since the researches about backward search mainly focus on the algorithms. The *complete* domain is defined as:

- *complete* domains should contain all necessary description to conduct forward search but should not mislead backward search when adopting these domains

One reasonable way to build the *complete* domains is modifying the current domains with additional restrictions. Although we have not formulated a series of rules to transfer the forward domains to *complete* domains, we use the *parking* domains as an example to demonstrate the necessary restrictions in *complete* domains.

In *parking* domain shown in A.4, the task is to park cars on the street with $n$ curb locations. One curb can park one car, and the other car can park in front of this car.

The goal is to find a solution to move from one configuration of parked cars to another configuration by moving cars from one curb location to another. The description of one action in *parking* domain is shown as follows.

> (:action *move-car-to-car* :parameters (*?car* - *car* *?carsrc* - *car* *?cardest* - *car*)
> :precondition (and (*car-clear ?car*) (*car-clear ?cardest*) (*behind-car ?car ?carsrc*) (*at-curb ?cardest*))
> :effect (and (*not* (*car-clear ?cardest*)) (*car-clear ?carsrc*) (*behind-car ?car ?cardest*) (not (*behind-car ?car ?carsrc*)) (*increase* (*total-cost*) 1)))

The action *move-car-to-car* can move the *car* which is in the front of *carsrc* to the front of *cardest*. Even if some incorrect actions such as (*move-car-to-car car*1 *car*1 *car*1), (*move-car-to-car car*1 *car*1 *car*0) and (*move-car-to-car car*1 *car*0 *car*1) would be generated based on this forward actions description, forward search still can perform very well since these incorrect actions would not be applied due to the unreachable precondition. However, in backward search, these incorrect actions would be applied. For example:

> (:action (move-car-to-car car1 car0 car1)
> :Precondition((car-clear car1), (behind-car car1car0), (at-curb car1))
> :Add ((car-clear car0), (behind-car car1 car1))
> :Del ((car-clear car1), (behind-car car1 car0)))

This action can be applied after the relevance and consistency check, but it generates the impossible fluent (*behind-car car*1 *car*1) in backward search and leads to no plan found finally.

One reasonable solution is modifying this action description to make it complete with additional restrictions that the moved *car* should not be the car behind itself, *carsrc*, and should not be the car at the moving destination, *cardest*. After that a *complete* action description is shown as following.

> (:action *move-car-to-car* :parameters (*?car* - *car* *?carsrc* - *car* *?cardest* - *car*)
> :precondition (and (*car-clear ?car*) (*car-clear ?cardest*) (*behind-car ?car ?carsrc*) (*at-curb ?cardest*)) **(*not* (=? *car ?cardest*)) (*not* (=? *car ?carsrc*))**
> :effect (and (*not* (*car-clear ?cardest*)) (*car-clear ?carsrc*) (*behind-car ?car ?cardest*) (not (*behind-car ?car ?carsrc*)) (*increase* (*total-cost*) 1)))

By adopting this *complete* action description, all incorrect actions mentioned above would not be generated. In the *parking* domain, another actions also should be perfected like *move-car-to-car* to ensure the whole domain is *complete*. In PDDL, the domain description contains the domain name, requirements, the object type definition, predicates, and actions. Based on our observation, only the forward action descriptions can cause the forward domain *incomplete*. Hence the *complete* domains can be regarded as a serial of *complete* actions in original forward domains.

### 7.2.4 Further Work

In this thesis, we mainly focus on backward width-based search while the performance of backward heuristic-based search would not be included. It is worth comparing backward heuristic-based search with backward width-based search so that we are able to analyse the influence of the regression state space on each search strategy. Meanwhile, in the $k$-BFWBS, the *check Close* method is not effective enough since we need to look through all states in the opposite *Close* list, and it also impacts on the number of solved problems compared with *check Head*. The further work should focus on finding more effective algorithms for the *check Close* such as only checking states with novelty 1 in the opposite *Close* list or using the symbolic representation like binary decision diagram. In addition, the results of applying two different $\#r$ and $\#g$ updating methods, *front-to-end* and *front-to-front*, in $k$-BFWBS have been discussed. Some further work should be done to improve these two updating methods in $k$-BFWBS to solve more problems and help the search meet in the middle. Finally, the methods to improve the mutex detection, partial state complement and complete domain description are also worthy of study. Although these three problems are more serious in backward search and can be ignored in forward search, we should not neglect these issues as they deserve further study.

