Search Behavior of Greedy Best-First Search

Inauguraldissertation
zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

MANUEL HEUSNER

aus Basel

Basel, 2019

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Malte Helmert
Universität Basel, Dissertationsleiter, Fakultätsverantwortlicher

Prof. Dr. Robert Holte
University of Alberta, Korreferent

Basel, den 26.03.2019

Prof. Dr. Martin Spiess
Dekan

Abstract

Greedy best-first search (GBFS) is a sibling of A* in the family of best-first state-space search algorithms. While A* is guaranteed to find optimal solutions of search problems, GBFS does not provide any guarantees but typically finds satisficing solutions more quickly than A*. A classical result of optimal best-first search shows that A* with admissible and consistent heuristic expands every state whose f-value is below the optimal solution cost and no state whose f-value is above the optimal solution cost. Theoretical results of this kind are useful for the analysis of heuristics in different search domains and for the improvement of algorithms. For satisficing algorithms a similarly clear understanding is currently lacking. We examine the search behavior of GBFS in order to make progress towards such an understanding.

We introduce the concept of high-water mark benches, which separate the search space into areas that are searched by GBFS in sequence. High-water mark benches allow us to exactly determine the set of states that GBFS expands under at least one tie-breaking strategy. We show that benches contain craters. Once GBFS enters a crater, it has to expand every state in the crater before being able to escape.

Benches and craters allow us to characterize the best-case and worst-case behavior of GBFS in given search instances. We show that computing the best-case or worst-case behavior of GBFS is NP-complete in general but can be computed in polynomial time for undirected state spaces.

We present algorithms for extracting the set of states that GBFS potentially expands and for computing the best-case and worst-case behavior. We use the algorithms to analyze GBFS on benchmark tasks from planning competitions under a state-of-the-art heuristic. Experimental results reveal interesting characteristics of the heuristic on the given tasks and demonstrate the importance of tie-breaking in GBFS.

Acknowledgements

I thank Malte Helmert who gave me the opportunity to write a Ph.D. thesis under his supervision. He pointed me to interesting research questions and provided valuable and constructive suggestions. I thank Martin Wherle and Thomas Keller for being my closest advisors. Their diverse expertise was helpful to consider my research questions from different angles. I thank Robert Holte and Martin Müller who invited me to collaborate with them at the University of Alberta. The time in Canada gave me the motivation to stay on track. Special thanks to Rob for co-reviewing my thesis. I thank Alex Fukunaga and Masataro Asai for inviting me to the University of Tokyo. I had many insightful discussions about my research topic with them. I thank all the group members, Salomé Eriksson, Patrick Ferber, Guillem Francèe, Cedric Geissmann, Florian Pommerening, Gabriele Röger, Jendrik Seipp and Silvan Sievers for being canteen buddies, reviewers and motivators. They all brought positive spirit into my working space. I thank Thomas Gehrig and Andreas Schneider for being my fellow students since the beginning of my studies in computer science in Basel. I will always remember our rooftop coffee breaks. I thank Andreas, Simon and Burri for being great friends. You are enriching my life with numerous social activities. I thank all my friends from blues and Lindy Hop dancing for all the dances. You are inspiring me - not only on the dance floors. I thank my family Verena, Werner, Malaika, Enrico and Tatjana for their ongoing support. A special thank goes to my girlfriend Judith. With all her love, kindness and support I am flourishing better than ever before.

Contents

1. Introduction

Many computational problems can be expressed with three pieces of information: an *initial state*, a *successor generator* and a *goal test function*. The initial state describes an initial situation in a given world. The successor generator produces neighbor states of a given state. The goal test function determines whether a state matches a goal situation. These pieces span a state space whose size depends on the underlying problem. A solution of a problem is a state from the state space that passes the goal test. Usually, the path from the initial state to a goal state is part of the solution as well. In this case, a problem may provide an additional piece of information: a *cost function* that defines the cost of transitioning from one state to another state. A problem that provides a cost function does often call for an optimal solution, i.e., a cheapest solution path.

Algorithms that search for solution paths in state spaces are called state-space search algorithms. Such algorithms start from the initial state, iteratively expand states by applying the successor function and explore the state space in this way until they reach a goal state. These algorithms are applied to typical search and optimization problems like 15-puzzle, Towers of Hanoi and traveling salesman problem. They are implemented in many planning systems that aim to find plans for planning tasks from domains like logistics, assembling or scheduling. Moreover, model checking systems use state-space search to verify that software and hardware systems match given specifications. All these applications have in common that they search in state spaces that grow exponentially in the size of the underlying problem description.

One approach to tackle large state spaces is to express further information about a problem in a *heuristic function* that estimates the cost-to-go or distance-to-go for reaching a goal from a given state. The family of state-space search algorithms that use heuristic functions is called heuristic best-first search. Representatives of this family are greedy best-first search (Doran and Michie, 1966), A* (Hart, Nilsson, and Raphael, 1968), Weighted A* (Pohl, 1970) and Iterative Deepening A* (Korf, 1985).

An essential characteristic of search algorithms is whether they guarantee that the solutions they produce are optimal. Optimal search algorithms in this family have a fairly well-developed theory (Dechter and Pearl, 1985). For example, we know that A* with admissible and consistent heuristic returns optimal solutions and never expands a state more than once. Moreover, we can easily characterize states that A* necessarily, never or potentially expands. We know when A* makes considerable progress towards finding a solution. We know that tie-breaking influences the number of states that A* expands during its run, and we can easily determine a best-case or worst-case tie-breaking policy.

Theoretical results of this kind are useful to show under which conditions there exists

no better algorithm than A* regarding the number of expanded states (Dechter and Pearl, 1985), to understand the limits of A* when using an almost perfect heuristic (Helmert and Röger, 2008), and to clarify misconceptions about when a heuristic dominates another one (Holte, 2010). On the practical side these results led to the development of an A* variant that is less memory intensive (Korf, 1985). They also shed light on cases where good tie-breaking becomes especially important and led to the development of more sophisticated tie-breaking strategies (Asai and Fukunaga, 2017b; Corrêa, Pereira, and Ritt, 2018).

For *satisficing* (non-optimal) algorithms in this family a comparably deep understanding is currently lacking. Many new algorithms based on greedy best-first search (GBFS) have been proposed in recent years (Imai and Kishimoto, 2011; Xie et al., 2014; Xie, Müller, and Holte, 2014; Valenzano et al., 2014; Asai and Fukunaga, 2017a; Cohen and Beck, 2018), all trying to handle the problem of local minima and search plateaus. Local minima and search plateaus are regions in a state space where a heuristic provides no guidance or even misguides a search. Our understanding of such regions and the behavior of satisficing algorithms is still quite limited.

First steps have been made. Xie, Müller, and Holte (2015) experimentally analyzed local minima and plateaus that appear during a search run with a focus on small and weakly connected sub-regions. Cohen and Beck (2018) did a similar analysis centered around the question of how the constrainedness of a search problem affects depths and sizes of local minima. Wilt and Ruml (2012, 2014, 2015) demonstrated cases in which A* performs better than GBFS given the same state space and heuristic, analyzed the behavior of GBFS under different kinds of heuristics and presented plausible reasons for their observations. Most interestingly, they were able to explain why improving the accuracy of an admissible heuristic is beneficial for A* but can be detrimental for GBFS. This is an insight that clearly shows how "conventional wisdom" for optimal search algorithms fails to apply to the satisficing case.

Basic theoretical questions that led to a profound understanding of optimal search algorithms remain unaddressed for satisfincing algorithms. In this thesis, we attempt to reduce this gap in knowledge by developing similar theoretical results for GBFS, the most basic and most commonly considered satisficing algorithm.

1.1. Outline

The central part of this thesis is structured as follows:

- Chapter 5 identifies theoretical questions that support the analysis of state-space search algorithms.
- Chapter 6 summarizes aspects that are known for the behavior of state-space search algorithms in general.

- Chapter 7 summarizes theoretical results from literature about the search behavior of A*. Moreover, it motivates our thesis by presenting the practical implications of these results.

- Chapter 8 offers an approach for comparing different heuristic best-first search algorithms and uses it to clarify the conditions under which A* behaves like GBFS.

- Chapter 9 translates some aspects of the search behavior of GBFS into a static environment that simplifies the further analysis. It also introduces high-water marks - the key to a better understanding of GBFS.

- Chapter 10 presents and generalizes a criterion from the literature that characterizes states which GBFS never expands based on the high-water mark. It also introduces the concept of high-water mark levels.

- Chapter 11 presents the most important discovery in this thesis. It identifies provable search progress of GBFS based on the high-water mark and introduces high-water mark benches, which formalizes what is commonly understood as search plateaus.

- Chapter 12 introduces craters, which formalize what is commonly understood as local minima. Moreover, it identifies states among which tie-breaking decisions have an impact on the search performance.

- Chapter 13 provides an alternative view on the behavior of GBFS which further deepens our understanding of GBFS.

- Chapter 14 characterizes the best-case and worst-case behavior of GBFS based on our insights about the behavior of GBFS. It shows that the problem of determining the best case and worst case is NP-complete in general and presents cases in which the problem becomes polynomial-time computable.

- Chapter 15 presents algorithms for extracting the information about the behavior of GBFS up to the best-case and worst-case behaviors in given search instances.

- Chapter 16 experimentally evaluates the algorithms on search instances from classical planning, and analyzes the extracted information about the search behavior of GBFS.

1.2. Publications

This thesis includes our contributions from following publications:

- Heusner, M.; Keller, T.; and Helmert, M. 2017. Understanding the search behaviour of greedy best-first search. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 47–55. AAAI Press.

  The publication won the SoCS2017 best paper award.

- Heusner, M.; Keller, T.; and Helmert, M. 2018a. Best-case and worst-case behavior of greedy best-first search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 1463–1470. AAAI Press.

- Heusner, M.; Keller, T.; and Helmert, M. 2018b. Search progress and potentially expanded states in greedy best-first search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 5269–5273. AAAI Press.

  This paper based on our first paper was invited for submission to the Sister Conference Best Paper Track.

2. State-Space Search

State-space search is often the first considered approach for finding solutions of computational problems and provides the main framework for any algorithm that searches in state spaces. This chapter formally introduces all the definitions and notations related to state-space search that are important for this thesis.

2.1. State Space

In practical applications, *state spaces* are often implicitly defined with state variables and rules in practical applications. State variables define states and the rules modify the states. Throughout this thesis, we only consider explicitly defined state spaces in which states and transitions do not reveal their connection to the underlying search problem.

Definition 2.1 (state space)**.** *A* state space *is a tuple* $\mathcal{S} = \langle S, s_{init}, S_{goal}, succ \rangle$*, where*

- S *is a finite set of* states*,*
- $s_{init} \in S$ *an* initial state*,*
- $S_{goal} \subseteq S$ *a set of* goal states*, and*
- $succ : S \to 2^S$ *a* successor function*.*

If $s' \in succ(s)$, we say that $s \to s'$ is a *state transition*, s' is a *successor* of s and s is a *predecessor* of s'. State spaces are often associated with a cost function $cost : S^2 \to \mathbb{R}_0^+$ that defines the cost of each transition in a state space. A state space is *undirected* iff for all pairs of states $s, s' \in S$ following condition holds: $s' \in succ(s)$ iff $s \in succ(s')$.

State spaces contain *state paths*.

Definition 2.2 (state path)**.** *Let* $\langle S, s_{init}, S_{goal}, succ \rangle$ *be a state space..*

A state path $\langle s_0, \ldots, s_n \rangle$ *of* length n *is a sequence of states on which* $s_i \in succ(s_{i-1})$ *holds for all states* s_i *with* $i > 0$*. An* s-path *is a path* $\langle s, \ldots, s_n \rangle$ *with* $s_n \in S_{goal}$*. A* solution path *is an* s_{init}*-path.*

A state space with a solution path is called *solvable* and *unsolvable*, otherwise. A state s without an s-path is called *dead end*. State s' is *reachable* from s iff there exists a path $\langle s, \ldots, s' \rangle$. When we say that s' is reachable, then we mean that s' is reachable from s_{init}. When ρ is a path, then we denote its length with $length(\rho)$. We say that path $\langle s_0, \ldots, s_n \rangle$

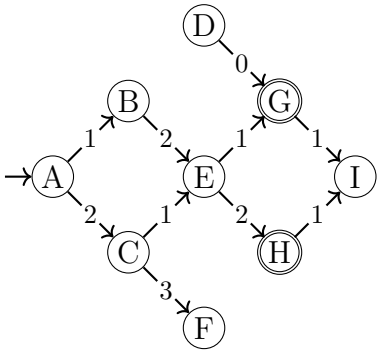


Figure 2.1: Example state space.

passes through state s iff s is on $\langle s_0, \ldots, s_n \rangle$, $s \neq s_0$ and $s \neq s_n$. A path ρ is *acyclic* iff it includes each state at most once.

If a cost function $cost$ is given, then the *cost of a state path* $\langle s_0, \ldots, s_n \rangle$ is

$$cost(\langle s_0, \ldots, s_n \rangle) = \sum_{i=0}^{n-1} cost(s_i, s_{i+1}).$$

A *cost-optimal* path from a state s to a state s' is a path of minimal cost among all possible paths from s to s'. We denote the cost of a cost-optimal path between state s and s' with $c^*(s, s')$. If s' is not reachable from s, then $c^*(s, s') = \infty$. The cost of a cost-optimal s-path is denoted with $c^*(s)$ and the cost of a cost-optimal solution path with c^*.

