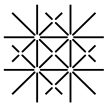


University of Basel

# NBS applied to Planning

Master Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch

Examiner: Dr. Gabriele Röger
Supervisor: Cedric Geissmann

Marvin Buff
marvin.buff@unibas.ch
2014-054-191

05/02/2019

# Acknowledgments

I would like to express my thanks to Cedric Geissmann, which not only supervised me but also provided his help in regards to Fast-Downward and writing up this thesis. I owe my thanks to Dr. Gabriele Röger who in place of Prof. Dr. Malte Helmert allowed me to write this thesis in their research group. And special thanks goes to Severin Wyss and Jonathan Aellen who proofread this thesis.

# Abstract

Heuristic forward search is the state-of-the-art approach to solve classical planning problems. On the other hand, bidirectional heuristic search has a lot of potential but was never able to deliver on those expectations in practice. Only recently the *near-optimal bidirectional search algorithm* (NBS) was introduces by Chen et al. [2017] and as the name suggests, NBS expands nearly the optimal number of states to solve any search problem. This is a novel achievement and makes the NBS algorithm a very promising and efficient algorithm in search. With this premise in mind, we raise the question of how applicable NBS is to planning. In this thesis, we inquire this very question by implementing NBS in the state-of-the-art planner Fast-Downward and analyse its performance on the benchmark of the latest international planning competition. We additionally implement *fractional meet-in-the-middle* and COMPUTEWVC [Shaham et al., 2017] to analyse NBS' performance more thoroughly in regards to the structure of the problem task.

The conducted experiments show that NBS can successfully be applied to planning as it was able to consistently outperform A*. Especially good results were achieved on the domains: *blocks*, *driverlog*, *floortile-opt11-strips*, *get-opt14-strips*, *logistics00*, and *termes-opt18-strips*. Analysing these results, we deduce that the efficiency of forward and backward search depends heavily upon the underlying implicit structure of the transition system which is induced by the problem task. This suggests that bidirectional search is inherently more suited for certain problems. Furthermore, we find that this aptitude for a certain search direction correlates with the domain, thereby providing a powerful analytic tool to a priori derive the effectiveness of certain search approaches.

In conclusion, even without intricate improvements the NBS algorithm is able to compete with A*. It therefore has further potential for future research. Additionally, the underlying transition system of a problem instance is shown to be an important factor which influences the efficiency of certain search approaches. This knowledge could be valuable for devising portfolio planners.

# Table of Contents

# 1 Introduction

Classical planning is about thinking before acting. The objective in this field is to contrive algorithms which achieve to deduce the best course of actions for a specific problem description. One of the first devised algorithms is the famous *Dijkstra algorithm* [Dijkstra, 1959], which builds a *decision tree* with the initial state as root. The result is a construct in which traversing the tree reflects a particular course of action. The algorithm finishes once the *optimal solution* is found. But with the ever growing problem sizes and complexities, it is not always feasible to search for the optimal solution in such an uninformed way. A logical next step is to let the search be guided by problem specific information in order to facilitate the progress. One of the resulting *informed algorithm*, which explores the solution space guided by a *heuristic function*, is called A*. But even for A*, the number of combinatorial permutations growths exponentially with the number of possibilities. To alleviate this problem, the fundamental idea of *bidirectional search* is to conduct two partial searches instead of a single one. The first search progresses in forward direction and the second one in backward direction, e.g. the search starts at the goal state and advances in direction of the initial state. Thereby, both searches only have to advance until they meet each other, reducing the effective search depth by a potential factor of 2. Thus, as the effort growths exponentially, the complexity is reduced from $2^n$ to a potential $2 \cdot 2^{\frac{n}{2}}$.

The concept of bidirectional search is nearly as old as search itself. But for a long time the advances in the field were stagnant. Non-trivial difficulties in bidirectional search are to first advance the searches towards each other in order for them to intersect and second, prove the optimality of a found solution. And although bidirectional search shows promise in theory, the performance in practice was not convincing for a long time. Only recently, Barker and Korf [2015] sparked new interest by analysing the properties of a theoretical bidirectional algorithm. Which was followed up with the practical implementation of the algorithm in question by Holte et al. [2016]. From there, different research papers led to the definition of *must-expand pairs*. These define state pairs from which either the first has to be expanded in forward direction, or the second in backward direction. These conditions enabled Chen et al. [2017] to devise the *near-optimal bidirectional search algorithm* (`NBS`). The authors achieved promising results with `NBS`, raising the question of its usability in classical planning. In this thesis, we want to inquire this exact question by implementing `NBS` in Fast-Downward and

evaluating its advantages and drawbacks with thorough experiments.

To achieve these goals, we implemented three different algorithms. First, the aforementioned promising bidirectional search algorithm NBS. Second, *fractional meet-in-the-middle* (fMM), a bidirectional search algorithm where both search frontiers meet at a specified point. We use fMM to explore the effectiveness of either search direction depending on the specific problem and determine the optimal combination of forward and backward search with the third algorithm computeWVC, which provides us with the optimal meeting point for fMM.

By running each algorithm over the optimal benchmark set of the latest international planning competition, we gathered the following insights. Using NBS to solve planning problems can lead to additional complexities which are neither present in unidirectional planning, nor in bidirectional search. Especially, the absence of a fully defined goal state has a negative effect on the search efficiency. However, the experiments show that NBS is a competitive alternative to A* as NBS outperforms it on many problem instances and sometimes even over whole domains. Further, by using fMM and computeWVC, we were able to show that the single most defining factor of the performance of NBS is the innate structure of the problem instance, which can enable it to surpass A* or significantly impair its performance. In conclusion, NBS is a compelling search algorithm which can successfully and efficiently be used in planning. But its performance is heavily influenced by the specific problem instance, which limits its usability for problems with unfavourable structure.

The thesis is structured in five parts. First, we start by introducing background information and terminology on planning and bidirectional search. Followed up by an overview of contemporary research in the field of bidirectional search with the goal to acquaint the reader with the necessary concepts and preliminary knowledge. In a third step, we describe in detail how we implemented the NBS algorithm and the theoretical background which was considered. Afterwards, we illustrate the setup and specify all conducted experiments, which includes a thorough evaluation and discussion. At last, we conclude the thesis by presenting the important findings and insights gathered throughout.

# 2
# Background

In this section, we provide a concise introduction of all necessary concepts and theories used within this thesis. We start with an overview of past research, going over to in-depth explanations of important findings and key aspects in the field of bidirectional search, and finishing up with a detailed analysis of the paper by Chen et al. [2017] about the NBS algorithm, which directly inspired this work. Thereby, we also introduce the corresponding terminology and background to facilitate a deeper understanding. Moreover, we discuss the relation between bidirectional search and classical planning, which will provide the foundation upon we build our own theories and hypotheses.

## 2.1 History of Bidirectional Search

The history of searching had its beginning with the famous Dijkstra's algorithm [Dijkstra, 1959]. It counts as a canonical best-first search, which uses the $g$-value of a search node as a tie-breaking criterion to decide whether to expand it. Bidirectional search was later introduced by Nicholson [1966], which adapted Dijkstra's algorithm to conduct the search from either direction. The biggest break through in searching was the utilization of heuristic functions, which builds the basis of the A* algorithm [Hart et al., 1968]. The usage of heuristic functions were quickly adapted to bidirectional search by Pohl [1969]. He showed that unidirectional heuristic search (Uni-HS) seems to outperform bidirectional heuristic search (Bi-HS) in practice. He further introduced the Bi-HS algorithm BHPA [Pohl, 1971]. However, the observed inefficiency of Bi-HS in practice lead to a stagnation in the development of this field.

Over the years, a few explanations for the bad performance of bidirectional search arose. Nilsson [1980] suggested that the search frontiers miss each other. But this turned out to be wrong, instead Kwa theorized that the frontiers cross each other without terminating, as a result he designed the BS* algorithm which focuses on clipping to prevent the frontiers from crossing [Kwa, 1989]. BS* is an improved Bi-HS algorithm, yet it was not able to outperform A*. An explanation for this was later provided by Kaindl and Kainz [1997], which suggested that a lot of resources are spent proving optimality of a solution found early on.

Only a few years ago, bidirectional search received new attention. Barker and Korf anal-

ysed the advantages between bidirectional brute force search (Bi-BS), Bi-HS and Uni-HS [Barker and Korf, 2015]. They reached the conclusion that in most cases front-to-end Bi-HS algorithm will be dominated by either Uni-HS or Bi-BS. But they additionally proved that there exist some pathological cases where Bi-HS is the dominant strategy. Barker and Korf relied on a few assumptions to make those claims. Most importantly, they defined Bi-HS as an algorithm which would not expand nodes further than half of the optimal cost, thus meet in the middle. This discovery inspired the development of the `MM` algorithm [Holte et al., 2016] which was the first bidirectional search algorithm to satisfy the imposed constraint, or specifically, whose search frontiers are guaranteed to meet in the middle.

We discuss contemporary research in the subsequent sections with more detail. However, to provide a holistic overview at this point, they are mentioned here as well. First, Eckerle et al. [2017], inspired by Dechter and Pearl [1985], defined sufficient conditions for node expansion[1] in bidirectional search. They used those conditions to derive a lower bound on the number of necessary expansions. The described lower bound is called |VC| and is equivalent with the number of nodes in the minimal vertex cover of the must-expand graph, which is purposefully designed to have this property. Upon those findings Chen et al. [2017] built the `NBS` algorithm which expands at most twice as many nodes, and furthermore, they showed that no better algorithms of this class can exist. Following up, Shaham et al. [2017] designed an algorithm, fractional `MM` (`fMM`), which only expands |VC| nodes, thus is optimally efficient. However, the use of `fMM` is restricted by its optimal meeting point $p^*$ which is not known a priori. Consequently, `fMM` is primarily of theoretical interest as computing the fraction $p^*$ is equally difficult as solving the search problem itself.

## 2.2 Terminology and Background

To discuss bidirectional search applied to planning in depth, we present an exhaustive introduction of the essential background and terminology regarding bidirectional search and planning.

We start by defining *classical planning* which aims to solve *planning problems* presented as *planning tasks*. We discuss how *heuristics* and *search* are used to solve *planning problems*. And at last, we present a *generic search algorithm* and the conditions which have to be fulfilled to be categorized as *DXBB* algorithm.

### 2.2.1 Classical Planning

> "Planning is the reasoning side of acting. It is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes." [Ghallab et al., 2004]

Classical planning is about determining the course of action which leads to the best possible outcome. The used framework consists of three different parts: the input, the algorithm, and

[1] They defined conditions which are sufficient to determine whether a search node must be expanded to prove optimality of a solution plan.

the output. A *problem instance* functions as the input and is described by the corresponding *planning task*. A *planning algorithm* solves the problem instance and computes the solution *plan* as output.

We distinguish between satisficing planning, where the objective is to find any solution at all (e.g. plan), and optimal planning, where many solutions may be easily deducible, but we are only interested in determining the optimal solution.

**Definition 1** (Planning Task)
*We are using an adaptation of the finite domain $SAS^+$ formalism introduced by Bäckström and Nebel [1995] to describe a planning task formally. A planning task is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where*

- *$\mathcal{V}$ is a finite set of **state variables**, each $v \in \mathcal{V}$ has an associated finite domain $D(v)$, which specifies valid variable assignments.*
- *A **partial state** $\hat{s}$ is a mapping of state variables $v \in \mathcal{V}$ to a value consistent with their defined domain. These assignments are written as $s[v] \in D(v)$. $var(s) \in \mathcal{V}$ is a finite set which lists all variables that have an assignment in the specified partial state.*
- *A partial state $s$ is called a **state**, if all variables $v \in \mathcal{V}$ have a valid assignment given by $s$, hence $var(s) = \mathcal{V}$.*
- *$\mathcal{A}$ is a finite set of actions, each $a \in \mathcal{A}$ is a triple $(pre(a), eff(a), cost(a))$, where*
  - *$pre(a)$ is a partial state defining the **preconditions**.*
  - *$eff(a)$ is a partial state defining the **effects**.*
  - *$cost(a) \in \mathbb{R}_0^+$ is the associated **cost**.*
- *$\mathcal{I}$ is a state defining the **initial state**.*
- *$\mathcal{G}$ is a partial state defining the **goal** conditions.*

A partial state $\hat{s}_1$ is a subset of another partial state $\hat{s}_2$ when $\hat{s}_1[v] = \hat{s}_2[v]$ for all $v \in var(\hat{s}_1)$ holds and is written as $\hat{s}_1 \subseteq \hat{s}_2$. A partial state *complies* with another partial state if one is a subset of the other. Action $a$ is *applicable* in state $s$ under the condition that $pre(a) \subseteq s$. Applying action $a$ on state $s$ is denoted by $s[\![a]\!]$ and modifies the state $s$ to comply with the partial state $eff(a)$. In classical planning we search for a sequence of actions, which when applied on the initial state $\mathcal{I}$ yield a resulting state $s$ where $\mathcal{G} \subseteq s$ holds. Such a sequence is called a *plan*: $\pi = (a^1, a^2, \ldots, a^n)$. The cost of a plan $\pi$ is given by the summed cost of all included actions: $cost(\pi) = \sum_{i=1}^{n} cost(a^i)$. If no plan exists then the planning task is *unsolvable*, otherwise, there are plans and thus at least one of those plans has the minimal cost compared to the others and is called the *optimal plan*. A commonly used method to determine the optimal plan from a planning task, is to first build the implicitly defined *transition system* and then find the plan by *heuristic search*.

**Definition 2** (Transition System)
*A **transition system** induced by a planning task $\Pi$ is 6-tuple $\mathcal{T}(\Pi) = \langle S, T, L, cost, s_0, s_G \rangle$, where*

- $S$ *is a finite set of* ***states*** *derived from* $\Pi$*, called the* ***state space****.*
- $T : S \times \mathcal{A} \mapsto S$ *is a finite set of* ***transitions*** *induced by the actions and states in* $\Pi$*.*
- $L : T \mapsto \mathcal{A}$ *is a* ***label*** *function defining from which action a transition is induced.*
- $cost : T \mapsto \mathbb{R}_0^+$ *is the* ***cost*** *function which maps the transition to the cost of the applied action.*
- $s_0 = \mathcal{I} \in S$ *is the* ***initial state****.*
- $s_G \subseteq S$ *is a finite set of* ***goal states*** *where each* $s \in S$ *which complies with* $\mathcal{G}$ *is a element of* $s_G$*.*

In practice, the full transition system is not required, as only reachable states need to be considered. The required transition system of a planning task is created by recursively expanding states, starting with the initial state, until the goal is reached and therefore only the required part of the system is mapped. *Expanding* a state $s$ is defined as applying all actions $a \in \alpha \subseteq \mathcal{A}$, where $\alpha$ is the set of applicable actions in $s$, to *generate* all *successors*, $s_a = s[\![a]\!]$ for all $a \in \alpha$. We denote the expansion of one state or a set of states as $s_{n+1} = s_n[\![\mathcal{A}]\!]$, or alternatively $s_{n+1} = expand(s_n)$, where $s_{n+1}$ is the set of possible successor states. To prevent cyclic expansion, states are generated with an associated $g$-value, which represents the cost of the shortest plan to reach a state $s$, denoted by $g(s)$. The $g$-value of a state $s$ is given by the $g$-value of its predecessor added to the cost of the applied action. Note that although the $g$-value represents the cost of the shortest plan, it is only the cheapest path found so far and has to be updated accordingly if a cheaper plan is found. The real cost of reaching a state $s$ is denoted by $cost(s)$. Moreover, $cost(s_a, s_b)$ is the cost of reaching state $s_b$ starting from state $s_a$. Usually, we insure that $g(s)$ is equal to $cost(s)$ by prioritizing the expansion of states with lower $g$-value. The reasoning is that as action costs can only be positive, it is impossible to reach a previous state with a new $g$-value that is lower than before. The function controlling the order in which states are expanded is called the *priority function*. Most commonly, the priority function not only considers the $g$-value, but the heuristic value of the state as well.

### 2.2.2 Heuristic Functions

In planning *heuristic functions* (simply referenced as heuristics) are used to guide the exploration of the transition system. Heuristics are functions

$$h : S \mapsto \mathbb{N}_0 \cup \{\infty\},$$

which map each state $s \in S$ to a non-negative number or infinity. The mapped value, called $h$-value, represents the estimated *goal distance*, or infinity if the goal is not reachable from the given state. The perfect heuristic $h^*$ maps each state to its minimal goal distance. *Informed algorithms* utilize heuristics to guide them. For instance, by prioritizing the expansion of states with low $f$*-value*, which is the sum of the $h$-value added to the $g$-value. There are a few important properties for heuristics. A heuristic $h$ is

- *safe* if $h(s) = \infty$ for states $s$ from which the goal is unreachable,
- *goal-aware* if $h(s) = 0 \mid \forall s \in \mathcal{G}$,
- *admissible* if $h(s) \leq h^*(s) \mid \forall s \in S$,
- and *consistent* if $h(s_n) \leq cost(s_n, s_{n+1}) + h(s_{n+1})$.

### 2.2.3 Search in Planning

Search can be used as a tool to solve optimal planning problems. The intuition behind search is to find the shortest path from the initial state to any goal state, by interpreting the transition system as graph where the nodes represent states and the edges transitions weighted by the corresponding cost. Based on this fundamental idea there exist different implementations which can be classified by four criteria:

- **Search Direction:** Progression, Regression, or Bidirectional.
- **Search Space Representation:** Explicit-states or Symbolic.
- **Search Algorithm:** Uninformed Search, Local Heuristic Search, Systematic Heuristic Search.
- **Search Control:** Heuristic Search, Pruning.

For example, the NBS algorithm is a bidirectional heuristic search algorithm with an explicit-state representation and no pruning method. Later on, we go into more detail and discuss the implication of this classification in more depth, but first the necessary terminology must be defined. The presented notation in the following paragraph is inspired by the notation of Eckerle et al. [2017] as we use it to explain the topics they covered in their work.

To discuss bidirectional search, the introduced notations must be refined to incorporate the backward component. In general, we distinguish forward direction from backward direction by a capital „F“, or respectively by a „B“ as index. For simplicity, omitting the index implicitly refers to the forward direction. If both directions are meant simultaneously, we denote it with a „D“ as index. By this notion, expanding a state $s_{n+1}$ in backward direction means to create the predecessors $s_n$ and is denoted by $s_n = expand_B(s_{n+1})$. A backward expanded state $s_B$ has an associated $g_B$-value and $h_B$-value, where $h_B$ is a heuristic which estimates the distance to the initial state. Depending on the heuristics, problem instances can be determined to belong to the set of instances $I_{AD}$ or $I_{CON}$. Derived from Dechter and Pearl [1985] we denote $I_{AD}$ to refer to the set of solvable problem instances with *bi-admissible* heuristics (meaning forward as well as backward heuristic are admissible). Similarly, $I_{CON}$ is the set of solvable problem instance with *bi-consistent* heuristics. It should be noted that the heuristics used for the different directions might not be the same.

At last, we introduce *paths* to simplify the theories presented in the next chapter. A forward path $U = (u_1, \ldots, u_n)$ is a sequence of states where each consecutive state pair denotes that there exists an action $a$ so that $u_{i+1} = u_i[\![a]\!]$. In the same manner, $V = (v_1, \ldots, v_n)$ is a backward path so that $v_i = v_{i+1}[\![a]\!]$. The reversion of a path $U$ is indicated

as $U^{-1}$ and changes the path from forward to backward direction, or vice versa. Moreover, the number of states in path $U$ is denoted as $|U|$ and the summed cost of all implied actions as $cost(U)$. A path $U$ is optimal, if there exists no path with lower cost from $U_1$ to $U_{|U|}$. The cost of the optimal path between two states is denoted by its *distance* $d(U_1, U_{|U|})$. Hence, $C^* = d(s_0, s_G)$ where $s_0$ and $s_G$ are defined by the induced transition system $\mathcal{T}(\Pi)$. Using bidirectional search, we pursue to find a *path pair* (U,V), which consists of a forward path $U$ and a backward path $V$, where $U_1 = s_1$, $V_1 = s_G$, and $U_{|U|} = V_{|V|}$. The *solution path* of such a path pair can be written as $UV^{-1}$. Finally, to be able to use paths intuitively, we define that a path inherits all state specific properties of the tailing state. E.g. the $g$-value of a path $U$ is defined by the $g$-value of $U_{|U|}$.

### 2.2.4 Bidirectional Search in Planning

With the notation for bidirectional search introduced, we can start to describe bidirectional search in detail. In practice, search algorithms can be used in different ways for planning. But for this thesis, we limit the scope to only include DXBB search algorithm.

**Definition 3** (DXBB algorithm)
*Following the definition by Eckerle et al. [2017], we define the class of deterministic, expansion-based, black box (DXBB) algorithm. They are deterministic as the search behaviour is reproducible. Expansion-based, insofar as their general structure revolves around expanding states. And, they only have a black box view regarding the functions* StoppingCondition, Choose, *and* Solution.

**Algorithm 1** DXBB algorithm in planning

**Input**: Planning task $\Pi$ and heuristic $h$
**Output**: A least-cost path from $s_0$ to $s_G$
1: $Open_F = s_0$
2: $Open_B = s_G$
3: $Closed_F = \{\}$
4: $Closed_B = \{\}$
5: **while** StoppingCondition not fulfilled **do**
6: $Dir,\ State =$ Choose$(Open_F, Open_B, h)$ ▷ $Dir$ can assume $F$ or $B$.
7: Add $State$ to $Closed_{Dir}$
8: $Successors = expand_{Dir}(State)$
9: Add $Successors$ to $Open_{Dir}$
10: **end while**
11: **return** Solution

In Algorithm 1, we present how a generic DXBB algorithm can be applied in planning. All possible algorithms which are classified as DXBB can only differ in the implementation of the StoppingCondition, Solution, and Choose function. To understand the impact those functions possess, we discuss in detail how the depicted algorithm operates.

The input is a problem instance and a forward and backward heuristic. Over the course of the algorithm, four lists are maintained. For each direction an *open* and a *closed* list are created. The open lists include states which are not yet expanded, but were generated by expanding other states, or in the beginning, the initial and goal state. States are moved

from the open to the closed list when they are expanded, which can be seen in line 6 and 7 of the algorithm. The core part of the algorithm is the while loop enveloping line 5 to 8, which successively expands states until the stopping condition is met. The important functions in the algorithm are the following. First, the StoppingCondition tries to determine whether a solution can be found, or if the found solution is optimal. Second, the Choose function controls the exploration of the state space by selecting which state is expanded in which direction. Hence, a Choose function which would always prioritize forward expanding would result in an unidirectional search. Usually, the Choose function determines the next state with the help of some *priority functions*. We will discuss different priority functions and the algorithms they are used in later on. Last, the Solution function extracts the correct solution plan from the expanded states unless the problem couldn't be solved, in which case *unsolvable* is returned.

# 3
# Related Work

In this section, we discuss contemporary research in the field of bidirectional search, with a focus on findings and novel theories preceding the NBS algorithm. First, we give short descriptions for three relevant algorithms: MM, fMM, and SymBA*. Second, we discuss key bidirectional search insights from the last few years. Finally, we introduce the NBS algorithm. The topics covered in this section are presented especially thorough as they are essential for the discussion in Chapter 4 and 5.

## 3.1 Bidirectional Search

Bidirectional search algorithms can be divided into two classes depending on the type of heuristics used: front-to-front or front-to-end heuristics. Front-to-front heuristics are unique to bidirectional search as they estimate the distance between the search frontiers. A few examples for bidirectional search algorithm using front-to-front heuristics are BHFFA2 [De Champeaux, 1983], SFBDS [Felner et al., 2010], and BIDA* [Manzini, 1995]. Contrary to those, the more common front-to-end heuristics[2] are used by MM [Holte et al., 2016], BHPA [Pohl, 1971], BS* [Kwa, 1989], and most importantly NBS [Chen et al., 2017]. We limit the discussion in this section to the most recent front-to-end search algorithms, as NBS classifies as one and as they are the most influential.