# Chapter 8

# Conclusion

In this thesis, the research about duality mapping enlightens us to explore the performance of width-based search on dual problems since they are quite difficult to solve for modern heuristic-based search. The experiment results suggest that the effective width of dual problems is reduced significantly compared with original problems when the goal state is restricted to a single fluent. We use the *blocks word* domain as an example to explain the reasons and forecast that dual problems with conjunctive goal fluents should be easily solved by *SIW*. However, *SIW* cannot solve the majority of dual problems because of the novelty, compared with heuristic-based search.

Then we build backward *SIW* and tow types of backward BFWS which are suitable for a wide range of IPCs benchmark domains. Backward search as an old strategy to search from the goal state has been widely adopted in the optimal planning, and related researches have proposed many issues which make backward search difficult. This thesis has studied and presents the relevant solutions in the *SIW*, BFWS($f5$) and $k$-BFWS($f5$). In the regression state space, the new goal consistency check, negative fluents generating, unreachable states pruning and heuristic updating methods are modelled via extensive literature review to ensure that backward search is manageable. Backward *SIW*, BFWS($f5$) and $k$-BFWS($f5$)) have been tested in the IPCs benchmark domains, and a sensitivity analysis is conducted to analyse the influence of regression modifications on forward search. The results suggest that backward search is uncompetitive with forward search in all *SIW*, BFWS($f5$) and $k$-BFWS($f5$). However, all backward *SIW*, BFWS($f5$) and $k$-BFWS($f5$) can outperform forward versions in some domains, such as the *floortile* which is used as an example to explain the reasons.

After that, we integrate forward and backward $k$-BFWS($f5$) into $k$-BFWBS with six different combinations by considering two different heuristic updating methods and two different goal check methods. Although these combinations cannot solve more problems than only running forward $k$-BFWS($f5$) or FB among tested IPCs benchmark domains, these six bidirectional search algorithms are distinctive. It is worth noting that FB, which runs forward search first and then run backward search if forward search cannot find the plan, does solve the most problems compared with other search strategies, and this result shows that backward search is helpful to solve more problems.

In the end, we discuss the positive attitudes that researchers should hold toward backward search even if it is uncompetitive compared with forward search. In addition, the present studies are lack of detailed analysis of the mutex computation, partial state complement and complete domain description. Even if we propose some methods to improve these missing fields in backward search, we sincerely hope further studies of backward search would be undertaken together with increasing implementation of it in the current planners to eliminate the existing problems.

# Bibliography

Alami, R., Fleury, S., Herrb, M., Ingrand, F., and Robert, F. (1998). Multi-robot cooperation in the martha project. *IEEE Robotics & Automation Magazine*, 5(1):36–47.

Alcázar, V., Borrajo, D., Fernández, S., and Fuentetaja, R. (2013). Revisiting regression in planning. In *Twenty-Third International Joint Conference on Artificial Intelligence*. Citeseer.

Alcázar, V., Fernández, S., and Borrajo, D. (2014). Analyzing the impact of partial states on duplicate detection and collision of frontiers. In *ICAPS*. Citeseer.

Alcázar, V. and Torralba, A. (2015). A reminder about the importance of computing and exploiting invariants in planning. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*.

Allen, J. and Ferguson, G. (2002). Human-machine collaborative planning. In *Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space*, pages 27–29.

Bacchus, F. (2001). Aips 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *Ai magazine*, 22(3):47–47.

Bernard, D., Doyle, R., Riedel, E., Rouquette, N., Wyatt, J., Lowry, M., and Nayak, P. (1999). Autonomy and software technology on nasa's deep space one. *IEEE Intelligent Systems and their Applications*, 14(3):10–15.

Betz, C. and Helmert, M. (2009). Planning with h+ in theory and practice. In *Annual Conference on Artificial Intelligence*, pages 9–16. Springer.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence. 2001 Jun; 129 (1-2): 5-33.*

Bylander, T. (1994). The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204.

de Greef, T., Oomes, A. H., and Neerincx, M. A. (2009). Distilling support opportunities to improve urban search and rescue missions. In *International Conference on Human-Computer Interaction*, pages 703–712. Springer.

Felner, A., Moldenhauer, C., Sturtevant, N. R., and Schaeffer, J. (2010). Single-frontier bidirectional search. In *AAAI*.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208.