Example 2.1. *Figure 2.1 shows the state space* $\mathcal{S} = \langle \{\mathrm{A}, \ldots, \mathrm{I}\}, \mathrm{A}, \{\mathrm{G}, \mathrm{H}\}, succ \rangle$ *that is provided with a cost function* $cost$ *and that is solvable.*

Circles with letters define the states. The circle with an arrow tip defines the initial state. Double lined circles define goal states. Arrows between states define the transitions of $succ$*. The numbers on the arrows define the transition cost function* $cost$*.*

$\langle \mathrm{A}, \mathrm{C}, \mathrm{F} \rangle$ *is a path with length* 2 *and cost* 5. $\langle \mathrm{C}, \mathrm{E}, \mathrm{H} \rangle$ *is a* C*-path with length* 2 *and cost* 3. $\langle \mathrm{C}, \mathrm{E}, \mathrm{G} \rangle$ *is a cost-optimal* C*-path with length* 2 *and cost* 2. $\langle \mathrm{A}, \mathrm{C}, \mathrm{E}, \mathrm{H} \rangle$ *is a solution path with length* 3 *and cost* 5. $\langle \mathrm{A}, \mathrm{B}, \mathrm{E}, \mathrm{G} \rangle$ *is a cost-optimal solution path with length* 3 *and cost* 4*.*

State F *is a dead end. State* D *is not reachable from any other state.* H *is reachable (from the initial state* A*) but not reachable from* F*.*

2.2. State-Space Search

In this thesis, we consider *forward uni-directed*, *expansion based*, *explicit* state-space search algorithms. Explicit means that each state is considered on its own. It stans in contrast to symbolic search (e.g. Bryant, 1986; McMillan, 1993), which simultaneously

considers a set of states. Expansion based means that a search only knows a subspace of the whole state space and increases its knowledge by applying the successor function to the known states. Forward uni-directed means that a search algorithm searches from an initial state towards goal states. It stands in contrast to bidirectional search (Kaindl and Kainz, 1997), which simultaneously searches from the initial state to the goal state and vice versa.

A *state expansion* in context of forward uni-directed, explicit state-space search algorithms extends the known state space by applying the successor function to a known state.

Definition 2.3 (state expansion)**.** *Let $\mathcal{S}$ be a state space with set of states S and successor function $succ$.*

A state expansion *of a state $s \in S$ is the application of $succ$ to s that involves the process of generating all the successor states $s' \in succ(s)$.*

We say that a search *expands* a state s and *generates* states $s' \in succ(s)$ in the expansion of s. State expansions are essential for state-space searches.

Definition 2.4 (state-space search)**.** *Let $\langle S, s_{init}, S_{goal}, succ \rangle$ be a state space.*

A state-space search *is the procedure that generates s_{init}, iteratively expands a generated state s, and stops if the expanded state s is a desired goal state from S_{goal}.*

A *desired* goal state is a goal state that fulfills a given requirement. Often the requirement is to reach a goal state along a cost-optimal solution path. We generally assume that all goal states are desired except if we state otherwise.

There is no universal agreement on whether the goal state counts as expanded in the literature. Some search algorithms are defined to already stop at the generation of a goal state. Some algorithm have the additional operation of selecting a state before expanding it. Then a goal state is only selected but not expanded. In this thesis, we count goal states as expanded because it considerably simplifies the analysis of search algorithms. We will discuss whenever our view conflicts with the general understanding of a considered algorithm.

State-space search algorithms keep track of paths that may eventually be extended to solution paths. For each generated state except for the initial state, it maintains a reference to a predecessor state. The path is then extracted by backtracking the predecessor states from the expanded goal state. We say that a state-space search algorithm applied to a solvable state space is *complete* iff it is guaranteed to find a solution path and *optimal* iff it is guaranteed to find a cost-optimal solution path. A complete algorithm that not optimal is called *satisficing*.

The execution of a state-space search algorithm on a state space results in a sequence of state expansions, which we call *search run*.

Definition 2.5 (search run)**.** *Let $\mathcal{S}$ be a state space $\langle S, s_{init}, S_{goal}, succ\rangle$. Let $\mathcal{A}$ be a state-space search algorithm that searches on $\mathcal{S}$.*

A search run *of $\mathcal{A}$ on $\mathcal{S}$ is a sequence of states $\langle s_1, \ldots, s_n\rangle$ of length n, where $s_1 = s_{init}$ and $\mathcal{A}$ expands state s_i in iteration step i, and for each state s_j that $\mathcal{A}$ expands in iteration step $j > 1$, there is a state s_i with $s_j \in succ(s_i)$ that $\mathcal{A}$ expands in an earlier iteration step $i < j$.*

We say that a search run is *successful* iff $s_n \in S_{\text{goal}}$ and *failed*, otherwise. When η is a search run, then we denote its length with $length(\eta)$. For a given search run $\eta = \langle s_1, \ldots, s_n\rangle$ we define the *search history* of η at iteration step i as $\eta[i] = \langle s_1, \ldots, s_i\rangle$. We define $\eta[0] = \langle\rangle$ and $\eta[n] = \eta$ for $n > length(\eta)$. We call $\langle s_{i+1}, \ldots, s_n\rangle$ of η the *search future* of η.

A state-space search develops its knowledge about the given state space during a search run. In this thesis, we will meet several functions that represent the knowledge of a state-space search along a search run. We present the two most basic ones here. Let H be the set of all possible search histories of an algorithm on a state space. The function $Expanded : H \to 2^S$ represents the set of states that a search has expanded in a search history and is defined as

$$Expanded(\eta[i]) = \{s \in \eta[i]\}.$$

The function $Generated : H \to 2^S$ represents the set of states that a search has generated in a search history and is defined as

$$Generated(\eta[i]) = \{s_{\text{init}}\} \cup \bigcup_{s \in Expanded(\eta[i])} succ(s).$$

Example 2.2. *$\eta = \langle \mathrm{A}, \mathrm{C}, \mathrm{F}, \mathrm{E}, \mathrm{H}\rangle$ is a successful search run of a state-space search in the state space of Figure 2.1. Its length is 5.*

$\eta[1] = \langle \mathrm{A}\rangle$ is a search history with search future $\langle \mathrm{C}, \mathrm{F}, \mathrm{E}, \mathrm{H}\rangle$ of η. The set of expanded states is $Expanded(\eta[1]) = \{\mathrm{A}\}$ and the set of generated states is $Generated(\eta[1]) = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$.

$\eta[4] = \langle \mathrm{A}, \mathrm{C}, \mathrm{F}, \mathrm{E}\rangle$ is a search history with search future $\langle \mathrm{H}\rangle$ of η. The set of expanded states is $Expanded(\eta[4]) = \{\mathrm{A}, \mathrm{C}, \mathrm{F}, \mathrm{E}\}$ and $Generated(\eta[4]) = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}\}$ is the set of generated states.

3. Heuristic Best-First Search

Heuristic best-first search is a family of state-space search algorithms. In this family a search is guided by a heuristic toward goal states. This chapter first presents the role of "best-first" in state-space search and then formally introduces heuristics in this context. Afterwards, it introduces the two of the most prominent representatives in this family that are the main subjects in this thesis.

3.1. Best-First Search

Best-first search is a state-space search that adds some notion of best for deciding which state to expand next.

Definition 3.1 (best-first search)**.** *Let $\langle S, s_{init}, S_{goal}, succ \rangle$ be a state space.*

A best-first search *is the procedure that generates s_{init}, iteratively expands a generated state s that is considered as a best state among all generated states, and stops if the expanded state s is a goal state from S_{goal}.*

A best-first search always expands a desired goal state out of all goal states. Otherwise, the algorithm designer would have to rethink about the correct implementation of the definition of best states. The definition of best states is often expressed with a *state evaluation function*.

Definition 3.2 (state evaluation function)**.** *Let S be a finite set of states.*

A state evaluation function *$f : A \to B$ is a function where $A = S$, $A = 2^S$, or $A = S^*$.*

We call a state evaluation function with $B = \mathbb{R}$ *numeric*. The cost function $cost$ is an example of a numeric state evaluation function. Functions $Expanded$ and $Generated$ are examples of non-numeric state evaluation functions.

We now define different properties that reflex different levels of dependencies of state evaluation functions. These properties will be useful to show that some aspects of best-first searches are easier to analyze than others. A state evaluation function f is

- *state-dependent* iff f depends on states,
- *path-dependent* iff f depends on paths,
- *search-run-dependent* iff f depends on search histories,

- *successor-ordering-dependent* iff f depends on the ordering of successor states in a successor function, and
- *random-variable-dependent* iff f depends on a random variable.

Successor-ordering dependency implies search-history dependency, search-history dependency implies path dependency, and path dependency implies state dependency. A state evaluation function that is state-dependent only is called *deterministic*.

3.2. Heuristic

A heuristic typically estimates the cost of a cost-optimal s-path of a state s. In this thesis, it suffices to regard the heuristic function as an arbitrary black-box, numeric, state evaluation function that assigns some non-negative real number to each state.

Definition 3.3 (heuristic)**.** *Let $\mathcal{S}$ be a set of states.*
A heuristic *is a state evaluation function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$.*

Heuristics have some properties. Let $\mathcal{S}$ be a state space with set of states S, goal states S_{goal} and successor function $succ$. Let h be a heuristic defined for states from S. Heuristic h is

- *goal aware* if $h(s) = 0$ for all $s \in S_{\text{goal}}$, and
- *safe* if $h(s) = \infty$ implies that state s a dead end.

If $\mathcal{S}$ is associated with a cost function $cost$, then h is

- *admissible* if $h(s) \leq c^*(s)$ for all $s \in S$, and
- *consistent* if $h(s) \leq h(s') + cost(s, s')$ for all $s, s' \in S$ with $s' \in succ(s)$.

These properties play an important role in ensuring the optimality and completeness of state-space search algorithms.

3.3. Greedy Best-First Search

Greedy best-first search (GBFS) (Doran and Michie, 1966) is a satisficing heuristic best-first search algorithm. It greedily searches for a goal state. It considers any goal state as a best state and as long as no goal state is generated it prefers states with smallest h-value among all the generated but not yet expanded states.

Definition 3.4 (greedy best-first search)**.** *Let $\langle S, s_{init}, S_{goal}, succ\rangle$ be a state space and h be a heuristic defined for states from S.*

A greedy best-first search *is the algorithm that generates s_{init}, iteratively expands a generated but not expanded state s, and stops if expanded state s is a goal state from S_{goal}. It prefers to expand goal states from S_{goal} if generated and, otherwise, a state s with minimum $h(s)$ among all generated but not expanded states.*

There exist suitable names for steps and states. GBFS *opens* a state s when it generates s for the first time and *closes* a state s when it expands s. A generated but not expanded state is called *open* and an expanded state is called *closed*. The set of open states is called *open list* and the set of closed states is called *closed list*.

Note that GBFS is often defined to stop as soon as a goal state is generated. In this thesis, we define GBFS to also expand the goal state because it simplifies our analysis. The difference between expanding and only generating a goal state is negligible.

3.4. A*

A* (Hart, Nilsson, and Raphael, 1968) is an optimal heuristic best-first search algorithm that minimizes the costs of paths from the initial state to states in a state space with the ultimate goal to find a cost-optimal solution path. It uses a heuristic in order to estimate for each state s the cost of a cost-optimal solution path that passes through s. In contrast to Dijkstra's algorithm (Dijkstra, 1959), A* focuses on finding a single cost-optimal solution path and is goal directed given a well informed heuristic.

Definition 3.5 (A*)**.** *Let $\langle S, s_{init}, S_{goal}, succ\rangle$ be a state space, $cost$ be a cost function and h be a heuristic defined for states from S.*

A maintains a function $g : S \to \mathbb{R}_0^+$ that is initialized with $g(s) = \infty$ for all $s \in S$. It is the algorithm that generates s_{init} and updates $g(s_{init})$ with 0. Afterwards it iteratively expands a generated state s and stops if the expanded state s is a goal state from S_{goal}. When it expands s, then it updates $g(s') = \min\{g(s) + cost(s, s'), g(s')\}$ for each $s' \in succ(s)$. It prefers to expand a state s whose solution path cost estimate $g(s) + h(s)$ is the smallest among all those generated states s' that have not been expanded since the last change of $g(s')$. It breaks ties in favor of a goal state from S_{goal}.*

There exist suitable names for steps and states. A* *opens* a state s when it generates s for the first time. It *closes* a state s when it expands s. A* *reopens* a state s when it generates s and the value of $g(s)$ has changed since the last expansion of s. A generated state s that has not been expanded since the last change of $g(s)$ is called *open*. All other generated states are called *closed*. The set of open states is called *open list* or *frontier* and the set of closed states is called *closed list*.

Note that A* is often defined to first select a state s before it expands s, and stops as soon as a goal state is selected. Consequently, a goal state is never expanded. In this thesis,

we count the goal state as expanded because it simplifies our analysis. The difference between expanding and only selecting a goal state is negligible because the difference in number of expansions in a search run is exactly one.

A* keeps track of cheapest known paths from the initial state to each state by assigning each state s with a reference to a predecessor state that lies on the current cheapest known path from the initial state to s.

Given an admissible heuristic, A* is guaranteed to find a cost-optimal solution path in a solvable state space. If the heuristic is consistent as well, then A* is guaranteed to never reopen a state (Hart, Nilsson, and Raphael, 1968).

4. Tie-Breaking

A best-first search algorithm expands one state in each iteration. Sometimes it faces a situation that offers more than one best state to chose from. Then it has to to break ties. Different tie-breaking decisions lead to different search runs. Therefore, best-first search algorithms are not well defined in general. We can say that each algorithm itself constitutes a family of algorithms. The algorithms in a family differ among each other in the applied tie-breaking strategies and policies. The following two sections formally introduce tie-breaking strategies and policies and discuss their subtle differences.

4.1. Strategy

An algorithm that applies a *tie-breaking strategy* may still not be well defined because a strategy can provide different tie-breaking options that result in different possible search runs.

Definition 4.1 (tie-breaking strategy)**.** *Let S be the set of states and let S' be a non-empty subset of S.*

A tie-breaking strategy *is a state evaluation function $\tau : 2^S \to 2^S$ that maps S' to a non-empty subset $S'' \subseteq S'$.*

Let τ_1 and τ_2 be tie-breaking strategies, then $\tau_2 \circ \tau_1$ defined as $\tau_2(\tau_1(S'))$ is a tie-breaking strategy.

We call states from S' *candidate* states and states from $\tau(S')$ *eligible* states. For a *combination* of tie-breaking strategies $\tau_n \circ \ldots \circ \tau_1$ we say τ_i is an *i-th round* tie-breaking strategy and τ_n is a *last-round* tie-breaking strategy.