### 3.1.1 The "Meet in the Middle" Algorithm (MM)

Barker and Korf [2015] analysed the node expansion of unidirectional heuristic search (Uni-HS), bidirectional heuristic search (Bi-HS) and bidirectional brute force search (Bi-BS). To limit the scope of the analysis they made the assumption that bidirectional search never expands states whose $g$-value exceeds half $C^*$. Such an algorithm did not exist at that time, thus Holte et al. [2016] were inspired to design an algorithm which complies to this constraint, hence it inherits the defined properties.

[2] The usually used heuristics are categorized as front-to-end heuristics as they estimate the distance from a state to the end. However, in bidirectional search there are additional options.

**The MM-algorithm** The MM-algorithm only expands nodes that have a $g$-value smaller than half $C^*$. This is achieved by a CHOOSE function which selects the next state to expand in either direction with the following priority functions:

$$pr_F(u) = \max(g_F(u) + h_F(u), 2 \cdot g_F(u)),$$
$$pr_B(v) = \max(g_B(v) + h_B(u), 2 \cdot g_B(v)),$$

where $u$ is a forward expanded state and $v$ a backward expanded state. The state with the lowest value from either priority function is expanded each iteration. This lowest value is denoted by $C$. The first condition of the priority function prioritizes the expansion of states which are most promising. The second condition enforces that no state is expanded whose $g$-value exceeds half $C^*$, this is achieved by doubling the $g$-value. An image illustrating the resulting search space is shown in Figure 3.1. Equally important as the priority functions is the STOPPINGCONDITION:

$$U \leq \max(C, fmin_F, fmin_B, gmin_F + gmin_B),$$

where $U$ is the cost of the cheapest solution found so far, $fmin_F$ is the minimal $f$-value which is present in the forward open list, similarly $fmin_B$ is the minimal $f$-value in the backward open list, and $gmin_F$ as well as $gmin_B$ are the lowest $g$-values in the open list in the respective direction. Each of the parameter on the right hand side of the equation are a lower bound on the cost of any solution that could be found if the search proceeds. Thus, if $U$ is smaller than or equal to the right hand side, the search stops as no better solution can be found any more, e.g. the optimal solution is found, hence $U = C^*$.[3]

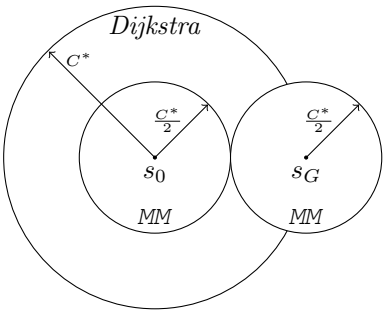



Figure 3.1: High-level illustration of the search space explored by Dijkstra or MM algorithm. [Sturtevant and Felner, 2018]

**Improved Version of MM** The MM$\epsilon$-algorithm introduced by Sharon et al. [2016] is a direct follow up to the MM algorithm [Holte et al., 2016], with a slight alteration of the priority functions. Instead of choosing the next node which has to be expanded solely on their $f$-value and $g$-value, they add the minimal action cost $\epsilon$ to the $g$-value. Thereby taking into consideration that a node with only expensive outgoing actions can be expanded after other nodes with higher $g$-value but cheaper outgoing actions.

[3] They also introduced $MM_0$, which is the brute-force version of MM. Both algorithm work exactly the same, however, the heuristic used with $MM_0$ always return 0.

Sharon et al. further made a proof sketch of MM$\epsilon$'s correctness and showed the performance of MM$\epsilon$ compared to MM and A*. Although there are conditions when MM$\epsilon$ expands more nodes than MM, in praxis they did not occur. MM$\epsilon$ outperformed MM up to a factor of 4. Compared with those algorithms, A* expands significantly more nodes, depending on how accurate the heuristic is.

### 3.1.2 Fractional Meet in the Middle (fMM)

Inspired from the MM-algorithm, Shaham et al. [2017] formalized a generalized version where the meeting point of forward and backward search is determined by the argument $p$. More precisely, in MM the two searches meet at the cost of $0.5 \cdot C^*$, whereas in fMM the forward search is expanded until the cost reaches $p \cdot C^*$, and the backward search until $(1-p) \cdot C^*$. This difference is reflected in the modified priority functions for fMM:

$$pr_F(u) = \max(g_F(u) + h_F(u), g_F(u)/p),$$
$$pr_B(u) = \max(g_B(u) + h_B(u), g_B(u)/(1-p)),$$

where $u$ is a forward expanded state and $v$ a backward expanded state. It is evident that fMM with $p = 0.5$ is equivalent to MM. Shaham et al. further deduced that fMM is not just a generalization of MM but also of A* and reverse A*, depending on the input argument $p$ of fMM. However, only when considering the expansion previous to the last layer. fMM is especially interesting, because there is a fraction $p^*$ for a which fMM is optimally efficient, meaning it exclusively expands the minimal number of states necessary. Although this would be a very competitive algorithm, $p^*$ is only computable a posteriori and is different for each problem instance. It can be determined by solving the following formula: $p^* = \underline{i}/C^*$, where $\underline{i}$ can be computed from the minimal vertex cover. However, $C^*$ is only known after the problem is already solved. Thus, fMM can only be used as an analysis tool. Furthermore, $p^*$ can give valuable insight on why a specific search direction is more efficient. Shaham et al. [2017] showed that there are domains where $p^*$ is close to 0, close to 1, or somewhere around the middle. With this calculation, one can explain why A*, reverse A*, or MM is more efficient than the other ones on the respective domains.

For our work, fMM is especially interesting as it gives us an estimation on how bidirectional search is expected to perform on a problem instance. Therefore, we can differentiate whether the reason for the performance of NBS on a specific instance is caused by the instance itself or the algorithm.

### 3.1.3 SymBA*

A last algorithm we want to discuss is the symbolic uniform-cost bidirectional heuristic search algorihm SymBA* introduced by Torralba et al. [2016]. This planning algorithm won the optimal-track award of the International Planning Competition 2014 [Vallati et al., 2015]. Thereby showing that at least symbolic bidirectional search has potential in planning. SymBA* is an intricate planning algorithm combining bidirectional search with abstraction heuristic, perimeter abstractions, and symbolic search. It starts with a uniform-cost bidirectional symbolic search, and expands states until computing the next iteration is deemed

infeasible. To make the search feasible again, a new bidirectional search is started in the abstract space around the current search frontiers. This improves the heuristic estimate and thus makes the primary search feasible again. The symbolic search is then continued with the guidance of the computed heuristic. This procedure is repeated until the search terminates. The idea behind the algorithm is that heuristic only sometimes improve bidirectional search, therefore, it only computes a heuristic estimate, when the search is infeasible otherwise.

## 3.2 Sufficient Conditions for State Expansion

An important question when analysing algorithms is whether there exists an improved version, given the same constraints. For A* this question was answered by Dechter and Pearl [1985]. They proved that there cannot exist an algorithm which outperforms A*, given problems of the $I_{CON}$ domain. They reasoned that A* expands all states with $f_F < C^*$, and that at least all of of those states have to be expanded by every optimal unidirectional algorithm, hence A* is optimally efficient.

The states with $f_F < C^*$ are determined by the sufficient conditions for state expansion. However, for bidirectional search, such conditions cannot exist, as every state can either be expanded in forward or backward direction. To get a similar notion of sufficient conditions, Eckerle et al. [2017] coined the new term *must-expand pair*.

**Definition 4** (Must-Expand Pair)
*A state pair $(u, v)$ is a must-expand pair under the following conditions:*

$$(1)\ f_F(u) < C^*,$$
$$(2)\ f_B(v) < C^*,$$
$$(3)\ cost(u) + cost(v) < C^*.$$

*Provided that $U$ is a forward path from $s_0$ to $u$ and $V$ is a backward path from $s_G$ to $v$, all solution paths including them can be written as $UTV^{-1}$, where $Z$ is a forward path from $u$ to $v$. Eckerle et al. [2017] proved that every DXBB-algorithm must expand at least one state of every must-expand pair to ensure that no superior solution may exist. We refer to a must-expand pair with at least one expanded state, as being expanded. E.g. pair $(u, v)$ is expanded if state $u$ or state $v$ is expanded.*

The three conditions for must-expand pairs characterize a lower bound on all possible solution paths of the form $UZV^{-1}$. Formula (1) and (2) from Definition 4 are complementary conditions for either forward or backward unidirectional search. E.g. a solution plan including state $u$ cannot be shorter than the $f$-value of $u$. Formula (3) corresponds to the case were $V$ is complementary to $U$, thus no other states have to be expanded. In this case the solution cost cannot be smaller than the sum of the cost of both paths individually. From those three conditions we define a lower bound on $C^*$, $lb$, by the following formulas:

$$lb(u,v) = \max\{f_F(u), f_B(v), g_F(u) + g_B(v)\}$$
$$lb(u,v) \leq C^*$$

Using this lower bound we can conclude that every DXBB algorithm must expand all must-expand pairs $(u,v)$, where $lb(u,v) < C^*$. It must not expand any pair with $lb(u,v) > C^*$. And lastly, we cannot make any assumption on pairs where $lb(u,v) = C^*$. In theory, a DXBB algorithm may expand a significant number of such pairs, which are not necessary to prove the optimality of a found solution. But these expansions may be required to find a solution in the first place. As we cannot make any assumptions about those state and given that in search the number of those states are negligible [Chen et al., 2017], we do not include them when discussing performance in this section. However, we will assess the impact of these pairs in the practical experiments concerning bidirectional search in planning. By the definition of must-expand pairs the question is raised on how to choose which state of each pair should be expanded in order to be optimal. To answer this question, Chen et al. [2017] introduced the *must-expand graph*.

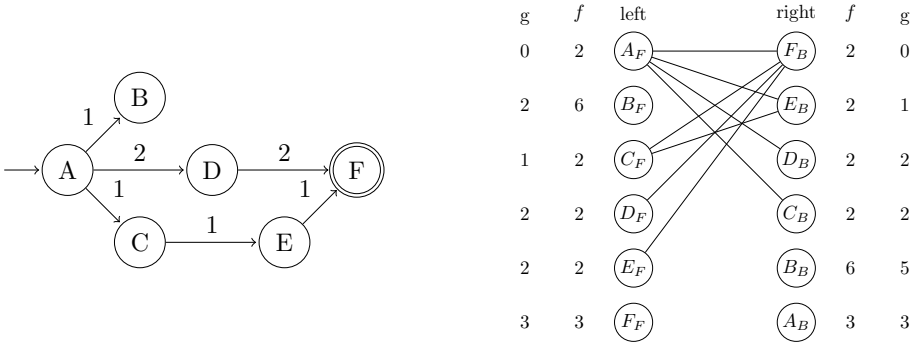



Figure 3.2: (Left) A simple problem instance. The initial state is $A$ and the goal is to reach state $F$. There are two possible solution paths with one optimum. (Right) Depicted is the $G_{MX}$ of the problem instance. The left partition of the graph includes all nodes expanded in forward direction with their corresponding $g$- and $f$-value. Complementary are the backward expanded nodes on the right partition.

**Definition 5** (Must-Expand Graph $G_{MX}$)
*The must-expand graph $G_{MX}$ of a problem instance illustrates how the must-expand pairs are entangled with each other. $G_{MX}$ is a undirected, unweighted bipartite graph defined as follows: for each state in $u \in G$, where $G$ is the associated state space of the problem instance, there exists a vertex in each partition of the graph, $u_F$ for the left partition and $u_B$ otherwise. Two vertices $u_F, v_B \in G_{MX}$ derived from state $u$ and $v$, are connected by an edge iff the state pair $(u,v)$ is a must-expand pair, in particular $lb(u,v) < C^*$ holds.*

By this definition, expanding the states included in a vertex cover of $G_{MX}$ is identical to expanding all must-expand pairs. Thus, the minimal number of necessary state expansions corresponds to the size of the minimal vertex cover of $G_{MX}$. We refer to the minimal vertex

cover of $G_{MX}$ by VC and the size of it by |VC|. By using |VC| as the minimal number of necessary expansions, Shaham et al. [2017] were able to prove that the `fMM`-algorithm did expand exactly |VC| states, provided the optimal meeting point $p^*$. Important to note is that $p^*$ is not known in advance, therefore, `fMM`($p^*$) is not classified as a DXBB-algorithm.

In conclusion, we defined an alternative way to describe the minimal number of state expansions in bidirectional search. By illustrating the search as a bipartite graph, we can deduce that finding the minimal vertex cover of said graph is identical with finding the optimal states to expand for all must-expand pairs. For our work, this is especially important as it directly inspired the design of the `NBS` algorithm.

### 3.2.1 Finding VC

In the last section we defined the bound |VC| and showed that it can be derived by finding the minimal vertex cover of $G_{MX}$. But even though the minimal vertex cover of a bipartite graph can be computed by a polynomial algorithm with the complexity of $O(E\sqrt{V})$ [Hopcroft and Karp, 1973][4], $G_{MX}$ itself may grow exponentially with the input size, as every new node may be connected to every already existing node. Therefore, finding |VC| using $G_{MX}$ might not be feasible. For this reason, Shaham et al. [2017] designed an improved algorithm, which only uses the significant information from $G_{MX}$ without explicitly creating it, see Algorithm 2. The fundamental idea is to abstract $G_{MX}$ by combining nodes with equal g-value, thus creating the abstract Graph $\hat{G}$, and subsequently determining the weighted vertex cover (WVC) by brute force searching over the whole solution space. Using brute force gives the impression that the algorithm cannot be efficient and is rather uninformed, but given a number of constraints on the solution space, the search itself is of linear complexity. At last, Shaham et al. showed that |WVC| of $\hat{G}$ is identical with |VC| of $G_{MX}$.

Algorithm COMPUTEWVC does compute |VC| efficiently, and it also provides us with $\underline{i}$, which was introduced in Section 3.1.2, and is essential for computing $p^*$. To understand the significance of this algorithm and the meaning of $\underline{i}$, we first have to establish $\hat{G}$ and by extend $G_F(i)$. The following definitions are derived from Shaham et al. [2017].

**Definition 6** ($G$-Value Group Sets)
*Abstracting from $G_{MX}$, we define the sets $G_F(i)$ and $G_B(j)$ where $G_F(i)$ is the set of all states $s \in G_{MX}$ with $g_F(s) = i$. Likewise, $G_B(j)$ includes all states with $g_B(s) = j$.*

Given that all states in $G_F(i)$ are derived from $G_{MX}$ we can deduce a number of properties. First, all $s \in G_F(i)$ have $f_F < C^*$, vice versa for states in $G_B(j)$. This follows directly from how $G_{MX}$ is derived, see Definition 5. As a result, a state-pair $(u, v)$ where $u \in G_F(i)$ and $v \in G_B(j)$ is a must-expand pair iff $i + j < C^*$ holds. As all states in $G_F(i)$ have the same $i$, we can infer that if there exists a pair $(u, v)$ that is a must-expand pair, then all possible pairs created from the same $g$-value group sets are must-expand pairs as well. Such group pairs are called *must-expand group pairs* (abbreviated by MEGP). $N(G_F(i))$ denotes the number of states combined within $G_F(i)$. Using the definition of MEGP, we can define $\hat{G}$.

[4] The complexity of $O(E\sqrt{V})$ is derived from the bipartite graph, where $E$ is the number of edges and $V$ the number of vertices.

**Algorithm 2** Calculate WVC [Shaham et al., 2017]

```
1: procedure CalculateWVC(Ĝ,C*)
2:     Run A* and find all N(G_F(i)) and N(G_B(j))
3:     i = 0
4:     j = dual(i)
5:     WVC = Σ_{y<j} N(G_{B_y})
6:     minWVC = WVC
7:     while i < C* do
8:         WVC = WVC + N(G_F(i)) − N(G_B(j))
9:         i++, j++
10:        if WVC < minWVC then
11:            minWVC = WVC
12:        end if
13:    end while
14:    return (minWVC)
15: end procedure
```

**Definition 7** (Must-expand Groups Graph $\hat{G}$)
*$\hat{G}$ is an abstraction of $G_{MX}$, where the different nodes of $G_{MX}$ are grouped by their g-value. $\hat{G}$ is a bipartite graph defined as follows: $\hat{G} = ((V_F, V_B), E)$, where*

- *$V_F$ are the vertices of one partition of $\hat{G}$. Each node $v_F \in V_F$ has a corresponding non empty $G_F(i)$ set, where i defines the associated g-value and the associated weight $w \in \mathbb{R}_0^+$ is given by $N(G_F(i))$. Consequently, this g-value defines which states are congregated into a specific node and the weight indicates how many states exist with this particular g-value.*
- *$V_B$ are the vertices of the other $\hat{G}$-partition. They are abstracted in the same fashion as $V_F$ with the difference that for $V_F$ only forward expanded states are considered and for $V_B$ only the backward expanded ones.*
- *$E$ are the edges connecting the partitions. There exists an edge $e(v_F, v_B)$ where $v_F \in V_F$ and $v_B \in V_B$ iff the corresponding $G_F(i)$ and $G_B(j)$ are MEPG.*

To use $\hat{G}$ as a substitute for $G_{MX}$, Shaham et al. [2017] proofed that instead of computing the minimal vertex cover (VC) of $G_{MX}$, it is equivalent to use the abstraction $\hat{G}$ and find the minimal weighted vertex cover (WVC). It is important to note that WVC can be computed more efficiently than VC, as there is no need to compute the must-expand graph $G_{MX}$.

**Definition 8** (Lowest Upper Bound $\underline{i}$)
*Let $\underline{i}$ be such that $g_F$ is the minimal value, given $g_F(\underline{i}) \in WVC$. Then all $g_F(i)$ with $i < \underline{i}$ are elements of WVC as well. Analogously, let $\underline{j}$ be such that $g_B$ is the minimal value, given $g_B(\underline{j}) \in WVC$. Furthermore, $\underline{j} = dual(\underline{i})$, where dual returns the minimal $j$ value given input $i$, where $i + j \geq C^*$ holds.*

Those two bounds, $\underline{i}$ and $\underline{j}$, enable us to find WVC efficiently. For each $\underline{i}$ value, there exists exactly one $\underline{j}$ value. Furthermore, $\underline{i}$ is lower bound by 0 and upper bound by $C^*$. Therefore, to find $\underline{i}$, we can iterate from 0 to $C^*$ and compare each resulting WVC candidate.

The algorithm COMPUTEWVC, shown in Algorithm 2, combines the theory presented in this section to compute WVC and $\underline{i}$. To give an overview and more insight, we shortly explain the algorithm. At the start, A* is run in forward and backward direction, thereby computing $C^*$ and the must-expand pair groups which build $\hat{G}$. Then, the algorithm iterates over all possible values for $\underline{i}$ and computes the size of the corresponding weighted vertex cover for each. At the end, the vertex cover with the minimal weight is returned. A step-by-step example of the algorithm and the associated theory is shown in Appendix A.1. With this, we introduced all the necessary theory to fully understand the following section in which we present the NBS algorithm.

## 3.3 Near-Optimal Front-to-End Bidirectional Search Algorithm (NBS)

In this section, we want to thoroughly introduce the algorithm which this thesis is focused on. NBS is a front-to-end bidirectional search algorithm introduced by Chen et al. [2017]. They designed the algorithm as a follow up to the previous paper of Eckerle et al. [2017], which established the necessary theory. The following paragraph recapitulates the previously introduced theoretical insights by highlighting their relation with the NBS algorithm.

**Background** Chen et al. use the notation of "surely expanded" (s.e.) which describes a state in unidirectional heuristic search whose $f$-value is smaller than the optimal solutions cost, thus, it needs to be expanded in order to find the optimal solution [Dechter and Pearl, 1985]. In combination with the sufficient conditions for node expansions given by Eckerle et al. [2017], they define *must-expand* pairs as a two-tuple, where either the first must be expanded in forward direction, or the second in backward direction. By expanding all *must-expand* pairs it can be shown that a given solution is optimal. The *Must-Expand Graph* $G_{MX}$ is designed to depict the relationship of must-expand pairs. $G_{MX}$ is a bipartite graph, with forward expanded nodes on the left side and backward expanded nodes on the right side. Those nodes are joined by an edge iff they are must-expand pairs. They further prove that the minimal vertex cover of $G_{MX}$ is equivalent to the minimal number of nodes that have to be expanded by an admissible algorithm to be optimal, thus establishing the lower bound |VC|. As computing $G_{MX}$ with the corresponding minimal vertex cover is more expensive than solving the problem itself, they adapted a known minimal vertex cover algorithm [Papadimitriou and Steiglitz, 1982] which approximates the optimal minimal vertex cover. Resulting in the formulation of the *Near-Optimal Bidirectional Heuristic Search* algorithm NBS.

**The Algorithm** In this paragraph we want to provide more technical insight in how the NBS algorithm functions. However, we will not provide any proof or formal description of its properties as they are already concisely and wholesomely written down in the paper by Chen et al. [2017]. Moreover, the main influences for this algorithm are described in the previous paragraph. Namely that the sufficient conditions for state expansions can be represented as a minimum vertex cover of the *Must-Expand Graph* $G_{MX}$. The bounded sub-optimality of the NBS algorithm is a direct consequence of the algorithm used to find the minimum vertex

cover. The approximation computes a cover by choosing edges randomly and including both adjacent vertices. This is repeated until all edges have at least one adjacent vertex which is included. Hence, at worst two times |VC| states will be expanded.

To discuss the algorithm in more depth we present high-level pseudocode in Algorithm 3. First, the open lists are initialized with the initial and goal state respectively. In the while-loop two states $u$ (forward expanded) and $v$ (backward expanded) are chosen, where $lb(u,v) = \max\{f_F(u), f_B(v), cost(u)+cost(v)\}$ is minimal. As a reminder, $lb(u,v)$ is a lower bound on the solution path cost $UZV^{-1}$, where $U$ is a forward path from $s_0$ to $u$, $V$ a backward path from $s_G$ to $v$, and $Z$ is a forward path from $U_{|U|}$ to $V_{|V|}$. The states with lowest $lb$-value express the minimal solution cost that can be found. In line 5 the algorithm tests whether a previously found solution has a smaller cost than the new lowest $lb$-value. If this is the case, then there exists no improved solution and consequently the previously found solution must be optimal. If it is not the case, both states are expanded, which is in accordance to the greedy approximation algorithm by Papadimitriou and Steiglitz [1982]. This loop is repeated until either a solution is proven to be optimal, or the problem is shown to be unsolvable.

**Algorithm 3** NBS [Sturtevant and Felner, 2018]

```
1:  Put s_0 in Open_F and s_G in Open_B
2:  while Open_F and Open_B are not empty do
3:      Among u ∈ Open_F and v ∈ Open_B
4:      Select the pair (u, v) with lowest lb(u, v)
5:      if lb(u, v) ≥ cost(U) then
6:          return U
7:      end if
8:      Expand both u and v
9:      if new path from s_0 to s_G is found then
10:         Update U if new path is better than previous
11:     end if
12: end while
```

We summarize that, NBS exhibits three important properties. (1) As an admissible algorithm, it is guaranteed to find the optimal solution or to show that no solution exists. (2) It does not expand more states than twice the sufficient number. (3) There cannot exist a DXBB-algorithm which has a better worst-case performance. Those properties define NBS as a powerful search algorithm with promising application.

# 4

# Planning with the NBS algorithm

The goal of this thesis is to inquire the applicability of the NBS algorithm in planning. To that end, we implemented three different algorithms into the state-of-the-art planner Fast-Downward: NBS, fMM, and COMPUTEWVC. In this section, we provide insight into the practical implementation of the aforementioned algorithm and hypothesize about the expected outcome.

## 4.1 Using Search in Planning

In Chapter 3 we introduced different bidirectional search algorithms. But for them to be applicable in planning, they have to be slightly adjusted. In this section, we list the necessary alterations in order for search to be applied in planning.

**Reversing the State Space** As defined in Chapter 2, a planning state space is a 6-tuple $\mathcal{T}(\Pi) = \langle S, T, L, c, s_0, s_G \rangle$, which is induced by the planning task $\Pi$. Whereas forward search operates on the directly induced planning state space, backward search requires a reversed state space.