Fukunaga, A., Rabideau, G., Chien, S., and Yan, D. (1997). Aspen: A framework for automated planning and scheduling of spacecraft control and operations. In *Proc. International Symposium on AI, Robotics and Automation in Space*.

García-Olaya, Á., Jiménez, S., and Linares López, C. (2011). The 2011 international planning competition.

Geffner, P. H. H. and Haslum, P. (2000). Admissible heuristics for optimal planning. In *Proceedings of the 5th Internat. Conf. of AI Planning Systems (AIPS 2000)*, pages 140–149.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668.

Green, C. (1981). Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

Hoffmann, J. and Edelkamp, S. (2005). The fourth international planning competition. *Journal of Artificial Intelligence Research*, 24(519-579):5.

Hoffmann, J. and Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

Hoffmann, J., Porteous, J., and Sebastia, L. (2004). Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278.

Kautz, H. and Selman, B. (1999). Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325.

Kautz, H. A., Selman, B., et al. (1992). Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer.

Keyder, E. (2010). *New Heuristics For Classical Planning With Action Costs.* PhD thesis, PhD thesis, Universitat Pompeu Fabra, 2010. 13, 40, 98.

Kuroiwa, R. and Fukunaga, A. (2020). Front-to-front heuristic search for satisficing classical planning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 4098–4105.

Lipovetzky, N. et al. (2013). *Structure and inference in classical planning.* PhD thesis, Universitat Pompeu Fabra.

Lipovetzky, N. and Geffner, H. (2012). Width and serialization of classical planning problems.

Lipovetzky, N. and Geffner, H. (2017). Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, pages 3590–3596.

Long, D. and Fox, M. (2003). The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59.

Malasky, J., Forest, L. M., Kahn, A. C., and Key, J. R. (2005). Experimental evaluation of human-machine collaborative algorithms in planning for multiple uavs. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 2469–2475. IEEE.

McDermott, D. (1996). A heuristic estimator for means-ends analysis in planning. In *AIPS'96 Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, pages 142–149.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). Pddl-the planning domain definition language.

McDermott, D. M. (2000). The 1998 ai planning systems competition. *AI magazine*, 21(2):35–35.

Newell, A., Shaw, J. C., and Simon, H. A. (1959). Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA.

Pearl, J. (1984a). Heuristics addison-wesley. *Reading, MA*.

Pearl, J. (1984b). Intelligent search strategies for computer problem solving. *Addision Wesley*.

Pednault, E. P. (1989). Adl: Exploring the middle ground between strips and the situation calculus. *Kr*, 89:324–332.

Pohl, I. (1970). First results on the effect of error in heuristic search. *Machine Intelligence*, 5:219–236.

Politowski, G. and Pohl, I. (1984). D-node retargeting in bidirectional heuristic search. In *AAAI*, pages 274–277.

Pozanco, A., Fernández, S., and Borrajo, D. (2018). Distributed planning and model learning for urban traffic control. *KEPS 2018*, page 20.

Richter, S. and Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

Russell, S. and Norvig, P. (2002). Artificial intelligence: a modern approach. pages 94–95.

Russell, S. J. and Norvig, P. (2010). Artificial intelligence-a modern approach, third international edition.

Sirin, E., Parsia, B., Wu, D., Hendler, J., and Nau, D. (2004). Htn planning for web service composition using shop2. *Journal of Web Semantics*, 1(4):377–396.

Suchman, L. and Suchman, L. A. (2007). *Human-machine reconfigurations: Plans and situated actions*. Cambridge university press.

Suchman, L. A. (1987). *Plans and situated actions: The problem of human-machine communication*. Cambridge university press.

Suda, M. (2013). Duality in strips planning. *Computer Science*, 53(491):1191–1196.

Thornburg, K. M. and Thomas, G. W. (2009). Robotic exploration utility for urban search and rescue tasks. *JCP*, 4(10):975–980.

Weld, D. S. (1994). An introduction to least commitment planning. *AI magazine*, 15(4):27–27.

Wood, K. (1993). Urban traffic control: Systems review. *Project report; 41*.

Yang, K., Tian, C., Zhang, N., Duan, Z., and Du, H. (2019). A temporal logic programming approach to planning. *Journal of Combinatorial Optimization*.