We now define blueprints for two classes of tie-breaking strategies. One is based on set of states and the other is based on a numeric state evaluation function. Let S' be a set of states for which we guarantee that states from the set win in a tie-break. We define the tie-breaking strategy which guarantees that states from S' become eligible as

$$\tau_{S'}(S) = \begin{cases} S \cap S' & \text{if } S \cap S' \neq \emptyset \\ S & \text{otherwise.} \end{cases}$$

For example, the tie-breaking strategy that prefers goal states S_{goal} is defined as $\tau_{S_{\text{goal}}}$.

Let $f : S \to \mathbb{R}$ be a numeric state evaluation function. A tie-breaking strategy that favors states s with smallest $f(s)$ among states from a set of states S is defined as

$$\tau_f(S) = \{s \in S \mid f(s) = \min_{s' \in S} f(s')\}.$$

For example, a tie-breaking strategy that prefers states with low h-values is defined as τ_h.

With $\mathcal{A}_\tau$ we denote the algorithm $\mathcal{A}$ that applies tie-breaking strategy τ. $\mathcal{A}_\tau$ is considered as a sub-family of algorithms from $\mathcal{A}$.

4.2. Policy

An algorithm that applies a *tie-breaking policy* determines a single search run because a policy defines a single option for each tie-breaking situation.

Definition 4.2 (tie-breaking policy)**.** *Let S be a set of states and let S' be a non-empty subset of S.*

A tie-breaking decision *$S' \to s$ is a mapping from S' to a state s from S'.*

A tie-breaking policy *is a state evaluation function $\pi : 2^S \to S$ that consist of a set of tie-breaking decisions.*

Let π be a tie-breaking policy and τ be a tie-breaking strategy, then $\pi \circ \tau$ defined as $\pi(\tau(S'))$ is a tie-breaking policy.

A tie-breaking decision $S' \to s$ with $|S'| = 1$ is called *trivial*. A policy is the special case of a strategy τ where $|\tau(S')| = 1$ for all non-empty $S' \subseteq S$.

We now present the most commonly considered tie-breaking policies that are used in actual implementations of search algorithms. Let S be a set of candidate states. The standard tie-breaking policies are

- π^{fifo}, which expands the earliest generated state from S,
- π^{lifo}, which expands the latest generated state from S, and
- π^{rand}, which expands a state from S at random.

With $\mathcal{A}_\pi$ we denote the algorithm $\mathcal{A}$ that applies tie-breaking policy π. Let Π be the set of all possible tie-breaking policies, then $\mathcal{A}$ is a family of algorithms that includes all algorithms from $\bigcup_{\pi \in \Pi} \mathcal{A}_\pi$. Algorithms from $\bigcup_{\pi \in \Pi} \mathcal{A}_\pi$ are well defined and determine single search runs. With $\eta_{\mathcal{A}_\pi}$, we denote that the run results from $\mathcal{A}$ with policy π. We write η_π if the algorithm is clear from the context. We write η if the algorithm and the policy are clear from the context or if the policy is not explicitly defined. When we write about an algorithm $\mathcal{A}$ without specifying a tie-breaking policy, then we always consider $\mathcal{A}$ under any possible tie-breaking policy.

Part I.

Search Behavior of State-Space Search Algorithms

This part of the thesis summarizes general aspects about the behavior of state-space search algorithms. It motivates this thesis by showing the practical implications from theoretical results. Moreover, it discusses why studying the behavior of GBFS is an important next step on the course of a better understanding of best-first search in general.

We first identify questions that are useful for developing a better understanding of how search algorithms behave. Then we address these questions to state-space search in general because the answers remain valid for each specialized state-space search algorithm. Afterwards, we present what is known about the behavior of A*, the most intensely studied algorithm in the family of heuristic best-first search algorithms. In the last section, we introduce an approach for comparing best-first search algorithms and clarify the relation between GBFS and A*.

5. Guiding Questions

In this chapter, we identify some basic questions from literature that are suitable for gaining a better understanding of state-space search algorithms.

Throughout this chapter, we consider the search behavior of an algorithm $\mathcal{A}$ on a fixed search instance $\mathcal{I}$. We consider $\mathcal{A}$ under any possible tie-breaking policy. An instance $\mathcal{I}$ consists of at least a state space but may also include a cost function and a heuristic.

5.1. Expanded States

A natural question to ask is whether a state-space search algorithm that runs on a given search instance expands a state and whether the expansion is necessary for finding the desired goal (e.g. Dechter and Pearl, 1985; Wilt and Ruml, 2014). The idea behind this question is to find answers that characterize states based on the information provided by the algorithm and the search instance. The answers then offers a clearer picture of how the algorithm behaves.

In this thesis we ask following main questions. Given a state-space search algorithm $\mathcal{A}$ and a search instance $\mathcal{I}$:

- Which states does $\mathcal{A}$ on $\mathcal{I}$ *never* expand?
- Which states does $\mathcal{A}$ on $\mathcal{I}$ *potentially* expand?
- Which states does $\mathcal{A}$ on $\mathcal{I}$ *necessarily* expand?

We formalize never expanded, potentially expanded and necessarily expanded states by using the definition of search runs (Definition 2.5).

Definition 5.1 (expanded states based on search runs)**.** *Let $\mathcal{A}$ be a state-space search algorithm and let $\mathcal{I}$ be a search instance with states S. Let R be the set of all search runs of $\mathcal{A}$ on $\mathcal{I}$ under any tie-breaking policy.*

- *$\mathcal{A}$ on $\mathcal{I}$* never *expands $s \in S$ if $\mathcal{A}$ expands s in none of the search runs from R.*
- *$\mathcal{A}$ on $\mathcal{I}$* potentially *expands $s \in S$ if $\mathcal{A}$ expands s in at least one search run from R.*
- *$\mathcal{A}$ on $\mathcal{I}$* necessarily *expands $s \in S$ if $\mathcal{A}$ expands s in all search runs from R.*

We could answer these questions by enumerating all possible search runs. However, this approach would be intractable for most of the algorithms on most of the search instances. Since we know the algorithm and its strategy to search through instances, we can develop compact characterizations of these states and will gain a better understanding of its search behavior.

5.2. Search Progress

Quantifying search progress is important for predicting search effort (e.g. Thayer, Stern, and Lelis, 2012; Lelis, Zilles, and Holte, 2012). Knowing the expected (remaining) runtime of an algorithm on a given search problem is a desired feature because it can support the selection of appropriate algorithms for the problem. We are interested in following question. Given a state-space search algorithm $\mathcal{A}$ and a search instance $\mathcal{I}$:

- When does $\mathcal{A}$ on $\mathcal{I}$ makes *search progress* during a search run?

We formally define search progress as a function over search histories and require that the values of the function monotonously increase with increasing iteration step numbers of a search run. A search algorithm then makes progress whenever the value of the function increases in an iteration.

Definition 5.2 (search progress)**.** *Let $\mathcal{A}$ be a state-space search algorithm and $\mathcal{I}$ be a search instance. Let R be the set of all search runs and let H be the set of all search histories of $\mathcal{A}$ on $\mathcal{I}$ under any possible tie-breaking policy.*

A progress function *is a function $p : H \to \mathbb{R}$ that maps each search history $\eta[i] \in H$ to a real value and satisfies $p(\eta[i-1]) \leq p(\eta[i])$ for $i > 0$ and each search run $\eta \in R$.*

We say that algorithm $\mathcal{A}$ makes search progress *in iteration step i of a search run η iff $p(\eta[i-1]) < p(\eta[i])$.*

This thesis will present several progress functions that capture search progress of search algorithms on different levels. We will see that some progress functions will represent more significant progress than others.

5.3. Best-Case and Worst-Case Search Runs

Knowing that an algorithm may expand many more states in its worst case than in its best case motivates the development of better tie-breaking strategies (e.g. Asai and Fukunaga, 2017b). Being able to characterize a best-case search run confirms a good understanding of a search algorithm and supports the development of better tie-breaking strategies (e.g. Corrêa, Pereira, and Ritt, 2018). Therefore, we ask following questions. Given a state-space search algorithm $\mathcal{A}$ and a search instance $\mathcal{I}$:

- Which is a best-case search run of $\mathcal{A}$ on $\mathcal{I}$?
- Which is a worst-case search run of $\mathcal{A}$ on $\mathcal{I}$?

We formally define the best-case and worst-case search runs by using the definition of search runs.

Definition 5.3 (best-case and worst-case search run)**.** *Let $\mathcal{A}$ be a state-space search algorithm and $\mathcal{I}$ be a search instance. Let R be the set of all search runs of $\mathcal{A}$ on $\mathcal{I}$ under any possible tie-breaking policy.*

- *A* best-case *search run of $\mathcal{A}$ on $\mathcal{I}$ is defined as* $\arg\min_{\eta \in R} length(\eta)$.
- *A* worst-case *search run of $\mathcal{A}$ on $\mathcal{I}$ is defined as* $\arg\max_{\eta \in R} length(\eta)$.

Enumerating all possible search runs in order to determine the perfect upper and lower bounds of search run lengths is intractable. A sufficiently detailed understanding of the search behavior of search algorithms is required to enable the characterization of best-case and worst-case search runs based on aspects of algorithms and search instances.

6. Search Behavior of State-Space Search

In this chapter, we consider the guiding questions for state-space search algorithms in general. The answers to the questions apply to all algorithms in the family of state-space search, including the algorithms which we investigate in more detail during this thesis.

Figure 6.1 shows a state space that is used as a running example during this chapter. Blue states are those ones which are expanded in all possible search runs and red states are those ones which are expanded in none of the search runs.

6.1. State Expansion and Generation

The expansion and generation of states is the main driver of state-space searches. Whenever a search expands or generates a state for the first time, it increases is its knowledge about the state space. Expanding a state for the first time reveals the successor states and the transitions. Generating a state for the first time increases the set of states that a search is aware of. We can create progress functions whose changing values can be interpreted as search progress because a state-space search explores new parts of the state space.

We define *state expansion progress* as

$$p^{\text{expa}}(\eta[i]) = |Expanded(\eta[i])|.$$

This progress function increases along a search run whenever a search expands a state

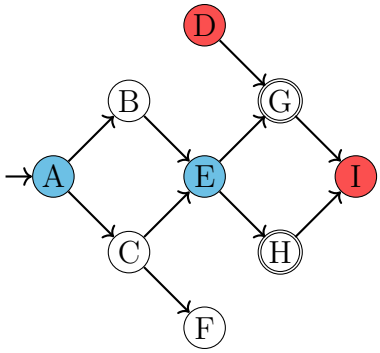


Figure 6.1: Example state space.

for the first time. The search also makes progress because it learns about successors states and possible new paths to successors states.

We define *state generation progress* as

$$p^{\text{gene}}(\eta[i]) = |Generated(\eta[i])|.$$

This progress function increases along a search run whenever a search generates a state for the first time. The search makes progress because it has found a first path to the state. Both kind of progress can be observed during a search. When we would know $p^{\text{expa}}(\eta)$ or $p^{\text{gene}}(\eta)$ of a complete search run η, then the functions could be used to report the remaining search effort during a search given the search history $\eta[i]$. There are techniques for estimating these values in order to predict the performance of search algorithms. Lelis, Zilles, and Holte (2012) introduced a method for predicting the search performance of IDA*. Thayer, Stern, and Lelis (2012) presented some methods for reporting search progress during a search based on estimates of the remaining search progress.

6.2. Unreachable States

Some parts of a state space are not reachable from the initial state. Consequently, these parts cannot be reached by any uni-directed state-space search algorithm that starts in the initial state. For example, any state-space search in our running example never expands state D because it is not reachable form the initial state A.

Proposition 6.1. *Let $\mathcal{S}$ be a state space with set of states S and initial state s_{init}. A state-space search on $\mathcal{S}$ never expands $s \in S$ if s is not reachable from s_{init}.*

State s is reachable from s_{init} iff there exists a path from s_{init} to s. A state-space search reaches s when it generates at least all the states on a path from s_{init} to s. Therefore, if there is no such path, then a state-space search cannot expand s.

6.3. Dead-End States

The expansion of dead-end states never contributes to finding a solution path. For example, state F from our running example is a dead-end state because there is no path from F to a goal state. As there is not such path, a state-space search could save this expansion. Moreover, dead ends can be harmful for state-space search algorithms, e.g., for enforced hill climbing search (Hoffmann, 2005).

Therefore, recognizing dead ends is an active branch of research in the domain of classical planning and heuristic search (Lipovetzky, Muise, and Geffner, 2016; Fickert and Hoffmann, 2017; Steinmetz and Hoffmann, 2017; Cserna et al., 2018).

The detection of dead-ends requires either a search run, an omniscient view on the state space or insights into the underlying search problem.

Whether a state expansion makes sense or not could have been another question among our guiding questions. However, we have not included this question because it is not as elementary as the other questions for gaining a better understanding of the search behavior of algorithms.

6.4. Goal States

A state-space search stops as soon as it expands a goal state. Consequently, a state-space search never expands a state that is only reachable from the initial state via goal states. For example, state I from our running example is never expanded by any state-space search because all paths to I pass through goal state G or H.

Proposition 6.2. *Let $\mathcal{S}$ be a state space with states S and goal states S_{goal}. A state-space search never expands a state $s \in S$ if all paths $\langle s_{init}, \ldots, s \rangle$ from s_{init} to s in $\mathcal{S}$ pass through a goal state from S_{goal}.*

This proposition holds because a state s can only be reached by a state-space search if there exists a path from s_{init} to s on which the search expands all states. Since the search stops after the expansion of a goal state, s can only be expanded when there is at least one path that does not pass through a goal state. This implies that s will never be expanded if it is only reachable via goal states.

Since a state-space search stops upon expansions of a goal state, we can interpret the event of expanding a goal state as search progress. It can be represented with following progress function that we call *goal progress*:

$$p^{\text{goal}}(\eta[i]) = \begin{cases} 1 & \text{if } \mathit{Expanded}(\eta[i]) \cap S_{\text{goal}} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

6.5. Bottleneck States

A state-space search needs to expand at least the initial state in order to find a goal state (since we count a goal state as expanded, we expand the initial state even if it is a goal state). It is due to the reason that all solution paths include the initial state. This is what is commonly understood as a bottleneck.