**Definition 9** (Reversed Planning State Space)
*A reversed state space resulting from reversing the induced planning state space $\mathcal{T}(\Pi)$ is defined as 6-tuple $\mathcal{T}'(\Pi) = \langle S, T', L', c', s'_0, s'_G \rangle$, where*

- $T' : S \times A \mapsto S$ *is a finite set of* ***reversed transitions****, for all $t' \in T'$ there exists a transition $t \in T$, which has the opposite effect.*
- $L' : T' \mapsto A$ *is a* ***label*** *function defining from which action the reversed transition is induced.*
- $c' : T' \mapsto \mathbb{R}_0^+$ *is the* ***cost*** *function.*
- $s'_0 = s_G$ *is the finite set of* ***initial states****.*
- $s'_G = s_0$ *is the* ***goal state****.*

The reversed transition system enables the backward search to traverse the state space in backward direction. But in contrast to the forward search, there may not be a single initial state. Which raises the question whether the theoretical bounds of NBS and fMM still hold in planning. To solve this problem, we introduce the *primary initial state*, which is contrived to create the set $s'_0$ when expanded. Hence, we call the set of initial goals in the backward search $s'_0$ as the *secondary initial states*. Given such a primary initial state, it is evident that a problem with multiple initial states can be reduced to a problem with one additional state, that functions as an artificial initial state. Before we discuss the problems going along with secondary initial states, we present the driverlog domain.

### 4.1.1 Driverlog

To give insight into theoretical properties of bidirectional search in planning, we introduce the driverlog domain, which serves as an example throughout this chapter. The driverlog domain is a part of the official benchmark for the international planning competition (IPC). It is a standard planning domain and includes *drivers*, *trucks*, *packages*, *locations*, *paths*, and *streets*. The streets and paths connect different locations. Streets can only be used by trucks and paths only by foot. A common problem in the driverlog domain has a solution where at least one driver has to find his way to the closest truck, board the truck, load the package, and then deliver the package at the right location. To accomplish the successful delivery, the following actions are available:

- *Load-truck* loads the package into the truck if both share the same location.
- *Unload-truck* unloads the package transported by the truck, at the truck's current location.
- *Board-truck* puts the driver behind the steering wheel given both are at the same location.
- *Disembark-truck* removes the driver from the truck and sets its location to the one of the truck.
- *Drive* moves the truck and its current driver between locations connected by a street.
- *Walk* lets a driver walk between locations connected by a path, unless he is currently driving a truck.

This example serves as a simple illustration on how a standard planning domain is structured. In the following explanations, we will use it as a foundation to give a more detailed description and analysis of the encountered phenomenon.

### 4.1.2 Secondary Initial States Explosion

A planning task defines variable assignments as goal conditions. Those conditions define which states are valid goal states. However, most often, they do not define a single state but a set of states. All the states in the set share the property that they fulfil the goal

conditions while differing in the variables which are undefined by the goal conditions. Each valid assignment combination of the undefined variables is a possible goal state. Hence, in backward search they are included in the set of secondary initial states. As all possible combinations have to be included, it is evident that the number of secondary initial states grows exponentially depending on the number and domain size of the undefined variables in the goal conditions. For example, if the goal conditions only dictate that the package must be delivered to a certain location, then it is unknown where the truck is located. Likewise, the position of the driver is unknown as well. Consequently, the secondary initial states include all states where the package is at the defined position and the truck and driver assume any valid position.

The number of secondary initial states is critical for the performance of bidirectional search and can be compared to a high branching factor. As expected, a high number of secondary initial states can reduce the performance of the backward search significantly. And similarly to a high branching factor in forward search, the problem is alleviated by using heuristics to select promising states. Furthermore, the number of secondary initial states is bounded by the number of undefined variables in the goal conditions. In the experiments, Chapter 5, we present which domains and problems display a high number of secondary initial states and how it affects the NBS algorithm. For the moment, it suffices to say, that in practice various cases occur and heuristics partially enable search to overcome the problems. However, as we show in the following worst-best case analysis, in theory no detailed assumption can be made regarding the impact of many secondary initial states.

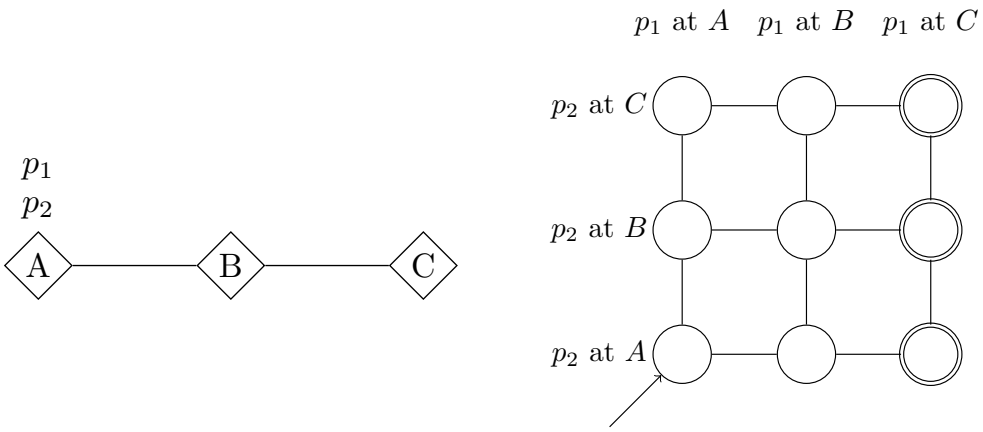



Figure 4.1: The *walker* problem with its corresponding state space.

Consider a planning problem of the introduced driverlog domain. The specific problem includes two drivers and three locations. Driver $d_1$ and $d_2$ are initially at location $A$. The goal is for driver $d_1$ to reach location $C$, which is connected to $A$ by $B$. Whereas the position of driver $d_2$ is irrelevant. The described problem and the corresponding state space is shown in Figure 4.1 and can be interpreted as driver $d_1$ walking to his truck. As can be seen in the Figure, the problem has three possible goal states. Those states are equivalent with the secondary initial states of the backward search. In Figure 4.2 the exploration of the search space is depicted with different search directions and heuristics. (1) Forward search

with a pattern database heuristic[5] (PDB) of size 1. (2) Bidirectional search with the same heuristic. (3) Bidirectional search with a PDB heuristic of size 2, which corresponds to the perfect heuristic in this instance, as there are only two variables present in total. The three depictions illustrate how different heuristic can influence the impact of the number of secondary initial states, by shaping the exploration of the search space. In the second example, the bidirectional search starts a backward search originating from each secondary initial state, these searches can be viewed as independent, at least in this example, as they run parallel to each other without intersecting. Showing that in a worst case, the secondary initial states may make the backward search more expensive for each additional state. Depending on the structure of the transition system, this effect might be amplified or weakened. In the worst case, a parallel chain of actions may open up a new part of the state space, which might appear to be more promising than the real solution. On the other hand, it might also be possible, that all secondary initial states directly lead back to a common state. For instance, in the previously introduced transportation example, the trucks might be positioned anywhere, however, as the package is at a specific place, the truck must always drive there first. Therefore, the solution path of all inferior secondary states lead through previously expanded states. The third example illustrates how a stronger heuristic can alleviate the problem successfully as it enables the algorithm to recognize suboptimal states. Showing that the accuracy of the applied heuristic is critical.

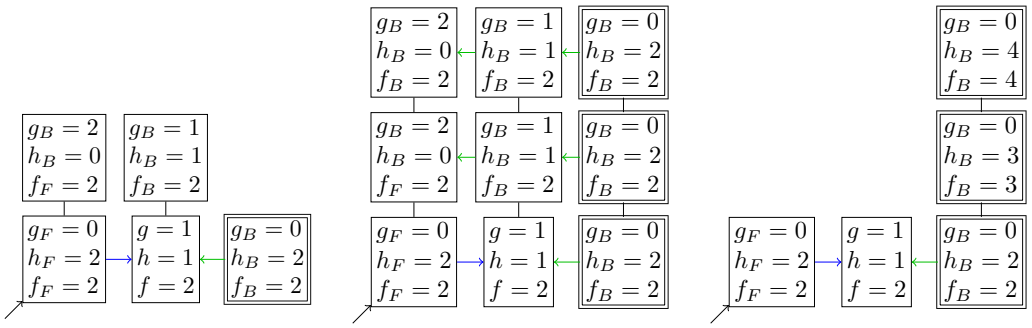



Figure 4.2: The search space of the *walker* problem for (1) forward search, (2) bidirectional search with PDB size 1, and (3) bidirectional search with PDB size 2. The color of the arrow denotes whether the indicated state is expanded in forward (blue) or backward (green) direction.

### 4.1.3 Using Heuristics

The last missing components for bidirectional search are heuristics. In this case, a front-to-end forward and backward heuristic. As the search is conducted in both directions simulta-

[5] Pattern Database Heuristics [Edelkamp, 2001] estimate the goal distance by solving an abstracted problem, which is a partial depiction of the original problem limited on the variables defined in the pattern. The size of the PDB determines how accurate and thus complicated the abstract depiction is. A PDB where the pattern include all variables solves the full problem to get an heuristic estimate, hence is equivalent with $h^*$

neously, there are some constraints on which heuristics are usable. Defined in the paper by Chen et al. [2017], the NBS algorithm expands at most twice the number of must-expand pairs. But only if the applied heuristics are admissible and consistent. Furthermore, we limit the scope of this thesis to always use the same forward as backward heuristics. In practice, it is possible that one heuristic is either more favourable, or only available for a certain direction. It is worth mentioning, that the heuristic quality of the same heuristic can differ between the directions, which has a direct influence on the performance of the corresponding search. Given the mentioned constraints and the choice to run the algorithm in Fast-Downward, we choose to apply the following heuristics.

- *Blind heuristic* ($h^1$) gives a estimate solely based on whether the state is a goal state or not. Using blind heuristic is equal to an uninformed brute force search and is used primarily to give a baseline to compare against.
- *Max heuristic* ($h^{max}$) [Bonet and Geffner, 2001] abstracts the problem by removing delete effects from all occurring actions, thereby creating the delete relaxation of the problem. Solving the delete relaxation of the problem returns an admissible heuristic. The max heuristic is not the most informative heuristic, but it is efficiently computable.
- *Critical path heuristic* ($h^m$) [Haslum and Geffner, 2000] is a generalization of $h^{max}$. In particular, $h^m$ with $m = 1$ is equivalent with $h^{max}$. Both heuristic estimate the heuristic value with the precision depending on the factor $m$. $h^{m=2}$ considers more information, which makes $h^{m=2}$ a stronger heuristic but also more expensive to compute. We use $h^m$ with $m = 2$, as it improves the estimate regarding $h^{max}$, but is still computable in a timely manner.
- *Landmark-Cut heuristic* ($h^{lmcut}$) [Helmert and Domshlak, 2009] is an inconsistent but very strong heuristic. To get a contrast to the other heuristics, we want to test the performance of NBS with a sophisticated and strong heuristic even tough we lose the bound on the number of guaranteed expansions.
- Heuristics not included in this list are either inadmissible, inconsistent, not implemented in Fast-Downward, not compatible with our implementation, or inferior to the chosen heuristics.

Outside of the field of planning, bidirectional search is well-researched with various reasoning justifying its performance. Most recently, Barker and Korf [2015] deduced that bidirectional heuristic search is never the optimal option to solve a given problem. Either unidirectional heuristic search, or bidirectional brute force search outperforms bidirectional heuristic search in all but a few special cases. Barker and Korf categorized problems on the distribution of states in the state space. For instance, a problem where more than half the states have a $g$-value higher than half the optimal cost, is very suited for unidirectional heuristic search. However, it is important to note that those characteristics only apply to the MM-algorithm. To derive similar properties, we run the experiment with various algorithms. Only considering the heuristics, we expect that bidirectional search has a clear advantage

if the available heuristic is weak. On the other hand, we expect A* to be more efficient if paired with a strong heuristic.

With this we conclude the theoretical discussion of bidirectional search in planning and continue with the practical approach we have taken, the problems that occurred, and the resulting properties.

## 4.2 Implementation Details

The preceding sections discuss the theoretical details of implementing bidirectional search in planning. In this section, we explain how the NBS, fMM, and COMPUTEWVC algorithms are implemented in or using Fast-Downward. The focus thereby lies on describing implementation details and clarifying ambiguities to enable reconstruction.

We implemented both the NBS algorithm and fMM into the state-of-the-art planner Fast-Downward introduced by Helmert [2006]. The planner is open-sourced, written in C++, and provides a framework for researcher to implement their algorithms or heuristics as plug-ins. The planning algorithm we implemented is composed of a monitor class, which controls a search with their respective search task for each direction. The conversion from forward to backward direction is encapsulated in our implementation of the BACKWARDTASK, which is used by NBS as well as fMM. For the COMPUTEWVC-algorithm, we created a python script which starts by running two instances of Fast-Downward to gather the size of all must-expand group pairs, and then processes the gained information to compute the desired solution.

### 4.2.1 Searching Backward

Preceding this section, we defined that bidirectional search traverses the state space. However, in practice it is rarely feasible to build the entire state space. Leading to a practical implementation of search as written in Section 2.2.4. Commonly, the search successively explores the state space guided by heuristics. Which means that the full state space is never created, and as a result it is not possible to invert it as described in Section 4.1. In practice, we invert the planning task and search in backward direction in the same fashion as forward. A planning task is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ consisting of state variables, actions, initial state, and goal conditions. For an algorithm like A*, this information is sufficient to create the initial state and expand the search space in the direction of the goal. However, in the case of bidirectional search, an additional backward search has to be conducted originating from the goal. Our approach to solving this problem, is to derive the *backward task* from the original task, thereby changing the backward search to a simple forward search on a reshaped task. We will now define how this conversion works.

**Reverse Actions** To create the backward task, all the available actions have to be reversed. The reversed action $a'$ to $s_1 = s_0[\![a]\!]$ is constructed such that $s_0 = s_1[\![a']\!]$. To ensure this behaviour, we consider following cases. We differentiate between variables appearing either in the (1) precondition, (2) effect, (3) or both.

- (1) Variables mentioned in only the precondition have the same value in state $s_0$ and $s_1$. Therefore, it suffices to keep them as preconditions.
- (2) Variables which are not mentioned in the preconditions but in the effect are problematic to handle, as the assignment of those variables are unknown for the predecessor state $s_0$. Therefore, we must construct an action for every possible variable assignment as effect, all having the original variable assignment as precondition.
- (3) Variables which appear in the precondition and effect can simply be reversed by switching the assignments.

**Primary Initial State** By reversing all actions, the backward algorithm is enabled to successfully traverse the problem backwards. But, it is not yet known from where to start and where to end. For that we have to define the initial state and goal conditions. Defining the backward goal conditions is simple, as it is equivalent with the forward initial state. But defining the backward initial state is more complex, as the explicit goal state is not known a priori. We implement this by adding the primary initial state and the corresponding actions to expand the secondary initial states.

**Action Explosion** When a forward action is reversed, it can occur that the number of generated backward actions increases exponentially. In the case of an action which has multiple effects without counterparts in the precondition, it is necessary to compute all combinations of variable assignments. The number of reversed actions generated from forward action $a$ amounts to the product of the domain sizes of the variables only present in $\mathit{eff}(a)$. For instance, consider the previous example, but supplemented with an additional action *drive-depot* and location *depot*. The described action moves a specified truck from any position to the *depot*. Reversing the *drive-depot* action requires to create a backward action for each possible prior location. Creating the backward task may result in an exponential growth of actions. However, as we show in our experiments, they rarely exceed twice the number of forward actions and are often equivalent.

**Illegal States** We discussed how the reversing of actions and the use of goal conditions to derive the secondary initial states can increase the total amount of actions in the backward task. So far we did not analyse whether the thereby generated states will be reachable or legal. In this paragraph, we investigate the nature of the generated states.

We differentiate between three different kinds of states. First, *legal states* have valid combinations of variable assignments and are included in the forward search state space. Second, *unreachable states* do not violate any logical constraints on the task, however, are unreachable by the forward search. For instance, the previous example could include two disjoint streets groups. On each group, one truck is available. It would be impossible for one truck to switch groups, as they are not connected, yet, it would be legal for a truck to be located in any city, even though the particular city is not reachable from the initial state. Third, every task has a number of mutually exclusive variable assignments (mutexes).

Although, we use the provided ones from Fast-Downward as constraints, the given list is not exhaustive, thus, it can occur that states are created which violate implicit mutexes.

In our implementation of bidirectional search, we mostly neglect illegal or unreachable states, albeit that they have an impact on performance. The dilemma stems from the individuality of the problem for each domain. Every domain displays a different behaviour regarding illegal states. As a consequence, it is difficult to devise a general solution. One approach we take is to not create actions which violate a known mutex in their effect. This already reduces the problem significantly. Another option, would be to close generated states, if they violate any mutex. However, in practice it was apparent that this procedure does not pay off as it increases the runtime significantly without equal compensation[6]. A different approach would be to preprocess the problem more thoroughly to either capture all mutexes, or only create reverse actions for the cases that actually occur in forward direction. This could be tried by generating the search space of the relaxed search, or using the *transition normal form* (TNF) of the problem task to remove all ambiguity. A $SAS^+$ task is in TNF if

- all actions list the same variables in the precondition and effect ($\forall a \in \mathcal{A}, vars(pre(a)) = vars(\mathit{eff}(a))$), and
- the goal conditions defines a single state ($vars(\mathcal{G}) = \mathcal{V}$).

As mentioned, those are concepts to solve the problem of illegal states, however, they were not implemented. We limit ourselves to evaluate bidirectional search in planning without applying exhaustive optimization options.

In conclusion, we discussed how the NBS algorithm introduced in Chapter 3 can be applied to planning. We identified that the main problematic results from an ambiguity in the planning task, which can lead to numerous secondary initial states. The additional effort resulting from those states is inherent to bidirectional search with explicit-state representation. While a symbolic state representation would not have to deal with this particular problem, it would have its own drawbacks. For instance, expanding a symbolic state is much more expensive than an expanding an explicit state.

In the next section, we present the conducted experiments which are designed to evaluate the claims proposed in this section. First, in order to test the performance of NBS we compare how efficient it solves problems compared with A*. Second, we investigate whether there are any measurable causal relations between the performance of NBS and the innate structure of the problem, which we assess with the help of fMM and computeWVC.

[6] Only in the *depot* domain states were closed by this method, but searching for them, takes up to a third of the whole run time in every domain.

# 5 Experiments and Evaluation

This chapter includes the setup, the execution, and the evaluation of the conducted experiments. The overarching goal of the experiments is to inquire the applicability of bidirectional search with explicit-state representation in planning. To achieve this, we first run our implementation of NBS over a set of problems and monitor the difference in performance depending on the problem instance and applied heuristic. Secondly, we run fMM and COMPUTEWVC over the same set of problems in order to find relations between the performance of NBS, the performance of fMM with different input argument, and $p^*$.

## 5.1 Environment

Before we discuss the two individual experiments, we specify the environment and setup which is shared by both. We use the state-of-the-art planning system Fast-Downward [Helmert, 2006] to run our implementation of NBS and fMM and combine them with various heuristics. To enable experiments with a reasonable scope, we utilize the scientific computing core facility at the University of Basel (sciCore[7]), which consists of a cluster of Intel Xeon E5-2660 processors running CentOS 6.5 at 2.2 GHz. The custom benchmark we use is a modified version of the most recent international planning competition (IPC) benchmark, which filters out all domains which are either designed for satisfying planning, including axioms, or using conditional operators.

## 5.2 Running NBS

In the first experiment, we run the NBS algorithm on our custom benchmark with various heuristics. Each run has a time limit of 30 minutes and a memory limit of 3.5 GB. For every problem in the benchmark, we run a total of 8 combinations of algorithms and heuristic. A* and NBS paired with either the blind, max, critical path, or landmark-cut heuristic. Allowing us to not only measure the performance of NBS, but also giving a comparison to A*, an unidirectional forward search. We focus on the following aspects during the ensuing

[7] http://scicore.unibas.ch

experiment.

- Measuring the number of additional actions in the backward task relative to its original number and evaluating the difference it makes in practice.
- Analysing the number of secondary initial states and its impact on the performance of NBS.
- Evaluating the performance of NBS and A* in regards to the applied heuristic.
- Determining whether the bounded number of expansions guaranteed by NBS is a good measure for the overall performance of NBS, or if a non-negligible portion of the expansions are persisting in the last search layer.

### 5.2.1 Results and Evaluation

The condensed results of the first experiment are shown in Table 5.1. It lists all the included domains and displays three different values for each applied heuristics. First, the number of problems which were solved by running A*, limited by the constraints listed in Section 5.1. Second, the number of solved problems by NBS. And third, the number of problems where NBS expanded less states than A*. More detailed and extended results can be found in the Appendix B.1.

The most important aspect visible in Table 5.1 is that there are many problem instances for which NBS expands less states than A* and some instances have only been solved by the NBS algorithm. This shows that NBS is not only applicable to planning, but is able to compete in practice with a well performing algorithm like A*. However, NBS does not consistently outperform A* or vice versa. There are different factors which influence the performance of either algorithm. In the following paragraphs we exemplify how the NBS algorithm performs in detail. We will conclude the NBS experiment by establishing how frequent and thus significant the described factors are.

**Performance of NBS** Before going into the result, we reiterate the advantages of bidirectional search in comparison with unidirectional search. The fundamental idea is that the search space grows exponentially with the search depth, thus bidirectional search should reduce the growth significantly as each conducted search only reaches half as deep. Leaving heuristics aside for the moment, it is evident from the results depicted in Table 5.1 that bidirectional search is indeed advantegous in certain cases. NBS achieves to solve more problem instances compared to A* in 9 domains and an equal number of problem instances in an additional 5 domains. The consequently arising question of these findings is: why does the NBS algorithm not strictly outperform A*? We argue that an important factor for the performance of either algorithm is structure of the induced state space. It defines the branching factor in forward and backward direction, dictating whether one algorithm is fundamentally more efficient than the other. Thereby included are the number of secondary initial states which are defined by the specific problem instance.

| Domain | Solved A* (#) | | | | Solved NBS (#) | | | | NBS < A* (#) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $h^1$ | $h^{max}$ | $h^m$ | $h^{lmcut}$ | $h^1$ | $h^{max}$ | $h^m$ | $h^{lmcut}$ | $h^1$ | $h^{max}$ | $h^m$ | $h^{lmcut}$ |
| blocks | 18 | 21 | 10 | 28 | **27** | **27** | **13** | **31** | 18 | 21 | 5 | 15 |
| depot | **4** | **6** | **2** | **7** | 3 | 2 | 1 | 2 | 0 | 1 | 0 | 0 |
| driverlog | 7 | 9 | 2 | **14** | **10** | **10** | **3** | 12 | 6 | 8 | 0 | 1 |
| elevators-opt08-strips | **14** | **19** | 0 | **22** | 12 | 13 | 0 | 11 | 8 | 5 | 0 | 0 |
| elevators-opt11-strips | **12** | **16** | 0 | **18** | 10 | 11 | 0 | 8 | 7 | 5 | 0 | 0 |
| floortile-opt11-strips | 2 | 6 | 0 | 7 | **10** | **12** | 0 | **10** | 2 | 6 | 0 | 5 |
| ged-opt14-strips | 15 | 15 | 5 | 15 | **19** | **20** | 5 | **19** | 10 | 10 | 0 | 10 |
| gripper | **8** | **8** | 3 | 7 | 7 | 7 | 3 | 7 | 7 | 6 | 1 | 4 |
| hiking-opt14-strips | **11** | **12** | **2** | **9** | 10 | 10 | 1 | 6 | 3 | 0 | 0 | 0 |
| logistics00 | 10 | 12 | 6 | 20 | **13** | **13** | 6 | 20 | 10 | 8 | 0 | 0 |
| miconic | **55** | **55** | **30** | **141** | 50 | 50 | 26 | 53 | 31 | 21 | 0 | 0 |
| nomystery-opt11-strips | 8 | 9 | **6** | 14 | **10** | **10** | 5 | 14 | 8 | 4 | 0 | 0 |
| openstacks-opt08-strips | **22** | **22** | **5** | **21** | 12 | 11 | 3 | 8 | 0 | 4 | 2 | 3 |
| openstacks-strips | **7** | **7** | **5** | **7** | 5 | 5 | 0 | 5 | 0 | 5 | 0 | 0 |
| pegsol-08-strips | 27 | 28 | **9** | 28 | **28** | 28 | 8 | 28 | 20 | 14 | 0 | 10 |
| pegsol-opt11-strips | 17 | 18 | 1 | 18 | **18** | 18 | 1 | 18 | 14 | 11 | 0 | 9 |
| psr-small | **49** | **49** | **40** | **49** | 42 | 42 | 29 | 40 | 0 | 11 | 10 | 7 |
| rovers | **6** | **6** | 4 | **8** | 4 | 5 | 4 | 4 | 0 | 0 | 0 | 0 |
| satellite | **6** | **6** | **3** | **7** | 4 | 4 | 2 | 4 | 2 | 1 | 0 | 0 |
| scanalyzer-08-strips | 12 | 9 | 3 | **16** | 12 | 9 | 3 | 11 | 12 | 6 | 0 | 3 |
| scanalyzer-opt11-strips | 9 | 6 | 1 | **12** | 9 | 6 | 1 | 8 | 9 | 5 | 0 | 3 |
| storage | **14** | **15** | **7** | **15** | 5 | 5 | 2 | 4 | 0 | 0 | 0 | 0 |
| termes-opt18-strips | 10 | 10 | 0 | 6 | **13** | **13** | 0 | **8** | 10 | 10 | 0 | 6 |
| tpp | **6** | **6** | **5** | **7** | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 |
| transport-opt08-strips | 11 | 11 | **6** | 11 | 11 | 11 | 5 | 11 | 5 | 0 | 0 | 0 |
| transport-opt11-strips | 6 | 6 | **1** | 6 | 6 | **7** | 0 | 6 | 5 | 0 | 0 | 0 |
| transport-opt14-strips | **7** | **7** | **1** | **6** | 6 | 6 | 0 | 5 | 5 | 1 | 0 | 0 |
| trucks-strips | **6** | **10** | 2 | **10** | 4 | 6 | 2 | 6 | 0 | 0 | 0 | 0 |
| zenotravel | 8 | 8 | **5** | **13** | 8 | 8 | 4 | 9 | 7 | 2 | 0 | 0 |

Table 5.1: A concise overview of all NBS runs on the custom benchmark. Focusing on the number of solved problems per domain for each algorithm and the comparison between the number of expansions by NBS and A*.