# Appendix A

# An Appendix

```
;;created by tomas de la rosa
;;domain for painting floor tiles with two colors
(define (domain floor-tile)
(:requirements :typing :action-costs)
(:types robot tile color - object)
(:predicates
(robot-at ?r - robot ?x - tile)
(up ?x - tile ?y - tile)
(down ?x - tile ?y - tile)
(right ?x - tile ?y - tile)
(left ?x - tile ?y - tile)
(clear ?x - tile)
(painted ?x - tile ?c - color)
(robot-has ?r - robot ?c - color)
(available-color ?c - color)
(free-color ?r - robot))
(:functions (total-cost))
(:action change-color
:parameters (?r - robot ?c - color ?c2 - color)
:precondition (and (robot-has ?r ?c) (available-color ?c2))
:effect (and (not (robot-has ?r ?c)) (robot-has ?r ?c2)
(increase (total-cost) 5))
```

```
(:action paint-up
:parameters (?r - robot ?y - tile ?x - tile ?c - color)
:precondition (and (robot-has ?r ?c) (robot-at ?r ?x) (up ?y ?x) (clear ?y))
:effect (and (not (clear ?y)) (painted ?y ?c)
(increase (total-cost) 2)))
(:action paint-down
:parameters (?r - robot ?y - tile ?x - tile ?c - color)
:precondition (and (robot-has ?r ?c) (robot-at ?r ?x) (down ?y ?x) (clear ?y))
:effect (and (not (clear ?y)) (painted ?y ?c)
(increase (total-cost) 2)))
(:action up
:parameters (?r - robot ?x - tile ?y - tile)
:precondition (and (robot-at ?r ?x) (up ?y ?x) (clear ?y))
:effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
(clear ?x) (not (clear ?y))
(increase (total-cost) 3)))
(:action down
:parameters (?r - robot ?x - tile ?y - tile)
:precondition (and (robot-at ?r ?x) (down ?y ?x) (clear ?y))
:effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
(clear ?x) (not (clear ?y))
(increase (total-cost) 1)))
(:action right
:parameters (?r - robot ?x - tile ?y - tile)
:precondition (and (robot-at ?r ?x) (right ?y ?x) (clear ?y))
:effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
(clear ?x) (not (clear ?y))
(increase (total-cost) 1)))
```

```
    (:action left
    :parameters (?r - robot ?x - tile ?y - tile)
    :precondition (and (robot-at ?r ?x) (left ?y ?x) (clear ?y))
    :effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
    (clear ?x) (not (clear ?y))
    (increase (total-cost) 1))))


(define (domain parking)
(:requirements :strips :typing :action-costs)
(:types car curb)
(:predicates,(at-curb ?car - car),
(at-curb-num ?car - car ?curb - curb),
(behind-car ?car ?front-car - car),
(car-clear ?car - car),
(curb-clear ?curb - curb) )
(:functions (total-cost) - number)
(:action move-curb-to-curb
:parameters (?car - car ?curbsrc ?curbdest - curb)
:precondition (and (car-clear ?car)(curb-clear ?curbdest)(at-curb-num ?car ?curbsrc))
:effect (and
(not (curb-clear ?curbdest))
(curb-clear ?curbsrc)
(at-curb-num ?car ?curbdest)
(not (at-curb-num ?car ?curbsrc))
(increase (total-cost) 1)))
```

```
(:action move-curb-to-car
:parameters (?car - car ?curbsrc - curb ?cardest - car)
:precondition (and (car-clear ?car)(car-clear ?cardest)(at-curb-num ?car ?curbsrc)(at-curb ?cardest) )
:effect (and
(not (car-clear ?cardest))
(curb-clear ?curbsrc)
(behind-car ?car ?cardest)
(not (at-curb-num ?car ?curbsrc))
(not (at-curb ?car))
(increase (total-cost) 1)))
(:action move-car-to-curb
:parameters (?car - car ?carsrc - car ?curbdest - curb)
:precondition (and (car-clear ?car)(curb-clear ?curbdest)(behind-car ?car ?carsrc))
:effect (and
(not (curb-clear ?curbdest))
(car-clear ?carsrc)
(at-curb-num ?car ?curbdest)
(not (behind-car ?car ?carsrc))
(at-curb ?car)
(increase (total-cost) 1)))
(:action move-car-to-car
:parameters (?car - car ?carsrc - car ?cardest - car)
:precondition (and (car-clear ?car)(car-clear ?cardest)(behind-car ?car ?carsrc)(at-curb ?cardest) )
:effect (and
(not (car-clear ?cardest))
(car-clear ?carsrc)
(behind-car ?car ?cardest)
(not (behind-car ?car ?carsrc))
(increase (total-cost) 1))))
```