Definition 6.1 (bottleneck state)**.** *Let $\mathcal{S}$ be a state space with set of states S.*
A bottleneck state *is a state $s \in S$ that is included in every solution path of $\mathcal{S}$.*

For example, states A and E are the only bottleneck states in our running example from Figure 6.1. Since bottleneck states appear on each solution path, a state-space search has to expand all the bottleneck states in order to reach a goal from the initial state. For example, the bottleneck states A and E from our running example are necessarily expanded by a state-space search.

Proposition 6.3. *Let $\mathcal{S}$ be a state space with states S. A state-space search necessarily expands state $s \in S$ if s is a bottleneck state in $\mathcal{S}$.*

Whether a state is a bottleneck state can be tested by removing the state from the state space and running a state-space search in order to determine if the state space remains solvable. Bottleneck states cannot be directly exploited during a search without considering the underlying search problem. When we consider the underlying search problem, then bottleneck states could be detected with landmarks (Porteous, Sebastia, and Hoffmann, 2001).

Bottleneck states constitute important milestones that a search has to pass. Therefore, we can use them to define search progress function. Assume we know the set B that contains all the bottleneck states from a given state space. Then we can define *bottleneck progress* as

$$p^{\text{bott}}(\eta[i]) = |B \cap \mathit{Expanded}(\eta[i])|.$$

Bottleneck progress can alternatively be based on generated states instead of expanded states.

7. Search Behavior of A*

In this chapter, we summarize what is known about the search behavior of A*. The many practical applications and subsequent theoretical results that stem from a profound understanding of A* motivate our goal to understand the behavior of other algorithms in this family.

Figures 7.1, 7.2 and 7.3 serve as running examples of search instances during this chapter. The instances cover heuristics of different properties: consistent & admissible, inconsistent & admissible and inadmissible. For each instance we determined characteristic states: blue states are those that are expanded in every possible search run and red states are those that are expanded in none of the possible search runs. Each of the Tables 7.1, 7.2 and 7.3 shows an example A* search run for one of the example search instances.

7.1. Path Cost Minimization

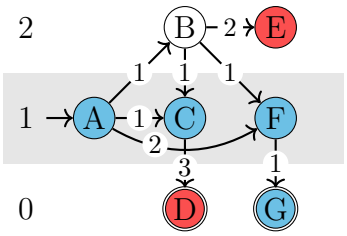
One aspect of A*'s search behavior is centered around the fact that A* minimizes $g(s)$ for each state s from a state space, i.e. it aims to find the cost of a cheapest path from the initial state to s. Each time when A* expands a state, then it has found a new cheaper path to s and expands s again only if it discovers another cheaper path to s.

The state expansion progress function from Section 6.1 may not change for several iteration steps along a search run because of re-expansions of already expanded states. We can use the changing values of g as an indicator of search progress. We write $g_{\eta[i]}$ to make clear that g changes along a search run and depends on a search history. Let q be a value that is larger than the most expensive path and S be the set of reachable states in a given state space. Then we define *path cost minimization progress* as

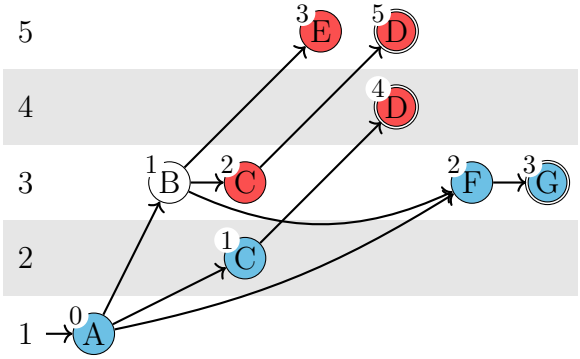
$$p^{\mathrm{g}}(\eta[i]) = -\Big(\sum_{s \in Generated(\eta[i])} g_{\eta[i]}(s) + |S \setminus Generated(\eta[i])| \cdot q \Big).$$

This function sums the path cost of all currently cheapest known paths to the generated states and adds q for each state that has not been generated yet. We add the negation to the given terms because they decrease with increasing i and the definition of progress function requires increasing values.

When A* expands a state after reaching it over a cost-optimal path from the initial state, it will never expand the state again. For example, A* expands state C in iteration 2 of the example search run from Table 7.1. As C was reached over a cost-optimal path from initial state A, A* will never expand C again.



(a) State space where states are on levels of different h-values and transitions are labeled with transition costs.



(b) Space of search nodes where each node is a state s labeled with the cost of a path to s and nodes are on levels of different f-values.

Figure 7.1: Search instance with consistent and admissible heuristic under a given cost function.

Iter.	ex.	*Open*	g
0.		$\{1 \to \{\mathrm{A}\}\}$	$\{\mathrm{A} \to 0\}$
1.	A	$\{2 \to \{\mathrm{C}\}, 3 \to \{\mathrm{B}, \mathrm{F}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 1, \mathrm{C} \to 1, \mathrm{F} \to 2\}$
2.	C	$\{3 \to \{\mathrm{B}, \mathrm{F}\}, 4 \to \{\mathrm{D}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 1, \mathrm{C} \to 1, \mathrm{D} \to 4, \mathrm{F} \to 2\}$
3.	B	$\{3 \to \{\mathrm{F}\}, 4 \to \{\mathrm{D}\}, 5 \to \{\mathrm{E}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 1, \mathrm{C} \to 1, \mathrm{D} \to 4, \mathrm{E} \to 3, \mathrm{F} \to 2\}$
4.	F	$\{3 \to \{\mathrm{G}\}, 4 \to \{\mathrm{D}\}, 5 \to \{\mathrm{E}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 1, \mathrm{C} \to 1, \mathrm{D} \to 4, \mathrm{E} \to 3, \mathrm{F} \to 2, \mathrm{G} \to 3\}$
5.	G	$\{4 \to \{\mathrm{D}\}, 5 \to \{\mathrm{E}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 1, \mathrm{C} \to 1, \mathrm{D} \to 4, \mathrm{E} \to 3, \mathrm{F} \to 2, \mathrm{G} \to 3\}$

Table 7.1: Expanded state, set of open states *Open* and g of A^* of each iteration step along search run $\langle \mathrm{A}, \mathrm{C}, \mathrm{B}, \mathrm{F}, \mathrm{G} \rangle$ in the search instance from Figure 7.1. States in *Open* are grouped by their f-values.

Proposition 7.1. *Let $\mathcal{I}$ be a search instance with state space $\mathcal{S}$, cost function $cost$ and a heuristic h that is admissible under $cost$. Let s_{init} be the initial state and S be the set of states from $\mathcal{S}$. Let $\eta[i]$ be a search history of A* on $\mathcal{I}$ and let s_i be the most recently expanded state in $\eta[i]$. Then A* never expands s_i in the search future of η if $g_{\eta[i]}(s_i) = c^\star(s_{init}, s_i)$.*

This proposition holds because s_i is reopened and possibly expanded again only if A* discovers a cheaper path to s_i. Since s_i has been expanded along a cheapest path to s_i, A* cannot find a cheaper path in the search future.

When A* uses a consistent and admissible heuristic, then $g_{\eta[i]}(s) = c^\star(s_{\text{init}}, s)$ always holds at the first expansion of s. Under such a heuristic, A* expands each state at most once (Hart, Nilsson, and Raphael, 1968). Otherwise, A* may re-expand states. Let us consider the example search run from Table 7.2 in the search instance from Figure 7.2. A* expands state C in iteration 2 but also expands C in iteration 4 because it reaches C on a cheaper path.

This event of expanding a state s after it has been reached on a cheapest path to s is search progress. The state s is then called settled as it never needs to be re-expanded because a cheapest path to s is found. Let $Settled(\eta[i])$ consists of all the states s_j that A* has expanded in iteration step $j \leq i$ of search history $\eta[i]$ and that satisfy $g_{\eta[j]}(s_j) = c^\star(s_{\text{init}}, s_j)$. We define *settled state progress* as

$$p^{\text{sett}}(\eta[i]) = |Settled(\eta[i])|.$$

Note that the analysis of this search progress requires knowing the cost-optimal path cost to states.

The phenomenon of re-expansions led to several in-depth theoretical analyses of the search behavior of A*. Martelli (1977) showed that the number of re-expansions can be $\mathcal{O}(2^n)$ in the number of states n from a given state space. Later Zhang et al. (2009) found that in the special case where the transition costs are independent of the number of states n from a state space, the number of re-expansions is $\mathcal{O}(n^2)$. These theoretical results have triggered the development of many techniques that aim to reduce the number of re-expansions by mitigating the effect of inconsistencies from an inconsistent heuristic to A* (Martelli, 1977; Bagchi and Mahanti, 1983; Mérő, 1984; Felner et al., 2005; Zhang et al., 2009).

7.2. Optimal Solution Path Cost

For A* with consistent and admissible heuristic there is a well known and easy understandable criterion that allows us to reason about the states expanded by A* based on the cost of a cost-optimal solution path.

Proposition 7.2. *Let $\mathcal{I}$ be a search instance with state space $\mathcal{S}$, cost function $cost$ and a heuristic h that is consistent and admissible under $cost$ in $\mathcal{S}$. Let $c^\star$ be the cost of a cost-optimal solution path in $\mathcal{S}$ under $cost$. Let S be the set of states and S_{goal} be the set of goal states from $\mathcal{S}$. Let $f(s)$ be defined as $c^\star(s_{init}, s) + h(s)$.*

- *A* on $\mathcal{I}$ never expands state $s \in S$ if $f(s) > c^\star$.*
- *A* on $\mathcal{I}$ necessarily expands state $s \in S$ if $f(s) < c^\star$.*

For example, Figure 7.1 shows a search instance where states A and C are necessarily expanded, and states D and E are never expanded by A*.

The given criterion is not perfect because it does not predict whether a state s with $f(s) = c^\star$ is expanded. Nevertheless, it goes a long way towards explaining the search behavior of A*. We remark that this proposition characterizes states based on optimal solution path cost. The results about never expanded and necessarily expanded states based on reachability and goal states from Chapter 6 still apply to A* and state-space search in general. What is not considered in this proposition is the fact that A* breaks ties in favor of goal states and may use other additional tie-breaking strategies.

Theoretical results of this kind are very useful to systematically investigate the behavior of A* under different state spaces, cost functions and heuristics. Dechter and Pearl (1985) discovered conditions of heuristics that induce non-pathological search instances in which A* only expands the necessarily expanded states under any possible tie-breaking policy. Helmert and Röger (2008) demonstrated that even almost perfect heuristics are not sufficient to tackle the exponential blow up of state spaces from planning tasks. As a result they suggest to focus on techniques that reduce the number of states that A* must consider. Although it is counterintuitive to what is often observed in practice, Holte (2010) showed that higher admissible h-values do not generally result in fewer state expansions. Hart, Nilsson, and Raphael (1968) already observed that states s with $h(s) = c^\star$ are critical for tie-breaking. Asai and Fukunaga (2017b) motivate the importance of tie-breaking for A* with the observation that the number of states s with $f(s) = c^\star$ increases with the number of zero-cost transitions.

For A* that does not breaks ties in favor of a goal state, we can easily characterize the best-case and worst-case search runs based on the optimal solution path cost.

Proposition 7.3. *Let $\mathcal{I}$ be a search instance with state space $\mathcal{S} = \langle S, s_{init}, S_{goal}, succ \rangle$, cost function $cost$ and a heuristic h that is consistent and admissible under $cost$ in $\mathcal{S}$. Let $c^\star$ be the cost of a cost-optimal solution path in $\mathcal{S}$ under $cost$. Let $f(s)$ be defined as $c^\star(s_{init}, s) + h(s)$. Let an* early layer *state be a state $s \in S$ with $f(s) < c^\star$ and let a* goal layer *state be a state $s \in S$ with $f(s) = c^\star$. Let ρ be a shortest path from a successor state of an early layer state to a goal state on which all states are goal layer states. Let G be the set of all goal layer states that are reachable from s_{init} on a path that only passes through non-goal states. Assume A* does not break ties by preferring goal states.*

- *A best-case search run of A**
  - *first expands all early layer states, and*
  - *then expands all states from ρ.*
- *A worst-case search run of A**
  - *first expands all early layer states,*
  - *then expands all non-goal states from G, and*
  - *finally expands a single goal state from G.*

For example, $\langle \mathrm{A}, \mathrm{C}, \mathrm{F}, \mathrm{G} \rangle$ is a best-case search run with length 4 and $\langle \mathrm{A}, \mathrm{C}, \mathrm{B}, \mathrm{F}, \mathrm{G} \rangle$ is a worst-case search run with length 5 in the example instance from Figure 7.1.

The difference between the behavior of A* in its best case and in its worst case only results from critical tie-breaking decisions among goal layer states. The understanding of the best-case behavior of A* helped Corrêa, Pereira, and Ritt (2018) to develop a perfect tie-breaking strategy for A*.

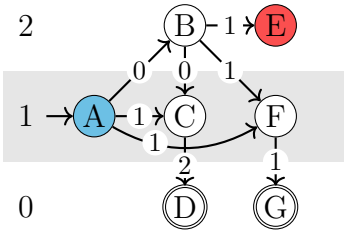
All the criteria presented in this section are restricted to consistent and admissible heuristics and to A* without tie-breaking. The criterion for never expanded states from Proposition 7.2 still holds for inconsistent and admissible heuristics. However, the answers to our guiding questions become more difficult when considering inconsistent or inadmissible heuristics, or when assuming that A* breaks ties in favor of a state with smallest heuristic value among eligible states. Asai and Fukunaga (2017b) showed that among states of same f-value A* behaves like GBFS. But we still lack a good understanding of the behavior of GBFS, which we address in this thesis.