**Secondary Initial States** The number of secondary initial states is one of the main parameters to estimate the performance of NBS. For example, we analyse the depot domain solved using blind heuristic. In the first problem, there are more secondary initial states than expansions by A*. By the design of NBS, it is assured that both search direction always expand the same number of states, with a maximal deviation of $\pm 1$. Thus, if NBS expands more than twice the number of states than A*, it is probable that the NBS algorithm only makes progress with the forward search but unnecessarily expands the same number of states in backward direction. Contrariwise, the problems in the elevator domain have thousands of secondary initial states, still NBS outperforms A*, because the amount of secondary initial states is not significant compared to the number of expansions. Both examples are shown in Table 5.2.

| *Problem* | *Ex.: A* (#)* | *Ex.: NBS (#)* | *Secondary Initial States (#)* |
|---|---|---|---|
| depot:p01 | 400 | 807 | 9216 |
| elevators:p01 | 158′861 | 94′702 | 8820 |

Table 5.2: The table includes a small sample of the experiment data. Two specified problems and the number of expansions by A* and NBS using blind heuristic and additionally the number of secondary initial states are illustrated.

A second factor alleviating the problem of too many secondary initial states is the use of heuristics. The additional information enables the algorithm to discard secondary initial states, given that they are distinguishable from the desired states. This is similar to the known problem of an high branching factor. In particular, NBS expands state pairs where the *lb*-value is smaller than $C^*$, therefore, the applied heuristic must increase the lower bound estimate high enough to be make previous must-expand pairs distinguishable from real ones. By the same token, using the information added from the heuristics decreases the number of must-expand state pairs. As discussed in Section 4.2.1, in theory we cannot claim any certain magnitude of improvement when using heuristics, but in practice, the effect is clearly visible. Going back to the example of the depot problem instance. Using $h^{max}$ instead of $h^1$ reduces the number of expansion by NBS significantly. As a result, A* does not expand half as many states as NBS any more, which shows that the backward search contributes to finding the solution.[8] This effect can be seen in Table 5.3.

| *Problem* | *Ex.: A* (#)* | | *Ex.: NBS (#)* | | *Secondary Initial States (#)* |
|---|---|---|---|---|---|
| | $h^1$ | $h^{max}$ | $h^1$ | $h^{max}$ | |
| depot:p01 | 400 | 140 | 807 | 222 | 9216 |
| depot:p02 | 15′463 | 3781 | 24′087 | 2065 | 147′456 |

Table 5.3: A small sample of the experiment data. Two examplary problems of the depot domain solved by A* and NBS using $h^1$ and $h^{max}$.

**Using Heuristics** The effect of heuristics on the search behaviour varies drastically between different problems and heuristic accuracies. A key attribute of NBS is that it expands all must-expand pairs. Heuristics complement NBS by reducing the number of must-expand pairs. This is achieved by raising the lower bound estimate of certain states, thus making it possible to discern must-expand states with more accuracy. Therefore, to prove optimality, less states must be expanded. In practice, the use of heuristics reduces the number of expanded states as expected. But interestingly enough, it influences A* rarely different than

[8] The assumption that backward search influences the search when NBS expands less than two times the expansions as A* is justified by the reasoning that a single search of the NBS algorithm has a suboptimal tie-breaking compared to A*. Because given same $f$-value, A* knows it should expand states with the highest $g$-value, whereas in NBS it is unclear as the search aims for the goal but needs to meet the frontiers.

NBS. Showing that heuristics behave similarly in both searches and therefore that heuristics are not an important factor to define the general performance of bidirectional search. Which does not imply that heuristics have no influence on bidirectional search, only that they improve the search regardless of the algorithm, hence heuristic and algorithms can be analysed independently. This claim is supported by the data we gathered in the experiment, to exemplify this, we present the first 9 problems of the blocks domain in Table 5.4. The heuristic improves the search significantly, but there is no evident pattern in regards to the used algorithm. The occurring influence is caused by the difference of the two underlying transition systems, therefore their innate structure and not the applied algorithm.

| *Blocks* | | *Expansions (#)* | | | |
|---|---|---|---|---|---|
| | *Algo* | $h^1$ | $h^{max}$ | $h^m$ | $h^{lmcut}$ |
| p4-0 | A* | 85 | 21 | 8 | 7 |
| | NBS | 19 | 10 | 10 | 9 |
| p4-1 | A* | 58 | 21 | 12 | 12 |
| | NBS | 22 | 13 | 13 | 13 |
| p4-2 | A* | 52 | 15 | 7 | 8 |
| | NBS | 19 | 10 | 8 | 7 |
| p5-0 | A* | 467 | 147 | 32 | 21 |
| | NBS | 43 | 35 | 24 | 21 |
| p5-1 | A* | 490 | 126 | 14 | 19 |
| | NBS | 43 | 31 | 18 | 25 |
| p5-2 | A* | 744 | 293 | 52 | 43 |
| | NBS | 65 | 55 | 56 | 54 |
| p6-0 | A* | 1794 | 263 | 24 | 17 |
| | NBS | 55 | 41 | 22 | 16 |
| p6-1 | A* | 3976 | 757 | 41 | 12 |
| | NBS | 53 | 36 | 20 | 16 |
| p6-2 | A* | 6526 | 2554 | 410 | 269 |
| | NBS | 235 | 243 | 224 | 212 |

Table 5.4: We present the number of expansions by NBS and A* with the indicated heuristics for the first 9 problems of the blocks domain.

**Non-Essential Complexity** A factor which has to be considered are complications due to how the NBS algorithm is implemented in practice. The occurrence of illegal states is discussed in Section 4.2.1. In accordance to the definition of Brooks [1987], we categorize the presence of illegal states as non-essential complexity. By that we emphasize that they exist in practice, but theoretically are not essential to the problem. To make an example,

we reintroduce the driverlog domain on which we illustrate this phenomenon on two actions. To drive a truck, a driver must first *board* a truck and afterwards he must *disembark* from it. In the problem file, those two actions are defined as follows. In order to board the truck, both entities have to be at the same location, and the truck must not be driven by another driver. Resulting in the driver being located within the truck, and the truck being marked as driven. Meanwhile, embarking from the truck, only checks whether the driver is located within the truck, and consequently sets his location to the same as the truck and the truck is marked as without driver. Hidden in this formulation, is that the truck does not have to be driven in order to embark from it. In forward search, this case is not reachable, hence it is not covered in the definition. However, when creating the backward actions, it is of vital importance to know that. It is evident that to provide the optimal conditions for bidirectional search, it is not sufficient to use the problem descriptions provided in the IPC benchmark as-is.

There are is one major factor, mitigating the impact of the non-essential complexity. Illegal states are always dead ends. For instance, in the previous example, if we apply the disembark action backwards without setting the truck as driven, the enter action is never applicable. What this means is that we cannot return to legal states from illegal ones. Because if we have a reversed action leading from an illegal state to a legal one, that would mean that the corresponding forward action is applicable in the legal state. Hence, the assumed illegal state is in fact legal, which contradicts the assumption. Given that from an illegal state we cannot reach any legal state, we can infer that safe heuristics will always recognize illegal states as dead ends. Thus, they have a minimal impact when they are generated, but they themself will never be expanded.

Presuming that illegal states are not the most impactful factor regarding the performance of `NBS`, we did not implement any sophisticated improvements, except of the aforementioned discarding of states which violate mutexes. This presumption is justified by the number of reverse actions that are created, which are shown in Table 5.5. In only two domains they reach over twice the original number. For most domains they stay constant for every problems. A correlation between additional actions and performance is also not apparent. Regardless, possible ways to reduce the number of illegal states include deriving a more thorough set of mutexes, creating the backward task manually, or using transition normal form to remove the ambiguity of actions.

**Bounded Worst Case** Having discussed multiple components influencing the performance of `NBS` in planning, the questions arises whether the bounded number of expansions is still valid and informative. To reiterate, the claim is that the `NBS` expands at most $2 \cdot |\text{VC}|$ in the $f$-layers preceding the last layer, which is an interesting bound because the expansions in the last layer are negligible in search [Chen et al., 2017]. Unlike in search, in planning the backward search is in practice not a mirrored search but a different one. This means that the bound is still valid, but is more loosely related to the number of expansions done by an exclusive forward search. Which raises the question whether the number of expansions in the last layer are still negligible. As can be seen in Table 5.6, in our experiments,

| *Domain* | *min.* | *avg.* | *max.* | *N* |
|---|---|---|---|---|
| blocks | 1.0 | 1.0 | 1.0 | 31 |
| depot | 1.0 | 1.0 | 1.0 | 2 |
| driverlog | 1.0 | 1.0 | 1.0 | 12 |
| elevators-opt08-strips | 1.0 | 1.0 | 1.0 | 11 |
| elevators-opt11-strips | 1.0 | 1.0 | 1.0 | 8 |
| floortile-opt11-strips | 1.0 | 1.0 | 1.0 | 10 |
| ged-opt14-strips | 1.0 | 1.0 | 1.0 | 19 |
| gripper | 1.0 | 1.0 | 1.0 | 7 |
| hiking-opt14-strips | 1.0 | 1.0 | 1.0 | 6 |
| logistics00 | 1.0 | 1.0 | 1.0 | 20 |
| miconic | 1.0 | 1.1 | 1.5 | 53 |
| nomystery-opt11-strips | 1.0 | 1.0 | 1.0 | 14 |
| openstacks-opt08-strips | 1.0 | 1.0 | 1.0 | 8 |
| openstacks-strips | 1.8 | 1.8 | 1.8 | 5 |
| pegsol-08-strips | 1.0 | 1.0 | 1.0 | 28 |
| pegsol-opt11-strips | 1.0 | 1.0 | 1.0 | 18 |
| psr-small | 1.9 | 2.3 | 2.7 | 40 |
| rovers | 1.3 | 1.4 | 1.4 | 4 |
| satellite | 1.1 | 1.2 | 1.2 | 4 |
| scanalyzer-08-strips | 1.1 | 1.4 | 1.5 | 11 |
| scanalyzer-opt11-strips | 1.1 | 1.4 | 1.5 | 8 |
| storage | 1.9 | 2.2 | 2.5 | 4 |
| termes-opt18-strips | 1.0 | 1.0 | 1.0 | 8 |
| tpp | 1.0 | 1.0 | 1.0 | 4 |
| transport-opt08-strips | 1.0 | 1.0 | 1.0 | 11 |
| transport-opt11-strips | 1.0 | 1.0 | 1.0 | 6 |
| transport-opt14-strips | 1.0 | 1.0 | 1.0 | 5 |
| trucks-strips | 1.0 | 1.0 | 1.0 | 6 |
| zenotravel | 1.0 | 1.0 | 1.0 | 9 |

Table 5.5: The number of actions existing in the backward task relative to the number of pre-existing actions in the forward task. We list the minimal, average, and maximal number of actions for each domain in the benchmark and additionaly we note the sample size as *N*, which are the problems solved by both A* and NBS.

the number of expansions in the last layer sometimes overshadowed the previous ones by multiple magnitudes. And although this phenomenon was rare, it nullifies the usefulness of the bounded worst case as it does not always hold.

| *Domain* | *min.* | *avg.* | *max.* | *N* |
|---|---|---|---|---|
| blocks | 0.6 | 0.9 | 1.0 | 27 |
| depot | 1.0 | 1.0 | 1.0 | 2 |
| driverlog | 0.2 | 0.9 | 1.0 | 10 |
| elevators-opt08-strips | 0.8 | 1.0 | 1.0 | 13 |
| elevators-opt11-strips | 0.9 | 1.0 | 1.0 | 11 |
| floortile-opt11-strips | 1.0 | 1.0 | 1.0 | 12 |
| ged-opt14-strips | 0.0 | 0.9 | 1.0 | 20 |
| gripper | 1.0 | 1.0 | 1.0 | 7 |
| hiking-opt14-strips | 0.7 | 1.0 | 1.0 | 10 |
| logistics00 | 1.0 | 1.0 | 1.0 | 13 |
| miconic | 0.2 | 0.9 | 1.0 | 50 |
| nomystery-opt11-strips | 0.9 | 1.0 | 1.0 | 10 |
| openstacks-opt08-strips | 0.0 | 0.3 | 0.6 | 11 |
| openstacks-strips | 1.0 | 1.0 | 1.0 | 5 |
| pegsol-08-strips | 0.0 | 0.8 | 1.0 | 28 |
| pegsol-opt11-strips | 0.4 | 0.9 | 1.0 | 18 |
| psr-small | 0.4 | 1.0 | 1.0 | 42 |
| rovers | 0.5 | 0.7 | 1.0 | 5 |
| satellite | 1.0 | 1.0 | 1.0 | 4 |
| scanalyzer-08-strips | 1.0 | 1.0 | 1.0 | 9 |
| scanalyzer-opt11-strips | 1.0 | 1.0 | 1.0 | 6 |
| storage | 0.1 | 0.2 | 0.4 | 5 |
| termes-opt18-strips | 1.0 | 1.0 | 1.0 | 13 |
| tpp | 0.6 | 0.7 | 0.9 | 4 |
| transport-opt08-strips | 0.6 | 0.9 | 1.0 | 11 |
| transport-opt11-strips | 1.0 | 1.0 | 1.0 | 7 |
| transport-opt14-strips | 1.0 | 1.0 | 1.0 | 6 |
| trucks-strips | 1.0 | 1.0 | 1.0 | 6 |
| zenotravel | 0.4 | 0.8 | 1.0 | 8 |

Table 5.6: The table presents the relative number of expansions in the layers preceding the last one compared to the expansions in the last layer. A zero would imply that all expansions were within the last layer, whereas a one would signify that the number of expansions in the last layer were insignificant. The showed number were created by running NBS with $h^{max}$.

### 5.2.2 Summary

The greatest advantage of bidirectional search is its potential reduction of expansions by conducting two searches. Given the results of the experiments, it is evident that bidirectional search can be very performant. But there are a few downsides to using it in planning, which

have to be taken into account. The most influential parameter is the number of secondary initial states. A problem which was expected and is inherently connected to the explicit-state space representation. The benefit of it is a reduced complexity of how to expand states. Hence, it is an act of balancing the benefits and drawbacks. Another smaller negative factor is the non-essential complexity, which is added by how the problems are defined and the algorithm is implemented. In conclusion, the performance of NBS depends on various factors of which the innate structure of the problem instance itself has by far the biggest influence. With this we mean to inquire the aptitude of problem tasks regarding the search direction. To that end, we first investigate whether $p^*$ is a reliable measurement for the structure of the transition system and second, we determine whether there is a causality between certain $p^*$ values and the performance of NBS.

## 5.3 Running fMM

The objective of this experiment is to inquire the innate structure of the search space in planning tasks, which in turn allows to choose a preferred search direction. We use the fMM algorithm with different inputs to get an impression on how the number of expanded states depends on where the two searches meet. Furthermore, we use the COMPUTEWVC algorithm to get the exact value of $p^*$ for each problem. We then investigate how these different parameters are connected. Taking these parts together, we give a convincing argument on how to explain the performance of NBS.

The second experiments were run on the same benchmark as the first one. On each problem we run 12 algorithms combined with the $h^1$, $h^{max}$, $h^m$, or $h^{lmcut}$ heuristic. The COMPUTEWVC algorithm and eleven instances of fMM with the input parameter $p$ evenly distributed from 0 to 1. Each fMM run has the same time and memory constraints as NBS before, namely a run time limit of 30 minutes and a memory limit of 3.5 GB. Allowing for a comparison between experiments. The COMPUTEWVC has the same constraints, with the exception that the conducted A* and reverse A* searches within each have a time limit of thirty minutes. Although we run the different algorithms with multiple heuristics, we mostly focus on the results from $h^{max}$, as it provides the biggest sample size of solved problems and therefore is the most reliable source of information. We especially omit the results from using $h^m$ heuristic, as the number of solved problems does not provide sufficient data to enable a meaningful analysis.

### 5.3.1 Results and Evaluation

The results of this experiment are presented throughout this section in various snippets as there is not enough space for the full volume. But to give an impression, we first list the computed $p^*$ values for each domain, which results in a concise and informative overview, shown in Table 5.7. Second, we present the different fMM runs in domain specific graphs which summarize all different runs of the specific domain. Due to their number, more domain specific fMM graphs using the $h^{max}$ heuristic can be found in the Appendix B.2.

Before going into detail, we want to present an overview over the most important findings

we obtained in these second experiments. First, our hypothesis that the innate structure of a problem instance determines the efficiency of different search directions was validated. Thereby, explaining the performance of NBS on different problems. Second, the distribution of $p^*$ correlates within each domain, suggesting a direct causality. Provided this correlation, it is possible to compute $p^*$ for a simple problem and infer the applicability of NBS, A*, and reverse A* for the whole domain from it. Finally, we were able to confirm the assumptions discussed in the previous experiments.

**Innate Structure** In the discussion about the NBS experiments[9], we hypothesized that the most influential factor for the performance of NBS is the innate structure of the problem. Which would mean that the problem instance defines which search direction will be most efficiently. To make a point in favour of our hypothesis we present an in depth example of the *logistics00* domain. Keep in mind, that although we only present instances of this single domain, they are chosen to exemplify the general mechanism. The same aspects are evident in other domains as well, as can be verified with the Figure B.2 in the the Appendix.

We start our explanation by introducing the *logistics00* domain, a standard planning domain where packages must be distributed to their defined destination. The available transportation options include travelling by truck or plane. Each problem consists of a description of where the trucks, planes, and packages are located initially. Furthermore, it defines how the location are connected with each other and where the packages must be delivered to. In Table 5.8 we list six problem instances of the *logistics00* domain with four corresponding values. First, the number of expansions by A* using $h^{max}$ used to solve the problem. Second, the same number but for the NBS algorithm. Third, the determined $p^*$ for the particular instance. And at last, the *meeting point* of the two searches conducted with the NBS algorithm. The number representing the meeting point denotes at which percentile the frontiers met regarding their $g$-value.

| *Logistics00* | *Ex.: A* (#)* | *Ex.: NBS (#)* | $p^*$ | *Meeting Point* |
|---|---|---|---|---|
| problem 4-0 | 4885 | 4355 | 0.60 | 0.60 |
| problem 4-1 | 4185 | 4223 | 0.63 | 0.58 |
| problem 5-0 | 74′693 | 43′409 | 0.59 | 0.56 |
| problem 5-1 | 6198 | 5387 | 0.65 | 0.59 |
| problem 6-0 | 202′229 | 87′839 | 0.60 | 0.60 |
| problem 6-1 | 3605 | 3951 | 0.64 | 0.57 |

Table 5.8: A small sample of experiment data, including the first six problems of the *logistics00* domain with the respective number of expansion by NBS and A* when solved with $h^{max}$. Additionally, $p^*$ and as comparison the actual meeting point by NBS is shown as well. The number of secondary initial states is 8 for all displayed instances.

The presented data shows that there is a correlation between $p^*$ and the meeting point.

[9] Can be found in Subsection 5.2.2.

| *Domain* | *min.* | *avg.* | *max.* | *st.dev.* | *N* |
|---|---|---|---|---|---|
| blocks | 0.00 | 0.32 | 0.50 | 0.19 | 17 |
| depot | 0.53 | 0.77 | 1.00 | 0.23 | 2 |
| driverlog | 0.47 | 0.64 | 1.00 | 0.19 | 5 |
| elevators-opt08-strips | 1.00 | 1.00 | 1.00 | 0.00 | 1 |
| floortile-opt11-strips | 0.00 | 0.18 | 0.29 | 0.13 | 3 |
| ged-opt14-strips | 0.50 | 0.72 | 1.00 | 0.23 | 13 |
| gripper | 0.41 | 0.45 | 0.49 | 0.03 | 6 |
| hiking-opt14-strips | 1.00 | 1.00 | 1.00 | 0.00 | 6 |
| logistics00 | 0.54 | 0.69 | 1.00 | 0.16 | 10 |
| miconic | 0.00 | 0.80 | 1.00 | 0.30 | 40 |
| nomystery-opt11-strips | 0.60 | 0.93 | 1.00 | 0.15 | 6 |
| openstacks-opt08-strips | 1.00 | 1.00 | 1.00 | 0.00 | 5 |
| openstacks-strips | 0.35 | 0.36 | 0.39 | 0.02 | 5 |
| pegsol-08-strips | 0.50 | 0.86 | 1.00 | 0.23 | 7 |
| pegsol-opt11-strips | 0.67 | 0.67 | 0.67 | 0.00 | 1 |
| psr-small | 0.43 | 0.87 | 1.00 | 0.19 | 41 |
| rovers | 1.00 | 1.00 | 1.00 | 0.00 | 4 |
| satellite | 0.69 | 0.86 | 1.00 | 0.14 | 4 |
| scanalyzer-08-strips | 0.45 | 0.74 | 1.00 | 0.26 | 6 |
| scanalyzer-opt11-strips | 0.45 | 0.64 | 1.00 | 0.26 | 3 |
| storage | 1.00 | 1.00 | 1.00 | 0.00 | 5 |
| termes-opt18-strips | 0.47 | 0.47 | 0.47 | 0.00 | 1 |
| tpp | 1.00 | 1.00 | 1.00 | 0.00 | 4 |
| transport-opt08-strips | 1.00 | 1.00 | 1.00 | 0.00 | 6 |
| transport-opt11-strips | 1.00 | 1.00 | 1.00 | 0.00 | 1 |
| transport-opt14-strips | 1.00 | 1.00 | 1.00 | 0.00 | 1 |
| trucks-strips | 1.00 | 1.00 | 1.00 | 0.00 | 2 |
| zenotravel | 0.00 | 0.77 | 1.00 | 0.37 | 6 |

Table 5.7: The minimum, average, maximum, standard deviation, and sample size of $p^*$ for each domain in the benchmark. The last column lists the sample size, e.g. the number of problems where the COMPUTEWVC algorithm using $h^{max}$ successfully computed $p^*$ whithin the time and memory constraints. Omitted in the list are domains with a sample size of zero.