| Domain | p | S | | | | T | | | | Q | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | F-k | B-k | F-f5 | B-f5 | F-k | B-k | F-f5 | B-f5 | F-k | B-k | F-f5 | B-f5 |
| agricola-sat18 | 20 | 0 | 0 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| airport | 50 | 37 | 25 | 37 | 25 | 13.26 | 36.06 | 14.36 | 35.29 | 116.24 | 112.56 | 116.30 | 112.56 |
| barman14 | 20 | 20 | 0 | 20 | 0 | 15.53 | 0.00 | 13.99 | 0.00 | 179.10 | 0.00 | 179.30 | 0.00 |
| blocks world | 50 | 30 | 30 | 30 | 30 | 17.05 | 88.67 | 22.88 | 84.23 | 112.87 | 192.77 | 114.75 | 198.10 |
| caldera18 | 20 | 0 | 0 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| caldera-split-18 | 20 | 0 | 0 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| childsnack14 | 20 | 0 | **6** | 2 | **4** | 0.00 | 266.23 | 15.02 | 177.36 | 0.00 | 58.67 | 56.00 | 64.25 |
| cybersec | 30 | 15 | 10 | 0 | 10 | 1327.15 | 137.94 | 0.00 | 130.40 | 39.80 | 51.60 | 0.00 | 51.60 |
| data-network18 | 20 | 8 | 0 | 8 | 0 | 195.76 | 0.00 | 255.86 | 0.00 | 59.88 | 0.00 | 66.88 | 0.00 |
| depot | 22 | 22 | 3 | 22 | 3 | 1.76 | 85.13 | 1.75 | 76.60 | 54.23 | 33.67 | 55.18 | 35.00 |
| driverlog | 20 | 20 | 20 | 20 | 20 | 0.79 | 9.11 | 0.77 | 8.86 | 52.40 | 72.20 | 52.85 | 72.55 |
| elevators11 | 20 | 20 | 13 | 20 | 13 | 40.66 | 323.57 | 43.74 | 331.86 | 249.65 | 345.69 | 249.80 | 345.46 |
| ferry | 30 | 30 | 30 | 30 | 30 | 0.01 | 0.02 | 0.01 | 0.02 | 24.73 | 30.17 | 24.83 | 30.10 |
| floortile14 | 20 | 2 | **20** | 2 | **20** | 1.08 | 0.16 | 2.99 | 0.14 | 42.00 | 107.30 | 43.00 | 105.10 |
| ged14 | 20 | 19 | 0 | 19 | 0 | 136.83 | 0.00 | 138.19 | 0.00 | 142.47 | 0.00 | 142.47 | 0.00 |
| gripper | 20 | 20 | 20 | 20 | 20 | 0.16 | 0.30 | 0.16 | 0.28 | 91.00 | 91.00 | 91.00 | 91.00 |
| hanoi | 30 | 4 | **6** | 11 | **13** | 2.30 | 0.00 | 45.50 | 94.82 | 6.50 | 23.50 | 371.18 | 1309.15 |
| hiking14 | 20 | 3 | 2 | 5 | 5 | 117.32 | 445.63 | 149.70 | 115.08 | 90.33 | 129.00 | 61.40 | 61.40 |
| logistics | 50 | 44 | 21 | 47 | 22 | 133.62 | 90.07 | 143.39 | 81.09 | 366.31 | 372.25 | 366.43 | 370.81 |
| miconic | 50 | 50 | 50 | 50 | 50 | 0.23 | 0.28 | 0.20 | 0.27 | 61.55 | 58.89 | 61.51 | 59.11 |
| mprime | 70 | 65 | 2 | 65 | 4 | 59.10 | 6.92 | 25.95 | 63.84 | 9.17 | 10.50 | 9.55 | 18.50 |
| mystery | 60 | 38 | 14 | 38 | 15 | 48.48 | 13.06 | 38.94 | 30.04 | 8.97 | 13.07 | 9.47 | 11.60 |
| no-mprime | 35 | 32 | 1 | 32 | 1 | 54.12 | 13.13 | 27.29 | 11.02 | 9.19 | 10.00 | 9.50 | 10.00 |
| no-mystery | 30 | 19 | 7 | 19 | 8 | 59.97 | 1.32 | 43.09 | 16.78 | 9.00 | 10.43 | 9.63 | 11.50 |
| nomystery11 | 20 | 13 | 10 | 14 | 11 | 220.90 | 28.90 | 77.37 | 138.12 | 40.46 | 36.70 | 39.43 | 35.45 |