7.3. Lower Solution Path Cost Bound

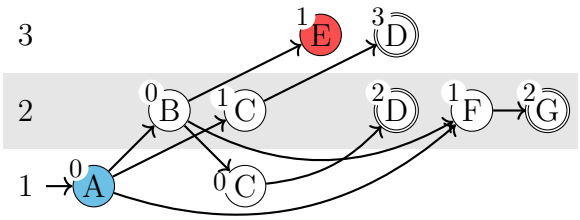
Whenever A* with admissible heuristic expands a state with f-value that is higher than that of any previously expanded state, then A* is said to make progress because a new lower bound on the optimal solution cost is found. We remind that function $f_{\eta[i]}(s) = g_{\eta[i]}(s) + h(s)$ estimates the cost of an optimal solution path that passes through s. We write $f_{\eta[i]}$ to make clear that f changes along a search run. We define *lower solution path cost bound progress* as

$$p^{\mathrm{f}}(\eta[i]) = \begin{cases} \max_{s \in \mathit{Expanded}(\eta[i])} f_{\eta[i]}(s) & \text{if } i > 0 \\ 0 & \text{otherwise.} \end{cases}$$

This progress function reflects the current lower bound on the optimal solution cost. The lower bound increases during a search run. For example, Table 7.1 shows that the highest f-value among expanded states increases when expanding A in the first iteration,



(a) State space where states are on levels of different h-values and transitions are labeled with transition costs.



(b) Space of search nodes where each node is a state s labeled with the cost of a path to s and nodes are on levels of different f-values.

Figure 7.2: Search instance with inconsistent and admissible heuristic under a given cost function.

Iter.	ex.	*Open*	g
0.		$\{1 \to \{\mathrm{A}\},\}$	$\{\mathrm{A} \to 0\}$
1.	A	$\{2 \to \{\mathrm{B},\mathrm{C},\mathrm{F}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 1, \mathrm{F} \to 1\}$
2.	C	$\{2 \to \{\mathrm{B},\mathrm{F}\}, 3 \to \{\mathrm{D}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 1, \mathrm{D} \to 3, \mathrm{F} \to 1\}$
3.	B	$\{1 \to \{\mathrm{C}\}, 2 \to \{\mathrm{F}\}, 3 \to \{\mathrm{D},\mathrm{E}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 0, \mathrm{D} \to 3, \mathrm{E} \to 1, \mathrm{F} \to 1\}$
4.	C	$\{2 \to \{\mathrm{D},\mathrm{F}\}, 4 \to \{\mathrm{E}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 0, \mathrm{D} \to 2, \mathrm{E} \to 1, \mathrm{F} \to 1\}$
5.	D	$\{2 \to \{\mathrm{F}\}, 4 \to \{\mathrm{E}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 0, \mathrm{D} \to 2, \mathrm{E} \to 1, \mathrm{F} \to 1\}$

Table 7.2: Expanded state, set of open states *Open* and g of A* of each iteration step along search run $\langle \mathrm{A}, \mathrm{C}, \mathrm{B}, \mathrm{C}, \mathrm{D} \rangle$ in the search instance from Figure 7.2. States in *Open* are grouped by their f-values.

expanding C in the second iteration and expanding B in the third iteration. This progress can be determined during an A* run without any previously extracted information from a given search instance. Moreover, it is robust on a given instance because progress happens at fixed iteration steps independently of tie-breaking.

The notion of progress based on lower solution cost bounds led to the development of new search algorithms that reduce memory consumption. Korf (1985) introduced IDA* a linear space variant of A*, which executes a depth-first search that is bounded by a lower bound on the cheapest solution path in each iteration. The algorithm increases the lower bound after each iteration allowing the search to make progress towards the optimal solution path. Korf and Zhang (2000) introduced divide-and-conquer frontier A*that discards expanded states in order to save memory. By discarding expanded states, the solution paths can get lost. A solution path is then reconstructed with a divide-and-conquer algorithm.

Lower solution path cost bound progress only applies to A* with admissible heuristic. Events that indicate progress of A* under weaker heuristic properties or under different tie-breaking strategies have not been presented to date.

7.4. Upper Solution Path Cost Bound

Some of the states that A* never expands can be pruned during a search run as soon as A* has generated some goal states because then we know an upper bound on the solution path cost. Assume $\hat{c}^\star$ is the smallest f-value of a goal state that A* has generated in the search history of a search run. Then A* does not expand a state with f-value larger than $\hat{c}^\star$ in the search future of the search run. It holds because we know that A* never expands a state with f-value above the optimal solution cost $c^\star$ (Proposition 7.2) and the solution path to a generated goal state can only be optimal or worse.

For example, the search run from Table 7.1 shows that A* generates goal state D in the second iteration. At this point we know that a goal state exists which can be reached on a solution path of cost 4. Since we know that a cost-optimal solution path will have a cost of at most 4, we can prune state E after reaching it on path of cost 5 in the third iteration.

Felner (2018) analyzed this fact by modifying A* to prune states with f-value larger than the currently best known upper solution path cost bound. Experimental results showed considerably reductions in the number of stored states for some search domains.

7.5. Minmax

Bagchi and Mahanti (1983) introduced minmax and used it to generalize the criterion from Proposition 7.2 to be applicable for arbitrary heuristics. The use of minmax goes beyond the characterization of necessarily or never expanded states. It is also used to explain why A* with inadmissible heuristic is not guaranteed to find cost-optimal solutions.

Minmax formalizes the highest f-value of a state that A* must expand in order to reach a desired goal state. It first determines the maximum f-value along a given path, which is called *pathmax*.

Definition 7.1 (pathmax)**.** *Let $\mathcal{S}$ be a state space with set of states S. Let $f : S \to \mathbb{R}$ be a state evaluation function. Let ρ be a path in $\mathcal{S}$.*

The pathmax *of ρ under f is defined as*

$$\mathit{pathmax}_f(\rho) = \max_{s \in \rho} f(s).$$

For example, in the search instance from Figure 7.2 we have $\mathit{pathmax}_f(\langle \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D} \rangle) = 2$ and $\mathit{pathmax}_f(\langle \mathrm{A}, \mathrm{C}, \mathrm{D} \rangle) = 3$.

Pathmax can be interpreted as a measure of resistance of a path. Minmax determines the minimum *pathmax*-value among given paths.

Definition 7.2 (minmax)**.** *Let $\mathcal{S}$ be a state space with set of states S. Let $f : S \to \mathbb{R}$ be a state evaluation function. Let P be a set of acyclic paths from $\mathcal{S}$.*

The minmax *of P under f is defined as*

$$minmax_f(P) := \begin{cases} \min_{\rho \in P}(\mathit{pathmax}_f(\rho)) & \text{if } P \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

For example, the minmax over paths $\langle A, B, C, D\rangle$ and $\langle A, C, D\rangle$ from Figure 7.2 is $minmax_f(\{\langle \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\rangle, \langle \mathrm{A}, \mathrm{C}, \mathrm{D}\rangle\}) = 2$.

Now, we can adapt Proposition 7.2 to be applicable for arbitrary heuristics.

Proposition 7.4. *Let $\mathcal{I}$ be a search instance with state space $\mathcal{S}$, cost function $cost$ and heuristic h. Let S be the set of states and s_{init} be the initial state from $\mathcal{S}$. Let $f(s)$ be defined as $g(s) + h(s)$ with g from the definition of A*. Let P be the set of all acyclic solution paths from $\mathcal{S}$. Let S' be the set of all states s that are reachable from s_{init} on a path on which all states (including s itself) satisfy $f(s) < minmax_f(P)$.*

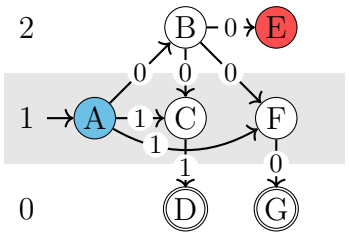
- *A* never expands state $s \in S$ if $f(s) > minmax_f(P)$ for all possible f-values of s.*
- *A* necessarily expands state $s \in S$ if $s \in S'$.*

Figure 7.2 shows a search instance where the $minmax$-value of all solution paths is 2. State A is characterized as necessarily expanded and state E is characterized as never expanded according to the criterion based on minmax. Note that in contrast to the criterion based on optimal solution path costs (Proposition 7.2), the adapted criterion correctly classifies C as a state that is not necessarily expanded. Moreover, state D is reached on two different paths with different costs that result in different f-values. Therefore, we need the condition in the criterion for never expanded states that any possible f-value of a state must be higher than $minmax_f(P)$. We are not sure whether Remark (vii) of Bagchi and Mahanti (1983) considers this fact. We leave this discussion for future work.

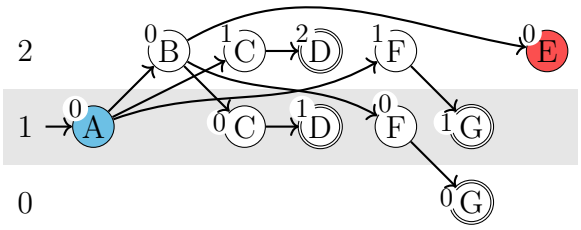
Bagchi and Mahanti (1983) used minmax to compactly formalize whether A* can find a cost-optimal solution path on a given search instance with inadmissible heuristic.

Proposition 7.5. *Let $\mathcal{I}$ be a search instance with state space $\mathcal{S}$, cost function $cost$ and heuristic h. Let S be the set of states from $\mathcal{S}$. Let $f(s)$ be defined as $g(s)+h(s)$ with g from the definition of A*. Let P be the set of all acyclic solution paths from $\mathcal{S}$. Let $P^\star$ be the set of all* cost-optimal *solution paths from $\mathcal{S}$ under $cost$. If $minmax_f(P^\star) > minmax_f(P)$ then A* cannot find a cost-optimal solution path in $\mathcal{S}$.*

This proposition holds because A* prefers states with low f-values and will find a solution path by only expanding states with f-value of at most $minmax_f(P)$ and stops before being able to find an optimal solution path. For finding the optimal solution path,



(a) State space where states are on levels of different h-values and transitions are labeled with transition costs.



(b) Space of search nodes where each node is a state s labeled with the cost of a path to s and nodes are on levels of different f-values.

Figure 7.3: Search instance with inadmissible heuristic under a given cost function.

Iter.	ex.	*Open*	g
0.		$\{1 \to \{\mathrm{A}\},\}$	$\{\mathrm{A} \to 0\}$
1.	A	$\{2 \to \{\mathrm{B},\mathrm{C},\mathrm{F}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 1, \mathrm{F} \to 1\}$
2.	B	$\{1 \to \{\mathrm{C},\mathrm{F}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 0, \mathrm{F} \to 0\}$
3.	C	$\{1 \to \{\mathrm{D},\mathrm{F}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 0, \mathrm{D} \to 1, \mathrm{F} \to 0\}$
4.	D	$\{1 \to \{\mathrm{F}\}\}$	$\{\mathrm{A} \to 0, \mathrm{B} \to 0, \mathrm{C} \to 0, \mathrm{D} \to 1, \mathrm{F} \to 0\}$

Table 7.3: Expanded state, set of open states *Open* and g of A* at the start of each iteration along search run $\langle \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D} \rangle$ in the search instance from Figure 7.3. States in *Open* are grouped by their f-values.

A* would need to expand a state with f-value of $minmax_f(P^\star)$. This proposition does not imply that A* is guaranteed to find a cost optimal solution path if $minmax_f(P) = minmax_f(P^\star)$. For example, Figure 7.3 shows a search instance with $minmax_f(P) = minmax_f(P^\star) = 2$, while Table 7.3 presents a search run of A* on the given instance that does not find an optimal solution path.

Based on their findings Bagchi and Mahanti introduced a new algorithm, called C, that finds solution paths that are as good as possible given an inadmissible heuristic. Algorithm C is designed to behave like uniform cost search after all states from S', as defined in Proposition 7.4, are expanded.

Dechter and Pearl (1985) made a first attempt towards a better understanding of the search behavior of best-first search in general. They used minmax to formalize sufficient and necessary conditions for a state to be expanded by a best-first search that uses an arbitrary but order preserving numeric state evaluation function. They formalized the conditions for state expansion under the assumptions that a state space is a tree on which some of the leaves are goal states and that tie-breaking favors the leftmost states on the tree. Their analysis is centered around the question whether a best-first search is able to reach a state s on a path $\langle s_{\text{init}}, \ldots, s \rangle$ from the initial state s_{init}. In a first step they defined the minmax of subtrees. If a subtree starts in state s, then the minmax of the subtree rooted

at s is defined to take the set of all s-paths. They used the criterion from Proposition 7.4 that a best-first search does not expand s if its $f(s)$ is above the $minmax$-value. They showed that the expansion of a state s depends on the smallest $minmax$-value among all successor states of states $s' \neq s$ from $\langle s_{\text{init}}, \ldots, s \rangle$. We denote this smallest value for state s with $b(s)$. Their final result says that a best-first search on a tree with state evaluation function f is able to expand state s if $f(s') < b(s')$ is satisfied for each state s' from $\langle s_{\text{init}}, \ldots, s \rangle$. In other words, a best-first search must be able to expand each state along a path from s_{init} to a state s in order to expand s. This is possible when all states on the path are on or below the bound given by the $minmax$-value that decreases along the path. As a tree has no transpositions and all the possible paths in a tree are acyclic, Dechter and Pearl were basically able to characterize states in a tree that a best-first search potentially expands. We will reconsider this statement in Section 13.1.

The contributions of this thesis are heavily based on the concept of minmax. We will only consider minmax with deterministic heuristics instead of arbitrary state evaluation functions and will define it for s-paths of states s instead of solution paths. We will then observe how the $minmax$-values change among different states along search runs.

8. Relations between Best-First Search Algorithms

We are interested in uncovering the relation between A* and GBFS. It is often said that GBFS is equal to A* when A* ignores transition costs. We clarify that there are more assumptions behind this statement. Our motivation stems from Holte (2010) who clarified a misconception by showing A* does not behave like breadth-first search on a search instance with unit cost function and a constant heuristic.

Dechter and Pearl (1985) showed for different classes of search instances whether A* is an optimal algorithm, in terms of number of expanded states, on a given instance by reasoning over any possible tie-breaking policy. In this chapter we ask in which search instances and under which assumptions GBFS equals A*.