This makes intuitively sense, because $p^*$ is influenced by how efficient the forward and backward searches are to enable `fMM` to be optimal when meeting at this point. Meanwhile, in `NBS` the searches are not manually guided to meet at a certain depth, but are given the same amount of expansions per search, hence it is logical that the search frontiers meet at a similar position as $p^*$ suggests. In a reverse conclusion, we can surmise that $p^*$ provides a rough estimation on where the search frontiers will meet in the `NBS` algorithm. Studying

the number of expansion presented in Table 5.8, we infer that there appears to be no visible correlation between the performance of A* and NBS in regards to $p^*$. To gain deeper insight into the structure of the problem instances, we show the different number of expansion by fMM with varying input $p$ for the same problem instances in Figure 5.1. Analysing the first two graphs for instance 4-0 and 4-1, it is evident that although $p^*$ is nearly equivalent, the behaviour around the optimum is different. For instance, the number of expansions does nearly reach 5000 in problem instance 4-0 with $p$ close to 1. Whereas in instance 4-1, the number of expansions is barely higher than 4000. This difference is clearly reflected in the experiments with A* and NBS, where NBS performs similar in both instances but A* is significantly less performant in instance 4-0.

Before drawing a conclusion, we discuss how the used algorithms are related to each other. fMM with $p = 1$ is a generalized version of A*. However, as fMM is a bidirectional algorithm, it cannot use the same tie-breaking rule as A*. In unidirectional search, it is obvious that when choosing states to expand it is favourable to prefer states with the same $f$-value but higher $g$-value, as they must be closer to a goal state. However, in bidirectional search, the objective is not to reach the goal, but for the frontiers to intersect, therefore there is no intuitively superior tie-breaking rule. As a result, A* expands only few more states than |VC| when $p^* = 1$. Whereas, fMM may expand arbitrarily many states in the last layer. Because of this, we always present the number of expansion preceding the last layer, when showing the fMM algorithms.

With the information from the previous paragraph in mind, we deduce that the performance of different algorithms can be predicted and explained by the underlying innate structure of the problem instance. The innate structure in turn can be inferred from the number of expansions by fMM. Furthermore, the smoothness of the presented graphs, enables the use of $p^*$ as a general measurement of the performance of fMM with different input. In conclusion, $p^*$ gives a meaningful estimate of the performance of fMM, thus contains information regarding the innate structure of a problem instance, and therefore can be used to determine which search algorithm should be applied.

**Similarities** In the previous paragraph, we showed that the suitability regarding search directions is determined by the innate structure a problem instance exhibits. And although this is an interesting property, it cannot be used a priori, thus it is primarily usable as an analysis tool. But interestingly enough, our experiments show that instances of the same domain share similar structures. As can be seen in Table 5.7, each domain has at least some bias for one search direction. Unfortunately for bidirectional search, the bias most often favours unidirectional forward search. Which is an effect we attribute to the number of secondary initial states. Generating those more meticulously could leverage backward search and with it bidirectional efficiency. But even without such improvements, domains like *block*, *gripper*, and so forth are already suited for bidirectional search and some domains like *floortile-opt11-strips* show a clear aptitude for backward search. A few different examples for the structure of the search space are shown in Figure 5.2 and moreover, we discuss the reasoning behind similar structures in the next Section 5.3.2. Most importantly, having a bias based on the domain is a very interesting property. As it enables us to gather

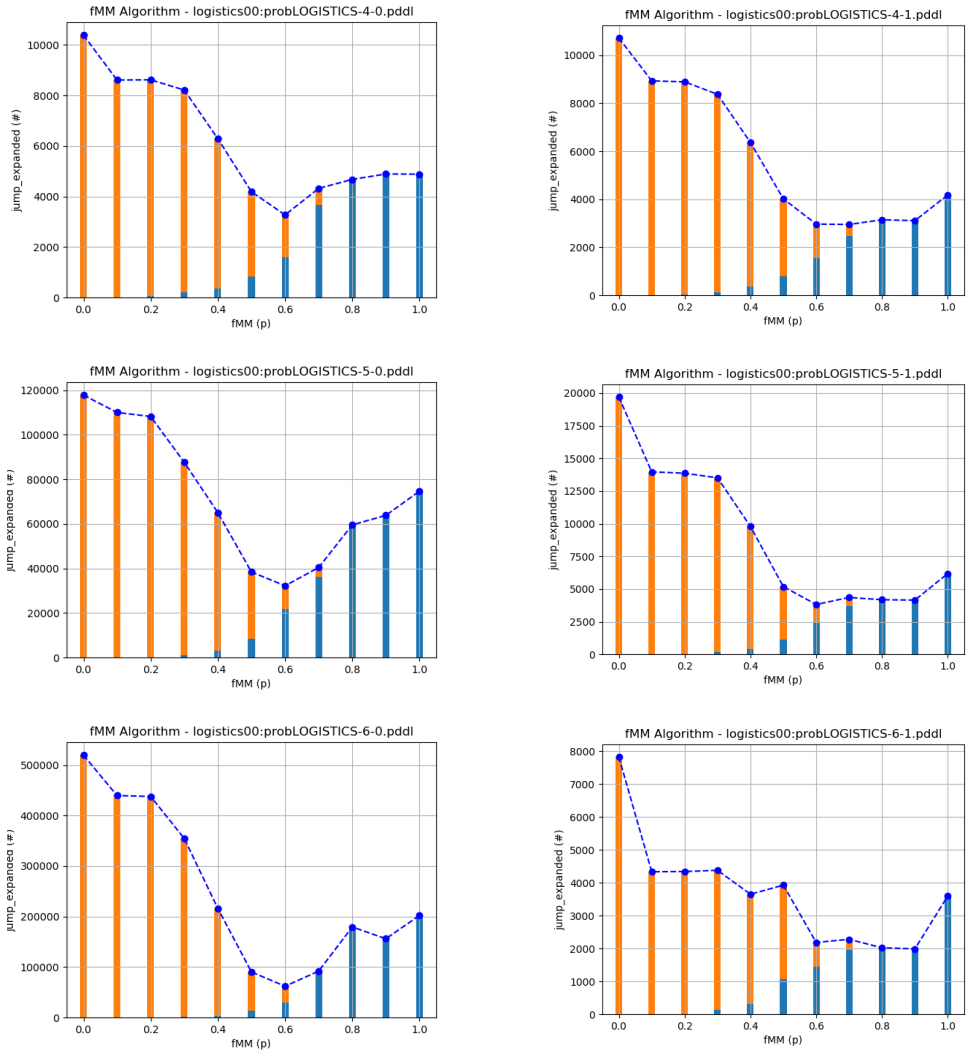



Figure 5.1: The number of expanded states preceding the final f-layer by `fMM` using $h^{max}$ with equidistant distributed $p$ values on the problems 4-0, 4-1, 5-0, 5-1, 6-0, and 6-1. The color denotes from which search each state is expanded. Orange for backward and blue for forward direction.

information on how to solve a problem instance, based on other instances of the same domain. Therefore, information about an expensive problem can be inferred from simpler problem instances, which could be used as an a-priori tool to decide upon the algorithmic approach.

**Revisit `NBS` Experiment** In the previous experiments about `NBS` we provided various hypothesis to explain the behaviourism and results of `NBS`. (1) First, we argued that a high number of secondary initial states impairs bidirectional search to the point where only the forward search portion achieves significant progress. Unless, there is a heuristic which is able to discern the promising states. (2) Second, we claimed that the number of secondary initial states is the main parameter influencing the search. Thus, problems with many secondary

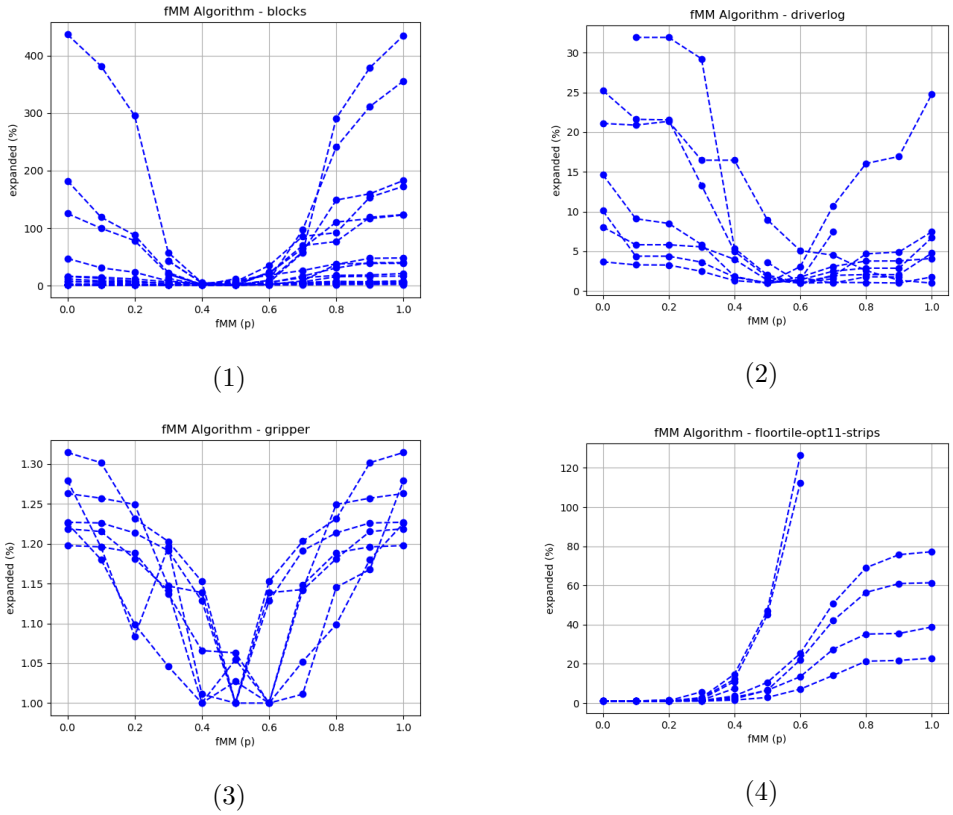



Figure 5.2: The number of expanded states before the last layer by the `fMM` algorithm using $h^{max}$ with different input $p$. The illustrated domains are (1) *blocks*, (2) *driverlog*, (3) *floortile*, and (4) *gripper*.

initial states have a $p^*$ shifted closer to 1. (3) Third, heuristics improve the search efficiency, regardless of its direction. Therefore, heuristics should not favour a specific algorithm.

- (1) To show that heuristics are able to discern many secondary initial states and therefore alleviate the problem with it, we presented the first two problem instances of the *depot* domain in Figure 5.3. As is clearly evident, the $h^{max}$ heuristic improves the backward search very significantly, far below the number of secondary initial states present in the problem instance.

- (2) Considering the previous example, we showed that the heuristic may alleviate the problem of secondary initial states significantly. Hence, the number of secondary initial states is a influencing factor, but the underlying innate structure of a problem instance seems to be the most defining factor.

- (3) We claimed that it is possible to analyse the performance of `NBS` and A* independent of the applied heuristic. Although we were not able to proof our claim, the gathered data supports it. Furthermore, we can reason about the influence they exert. Heuristics have different effects on different problems and even the backward search and the forward search of a problem instance are not identical. Therefore, heuristics have different behaviour and accuracy between problems and search directions. This

reinforces the previous point about $h^{max}$ improving the backward search more than the forward search. As the backward search contains more states, it is logical that heuristics might be able to discern more states and therefore improve the searches unevenly. But although this difference persists, the experiment data suggests that in most cases, the effect evens itself out when observed over the whole problem. A representative example is shown in Figure 5.4. It shows that the general bias of the problem instances is indifferent to the change in heuristics.

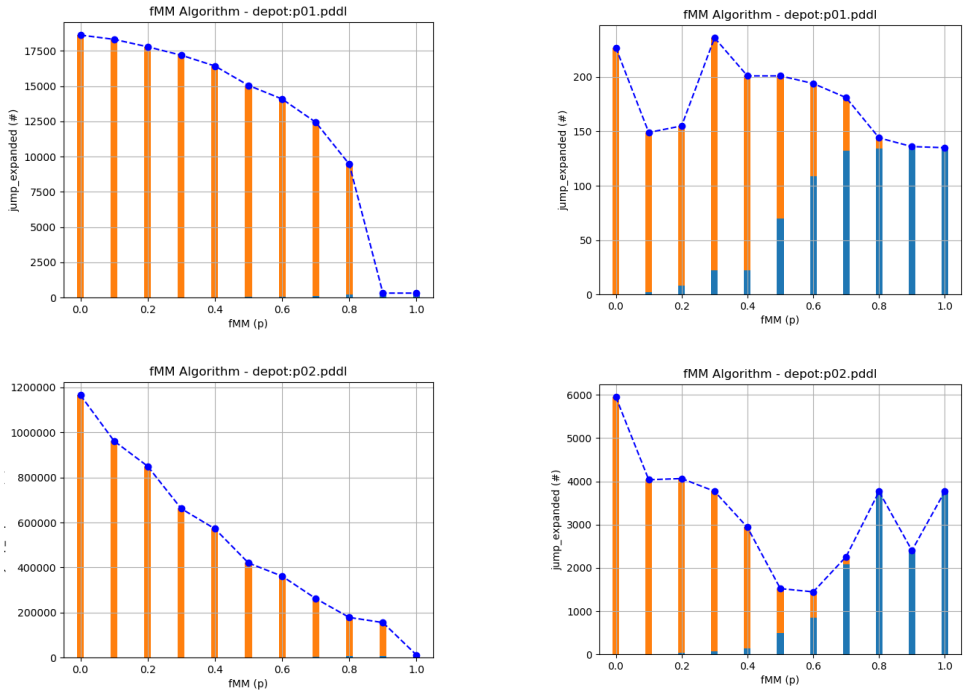
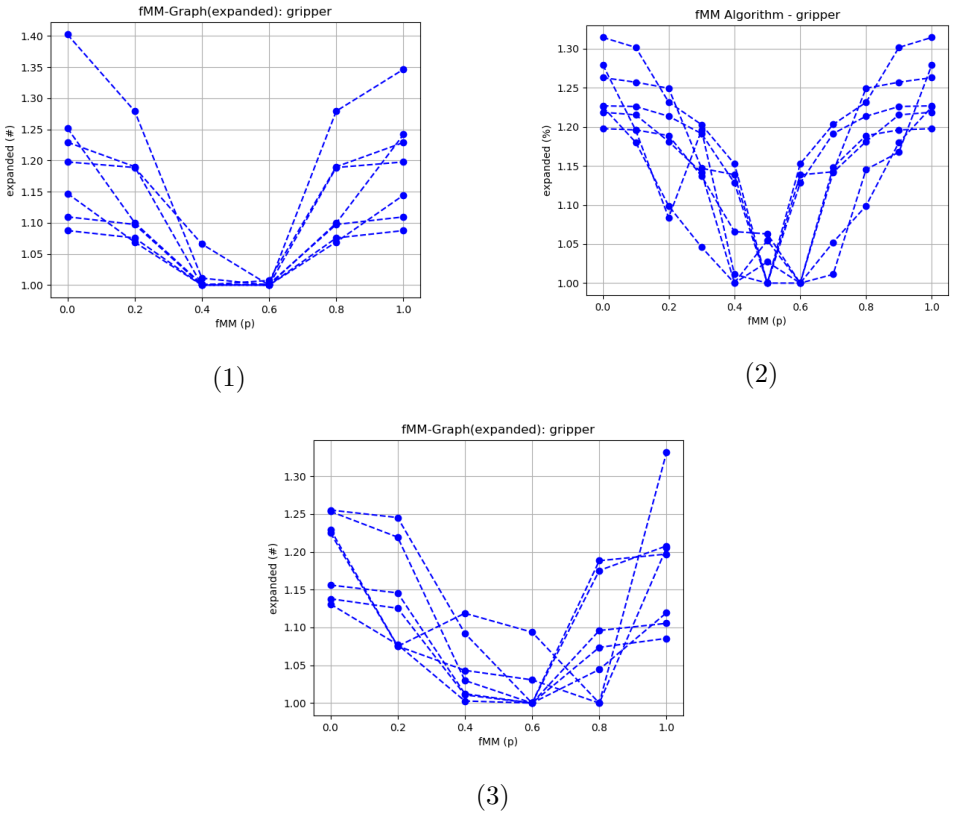



Figure 5.3: The number of expanded states before the last layer by the `fMM` algorithm with different input $p$. (Left) `fMM` using the blind heuristic. (Right) `fMM` using the $h^{max}$ heuristic.




(1) (2)

(3)

Figure 5.4: The number of expansions by `fMM` over the whole *gripper* domain. Each line connects the same problem instance with various $p$ parameter. The `fMM` algorithm uses a different heuristic in each graph: (1) $h^1$, (2) $h^{max}$, (3) $h^{lmcut}$.

### 5.3.2 Analysing the Innate Structure

In Section 5.3.1, we discuss how the performance of NBS is defined by the underlying structure of the induced transition system. Based on the data we gathered in the experiments, we remark a correlation between the bias expressed by a problem task regarding the search direction and the domain it belongs to. However, we do not go into much detail in respect to an intuitive explanation to justify our findings. In this section, we want to add to that discussion and thereby provide a reasoning why problems of the same domain express a similar bias. To that end, we introduce the *blocks* domain and analyse its various behaviourism in a case study.

In the *blocks* domain the objective is to arrange a number of blocks to match a given configuration. Each problem in the domain defines a number of available blocks and the initial configuration as well as the goal configuration of those blocks. Every block is marked alphabetically and is put on the ground or stacked on top of another specified block. A block with no other blocks on top of it is denoted as clear. Finally, we have a gripper which can grab a clear block and put it either on another clear block or on the ground.

**Case Study** Presented in Figure 5.5 to 5.7 are problem instances of the *blocks* domain[10]. For each instance, we illustrate the initial configuration as well as the goal configuration. Additionally, on the right side is the corresponding fMM graph shown. Note that there is an apparent correlation between the initial state structure and the efficiency of the backward search. If the blocks are stacked as a single column, then a backward search would be very expensive. On the other hand, if all blocks are placed directly on the table, then the backward search is nearly optimal. We expect that the same relation holds true for the forward search and the goal configuration. However, in practice the goal is always the same, namely to stack the blocks with some order, which explains why the *blocks* domain is very suited for bidirectional or backward search.

The explanation for the relation between efficiency of a particular search and the specific initial and goal state can be found in the transition system. Consider the problem instance 4-0. In the initial state, we could move each of the blocks on top of another one, resulting in a total of $3 \cdot 4$ possible options, whereas in instance 4-1, we can only put the uppermost block on the table. In other words, the branching factor is higher when the blocks have to be stacked. This relates directly to the Figures in 5.5 to B.30 where stacked initial configurations are unsuited for backward search, unstacked configurations are suited for bidirectional search, and a combination of the two shows mixed results.

In conclusion, on basis of experiments, we hypothesize that the similar innate structure which defines the efficiency of the search directions is justified by the similarity of how the problems are created. Provided with this information, we could determine the most efficient search algorithm from our collection for any problem of the *blocks* domain based solely on its description.

[10] Additional Figures can be found in Appendix B.3.

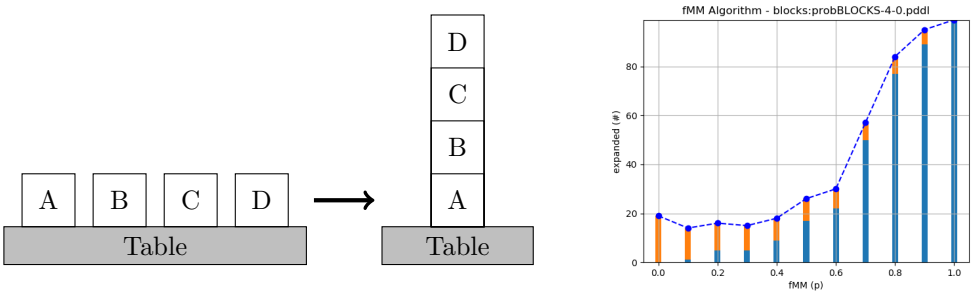



Figure 5.5: The problem 4-0 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

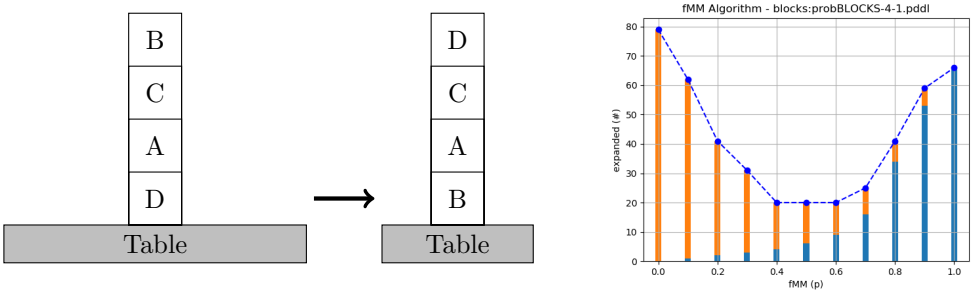



Figure 5.6: The problem 4-1 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

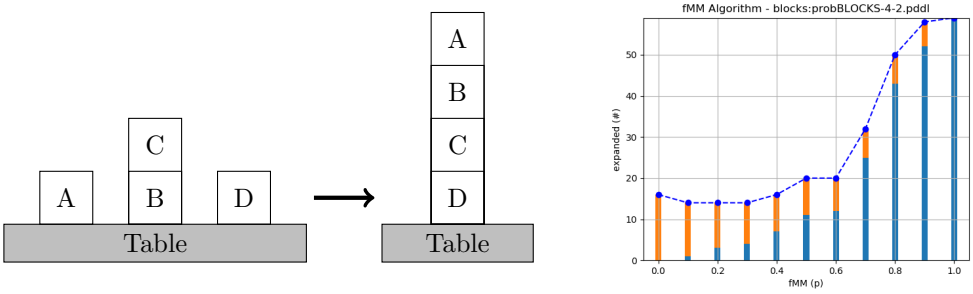



Figure 5.7: The problem 4-2 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

### 5.3.3 Summary

In conclusion of the second experiments, we run the algorithms fMM and COMPUTEWVC in order to determine practical attributes of NBS and the problem instances. We found that the implicit state space defined by a problem instance is the most defining aspect when looking at the efficiency of different search directions. We further observed that different problems of the same domain share the same structure to a certain extend, hence, display the same bias for or against a particular search approach. Finally, we provide an overview of which domains display a certain bias in Table 5.9.

| *Domain* | $h^1$ | | $h^{max}$ | | $h^{lmcut}$ | |
|---|---|---|---|---|---|---|
| | $p^*$ | *NBS* | $p^*$ | *NBS* | $p^*$ | *NBS* |
| blocks | 0.37 | 18 | 0.32 | 21 | 0.42 | 15 |
| depot | 1.00 | 0 | 0.77 | 1 | 1.00 | 0 |
| driverlog | 0.65 | 6 | 0.64 | 8 | 0.75 | 1 |
| elevators-opt08-strips | 0.86 | 8 | 1.00 | 5 | - | 0 |
| elevators-opt11-strips | - | 7 | - | 5 | - | 0 |
| floortile-opt11-strips | - | 2 | 0.18 | 6 | 0.00 | 5 |
| ged-opt14-strips | 0.90 | 10 | 0.72 | 10 | 0.72 | 10 |
| gripper | 0.45 | 7 | 0.45 | 6 | 0.68 | 4 |
| hiking-opt14-strips | 1.00 | 3 | 1.00 | 0 | 1.00 | 0 |
| logistics00 | 0.59 | 10 | 0.69 | 8 | 1.00 | 0 |
| miconic | 0.68 | 31 | 0.80 | 21 | 0.89 | 0 |
| nomystery-opt11-strips | 0.67 | 8 | 0.93 | 4 | 1.00 | 0 |
| openstacks-opt08-strips | 1.00 | 0 | 1.00 | 4 | 1.00 | 3 |
| openstacks-strips | 0.00 | 0 | 0.36 | 5 | 1.00 | 0 |
| pegsol-08-strips | 0.69 | 20 | 0.86 | 14 | 0.93 | 10 |
| pegsol-opt11-strips | 0.67 | 14 | 0.67 | 11 | 0.67 | 9 |
| psr-small | 1.00 | 0 | 0.87 | 11 | 0.89 | 7 |
| rovers | 1.00 | 0 | 1.00 | 0 | 1.00 | 0 |
| satellite | 0.90 | 2 | 0.86 | 1 | 1.00 | 0 |
| scanalyzer-08-strips | 0.47 | 12 | 0.74 | 6 | 0.57 | 3 |
| scanalyzer-opt11-strips | 0.46 | 9 | 0.64 | 5 | 0.50 | 3 |
| storage | 1.00 | 0 | 1.00 | 0 | 1.00 | 0 |
| termes-opt18-strips | 0.46 | 10 | 0.47 | 10 | 0.47 | 6 |
| tpp | 0.00 | 0 | 1.00 | 0 | 1.00 | 0 |
| transport-opt08-strips | 0.99 | 5 | 1.00 | 0 | 1.00 | 0 |
| transport-opt11-strips | 0.94 | 5 | 1.00 | 0 | - | 0 |
| transport-opt14-strips | 0.83 | 5 | 1.00 | 1 | 1.00 | 0 |
| trucks-strips | 1.00 | 0 | 1.00 | 0 | 1.00 | 0 |
| zenotravel | 0.83 | 7 | 0.77 | 2 | 0.86 | 0 |

Table 5.9: The table presents an overview over all domains regarding their aptitude towards bidirectional search. For each heuristic, we list the average $p^*$ and the number of problems where NBS expanded less states than A*. The second number has to be interpreted with care, as it is an absolute number and not relative to the total number of solved problems.