| openstacks | 30 | 27 | 26 | 27 | 26 | 38.03 | 58.77 | 41.04 | 58.26 | 127.04 | 111.04 | 126.85 | 110.92 |
| openstacks14 | 20 | 15 | 0 | 15 | 0 | 571.53 | 0.00 | 567.58 | 0.00 | 774.67 | 0.00 | 774.67 | 0.00 |
| parcprinter11 | 20 | 13 | **18** | 13 | **18** | 35.60 | 2.61 | 44.26 | 2.53 | 66.92 | 71.56 | 66.92 | 71.56 |
| parking14 | 20 | 20 | 0 | 20 | 0 | 11.42 | 0.00 | 11.37 | 0.00 | 64.95 | 0.00 | 64.95 | 0.00 |
| pathways | 30 | 24 | 5 | 24 | 6 | 6.30 | 24.59 | 6.99 | 21.09 | 129.88 | 31.40 | 129.96 | 32.00 |
| pegsol11 | 20 | 9 | 1 | 20 | 5 | 0.13 | 1.48 | 9.45 | 96.32 | 28.44 | 29.00 | 29.55 | 36.60 |
| pipesworld06 | 50 | 30 | 6 | 34 | 6 | 196.56 | 19.65 | 258.73 | 20.86 | 37.47 | 47.50 | 40.35 | 58.50 |
| pipesworld-notankage | 50 | 50 | 15 | 50 | 15 | 30.74 | 7.44 | 20.68 | 65.77 | 47.30 | 58.33 | 47.08 | 68.53 |
| pipesworld-tankage | 50 | 30 | 6 | 34 | 6 | 228.80 | 17.58 | 262.25 | 18.60 | 37.47 | 47.50 | 40.35 | 58.50 |
| rovers | 20 | 20 | 19 | 20 | 19 | 0.32 | 1.26 | 0.29 | 0.82 | 35.70 | 43.26 | 35.30 | 44.42 |
| scanalyzer11 | 20 | 18 | **20** | 18 | **20** | 5.39 | 4.01 | 5.62 | 5.25 | 35.61 | 44.05 | 34.94 | 45.15 |
| settlers18 | 20 | 0 | 0 | 0 | 0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| snake18 | 20 | 11 | 0 | 10 | 0 | 385.34 | 0.00 | 336.88 | 0.00 | 79.00 | 0.00 | 81.40 | 0.00 |
| sokoban11 | 20 | 5 | 1 | 15 | 1 | 8.01 | 0.05 | 68.68 | 0.05 | 190.00 | 429.00 | 214.47 | 429.00 |
| spider18 | 20 | 13 | 0 | 13 | 0 | 185.96 | 0.00 | 184.19 | 0.00 | 240.31 | 0.00 | 240.31 | 0.00 |
| storage | 30 | 29 | 14 | 29 | 15 | 18.73 | 1.49 | 20.25 | 5.90 | 41.62 | 21.86 | 41.45 | 23.20 |
| termes18 | 20 | 1 | 2 | 10 | 8 | 3.47 | 5.06 | 241.08 | 130.44 | 245.00 | 433.00 | 622.20 | 515.25 |
| thoughtful14 | 20 | 20 | 5 | 20 | 5 | 31.36 | 44.42 | 38.09 | 27.88 | 92.75 | 43.80 | 93.70 | 44.60 |
| tpp | 30 | 30 | 10 | 30 | 12 | 23.58 | 25.23 | 24.18 | 41.56 | 127.80 | 41.90 | 127.57 | 49.83 |
| transport14 | 20 | 7 | 0 | 6 | 0 | 278.16 | 0.00 | 144.73 | 0.00 | 286.00 | 0.00 | 243.00 | 0.00 |
| trucks | 30 | 7 | 14 | 9 | 14 | 9.45 | 70.16 | 45.49 | 35.50 | 27.57 | 35.14 | 29.00 | 34.93 |
| tyreworld | 30 | 30 | 12 | 30 | 12 | 10.23 | 33.38 | 10.66 | 30.13 | 181.10 | 144.33 | 181.03 | 141.50 |
| visitall14 | 20 | 19 | 17 | 19 | 17 | 26.72 | 596.73 | 33.57 | 569.97 | 2843.00 | 2655.41 | 2843.00 | 2655.41 |
| woodworking11 | 20 | 20 | 11 | 20 | 11 | 22.93 | 3.52 | 24.00 | 3.25 | 69.10 | 79.18 | 69.10 | 79.18 |
| zenotravel | 20 | 20 | 20 | 20 | 20 | 1.85 | 7.10 | 2.01 | 7.84 | 43.35 | 70.50 | 45.15 | 72.15 |
| summary | 1417 | 979 | 512 | 1017 | 540 | 101.70 | 66.78 | 76.96 | 66.79 | 169.29 | 163.10 | 185.62 | 199.07 |