We first introduce an alternative view on best-first search and then define equivalence of state-space search algorithms, which allows us to formally compare GBFS with A*.

8.1. Best-First Search as Tie-Breaking Strategy

A best-first search algorithm defines which states are the best states among all generated states. It can be understood as a combination of tie-breaking strategies where the first strategy is applied to all states from the state space and the strategy selects all the generated states. Let us formalize this with $\tau_{Generated}$ where $Generated$ contains all generated states and $\tau_{Generated}$ is $\tau_{S'}$ with $S' = Generated$ as defined in Section 4.1. The second tie-breaking round of GBFS selects all states that have not been expanded so far, which we can formalize with τ_{Open} where $Open$ contains all generated but not expanded states. With $\tau_{S_{\text{goal}}}$ and τ_h as defined in Section 4.1, we can represent GBFS as a best-first search with tie-breaking strategy

$$\tau^{\text{GBFS}} = \tau_h \circ \tau_{S_{\text{goal}}} \circ \tau_{Open} \circ \tau_{Generated}.$$

Strategy $\tau^{\text{GBFS}}(S)$ returns those states among all states from a set of states S that are eligible for expansion in the current iteration of a search run.

Let f be the sum of g and h, with heuristic h and function g as defined in the definition of A*. Let $Open'$ be the set of states s that has not been expanded since the last change of $g(s)$. Then A* can be represented as a best-first search with tie-breaking strategy

$$\tau^{\text{A}^*} = \tau_{S_{\text{goal}}} \circ \tau_f \circ \tau_{Open'} \circ \tau_{Generated}.$$

These representations of GBFS and A* as combinations of tie-breaking strategies help to analyze their difference.

8.2. Specialization and Equivalence

In order to establish a formal basis for comparing A* and GBFS to each other, we define specialization and equivalence of state-space search algorithms. The definition is not to be confused with concept of dominance (Dechter and Pearl, 1985).

Definition 8.1 (specialization and equivalence of state-space search algorithms)**.** *Let $\mathcal{I}$ denote a search instance. Let A and B be two state-space search algorithms. Let R_A and R_B be the sets of all search runs of A and B on $\mathcal{I}$ under any possible tie-breaking policy.*

- *B* specializes *A on $\mathcal{I}$, denoted with $B \subseteq_{\mathcal{I}} A$, iff $R_B \subseteq R_A$.*
- *B is equivalent to A on $\mathcal{I}$, denoted with $B \equiv_{\mathcal{I}} A$, iff $B \subseteq_{\mathcal{I}} A$ and $A \subseteq_{\mathcal{I}} B$.*

We write $B \subseteq A$ iff $B \subseteq_{\mathcal{I}} A$ for all $\mathcal{I}$ and $B \equiv A$ iff $B \equiv_{\mathcal{I}} A$ for all $\mathcal{I}$.

We show a straight forward example based on A*. Let τ^{A^*} be A* represented as tie-breaking strategy and let τ_h be the tie-breaking strategy that prefers states with smaller heuristic values. Then it is easy to see that $\tau_h \circ \tau^{\text{A}^*} \subseteq \tau^{\text{A}^*}$. Another example: let $\mathcal{I}$ be a search instance with a state space that contains a single solution path and only consists of states and transitions from the solution path, then it is easy to see that GBFS $\equiv_{\mathcal{I}}$ A* on $\mathcal{I}$.

8.3. Relation between A* and GBFS

We now clarify the relation between GBFS and A*. Instead of modifying A* such that it becomes GBFS we identify a search instance on which both algorithms express the same behavior. Viewing the search algorithms as tie-breaking strategies for best-first search helps to structure the proof.

Proposition 8.1. *Let $\mathcal{I}$ be a search instance with state space $\mathcal{S}$, cost function $cost$ and heuristic h. Let S be the states from $\mathcal{S}$. Let $cost(s, s') = 0$ with $s' \in succ(s)$ for all $s, s' \in S$ and let h be goal-aware. Assume $\mathcal{I}$ is given and A* breaks ties in favor of goal states, then GBFS $\equiv_{\mathcal{I}}$ A*.*

Proof. We use the representations of GBFS and A* as tie-breaking strategies

$$\tau^{\text{GBFS}} = \tau_h \circ \tau_{S_{\text{goal}}} \circ \tau_{Open} \circ \tau_{Generated}$$

and

$$\tau^{\text{A}^*} = \tau_{S_{\text{goal}}} \circ \tau_f \circ \tau_{Open'} \circ \tau_{Generated}.$$

We use following notation $\tau^{\text{GBFS}} \equiv_{\mathcal{I}} \tau^{\text{A}^*}$ analogous to GBFS $\equiv_{\mathcal{I}}$ A*.

As $\tau_{Generated}$ is equal for both algorithms, it remains to show that $\tau_h \circ \tau_{S_{\text{goal}}} \circ \tau_{Open} \equiv_{\mathcal{I}} \tau_{S_{\text{goal}}} \circ \tau_f \circ \tau_{Open'}$.

We know that $s \in Open'$ of A* if s has not been expanded since the last change of $g(s)$, and $g(s)$ is the cost of a path from the initial state to a state s under $cost$ or ∞ if s has not been generated yet. Since $cost(s, s') = 0$ with $s' \in succ(s)$ for all $s, s' \in S$, we have $g(s) = 0$ for all s if s is generated and $g(s) = \infty$, otherwise. Consequently, there is only one change of $g(s)$, which is from ∞ to 0 and it can only happen at the first generation of s. As $g(s)$ is updated at most once we can say that $s \in Open'$ if s is generated but not expanded, which is equal to the semantic of $s \in Open$ of GBFS. Consequently, τ_{Open} is equivalent to $\tau_{Open'}$ under the given cost function $cost$. It remains to show $\tau_h \circ \tau_{S_{\text{goal}}} \equiv_{\mathcal{I}} \tau_{S_{\text{goal}}} \circ \tau_f$.

We know that $f(s) = g(s) + h(s)$ and we know that $g(s) = 0$ if s is generated due to the given cost function $cost$ with $cost(s, s') = 0$ with $s' \in succ(s)$ for all $s, s' \in S$. As s passed the first round strategy $\tau_{Generated}$, we know that s is generated, which results in $f(s) = h(s)$. Consequently, τ_f is equivalent to τ_h under the given cost function $cost$. Therefore, we can replace τ_f with τ_h, which results in $\tau_h \circ \tau_{S_{\text{goal}}} \equiv_{\mathcal{I}} \tau_{S_{\text{goal}}} \circ \tau_h$. It remains to show that this equivalence relation is valid.

As h is goal-aware, we know that $h(s) = 0$ if $s \in S_{\text{goal}}$. Let S' be a set of states and let s be a state from S'. We know that $s \in \tau_h(S')$ if $h(s) = 0$ because τ_h minimizes the h-value among all states from S' and 0 is the lowest possible h-value. We also know that $s \in \tau_{S_{\text{goal}}}(S')$ if $s \in S_{\text{goal}}$. Consequently, if $s \in S_{\text{goal}}$, then $s \in \tau_{S_{\text{goal}}}(S')$ and $s \in \tau_h(S')$. In this case, $\tau_h \circ \tau_{S_{\text{goal}}}$ is equivalent to $\tau_{S_{\text{goal}}} \circ \tau_h$. We now show the other case. If $s \notin S_{\text{goal}}$, then $s \in \tau_{S_{\text{goal}}}(S')$ if S' contains no goal state or $s \notin \tau_{S_{\text{goal}}}(S')$, otherwise. $\tau_{S_{\text{goal}}}$ either returns all states from S' or only the goal states by definition. A strategy that returns all states has no effect on the combined strategy independent of the round in which it appears. In this case, $\tau_h \circ \tau_{S_{\text{goal}}}$ is equivalent to $\tau_{S_{\text{goal}}} \circ \tau_h$. The case, where $\tau_{S_{\text{goal}}}$ only returns the goal states was shown in case $s \in S_{\text{goal}}$. Therefore, $\tau_h \circ \tau_{S_{\text{goal}}}$ is equivalent to $\tau_{S_{\text{goal}}} \circ \tau_h$ under any goal-aware heuristic.

As we showed that the combinations of tie-breaking strategies are equivalent, we conclude that $\tau^{\text{GBFS}} \equiv_{\mathcal{I}} \tau^{\text{A}^*}$. □

Since GBFS ignores the cost function, any criterion that characterizes the behavior of A* based on transition cost turns out to not be applicable to GBFS. Nevertheless, we believe that a better understanding of the behavior of GBFS might support our understanding of A*'s search behavior under inadmissible heuristics, e.g. weighted A* (Pohl, 1970). It may lead to a better understanding of experimental results that compare weighted A* under different weights (e.g. Wilt and Ruml, 2012).

Part II.

Search Behavior of Greedy Best-First Search

In this central part of this thesis, we study the search behavior of GBFS by focussing on our guiding questions from Section 5. Our goal is to characterize states that GBFS never, necessarily, or potentially expands. Moreover, we are interested in determining the best-case and worst-case search runs of GBFS in a given state space and with a given heuristic.

We begin with introducing the state space topology, which visualizes aspects of the behavior of GBFS that can be represented as a static environment. Then we present high-water marks that are characteristic values of states based on s-paths and that perfectly reflect the behavior of GBFS locally in a state space topology. We show that the high-water mark determines states that GBFS ignores. We observe that the high-water mark decreases during a search and that a decrease of the high-water mark indicates search progress. The notion of search progress reveals that a search run can be understood as a sequence of episodes where each episode searches in a subspace of the state space, which we call bench. Afterwards we study the behavior of GBFS on benches and show that a search can be trapped in subspaces, which we call craters. Finally, we sufficiently understand the behavior of GBFS such that we are able to characterize and determine best-case and worst-case search runs.

9. Representation of Greediness

We often find the statement that GBFS is greedy because it only considers the heuristic and ignores transition costs (e.g. Russell and Norvig, 2003). However, we find some more reasons for the greediness of GBFS that we want to clarify.

GBFS expresses its greediness by stopping as soon as a goal state is generated. For simplifying our analysis we say that GBFS stops after expanding a generated goal state. Nevertheless, GBFS will not expand a non-goal state after a goal state has been generated because it always first expands the goal state. We can say that GBFS searches more greedily for goal states than for states with small heuristic values.

GBFS also expresses its greediness by forbidding multiple expansions of states. It stands in contrast to A* that may expand states several times. However, A* with consistent and admissible heuristic A* expands each state at most once (Hart, Nilsson, and Raphael, 1968). In contrast to GBFS this is due to the behavior of A* under consistent and admissible heuristic and not due to the definition of A*.

We will represent the preference of GBFS for goal states and states with small heuristic values in a static environment called state space topology and as a function called high-water mark. Afterwards, we discuss how we include the fact in our analysis that GBFS expands each state at most once.

9.1. State Space Topology

A *state space topology* is a state space combined with a state evaluation that can be a heuristic (Hoffmann, 2005). It is often viewed as a landscape with hills, valleys and plateaus.

Definition 9.1 (state space topology)**.** *Let $\mathcal{S}$ be a state space with set of states S. Let $f : S \to \mathbb{R}$ be a deterministic state evaluation function.*

A state space topology *$\mathcal{T}$ is defined as $\langle \mathcal{S}, f \rangle$.*

Example 9.1. *Figure 9.1 shows a state space topology $\langle \mathcal{S}, h \rangle$ that we use as a running example throughout this part of this thesis. It has state space $\mathcal{S} = \langle \{\mathrm{A}, \ldots, \mathrm{Z}\}, \mathrm{B}, \{\mathrm{Y}, \mathrm{Z}\}, succ \rangle$ and uses heuristic h as state evaluation function. The numbered layers define the h-values of states.*

By requiring a deterministic state evaluation function, we restrict a topology to be static. This restriction excludes any path-dependent state evaluation function from be-

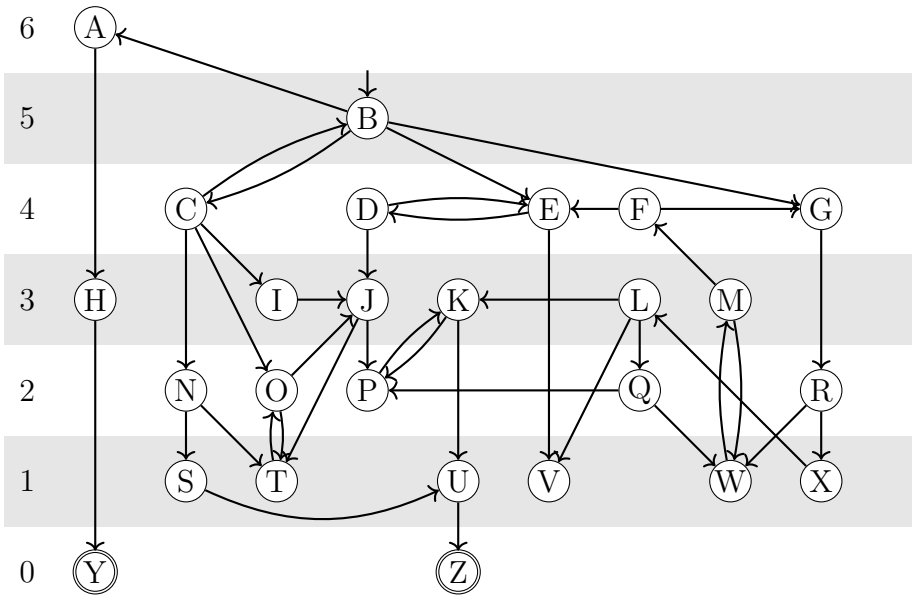


Figure 9.1: Example state space topology.

ing involved in a topology. In this thesis, we analyze GBFS in it simplest form by always assuming a deterministic heuristic.

The role of a state evaluation function in a state space topology is to represent the preferences of a best-first search. The function assigns more preferred states with smaller values than those of less preferred ones. While a topology can represent the preferences of arbitrary best-first search algorithms, we focus on the preferences of GBFS.