# 6
# Conclusion

In this thesis we investigated the practicality of using bidirectional search with explicit-state space representation, specifically NBS algorithm, to solve planning problems. The work is inspired by the recent introduction of a promising bidirectional search algorithm by Chen et al. [2017]. Apart from implementing the NBS algorithm in the state-of-the-art planner Fast-Downward, we run experiments with the fMM algorithm [Holte et al., 2016] to evaluate the displayed performance of NBS. With the experiments, we were able to find three important properties.

1. Our implementation of NBS was able to successfully outperform A* on some problem instances.

2. Whether A* or NBS is more efficient depends mainly on the implicit structure of the search space defined by the problem instance.

3. The predisposition towards certain search approaches is shared by problems of the same domain.

Smaller findings include that heuristic functions are not significantly different for unidirectional search and bidirectional search, that the worst case bound of NBS is invalid as the forward and backward search are not run on an identical state space, and that a sophisticated method to derive the backward task could improve bidirectional search significantly.

Regarding the performance of NBS on the international planning competition benchmark, we inferred a few helpful properties. First, simple domains are more intuitive to reverse as there are less complex actions and variable combinations to consider. Second, clearly defined goal conditions are very important to keep the number of secondary initial states low. Third, symmetries can be very problematic if they are not distinguishable by the heuristic. Combining these properties, we found that the underlying transition system varies between domains and that all of those properties have an additional influence on the resulting performance. In particular, the different NBS runs with the *blocks*, *floortile-opt11-strips*, *ged-opt14-strips*, and *termes-opt18-strips* problem instances yielded the best results compared to A*.

To put our work into perspective with contemporary research, we list the relevance of our work compared to others and discuss potential future work. We achieved to use bidirectional search successfully for planning. Aside from SymBA*, which takes a fundamentally different approach, this is a novelty and shows that the promise linked to the initial idea of bidirectional search is valid. With a refined algorithm such as NBS, we can outperform A* on certain problems. Together with the uniformity of predisposition within a domain, we built the foundation of using NBS in combination with COMPUTEWVC in portfolio planners. The displayed efficiency of NBS further promotes the research of how to apply bidirectional search in planning. Especially interesting are the question of how to eliminate the encountered non-essential complexity and how to reduce the number of secondary initial states. For instance, the secondary initial states are constructed by going over all permutations of variable assignment combinations, which creates symmetries. Hence, contemporary research of how to handle symmetries might provide methods to reduce the number of secondary initial states effectively. Future work might include more experiments comparing NBS algorithm to different algorithm like SymBA* and reverse A* with the use of different heuristics.

In conclusion, NBS is a bidirectional search algorithm which can be used to efficiently solve planning problems. Its performance depends on factors like the innate structure of the problem instance or how efficiently the backward search is designed. Especially the number of secondary initial states is a limiting factor. However, this can potentially be significantly improved in future work. Furthermore, the innate structure of a problem instance can be anticipated by analysing other problems of the same domain by using COMPUTEWVC. This could be a powerful tool for a portfolio planner.

# Bibliography


Bäckström, C. and Nebel, B. (1995). Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655.

Barker, J. K. and Korf, R. E. (2015). Limitations of front-to-end bidirectional heuristic search. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 1086–1092. AAAI Press.

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

Brooks, Jr., F. P. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19.

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

Chen, J., Holte, R. C., Zilles, S., and Sturtevant, N. (2017). Front-to-end bidirectional heuristic search with near-optimal node expansions. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pages 489–495. AAAI Press.

De Champeaux, D. (1983). Bidirectional heuristic search again. *Journal of the ACM (JACM)*, 30(1):22–32.

Dechter, R. and Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, 32(3):505–536.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Eckerle, J., Chen, J., Zilles, S., and Holte, R. C. (2017). Sufficient conditions for node expansion in bidirectional heuristic search. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pages 79–87. AAAI Press.

Edelkamp, S. (2001). Planning with pattern databases. In *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, pages 13–24.

Edelkamp, S. and Schrödl, S. (2011). *Heuristic Search: Theory and Applicaitons*. Elsevier.

Felner, A., Moldenhauer, C., Sturtevant, N., and Schaeffer, J. (2010). Single-frontier bidirectional search. In *Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 59–64. AAAI Press.

Ghallab, M., Nau, N., and Traverso, P. (2004). *Automated Planning: Theory & Practice.* 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.

Haslum, P. and Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, pages 140–149. AAAI Press.

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

Helmert, M. (2010). Landmark heuristics for the pancake problem. In *Third Annual Symposium on Combinatorial Search (SoCS 2016)*, pages 109–110.

Helmert, M. and Domshlak, C. (2009). Landmarks, critical paths and abstractions: what's the difference anyway? In *Proceedings of the Nine-Teenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169. AAAI Press.

Holte, R. C., Felner, A., Sharon, G., and Sturtevant, N. (2016). Bidirectional search that is guaranteed to meet in the middle. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, pages 3411–3417. AAAI Press.

Hopcroft, J. E. and Karp, R. M. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231.

Kaindl, H. and Kainz, G. (1997). Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research (JAIR)*, 7:283–317.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

Kwa, J. B. H. (1989). BS*: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38(1):95–109.

Manzini, G. (1995). BIDA*: an improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360.

Nicholson, T. A. J. (1966). Finding the shortest route between two points in a network. *The Computer Journal*, 9(3):275–280.

Nilsson, N. J. (1980). *Principles of artificial intelligence.* Springer.

Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial optimization: algorithms and complexity.* Courier Corporation.

Pohl, I. (1969). *Bi-directional and Heuristic Search in Path Problems*. PhD thesis, Stanford University.

Pohl, I. (1971). Bi-directional search. *Machine Intelligence*, 6:127–149.

Seipp, J., Pommerening, F., Sievers, S., and Helmert, M. (2017). Downward Lab. https://doi.org/10.5281/zenodo.790461.

Shaham, E., Felner, A., Chen, J., and Sturtevant, N. (2017). The minimal set of states that must be expanded in a front-to-end bidirectional search. In *Proceedings of the Tenth International Symposium on Combinatorial Search (SoCS 2017)*, pages 82–90. AAAI Press.

Shaham, E., Felner, A., Sturtevant, N., and Rosenschein, J. S. (2018). Minimizing node expansions in bidirectional search with consistent heuristics. In *Proceedings of the Eleventh International Symposium on Combinatorial Search (SoCS 2018)*, pages 81–89. AAAI Press.

Sharon, G., Holte, R. C., Felner, A., and Sturtevant, N. (2016). An improved priority function for bidirectional heuristic search. In *Ninth Annual Symposium on Combinatorial Search (SoCS 2016)*, pages 139–140. AAAI Press.

Sturtevant, N. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144–148.

Sturtevant, N. and Felner, A. (2018). A brief history and recent achievements in bidirectional search. In *Proceedings of the 32th AAAI Conference on Artificial Intelligence (AAAI 2018)*, pages 8000–8006. AAAI Press.

Torralba, A., López, C. L., and Borrajo, D. (2016). Abstraction heuristics for symbolic bidirectional search. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 3272–3278. AAAI Press.

Vallati, M., Chrpa, L., Grześ, M., McCluskey, T. L., Roberts, M., Sanner, S., et al. (2015). The 2014 international planning competition: Progress and trends. *Ai Magazine*, 36(3):90–98.

# A Examples

## A.1 From the Problem Task to VC

This section should provide a supplement to the theory discussed in Section 3.2.1. We present a simple planning problem and show over multiple steps how to derive the lower-bound |VC| and the perfect fraction $p^*$. The steps go along with adequate explanations and special focus on how the $G_{MX}$- and $\hat{G}$-graph look in practice.

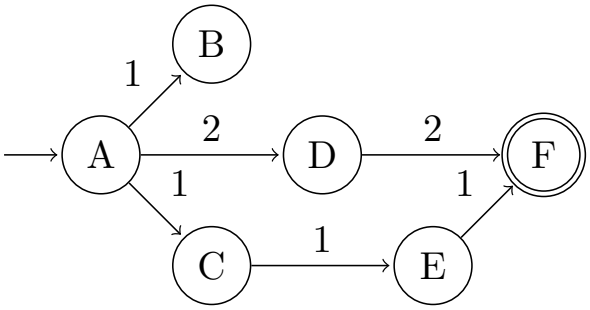



Figure A.1: Simple path-finding problem example.

**Path-finding problem** The problem shown in Figure A.1 depicts a simple path-finding problem, where cities are connected by routes. It starts in city A and aims to reach city F. It is a optimization problem, as the objective is to find the optimal path, e.g. the path with minimal cost. For this example we use the artificial heuristic shown in the following table.

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
|---|---|---|---|---|---|---|
| $h_F$ | 2 | 4 | 1 | 0 | 0 | 0 |
| $h_B$ | 0 | 1 | 0 | 0 | 1 | 2 |

**Must-Expand Graph $G_{MX}$** To derive the corresponding must-expand graph, a bipartite graph is created with all possible states on either side. On the left side, are forward expanded states, analogously, backward expanded states are on the right side. Two opposite states

$u$ and $v$ are joined by an edge iff $lb(u,v) \leq C^*$ holds[11]. The resulting $G_{MX}$, with $g$- and $f$-value for all states, is shown in Figure A.2. Important to mention is that even though all states are visualized in the shown $G_{MX}$ representation, only states which have at least one connection are actually included in $G_{MX}$.

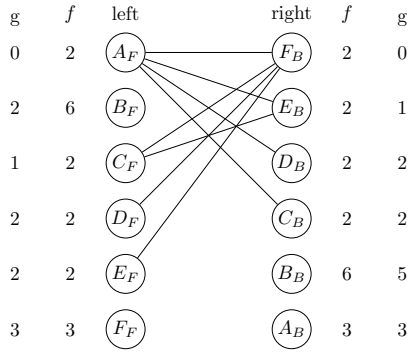



Figure A.2: The corresponding $G_{MX}$.

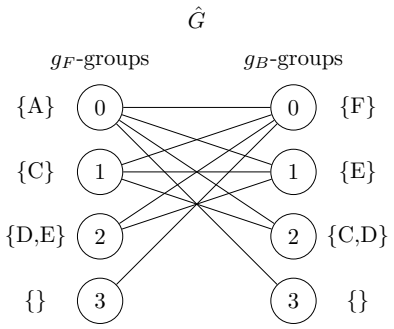



Figure A.3: The corresponding $\hat{G}$.

**The Abstract Graph $\hat{G}$** The abstraction resulting from grouping the states included in $G_{MX}$ with equal $g$-value, can be created by using $G_{MX}$ explicitly as basis or implicitly from the number of states with the same $g$-value encountered when running A* in either direction. The abstraction of the must-expand graph shown in Figure A.2 is presented in Figure A.3. A notable property of $\hat{G}$ is that all states included in the groups have an $f$-value smaller than $C^*$ and all edges join two groups with summed $g$-value smaller than $C^*$. Thus, all states belonging to the same group can be combined to build must-expand pairs with each state of the other group. Moreover, Shaham et al. [2017] proved that if one state of specific $g$-value is included in VC, then all other states with equal $g$-value and expansion direction must be elements of VC as well. Hence, the minimal weighted vertex cover of $\hat{G}$ is equivalent with the minimal vertex cover of $G_{MX}$.

**VC and $p^*$** In a last step, interesting properties like |VC|, $p^*$, and $\underline{i}$ can be computed from $\hat{G}$. Given the constraints discussed in Section 3.2.1, we can deduce that only a number of possible weighted vertex cover exist, which is equal to $C^* + 1$. E.g. in the presented example $C^*$ is 3, therefore the loop variable $i \in \{0, 1, 2, 3\}$. As a reminder, the weighted vertex cover with $i = 2$ defines that all included $g_F$-groups have a $g_f$-value smaller than 2 and all included $g_B$-groups have a $g_B$ value smaller than $j$, which is 1.

Concluding, the minimal weighted vertex cover for the presented problem are created with $i \in \{1, 2\}$ and $j \in \{2, 1\}$. The corresponding $p^* \in \{\frac{1}{3}, \frac{2}{3}\}$ and VC = 3. Inserting this knowledge into the initial problem, we can say that to prove the optimality of a found solution with cost = 3, we have to expand at least state A, F, and C or E. Given those states, it is clear that no solution can be found with a cost less than 3, therefore, the found solution is optimal. Furthermore, for bidirectional search $p^*$ is of high importance, as depending on $p^*$ different search tactics are more favourable.

[11] $lb(u,v) = \max\{f_F(u), f_B(v), g_F(u) + g_B(v)\}$

## A.2 Single Goal Experiment

One of the main influential factors regarding the performance of the NBS algorithm is the number of secondary initial states. We discussed in Section 4.1.2 how they occur and what impact they potentially have. To test those theories, we devised an additional experiment which we omitted from Chapter 4. The experiment is identical to the initial NBS experiment in Section 5.2 with the exception that we modify the custom benchmark. We alter a number of problem instances by narrowing the goal conditions to include only a single state. The plan is to measure the potential improvement of reducing the complexity stemming form the secondary initial states. In this section, we want to discuss the results of this experiment and comment on why it is not included in the main part of the thesis.

**Case Study: Driverlog** To explain the results of the single goal experiments, we make an in depth example of the first instance in the *driverlog* domain[12]. The first instance includes six objects: two drivers $d_1$ and $d_2$, two trucks $t_1$ and $t_2$, and two packages $p_1$ and $p_2$. All the objects can be moved between the locations $l_1$, $l_2$, and $l_3$. The drivers are additionally able to move to the paths $p_{1\text{-}2}$ and $p_{2\text{-}3}$. The locations and paths are connected as depicted in Figure A.4. The initial location of the objects is shown by the boxes beside the location. Whereas a dashed box indicates that the goal conditions. In this first instance, the optimal path is to walk driver $d_1$ from $l_3$ to $l_1$ using the available paths. There he enters truck $t_1$ and drives it to $l_2$ where he disembarks again. These seven actions conclude the optimal solution.

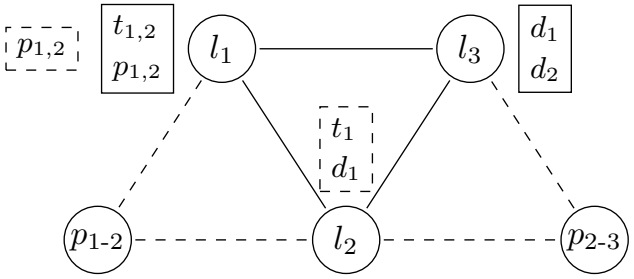



Figure A.4: The figure depicts the first instance of the *driverlog* domain. Locations are marked as circle, streets as unbroken lines, and paths as dotted lines. The initial and goal configuration is indicated by the labels in the boxes.

There are 84 states which satisfy the goal conditions. All of these state have a defined position for the driver $d_1$, truck $t_1$, and both packages. The only objects omitted are the second driver and second truck. To get an understanding from where those 84 states origin, we have to consider how Fast-Downward represents the problem internally. In this instance, Fast-Downward additionally stores a boolean variable for each truck, which defines whether the truck is being driven at the moment. As a result, we have 4 undefined variables:

- $t_2$-*at*: The position of the second truck. The domain of the variable includes the three possible locations.

[12] A introduction of the *driverlog* domain can be found in Section 4.1.1.

- $d_2$-*at*: The position of the second driver. The driver can either be driving one of the trucks, stand at one of the locations, or be walking on a path. Resulting in a domain size of 7.
- $t_1$-*empty*: This boolean variable denotes whether $t_1$ is currently being driven.
- $t_2$-*empty*: The corresponding boolean variable for truck $t_2$.

The number of possible combinatorial permutations with these four variables is 84. From those states, only 21 are legal states. The other 63 states are illegal. The mutexes derived by Fast-Downward allow us to detect 12 of those states, which results in 51 illegal states which are not discarded. However, as we discussed in Paragraph 4.2.1, illegal states are always dead ends. And in fact, the $h^{max}$ heuristic recognizes all of them as dead ends. In the end, only the 21 legal goal states are being expanded and considered by our implementation of the NBS algorithm, but still the algorithm computes the heuristic value of all non-discarded illegal states.
To summarize, we discussed how many essential secondary initial states there are in the first instance of the *driverlog* domain and where they come from. In the next paragraph, we present the results of the single goal experiments and explain what effects occur when the secondary initial states are reduced to a single state.

**Results and Evaluation** As expected, the reduced number of secondary initial states improves the bidirectional search significantly. In Table A.1 and A.2 we list detailed number for the first six instances of the *driverlog* domain. Table A.1 lists the number of expansions using a single secondary initial state and Table A.2 shows the result from the NBS experiment. One unexpected result is the apparent difference in expansions by A*. Intuitively, A* should not be affected by a change in goal states. One explanation to the contrary is that the computed heuristic changes when the goal is more constricted. For instance, if it is given that at the end all trucks must be empty, than entering one may be regarded as an unfavourable action, which would result in a wider exploration of the state space. On the contrary, knowing that the second truck does not move from its initial position, enables the search to only focus on moving the first truck. While this experiment proves that reducing the number of secondary initial states affects the search mostly positively, we cannot argue about the magnitude as there are various unmeasurable factors at play at once. For those reasons, we omitted this experiment from the main part of the thesis.

| Driverlog | *Ex.: A** | *Ex.: NBS* | *MP* | $p^*$ | *SIS* | $C^*$ |
|---|---|---|---|---|---|---|
| problem 01 | | | | | 1 | 7 |
| $h^1$ | 177 | 91 | ~0.43 | 0.57 | | |
| $h^{max}$ | 11 | 28 | ~0.71 | 1.00 | | |
| $h^{lmcut}$ | 8 | 13 | ~0.86 | 1.00 | | |
| problem 02 | | | | | 1 | 19 |
| $h^1$ | 68'751 | 10'305 | ~0.47 | ~0.47 | | |
| $h^{max}$ | 54'281 | 10'307 | ~0.47 | ~0.47 | | |
| $h^{lmcut}$ | 8026 | 7685 | ~0.47 | 0.00 | | |
| problem 03 | | | | | 1 | 12 |
| $h^1$ | 15'090 | 1287 | 0.50 | ~0.58 | | |
| $h^{max}$ | 2461 | 1347 | ~0.58 | ~0.58 | | |
| $h^{lmcut}$ | 24 | 179 | ~0.67 | 1.00 | | |
| problem 04 | | | | | 1 | 16 |
| $h^1$ | 1'137'000 | 25'957 | 0.50 | 0.50 | | |
| $h^{max}$ | 393'674 | 25'957 | 0.50 | 0.50 | | |
| $h^{lmcut}$ | 1726 | 1437 | ~0.38 | 0.00 | | |
| problem 05 | | | | | 1 | 18 |
| $h^1$ | 5'505'872 | 101'805 | 0.50 | 0.50 | | |
| $h^{max}$ | 1'707'212 | 101'805 | ~0.56 | 0.50 | | |
| $h^{lmcut}$ | 329 | 3353 | ~0.67 | 1.00 | | |
| problem 6 | | | | | 1 | 11 |
| $h^1$ | 887'408 | 7799 | ~0.45 | ~0.45 | | |
| $h^{max}$ | 53'778 | 6814 | ~0.45 | ~0.45 | | |
| $h^{lmcut}$ | 96 | 542 | ~0.63 | 1.00 | | |

Table A.1: The table includes the relevant information regarding the first six problem instances of the *driverlog* domain which are modified to define a single goal state. First, the secondary initial states (SIS) and corresponding $C^*$ is shown for each instance . Second, for each applied heuristic, we list the number of expansions by A*, expansions by NBS, the meeting point (MP) of the search frontiers of NBS, and $p^*$.

| Driverlog | *Ex.: A** | *Ex.: NBS* | *MP* | $p^*$ | *SIS* | $C^*$ |
|---|---|---|---|---|---|---|
| problem 01 | | | | | 21 | 7 |
| $h^1$ | 190 | 375 | ~0.86 | 1.00 | | |
| $h^{max}$ | 22 | 93 | ~0.71 | 1.00 | | |
| $h^{lmcut}$ | 9 | 19 | ~0.86 | 1.00 | | |
| problem 02 | | | | | 1 | 19 |
| $h^1$ | 68'751 | 18'349 | ~0.52 | ~0.53 | | |
| $h^{max}$ | 54'281 | 10'305 | ~0.47 | ~0.47 | | |
| $h^{lmcut}$ | 8103 | 7698 | ~0.47 | 0.00 | | |
| problem 03 | | | | | 8 | 12 |
| $h^1$ | 16'918 | 2893 | ~0.67 | ~0.67 | | |
| $h^{max}$ | 2500 | 2437 | ~0.58 | ~0.67 | | |
| $h^{lmcut}$ | 24 | 213 | 0.75 | 1.00 | | |
| problem 04 | | | | | 1 | 16 |
| $h^1$ | 1'134'510 | 35'977 | 0.50 | 0.50 | | |
| $h^{max}$ | 393'674 | 25'957 | 0.50 | 0.50 | | |
| $h^{lmcut}$ | 1744 | 2490 | ~0.33 | 0.00 | | |
| problem 05 | | | | | 62 | 18 |
| $h^1$ | 5'763'654 | 306'497 | ~0.61 | - | | |
| $h^{max}$ | 1'724'608 | 306'497 | ~0.61 | - | | |
| $h^{lmcut}$ | 731 | 7657 | ~0.72 | 1.00 | | |
| problem 6 | | | | | 8 | 11 |
| $h^1$ | 840'126 | 25'281 | ~0.55 | ~0.55 | | |
| $h^{max}$ | 54'454 | 16'915 | ~0.45 | ~0.55 | | |
| $h^{lmcut}$ | 148 | 919 | ~0.63 | 1.00 | | |

Table A.2: The table includes the relevant information regarding the first six problem instances of the *driverlog* domain. First, the secondary initial states (SIS) and corresponding $C^*$ is shown for each instance . Second, for each applied heuristic, we list the number of expansions by A*, expansions by NBS, the meeting point (MP) of the search frontiers of NBS, and $p^*$.