TABLE A.1: Backward BFWS(*f5*) vs. forward BFWS(*f5*); backward *k*-BFWS vs. forward *k*-BFWS where k=2. F-k is the forward *k*-BFWS; B-k is the backward *k*-BFWS, F-f5 is the forward BFWS(*f5*); B-f5 is the backward BFWS(*f5*). P is the number of problems in each domain. S is the number of solved problems. $Q$ is average plan length and $T$ is average time in seconds.

| Domain | p | Solved | |
|---|---|---|---|
| | | Original | Dual |
| agricola18 | 20 | 0 | 0 |
| airport | 50 | 49 | 0 |
| barman14 | 20 | 0 | 0 |
| blocks world | 50 | 37 | 0 |
| caldera18 | 20 | 10 | 10 |
| caldera-split-18 | 20 | 0 | 0 |
| childsnack14 | 20 | 0 | 0 |
| cybersec | 30 | 0 | 0 |
| data-network18 | 20 | 3 | 0 |
| depot | 22 | 18 | 0 |
| driverlog | 20 | 7 | 0 |
| elevators11 | 20 | 18 | 0 |
| ferry | 30 | 30 | 0 |
| floortile14 | 20 | 0 | 0 |
| ged14 | 20 | 0 | 0 |
| gripper | 20 | 20 | 0 |
| hanoi | 30 | 3 | 1 |
| hiking14 | 20 | 0 | 0 |
| logistics | 50 | 26 | 0 |
| miconic | 50 | 50 | 5 |
| mprime | 70 | 60 | 0 |
| mystery | 60 | 33 | 0 |
| no-mprime | 35 | 30 | 0 |
| no-mystery | 30 | 16 | 0 |
| nomystery11 | 20 | 1 | 0 |
| openstacks | 30 | 0 | 0 |
| openstacks14 | 20 | 0 | 0 |
| parcprinter11 | 20 | 20 | 0 |
| parking14 | 20 | 20 | 0 |
| pathways | 30 | 15 | 0 |
| pegsol11 | 20 | 0 | 0 |
| pipesworld06 | 50 | 24 | 0 |
| pipesworld-notankage | 50 | 19 | 0 |
| pipesworld-tankage | 50 | 24 | 0 |
| rovers | 20 | 20 | 1 |
| scanalyzer11 | 20 | 17 | 0 |
| settlers18 | 20 | 0 | 0 |
| snake18 | 20 | 0 | 0 |
| sokoban11 | 20 | 1 | 0 |
| spider18 | 20 | 0 | 0 |
| storage | 30 | 23 | 1 |
| termes18 | 20 | 0 | 0 |
| thoughtful14 | 20 | 15 | 0 |
| tpp | 30 | 10 | 4 |
| transport14 | 20 | 14 | 0 |
| trucks | 30 | 3 | 0 |
| tyreworld | 30 | 26 | 0 |
| visitall14 | 20 | 20 | 0 |
| woodworking11 | 20 | 19 | 0 |
| zenotravel | 20 | 19 | 0 |
| summary | 1417 | 720 | 22 |

TABLE A.2: *SIW* solved original problems vs. dual problems. P is the number of problems in each domain. Original is the solved original problems; Dual is the solved dual problems