A heuristic already assigns more preferred states with lower values. It remains to encode the fact that GBFS prefers goal states. Preferring goal states is a tie-breaking strategy and tie-breaking strategies induce rankings among states.

Definition 9.2 (state ranking under tie-breaking strategy)**.** *Let S be a set of states and τ be a tie-breaking strategy.*

The state ranking under τ is a state evaluation function $rank_\tau : S \to \mathbb{R}$ that maps a state $s \in S$ to a value $r \in \mathbb{R}$ and has the following semantic: for a pair of states $s, s' \in S$, state s has a higher rank (lower ranking value) than s' under τ iff $rank_\tau(s) < rank_\tau(s')$.

Example 9.2. *The state ranking of states s under a tie-breaking strategy τ_h based on heuristic h can be defined as $rank_{\tau_h}(s) = h(s)$. The state ranking of states under a tie-breaking strategy τ_{goal} that prefers goal states from S_{goal} can be defined as $rank_{\tau_{goal}}(s) = 0$ if $s \in S_{goal}$ and $rank_{\tau_{goal}}(s) = 1$, otherwise.*

Ranking function $rank_{\tau_h}$ represents the preference of GBFS for small h-values and ranking function $rank_{\tau_{goal}}$ represents the preference for goal states. We now combine these ranking functions to formalize that GBFS prefers goal states over low h-values. For

this, we extend the definition of state ranking to consider combinations of tie-breaking strategies.

Definition 9.3 (state ranking under a combination of tie-breaking policies)**.** *Let $\tau_n \circ \ldots \circ \tau_1$ be a combination of tie-breaking strategies.*

A state ranking *under $\tau_n \circ \ldots \circ \tau_1$ is a state evaluation function $rank_{\tau_n \circ \ldots \circ \tau_1} : S \to \mathbb{R}^n$ that maps each state $s \in S$ to the tuple $\langle rank_{\tau_1}(s), \ldots, rank_{\tau_n}(s) \rangle$.*

For two values $\langle a_1, \ldots, a_m \rangle \in \mathbb{R}^m$ and $\langle b_1, \ldots, b_n \rangle \in \mathbb{R}^n$, the relation $\langle a_1, \ldots, a_m \rangle < \langle b_1, \ldots, b_n \rangle$ is defined to be true if $\langle a_1, \ldots, a_m \rangle$ precedes $\langle b_1, \ldots, b_n \rangle$ in an ascending natural ordering.

Example 9.3. *Continuing the previous example, the ranking of a state s under $\tau_h \circ \tau_{goal}$ is defined as $rank_{\tau_h \circ \tau_{goal}}(s) = \langle rank_{\tau_{goal}}(s), rank_{\tau_h}(s) \rangle$.*

The state ranking $rank_{\tau_h \circ \tau_{goal}}(s)$ reflects the primary preference of GBFS for goal states and the secondary preference for small h-values.

A ranking of states can be used as a state evaluation function for a state space topology by mapping the multidimensional *rank*-values into single real values. In the remainder of this thesis, we assume that a goal state always has a smaller h-value than that of all the non-goal states. This assumption saves the encoding of the preference of GBFS for goals into a numeric state evaluation function via a state ranking function. Nevertheless, we clarify that state ranking functions can help to consider GBFS with additional tie-breaking strategies. For example, we could analyze the effect of an additional heuristic as tie-breaker. Moreover, we think that this approach to represent different aspects of an algorithm in a state space topology is also an important tool for the analysis of A* with different tie-breaking strategies and the analysis of A* with inadmissible heuristics.

9.2. High-Water Mark

High-water marks indicate how water flows in a landscape. They are an analogy that describes how a GBFS proceeds in a state space topology. Water flows along a steepest descent to a lower point in a landscape due to gravity, and pools when no lower point is reachable. Similarly, GBFS expands states with small values due to preferring small heuristic values and expands states with higher values when no state with smaller value is available.

The high-water mark was first used for the analysis of the search behavior of GBFS in undirected state spaces by Wilt and Ruml (2014). However, it was introduced earlier as minmax by Bagchi and Mahanti (1983) and used by Dechter and Pearl (1985) for the analysis of other algorithms from the family of heuristic best-first search.

The high-water mark represents the greedy behavior of GBFS in context of a state space topology. It is the highest h-value of a state that GBFS starting in a state s expands in order to reach a goal state from s. Due to preferring states with low h-values, GBFS

will never expand a state above the high-water mark. Independent of GBFS and based on paths, the high-water mark of a state s is the maximum h-value of a state s' on an s-path, while there is no other s-path on which the maximum h-value of a state is smaller than that of s'. We define high-water mark as a specialization of minmax (Section 7.5).

Definition 9.4 (high-water mark of state)**.** *Let $\mathcal{T}$ be a state space topology $\langle \mathcal{S}, h \rangle$ and let S be states from $\mathcal{S}$. Let s be a state from S and let $P(s)$ be the set of all acyclic s-paths in $\mathcal{S}$.*

The high-water mark *of s in $\mathcal{T}$ is defined as*

$$hwm(s) = minmax_h(P(s)).$$

We say that an s-path *determines* $hwm(s)$ if it minimizes the *pathmax*-value among all s-paths. Moreover, we say s' *determines* $hwm(s)$ if it maximizes the h-value among all states from an s-path that determines $hwm(s)$.

Example 9.4. *As an example, we determine the high-water mark of* Q*. The set of acyclic Q-paths is $P = \{\rho_0, \rho_1, \rho_2\}$ with*

$$\begin{aligned}\rho_0 =& \langle \mathrm{Q}, \mathrm{P}, \mathrm{K}, \mathrm{U}, \mathrm{Z} \rangle \\ \rho_1 =& \langle \mathrm{Q}, \mathrm{W}, \mathrm{M}, \mathrm{F}, \mathrm{G}, \mathrm{R}, \mathrm{X}, \mathrm{L}, \mathrm{K}, \mathrm{U}, \mathrm{Z} \rangle \\ \rho_2 =& \langle \mathrm{Q}, \mathrm{W}, \mathrm{M}, \mathrm{F}, \mathrm{E}, \mathrm{D}, \mathrm{J}, \mathrm{P}, \mathrm{K}, \mathrm{U}, \mathrm{Z} \rangle\end{aligned}$$

The pathmax-values are as follows: pathmax$(\rho_0) = 3$ is determined by K *with $h(\mathrm{K}) = 3$, pathmax$(\rho_1) = 4$ is determined by* F *or* G *with $h(\mathrm{F}) = h(\mathrm{G}) = 4$, and pathmax$(\rho_2) = 4$ is determined by D, E or* F *with $h(\mathrm{D}) = h(\mathrm{E}) = h(\mathrm{F}) = 4$.*

The high-water mark $hwm(\mathrm{Q}) = 3$ is determined by ρ_0 because pathmax(ρ_0) is the minimum pathmax-value among all Q-paths from P.

Figure 9.2 annotates all states from the example state space topology with their hwm-values.

To reflect the fact that GBFS often chooses among several states, we extend the definition of high-water mark to include set of states. Dechter and Pearl (1985) have already considered $minmax$-values among several states in their analysis. The high-water mark of a set of states S' is simply the lowest high-water mark among states from S'. This is because we know that $minmax$ minimizes the *pathmax*-values among a set of paths P. We can define P to include all acyclic s-paths of each state s form S'. Alternatively, we can define a separate set of acyclic s-paths $P(s)$ for each s from S' and then first minimize the *pathmax*-values among paths from $P(s)$ for each s individually before minimizing the values among the different s from S'.

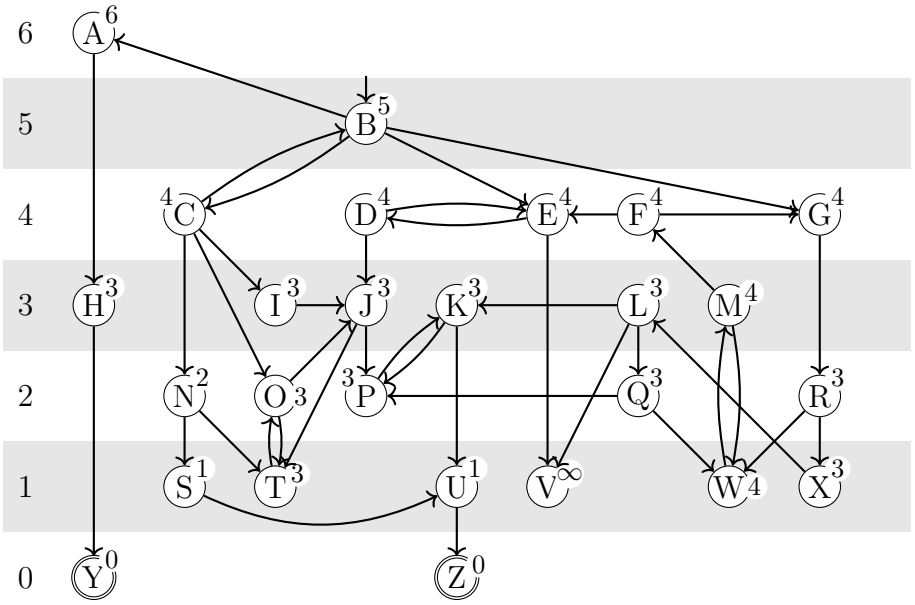


Figure 9.2: Example state space topology where states are annotated with their hwm-values.

Definition 9.5 (high-water mark of set of states)**.** *Let $\mathcal{T}$ be a state space topology with states S and heuristic h. Let S' be a subset from S.*

The high-water mark *of S' in $\mathcal{T}$ is defined as*

$$hwm(S') = \begin{cases} \min_{s \in S'} hwm(s) & \text{if } S' \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

We say that a state s *determines* $hwm(S')$ if s minimizes $\min_{s \in S'} hwm(s)$.

Example 9.5. *As an example, we determine the high-water mark of $\{W, X\}$. The high-water mark of the single states are $hwm(\mathrm{W}) = 4$ and $hwm(\mathrm{X}) = 3$. The high-water mark $hwm(\{\mathrm{W}, \mathrm{X}\}) = 3$ is determined by state X because $hwm(\mathrm{X})$ is smaller than $hwm(\mathrm{W})$.*

Functional Equation We introduce the functional equation for the high-water mark of a state because it is important for the computation of high-water marks and deepens our understanding of the search behavior of GBFS. It uses the fact that the high-water mark of a state only depends on the state and its successor states. The solutions of the functional equations are initially known for state s if s is a goal state or if s has no successor states. The set of functional equations of all states from a state space constitutes a system of functional equations. The solution of the functional equation system determines the high-water mark of all the states.

Definition 9.6 (functional equation of high-water mark of state)**.** *Let $\mathcal{T}$ be a state space topology with states S, goal states S_{goal}, successor function $succ$ and heuristic h.*

The functional equation for high-water mark of s in $\mathcal{T}$ is defined as

$$hwm(s)_{fe} = \begin{cases} h(s) & \text{if } s \in S_{goal} \\ \infty & \text{if } s \notin S_{goal} \text{ and } succ(s) = \emptyset \\ \max(h(s), \min_{s' \in succ(s)} hwm_{fe}(s')) & \text{otherwise.} \end{cases}$$

We show that the functional equation for the high-water mark of a state matches the definition of high-water mark.

Theorem 9.1. *Let $\mathcal{T}$ be a state space topology with states S, then $hwm(s) = hwm_{fe}(s)$ for all $s \in S$.*

Proof. We first prove the base cases.

- Case $s \in S_{\text{goal}}$: $s \in S_{\text{goal}}$ implies that $\langle s \rangle$ is an s-path and $\langle s \rangle$ determines $hwm(s)$ independently of any other s-path. Therefore, $hwm(s) = h(s) = hwm_{\text{fe}}(s)$ holds for all $s \in S_{\text{goal}}$.
- Case $s \notin S_{\text{goal}}$ and $succ(s) = \emptyset$: $s \notin S_{\text{goal}}$ and $succ(s) = \emptyset$ imply that there is no s-path. The hwm-value over an empty set of paths is ∞ by definition of high-water mark. Therefore, $hwm(s) = \infty = hwm_{\text{fe}}(s)$ holds for all $s \in S$ with $s \notin S_{\text{goal}}$ and $succ(s) = \emptyset$.
- All other cases:

$$\begin{aligned} hwm(s) &= \min_{\langle s,s',\ldots,s^* \rangle \in P(s)} \Big(\max_{s'' \in \langle s,s',\ldots,s^* \rangle} h(s'') \Big) \\ &= \min_{\langle s,s',\ldots,s^* \rangle \in P(s)} \big(\max\{h(s), \max_{s'' \in \langle s',\ldots,s^* \rangle} h(s'')\} \big) \\ &\overset{(i)}{=} \max\{h(s), \min_{\langle s,\ldots,s^* \rangle \in P(s)} \Big(\max_{s'' \in \langle s',\ldots,s^* \rangle} h(s'') \Big)\} \\ &\overset{(ii)}{=} \max\{h(s), \min_{s' \in succ(s)} \Big(\min_{\langle s',\ldots,s^* \rangle \in P(s')} \Big(\max_{s'' \in \langle s',\ldots,s^* \rangle} h(s'') \Big) \Big)\} \\ &\overset{(iii)}{=} \max\{h(s), \min_{s' \in succ(s)} hwm(s')\} \\ &= hwm_{\text{fe}}(s) \end{aligned}$$

(i) If s with $h(s)$ determines the *pathmax*-value of at least one s-path, then such an s-path minimizes the *pathmax*-values among all s-paths because other s-paths can only have *pathmax*-values equal or larger than $h(s)$.

If s does not determine the *pathmax*-value of any s-path, then the *pathmax*-value of each s-path is determined by a state $s' \neq s$ with $h(s') > h(s)$. This implies that the minimum *pathmax*-value among s-paths is larger than $h(s)$.

Consequently, either $h(s)$ is the maximum or there is another state that contributes to the maximum.

(ii) Each s-path passes through at least one of the successor states s' of s. Therefore, we can first minimize the *pathmax*-value among s'-paths for each $s' \in succ(s)$ separately and then further minimize the value among s'.