# B

# Extended Results

## B.1 Extended NBS Results

This section provides more detailed data on the experiments discussed in Section 5.2.

| Domain | Solved (#) | | Problems (#) | | | Secondary Initial States (#) | | |
|---|---|---|---|---|---|---|---|---|
| | A* | NBS | NBS < A* | NBS < 2×\|VC\| | $f_B > f_F$ | min. | med. | max. |
| blocks | | | | | | 1 | 1 | 1 |
| $h^1$ | 18 | 27 | 0 | 0 | 26 | | | |
| $h^{max}$ | 21 | 27 | 21 | 21 | 27 | | | |
| $h^{m=2}$ | 10 | 13 | 0 | 0 | 13 | | | |
| $h^{lmcut}$ | 28 | 31 | 15 | 25 | 30 | | | |
| depot | | | | | | 9 | 9 | 9 |
| $h^1$ | 4 | 3 | 0 | 3 | 0 | | | |
| $h^{max}$ | 6 | 2 | 1 | 2 | 0 | | | |
| $h^{m=2}$ | 2 | 1 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 7 | 2 | 0 | 0 | 0 | | | |
| driverlog | | | | | | 1 | 21 | 1029 |
| $h^1$ | 7 | 10 | 0 | 0 | 1 | | | |
| $h^{max}$ | 9 | 10 | 0 | 0 | 3 | | | |
| $h^{m=2}$ | 2 | 3 | 0 | 1 | 1 | | | |
| $h^{lmcut}$ | 14 | 12 | 1 | 12 | 2 | | | |
| elevators-opt08-strips | | | | | | 1500 | 21'000 | 441'000 |
| $h^1$ | 14 | 12 | 0 | 0 | 0 | | | |
| $h^{max}$ | 19 | 13 | 5 | 11 | 0 | | | |
| $h^{m=2}$ | 0 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 22 | 11 | 0 | 0 | 0 | | | |
| elevators-opt11-strips | | | | | | 1500 | 21'000 | 441'000 |
| $h^1$ | 12 | 10 | 7 | 7 | 0 | | | |
| $h^{max}$ | 16 | 11 | 5 | 10 | 0 | | | |
| $h^{m=2}$ | 0 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 18 | 8 | 0 | 2 | 0 | | | |
| floortile-opt11-strips | | | | | | 24 | 48 | 48 |
| $h^1$ | 2 | 10 | 2 | 2 | 10 | | | |
| $h^{max}$ | 6 | 12 | 0 | 0 | 12 | | | |
| $h^{m=2}$ | 0 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 7 | 10 | 0 | 0 | 10 | | | |
| ged-opt14-strips | | | | | | 4 | 8 | 10 |
| $h^1$ | 15 | 19 | 0 | 0 | 7 | | | |
| $h^{max}$ | 15 | 20 | 10 | 12 | 12 | | | |
| $h^{m=2}$ | 5 | 5 | 0 | 0 | 2 | | | |
| $h^{lmcut}$ | 15 | 19 | 10 | 12 | 12 | | | |

Continued on next page

| Domain | Solved (#) | | Problems (#) | | | Secondary Initial States (#) | | |
|---|---|---|---|---|---|---|---|---|
| | A* | NBS | NBS < A* | NBS < 2×\|VC\| | $f_B > f_F$ | min. | med. | max. |
| gripper | | | | | | 2 | 2 | 2 |
| $h^1$ | 8 | 7 | 7 | 7 | 3 | | | |
| $h^{max}$ | 8 | 7 | 6 | 7 | 2 | | | |
| $h^{m=2}$ | 3 | 3 | 0 | 0 | 2 | | | |
| $h^{lmcut}$ | 7 | 7 | 4 | 7 | 7 | | | |
| hiking-opt14-strips | | | | | | 384 | 27'034 | 1'278'080 |
| $h^1$ | 11 | 10 | 3 | 10 | 0 | | | |
| $h^{max}$ | 12 | 10 | 0 | 10 | 0 | | | |
| $h^{m=2}$ | 2 | 1 | 0 | 0 | 1 | | | |
| $h^{lmcut}$ | 9 | 6 | 0 | 0 | 0 | | | |
| logistics00 | | | | | | 8 | 8 | 24 |
| $h^1$ | 10 | 13 | 10 | 10 | 0 | | | |
| $h^{max}$ | 12 | 13 | 0 | 0 | 0 | | | |
| $h^{m=2}$ | 6 | 6 | 0 | 6 | 0 | | | |
| $h^{lmcut}$ | 20 | 20 | 0 | 0 | 0 | | | |
| miconic | | | | | | 4 | 768 | 20480 |
| $h^1$ | 55 | 50 | 31 | 50 | 0 | | | |
| $h^{max}$ | 55 | 50 | 0 | 0 | 0 | | | |
| $h^{m=2}$ | 30 | 26 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 141 | 53 | 0 | 33 | 0 | | | |
| nomystery-opt11-strips | | | | | | 80 | 375 | 1224 |
| $h^1$ | 8 | 10 | 8 | 8 | 0 | | | |
| $h^{max}$ | 9 | 10 | 4 | 9 | 0 | | | |
| $h^{m=2}$ | 6 | 5 | 0 | 5 | 0 | | | |
| $h^{lmcut}$ | 14 | 14 | 0 | 0 | 0 | | | |
| openstacks-opt08-strips | | | | | | 5 | 10 | 15 |
| $h^1$ | 22 | 12 | 0 | 0 | 0 | | | |
| $h^{max}$ | 22 | 11 | 4 | 11 | 0 | | | |
| $h^{m=2}$ | 5 | 3 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 21 | 8 | 3 | 8 | 0 | | | |
| openstacks-strips | | | | | | 5 | 5 | 5 |
| $h^1$ | 7 | 5 | 0 | 5 | 0 | | | |
| $h^{max}$ | 7 | 5 | 0 | 0 | 5 | | | |
| $h^{m=2}$ | 5 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 7 | 5 | 0 | 5 | 0 | | | |
| pegsol-08-strips | | | | | | 2 | 2 | 2 |
| $h^1$ | 27 | 28 | 0 | 0 | 6 | | | |
| $h^{max}$ | 28 | 28 | 14 | 26 | 10 | | | |
| $h^{m=2}$ | 9 | 8 | 0 | 6 | 2 | | | |
| $h^{lmcut}$ | 28 | 28 | 10 | 26 | 8 | | | |
| pegsol-opt11-strips | | | | | | 2 | 2 | 2 |
| $h^1$ | 17 | 18 | 14 | 17 | 1 | | | |
| $h^{max}$ | 18 | 18 | 0 | 0 | 5 | | | |
| $h^{m=2}$ | 1 | 1 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 18 | 18 | 0 | 0 | 2 | | | |
| psr-small | | | | | | 2 | 8 | 9640 |
| $h^1$ | 49 | 42 | 0 | 42 | 0 | | | |
| $h^{max}$ | 49 | 42 | 11 | 42 | 0 | | | |
| $h^{m=2}$ | 40 | 29 | 0 | 0 | 6 | | | |
| $h^{lmcut}$ | 49 | 40 | 0 | 0 | 2 | | | |
| rovers | | | | | | 16 | 384 | 20763 |
| $h^1$ | 6 | 4 | 0 | 4 | 0 | | | |
| $h^{max}$ | 6 | 5 | 0 | 0 | 0 | | | |
| $h^{m=2}$ | 4 | 4 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 8 | 4 | 0 | 3 | 0 | | | |
| satellite | | | | | | 28 | 960 | 1792 |
| $h^1$ | 6 | 4 | 0 | 0 | 0 | | | |
| $h^{max}$ | 6 | 4 | 1 | 4 | 0 | | | |
| $h^{m=2}$ | 3 | 2 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 7 | 4 | 0 | 3 | 0 | | | |

Continued on next page

| Domain | Solved (#) | | Problems (#) | | | Secondary Initial States (#) | | |
|---|---|---|---|---|---|---|---|---|
| | A* | NBS | NBS < A* | NBS < 2×\|VC\| | $f_B > f_F$ | min. | med. | max. |
| scanalyzer-08-strips | | | | | | 1 | 1 | 1 |
| $h^1$ | 12 | 12 | 0 | 0 | 12 | | | |
| $h^{max}$ | 9 | 9 | 6 | 9 | 6 | | | |
| $h^{m=2}$ | 3 | 3 | 0 | 3 | 0 | | | |
| $h^{lmcut}$ | 16 | 11 | 0 | 0 | 11 | | | |
| scanalyzer-opt11-strips | | | | | | 1 | 1 | 1 |
| $h^1$ | 9 | 9 | 9 | 9 | 9 | | | |
| $h^{max}$ | 6 | 6 | 0 | 0 | 5 | | | |
| $h^{m=2}$ | 1 | 1 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 12 | 8 | 3 | 6 | 8 | | | |
| storage | | | | | | 5 | 340 | 5156 |
| $h^1$ | 14 | 5 | 0 | 0 | 0 | | | |
| $h^{max}$ | 15 | 5 | 0 | 2 | 0 | | | |
| $h^{m=2}$ | 7 | 2 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 15 | 4 | 0 | 1 | 0 | | | |
| termes-opt18-strips | | | | | | 4 | 9 | 11 |
| $h^1$ | 10 | 13 | 10 | 10 | 9 | | | |
| $h^{max}$ | 10 | 13 | 0 | 0 | 9 | | | |
| $h^{m=2}$ | 0 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 6 | 8 | 0 | 0 | 7 | | | |
| tpp | | | | | | 12 | 432 | 2592 |
| $h^1$ | 6 | 4 | 0 | 4 | 0 | | | |
| $h^{max}$ | 6 | 4 | 0 | 4 | 0 | | | |
| $h^{m=2}$ | 5 | 4 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 7 | 4 | 0 | 4 | 0 | | | |
| transport-opt08-strips | | | | | | 144 | 1024 | 2304 |
| $h^1$ | 11 | 11 | 5 | 10 | 0 | | | |
| $h^{max}$ | 11 | 11 | 0 | 0 | 0 | | | |
| $h^{m=2}$ | 6 | 5 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 11 | 11 | 0 | 0 | 0 | | | |
| transport-opt11-strips | | | | | | 1024 | 2304 | 2704 |
| $h^1$ | 6 | 6 | 0 | 0 | 0 | | | |
| $h^{max}$ | 6 | 7 | 0 | 2 | 0 | | | |
| $h^{m=2}$ | 1 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 6 | 6 | 0 | 0 | 0 | | | |
| transport-opt14-strips | | | | | | 400 | 2304 | 14'400 |
| $h^1$ | 7 | 6 | 0 | 0 | 0 | | | |
| $h^{max}$ | 7 | 6 | 1 | 4 | 0 | | | |
| $h^{m=2}$ | 1 | 0 | 0 | 0 | 0 | | | |
| $h^{lmcut}$ | 6 | 5 | 0 | 2 | 0 | | | |
| trucks-strips | | | | | | 18 | 864 | 1728 |
| $h^1$ | 6 | 4 | 0 | 4 | 0 | | | |
| $h^{max}$ | 10 | 6 | 0 | 6 | 0 | | | |
| $h^{m=2}$ | 2 | 2 | 0 | 2 | 0 | | | |
| $h^{lmcut}$ | 10 | 6 | 0 | 6 | 0 | | | |
| zenotravel | | | | | | 7 | 196 | 8575 |
| $h^1$ | 8 | 8 | 0 | 0 | 0 | | | |
| $h^{max}$ | 8 | 8 | 2 | 7 | 0 | | | |
| $h^{m=2}$ | 5 | 4 | 0 | 3 | 0 | | | |
| $h^{lmcut}$ | 13 | 9 | 0 | 6 | 0 | | | |

Table B.1: A wholesome overview of the experiments with the NBS algorithm. The listed attributes, from left to right, describe the following information. (1) The domain and the various heuristics. (2) The number of solved problems by either A* or NBS. (3) A comparison between A* and NBS. The number of problems which fulfil the displayed condition: either where the number of expansion is smaller for NBS, where the number of expansions in NBS prior to the last layer is smaller than twice the number of expansion in A*, or whether the $g$-value at the meeting point of forward and backward search is greater in backward direction. (4) The number of secondary initial states for each domain. Excluding the states which are detected as dead ends by the $h^{max}$ heuristic.

## B.2 Extended fMM Results

In this section we present results regarding the experiments described in Section 5.3. The goal is to illustrate the structure of the underlying transition system for as many domains as possible. To that end we provide two graphs for each domain. First, an overview graph which combines all the individual problem graphs. Second, a hand picked problem graph, which should be representative of the whole domain. Note that the y-axis denotes the number of expansions relative to the minimal occurrence as the absolute number varies greatly over the different problems. However, even the relative number can be vastly different. Because of this, we additionally provide a problem graph which represents the whole domain.

The following graphs were created by running different instances of fMM with evenly distributed input argument $p$ and $h^{max}$ heuristic.

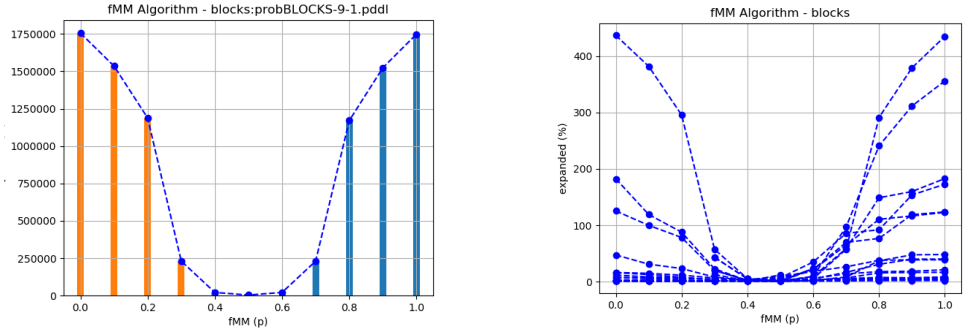



Figure B.1: As is evident by the graphs, the *blocks* domain is very suited for bidirectional search. In a few problems, unidirectional backward search would be favoured, but never unidirectional forward search. On the right side, the domain-wide fMM graph is depicted. The uppermost points denote that the specified search expands more than 400 times more states than the optimum. As a result, the other lines look rather flat. On the left side, we present the best-case example which obscures the results of the other graph. It exemplifies how much more efficient the bidirectional search can potentially be.

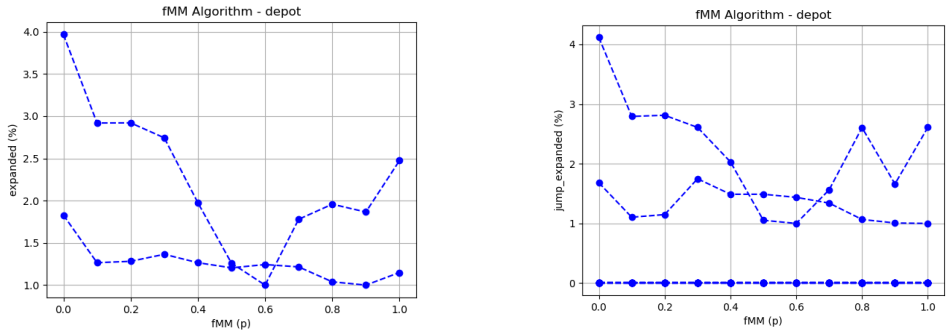



Figure B.2: Due to the complexity of the *depot* domain we were only able to solve the first two problems. One the left graph are the total number of expansions depicted, whereas on the right graph only the expansions preceding the last layer are counted. As can be seen, the structure in both graphs are very similar, meaning that the number of expansions in the last layer is very small or at least proportional to the rest.

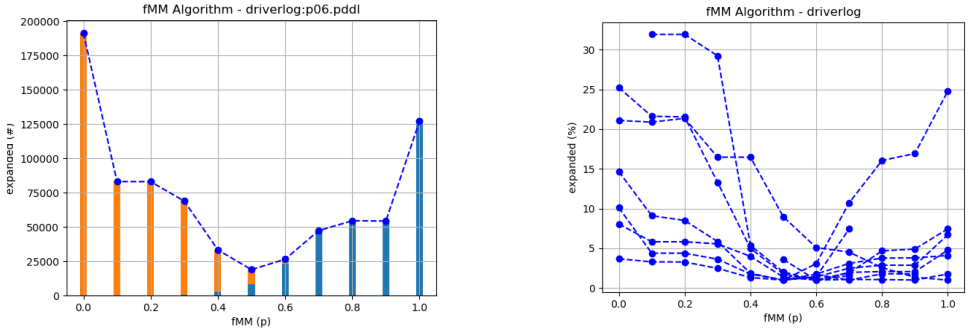



Figure B.3: The *driverlog* domain shows mixed results, where sole backward search is never optimal, sole forward search sometimes, and bidirectional search most often. This can be seen in the right graph, although the scale is distorted to include the extremes. On the left is a typical graph of the drivelog domain. The forward search is favoured over the backward search, but most times bidirectional search is optimal. We reason that the backward search performs badly because it has to consider many secondary initial states due to symmetries.

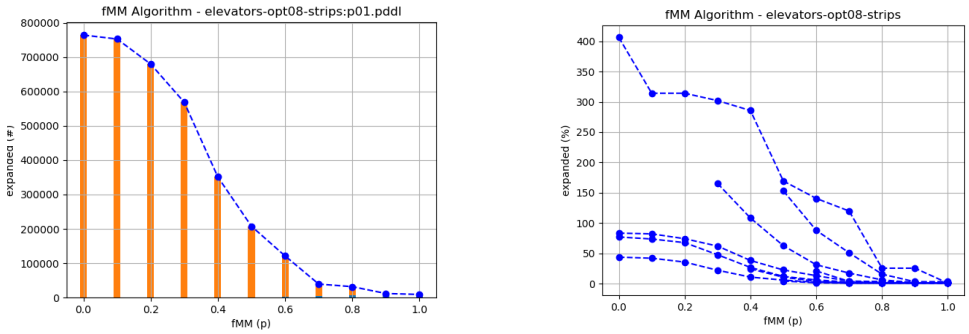



Figure B.4: The *elevator* domain is not suited for backward search. Therefore, unless the problem size is very big, bidirectional search will not be more efficient than forward search. However, NBS achieves to expand less states then A* for a few problems.

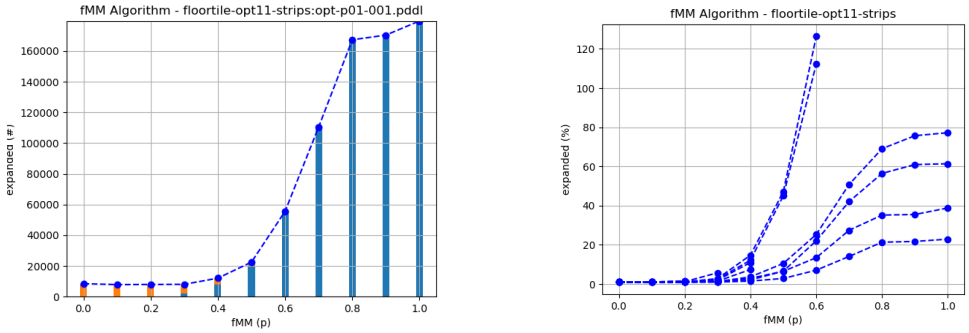



Figure B.5: The *floortile* domain exhibits the very opposite aptitude as the previous *elevator* domain. The domain clearly favours backward search. Interestingly, similar to the *elevator* domain, bidirectional search is more efficient than unidirectional for the biggest problems.

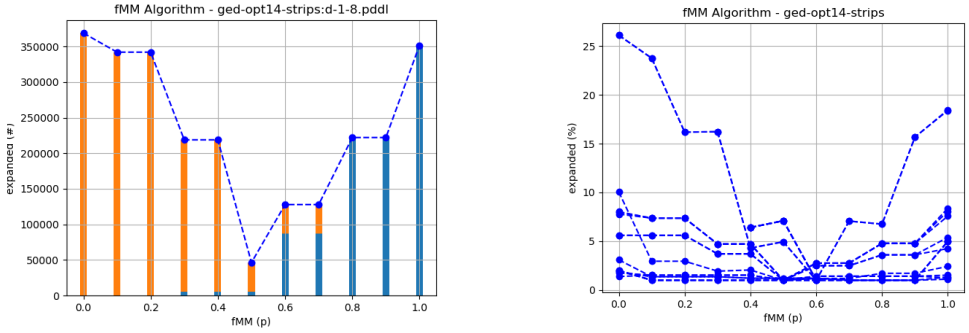



Figure B.6: The *get-opt* domain is a special domain because the number of expansions seem to change drastically at certain point. Nevertheless, the domain is very suited for bidirectional search.

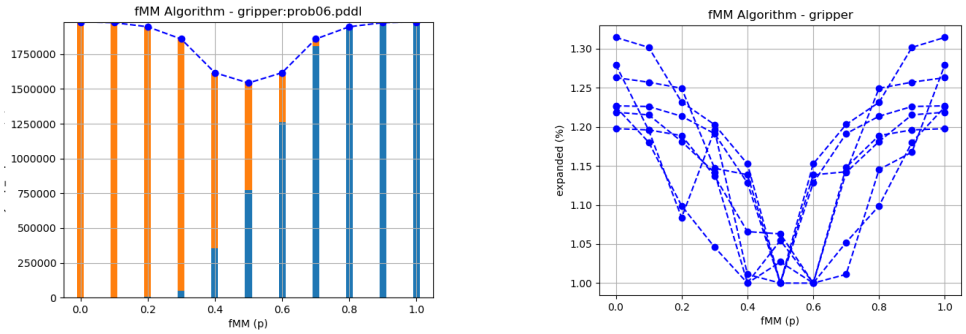



Figure B.7: The *gripper* domain is mostly symmetric and therefore suited for bidirectional search. However, the difference between the search directions is of a low magnitude. This is clearly visible in the right image.

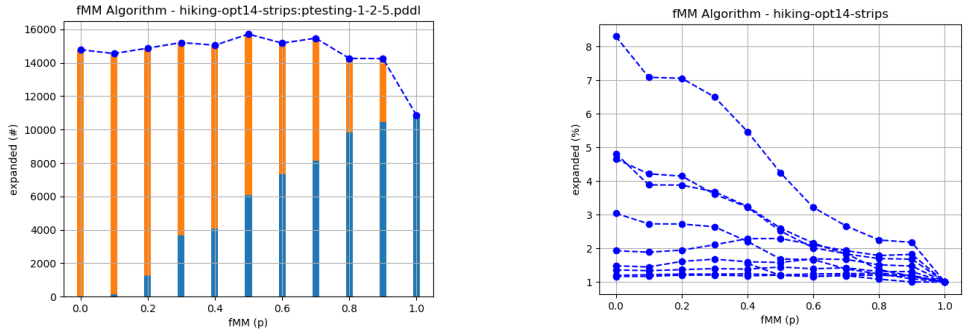



Figure B.8: The *hiking* domain has many domains where the search direction only marginally influences the number of expansions, however, the overarching trend is still preferable to unidirectional forward search.

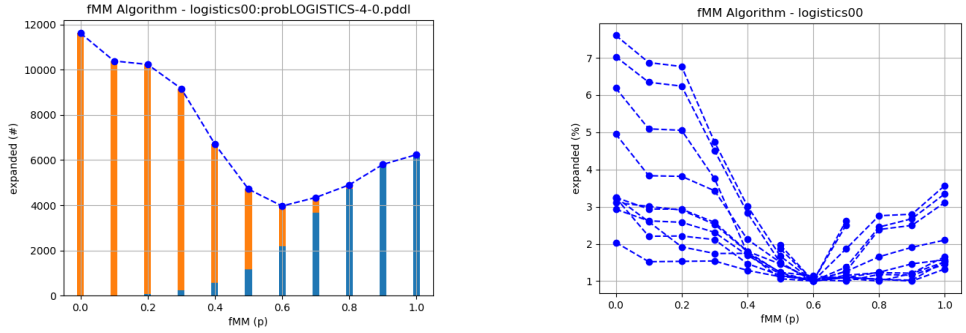



Figure B.9: The problem of the *logistics* domain show a very consistent `fMM` graph, where the backward search is worse than the forward search, but bidirectional search is optimal. This is reflected in a $p^*$ value with low standard deviation over the whole domain.

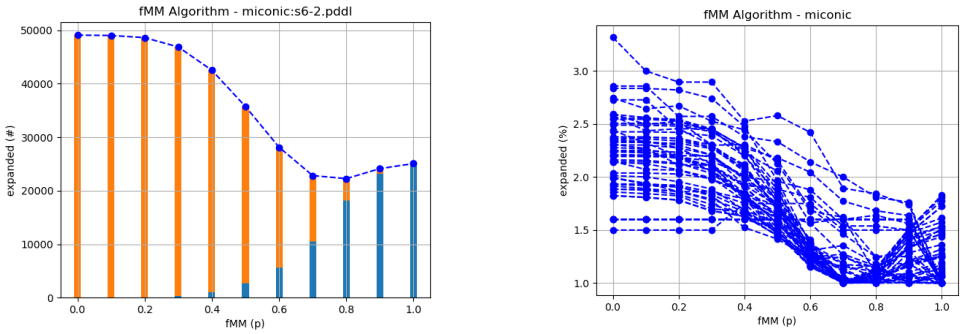



Figure B.10: The *miconic* domain is very similar to the *logistics* domain. Both have an average $p^*$ value around 0.75 with a relatively low standard deviation.

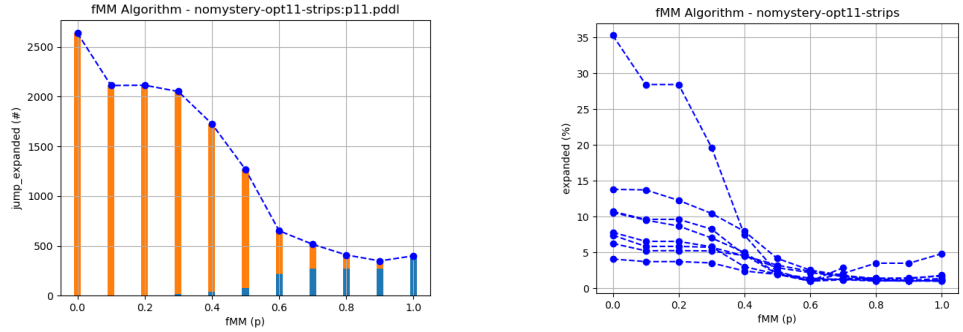



Figure B.11: The *nomystery* domain shows a similar aptitude as *logistics* and *miconic*, but less extreme.

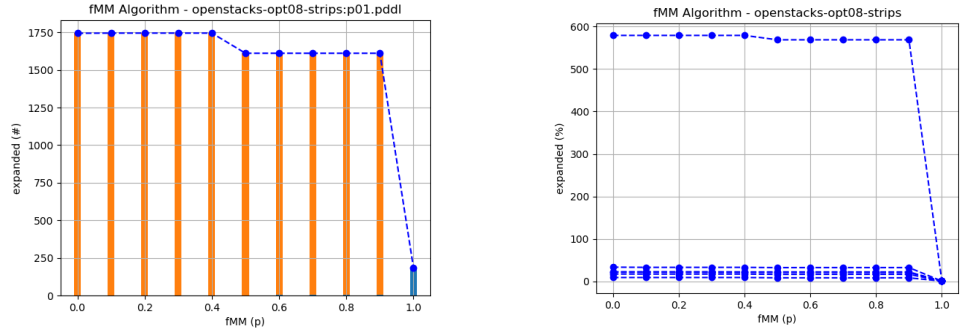



Figure B.12: Because to the high number of secondary initial states is the *openstacks* domain not suited to be solved with backward search. This is very well reflected in both graphs. But once again, the right graph is distorted by an extreme outlier.

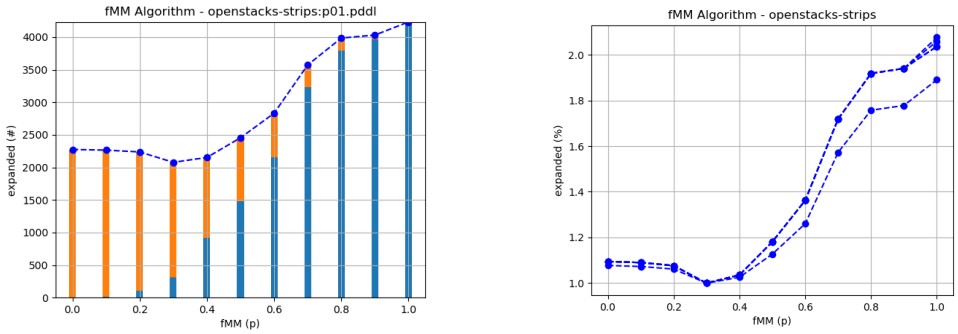



Figure B.13: The *openstacks-strips* domain shows a clear preference for bidirectional search. A noteworthy detail in this domain is that all `fMM` graphs are very similar. On the right side, the five different graphs are nearly not discernible from each other.

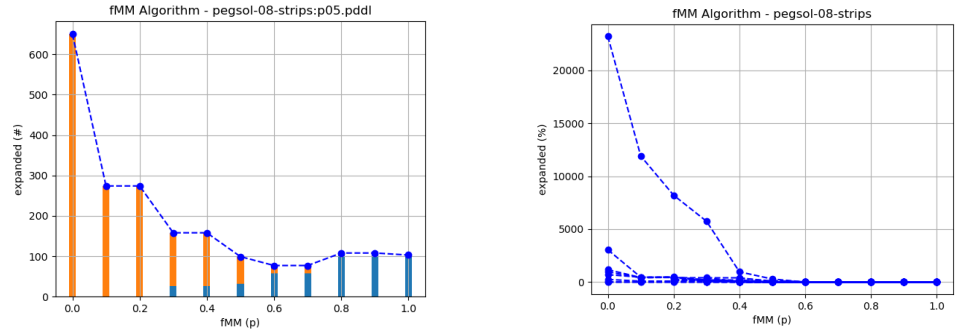



Figure B.14: The *pegsol* domain exhibits a preference for unidirectional forward search.

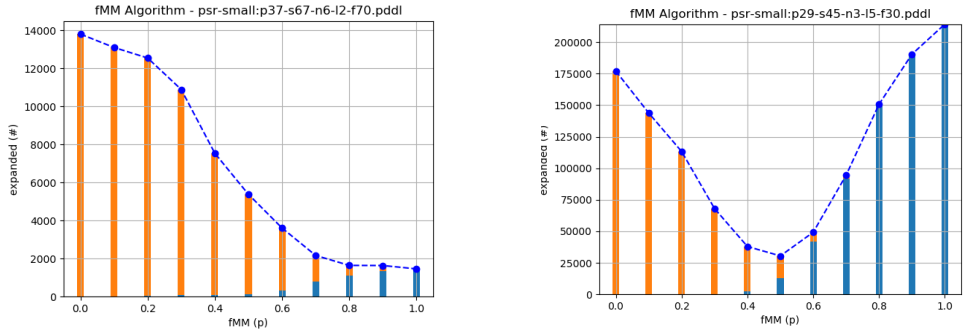



Figure B.15: The domain wide graph of the *psr* domain is not very expressive. We present two representative problem instances. Most instances share the structure depicted in the left image. Some rare instances are very suited for bidirectional search and have a similar structure as shown on the right side.

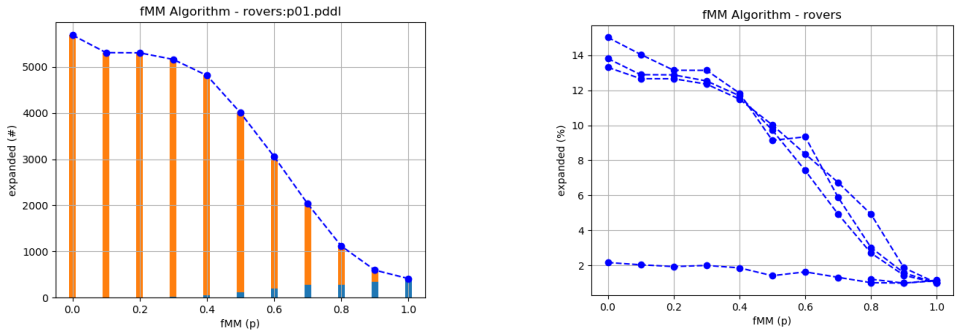



Figure B.16: The *rover* domain is exclusively suited for forward search.

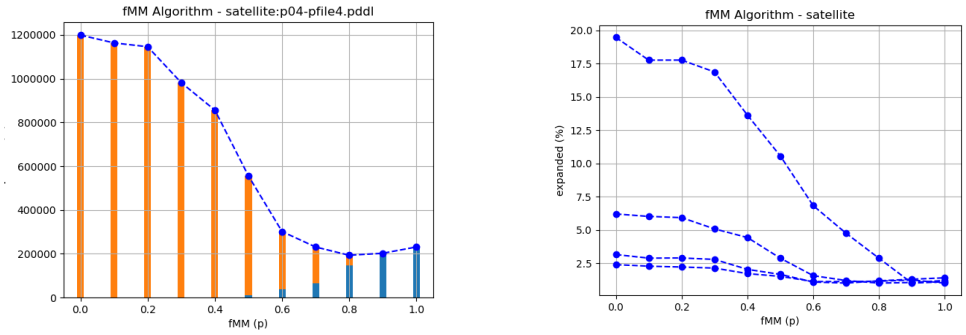



Figure B.17: The *satellite* domain shows an aptitude for unidirectional forward search.

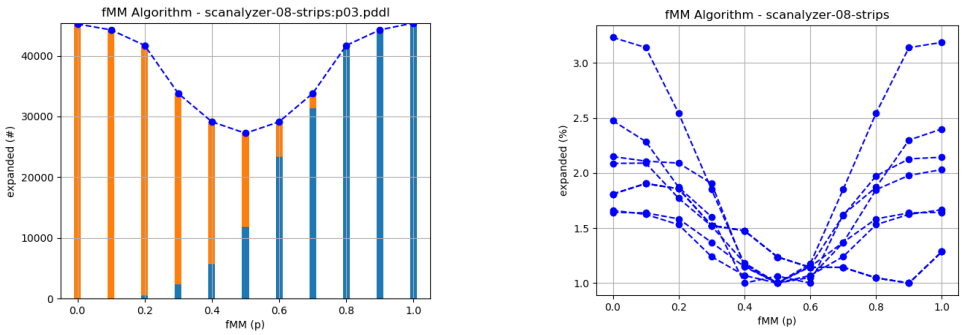



Figure B.18: The *scanalyzer* domain is mostly suited for bidirectional search.

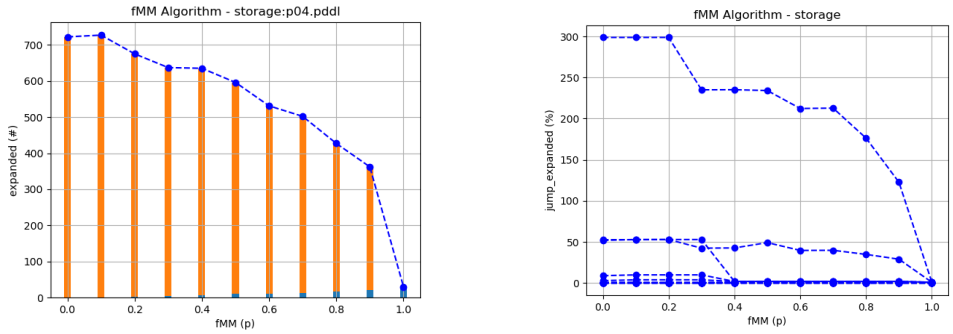



Figure B.19: The *storage* domain has many secondary initial states, which reduce the performance of the backward search significantly.

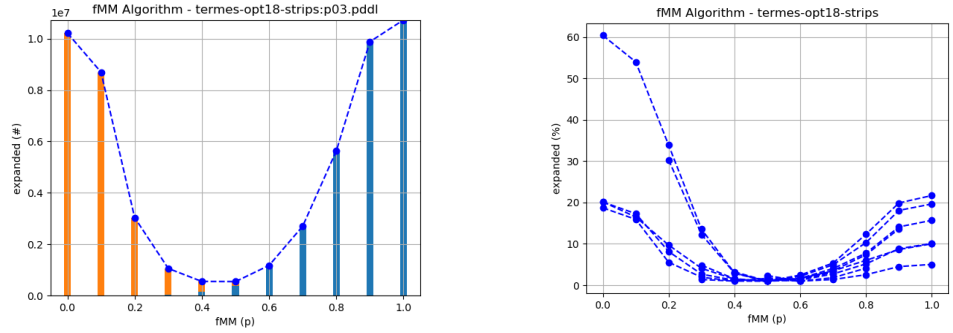



Figure B.20: The *termes* domain is very suited for bidirectional search and is one of the domains where NBS solved more problems than A*.

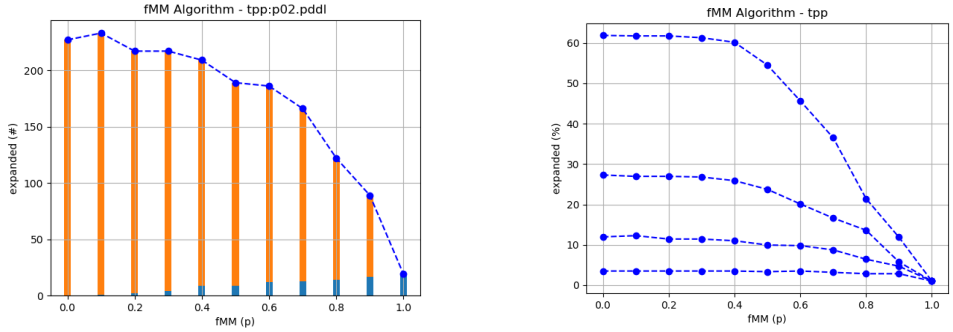



Figure B.21: The *tpp* domain suffers from too many secondary initial states.

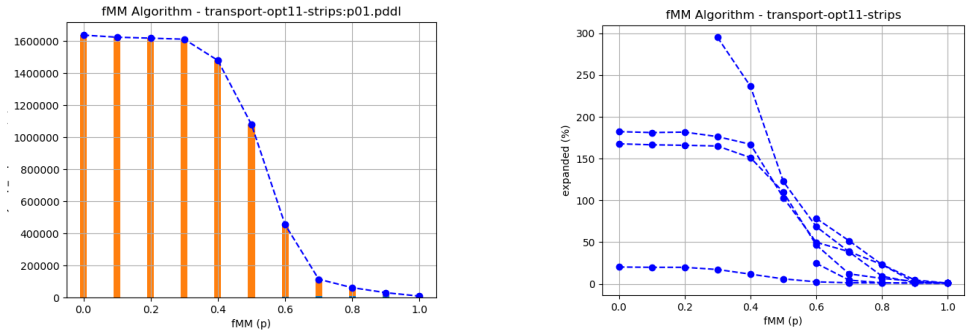



Figure B.22: The *transport* domain has a clear aptitude for forward search, however, bidirectional search can be effective in rare cases.

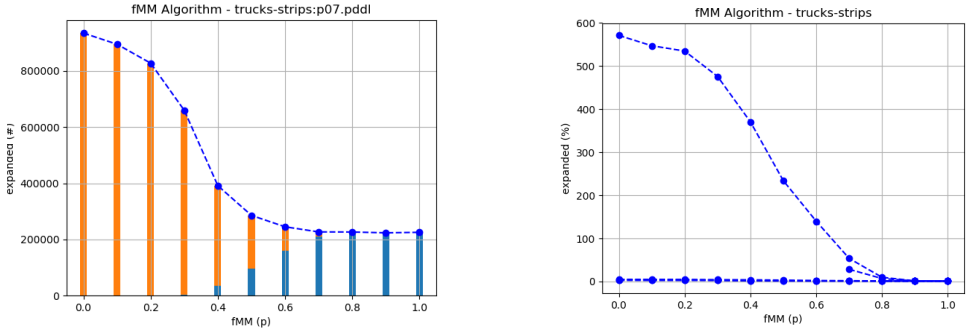



Figure B.23: The *trucks* domain is similar to the *transport* domain. A best case example is shown on the left side.

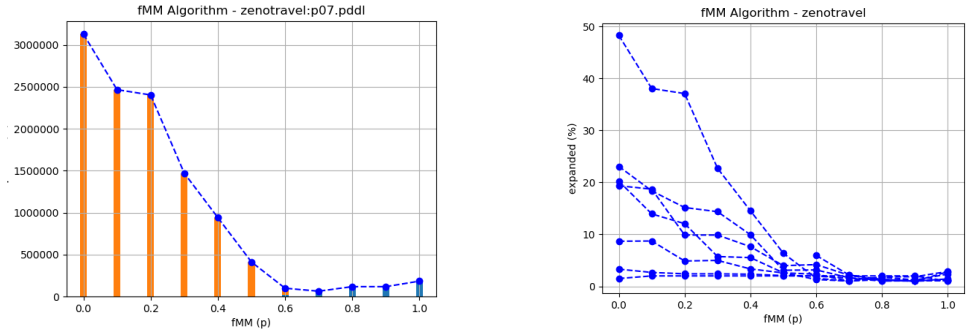



Figure B.24: The *zenotravel* has a few domains which can be efficiently solved with NBS, but most often A* is more performant.

## B.3 Extended Case Study

The graphs depicted in this section provide additional examples for the material discussed in Section 5.3.2

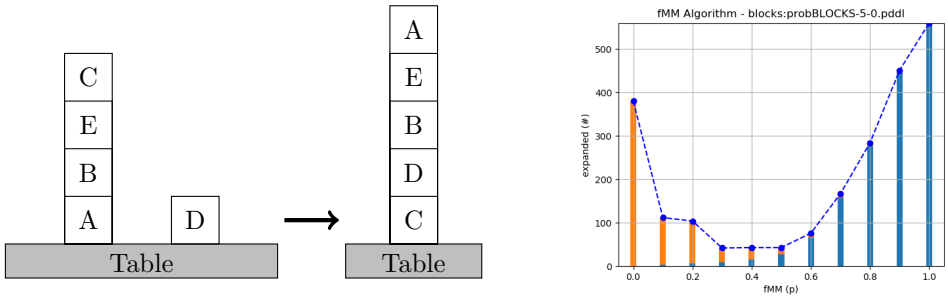



Figure B.25: The problem 5-0 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

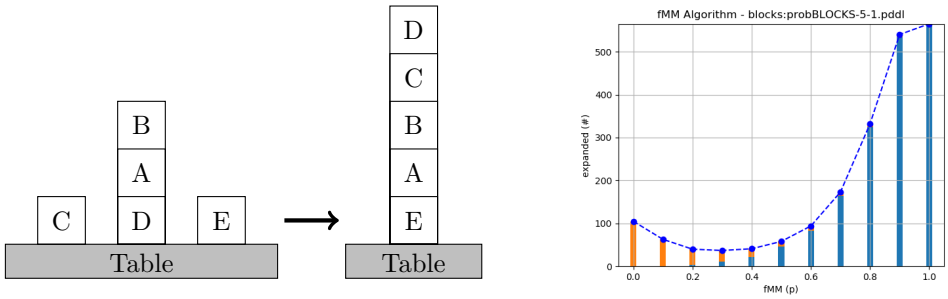



Figure B.26: The problem 5-1 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

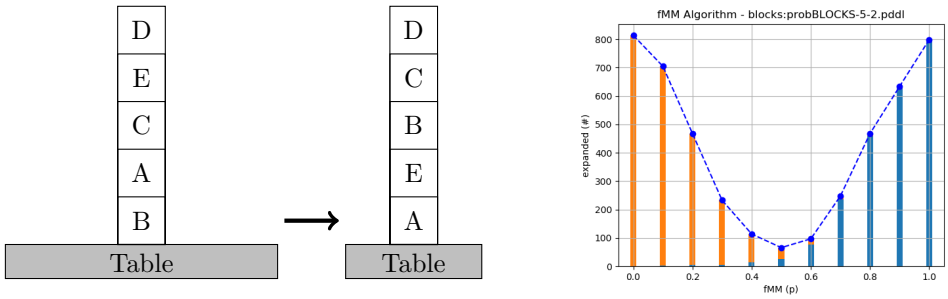



Figure B.27: The problem 5-2 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

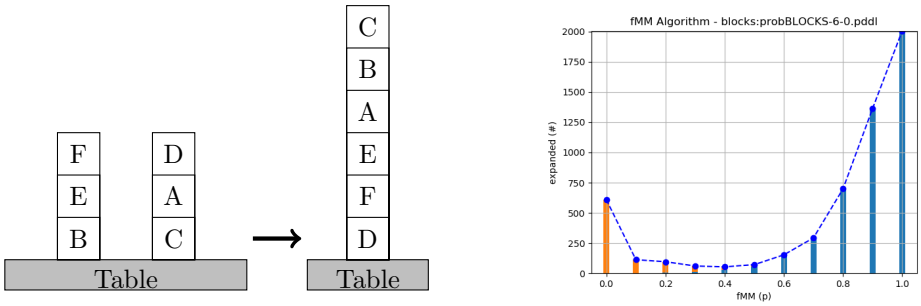



Figure B.28: The problem 6-0 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

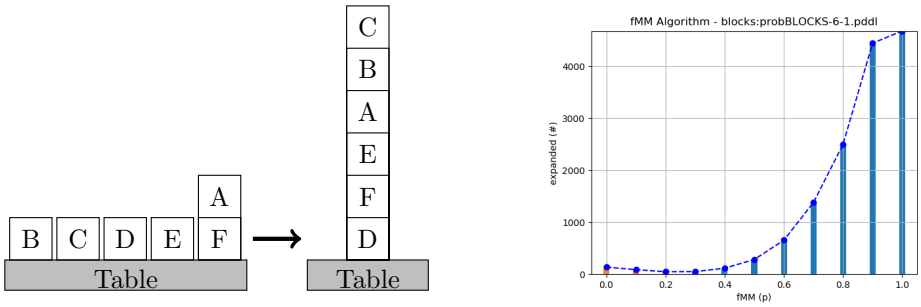



Figure B.29: The problem 6-1 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

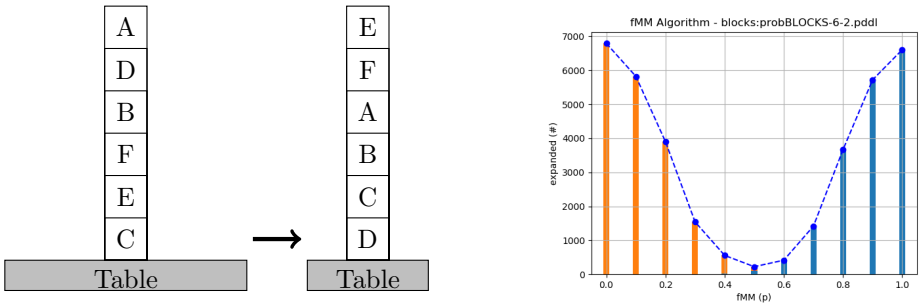



Figure B.30: The problem 6-2 of the *blocks* domain. On the left is the initial and goal configuration depicted, whereas on the right is the `fMM` graph of the problem instance solved with $h^1$.

# Declaration on Scientific Integrity
# Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

**Author — Autor**
Marvin Buff

**Matriculation number — Matrikelnummer**
2014-054-191

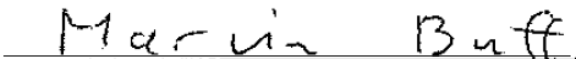
**Title of work — Titel der Arbeit**
NBS applied to Planning

**Type of work — Typ der Arbeit**
Master Thesis

**Declaration — Erklärung**
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 05/02/2019



**Signature — Unterschrift**