(iii) By definition of high-water mark (Definition 9.4).

We have shown that $hwm(s) = hwm_{\text{fe}}(s)$ in all different cases for all $s \in S$. □

The recursive term of the functional equation suggest a recursive procedure for solving the functional equation system. However, cycles in the state space induce dependencies that are not easy to resolve in a recursive procedure. Fortunately, we can show that high-water marks are monotonic and obey *Bellman's Principle of Optimality* (Bellman, 1957) such that there exists a topological ordering of states that is suitable for solving the functional equation system with a successive approximation method.

Theorem 9.2. *Let $\mathcal{T}$ be a state space topology with states S, goal states S_{goal} and successor function $succ$. Then $hwm(s) \geq \min_{s' \in succ(s)} hwm(s')$ for all states $s \in S$ with $s \notin S_{goal}$.*

Proof. We show the statement by using the recursive term of the functional equation of the high-water mark of state (Definition 9.6).

$$
\begin{aligned}
hwm(s) &= \max(h(s), \min_{s' \in succ(s)} hwm(s')) \\
&= \begin{cases} h(s) & \text{if } h(s) > \min_{s' \in succ(s)} hwm(s') \\ \min_{s' \in succ(s)} hwm(s') & \text{if } h(s) \leq \min_{s' \in succ(s)} hwm(s') \end{cases} \\
&\text{iff } \left(hwm(s) > \min_{s' \in succ(s)} hwm(s')\right) \text{ or } \left(hwm(s) = \min_{s' \in succ(s)} hwm(s')\right) \\
&\text{iff } hwm(s) \geq \min_{s' \in succ(s)} hwm(s')
\end{aligned}
$$

□

The monotonicity implies that the high-water mark of all states in a state space topology can be computed in time and space polynomial in the number n of states from a state space topology with dynamic programming. In a first step, we exhaustively explore the reachable state space, which requires at most n expansions. Then we backtrack from the goal states such that states with low hwm-values are processed first. In this way each of the n states is processed at most once.

Iter.	expan.	*Open*	$hwm(Generated)$
0.		$\{5 \mapsto \{\mathrm{B}\}\}$	5
1.	B	$\{4 \mapsto \{\mathrm{C}, \mathrm{E}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	4
2.	E	$\{1 \mapsto \{\mathrm{V}\}, 4 \mapsto \{\mathrm{C}, \mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	4
3.	V	$\{4 \mapsto \{\mathrm{C}, \mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	4
4.	C	$\{2 \mapsto \{\mathrm{N}, \mathrm{O}\}, 3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	2
5.	O	$\{1 \mapsto \{\mathrm{T}\}, 2 \mapsto \{\mathrm{N}\}, 3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	2
6.	T	$\{2 \mapsto \{\mathrm{N}\}, 3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	2
7.	N	$\{1 \mapsto \{\mathrm{S}\}, 3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	1
8.	S	$\{1 \mapsto \{\mathrm{U}\}, 3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	1
9.	U	$\{0 \mapsto \{\mathrm{Z}\}, 3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	0
10.	Z	$\{3 \mapsto \{\mathrm{I}\}, 4 \mapsto \{\mathrm{D}, \mathrm{G}\}, 6 \mapsto \{\mathrm{A}\}\}$	0

Table 9.1: Iteration step numbers, expanded states, sets of open states $Open$ and high-water marks of set of generated states $Generated$ along search run $\langle \mathrm{B}, \mathrm{E}, \mathrm{V}, \mathrm{C}, \mathrm{O}, \mathrm{T}, \mathrm{N}, \mathrm{S}, \mathrm{U}, \mathrm{Z} \rangle$ of GBFS in our example state space topology from Figure 9.1. States in $Open$ are grouped by their h-values.

9.3. Expanded States

GBFS expands each state at most once. For example, consider the search run listed in Table 9.1. In this run GBFS expands state V in the third iteration. Since GBFS never expands V again during the rest of its run, it is able to continue the search by expanding state C instead of expanding V infinitely often. We highlight this well known behavior of GBFS in following proposition.

Proposition 9.1. *Let $\mathcal{T}$ be a state space topology with states S. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$ and let $Expanded(\eta[i])$ be the set of states that GBFS has expanded in $\eta[i]$. Then GBFS never expands $s \in Expanded(\eta[i])$ in the search future of η.*

Note that we aim to eventually find a criterion for states that GBFS never expands in all search runs independently of a concrete search history. Nevertheless, this conditional criterion is an important aspect of GBFS. As a state becomes a never expanded state through its expansion, we can consider this event as search progress. We have already discussed this kind of progress in Section 5.1 and called *state expansion progress*.

Expanding each state at most once can lead to a full exploration of the state space and, therefore, is an important ingredient for GBFS in being a complete search algorithm.

In this thesis, we only consider state space topologies that contain solvable state spaces because the behavior of GBFS is trivial in unsolvable state spaces: GBFS necessarily expands all reachable states.

To simplify our further analysis, we always assume Proposition 9.1 as given and do not explicitly restate it. For example, we will find a criterion based on a search history for states that GBFS necessarily expands in the search future. We avoid to restate that

GBFS will only necessarily expand those states that have not been expanded in the search history.

10. Pruning

As the high-water mark reflects some aspects about the behavior of GBFS in a state space topology, it can be used to reason about which states GBFS never expands. We start with presenting a criterion from literature that characterizes never expanded states based on the high-water mark of the initial state. Afterwards, we extend the criterion to additionally consider the high-water marks of all states that GBFS has generated so far in a search run. Finally, we identify states that GBFS potentially expands locally.

10.1. Initial State

We observe that GBFS never expands some states under any possible tie-breaking policy. For example, GBFS generates state A in the first iteration of the example search run from Table 9.1 but never expands it during the rest of the run. One can check that GBFS neither expands state A in any other possible search run in the state space topology from Figure 9.2. Wilt and Ruml (2014) formalized this observation based on high-water mark. Similar formalizations can be found in the works of Bagchi and Mahanti (1983) and Dechter and Pearl (1985) for other algorithms from the family of heuristic best-first searches.

Theorem 10.1 (Wilt and Ruml, 2014)**.** *Let $\mathcal{T}$ be a state space topology with states S, initial state s_{init} and heuristic h. Then GBFS on $\mathcal{T}$ never expands $s \in S$ if $h(s) > hwm(s_{init})$.*

This theorem holds because the high-water mark accurately models the greediness of GBFS. The high-water mark $hwm(s)$ of a state s is the maximum h-value of a state s' on an s-path such that there is no other s-path on which a state has a smaller maximum h-value than that of s'. Since GBFS preferably expands states with low h-values, it finds an s-path by only expanding states with h-values of at most $hwm(s)$. Moreover, as GBFS stops as soon as it selects a goal state, it will never expand a state above $hwm(s)$.

We observe that GBFS never expands some states even though their h-values are below the hwm-value of the initial state. This is because all paths from s_{init} to the states pass through other states that are known to be never expanded due to having h-values above the hwm-value of the initial state. For example, GBFS never expands state H despite of $h(\mathrm{H})$ being smaller than $hwm(\mathrm{B})$ of initial state B because the only path $\langle \mathrm{B}, \mathrm{A}, \mathrm{H} \rangle$ from initial state B to H passes through state A that is known to be never expanded due to $h(\mathrm{A}) > hwm(\mathrm{B})$.

We formalize our observation by introducing the *apex* of a state s, which is the maximum h-value of a state s' on a path from the initial state to s, while there is no other path from the initial state to s on which the maximum h-value of a state is smaller than that of s'.

Definition 10.1 (apex of state)**.** *Let $\mathcal{T}$ be a state space topology $\langle \mathcal{S}, h \rangle$ with states S and initial state s_{init}. Let s be a state from S and let $P(s_{init}, s)$ be the set of all acyclic paths from s_{init} to s in $\mathcal{S}$.*

The apex *of s in $\mathcal{T}$ is defined as*

$$apex(s) = minmax_h(P(s_{init}, s)).$$

We now prove a similar result to Theorem 10.1 based on the apex of a state.

Theorem 10.2. *Let $\mathcal{T}$ be a state space topology with states S and initial state s_{init}. Then GBFS on $\mathcal{T}$ never expands $s \in S$ if $apex(s) > hwm(s_{init})$.*

Proof. Every path from s_{init} to s contains a state s' with $h(s') \geq apex(s)$ by Definition 10.1, which implies $h(s') \geq apex(s) > hwm(s_{\text{init}})$. With Theorem 10.1, we know that GBFS never expands s' with $h(s') > hwm(s_{\text{init}})$. Since it holds for every path from s_{init} to s, GBFS never expands s. □

Note that GBFS is not able to find the shortest solution path $\langle \mathrm{B}, \mathrm{A}, \mathrm{H}, \mathrm{Y} \rangle$ in our example state space topology due to its $apex$-value. The apex is a generalization of the second discriminant that Bagchi and Mahanti (1983) have introduced in order to determine whether A* with inadmissible heuristic is able to find a cost-optimal solution path (see discussion in Section 7.5).

Our new theorem is able to recognize many more never expanded states than Theorem 10.1. For example, our new theorem excludes state $\mathrm{A}, \mathrm{H}, \mathrm{Y}$, while Theorem 10.1 only excludes A in the example state space topology from Figure 9.2.

Theorem 10.3. *Let $\mathcal{T}$ be a state space topology with states S and initial state s_{init}. Let $\bar{S}_{hwm} = \{s \in S \mid h(s) > hwm(s_{init})\}$ and $\bar{S}_{apex} = \{s \in S \mid apex(s) > hwm(s_{init})\}$. Then $\bar{S}_{hwm} \subseteq \bar{S}_{apex}$.*

Proof. We prove the statement by showing that $apex(s) \geq h(s)$ for all states $s \in S$. If s is unreachable (i.e., $P(s_{\text{init}}, s) = \emptyset$), it holds trivially because $apex(s) = \infty$. Otherwise, it holds because every path ρ from s_{init} to s must include s, and hence the maximum h-value of a state s' from ρ is larger or equal $h(s)$, which results in $apex(s) \geq h(s)$. Therefore, Theorem 10.2 is a proper generalization of Theorem 10.1 and $\bar{S}_{\text{hwm}} \subseteq \bar{S}_{\text{apex}}$ holds. □

So far we only determined criteria for states that GBFS never expands based on the high-water mark of the initial state. In the next section, we generalize the criteria to consider any state that GBFS has generated during a search run so far.

10.2. Generated States

We observe that there are states that GBFS never expands in any possible search run and that are not recognized by the criteria from Theorems 10.1 and 10.2. For example, GBFS generates state I in the fourth iteration in the example search run from Table 9.1 on the state space topology from Figure 9.2 but never expands it during the rest of the search run. When GBFS expands state C in the fourth iteration, it generates state N alongside with I. The high-water mark of N is smaller than the h-value of I. Therefore, GBFS finds a path from C to a goal state without ever expanding I. We learn from this observation that the high-water mark of a generated state determines the states that will be pruned for the rest of a search run. We formalize this observation in a theorem that generalizes Theorem 10.1 by considering an arbitrary generated state instead of the initial state.

Theorem 10.4. *Let $\mathcal{T}$ be a state space topology with states S and heuristic h. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Let s be a state from $Generated(\eta[i])$. Then GBFS never expands state $s' \in S$ in the search future of η if $h(s') > hwm(s)$.*

Proof. This theorem differs from Theorem 10.1 in that a generated state s is given instead of the (generated) initial state s_{init} and that the condition holds for the search future $\eta[i]$ instead of a complete run η. Therefore, the proof arguments of Theorem 10.1 apply to this theorem. □

We observe that GBFS generates states during its run and that the smallest hwm-value among these states determines which states GBFS ignores during the rest of the search run. For example, GBFS generates states $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{G}, \mathrm{I}, \mathrm{N}, \mathrm{O}$ and V before the fifth iteration of the example search run. The hwm-value of these states is determined by state N with $hwm(\mathrm{N}) = 2$ and GBFS never expands a state with h-value above s during the rest of its run. We formalize this observation in a theorem that generalizes Theorem 10.4 by considering the high-water mark of all states that GBFS has generated in a search history instead of a single state.

Theorem 10.5. *Let $\mathcal{T}$ be a state space topology with states S and heuristic h. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Then GBFS never expands $s \in S$ in the search future if $h(s) > hwm(Generated(\eta[i]))$.*

Proof. This theorem holds because the high-water mark of a set of states is defined to minimize the high-water mark of each state in the set (Definition 9.5). The application of Theorem 10.4 to a state that determines $hwm(Generated(\eta[i]))$, by minimizing $hwm(s)$ among states s from $Generated[i]$, proves this theorem. □

We have seen that the lowest high-water mark among generated states establishes a bound such that GBFS ignores states with higher h-value than the bound. In the next section, we will show which of the states with h-value below the bound are possibly expanded by GBFS.

10.3. High-Water Mark Level

We observe that GBFS is able to expand generated states because their h-values are smaller or equal to the hwm-value of the set of generated states. For example, GBFS generates states $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{G}$ and V before the third iteration of our example search run from Table 9.1 on the state space topology from Figure 9.2. The smallest hwm-value among these states is 4. GBFS potentially expands states $\mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{G}$ and V because it has generated them and their h-values are equal or smaller 4.

Moreover, we can see that GBFS expands at least one state whose h-value equals the hwm-value of the set of generated states. This is probably a consequence of the definition of high-water mark. For example, GBFS expands at least one of $\mathrm{C}, \mathrm{D}, \mathrm{E}$ or G in order to reach a goal state because these states all contribute to the hwm-value.

We can also see that GBFS expands states with h-value below the hwm-value of all generated states. There are two simultaneous reasons for their expansion. One reason is that these states are expanded because GBFS prefers to expand states with low h-values. The other reason is that GBFS have to expand these generated states before being able to expand another state with h-value equal to the hwm-value of generated states in order to reach a goal state. For example, GBFS expands state V as soon as V is generated because the h-value of V is 1, which is smaller than the hwm-value of the set of generated states, which is 4. Its expansion is necessary in order to expand C in the given search run which leads to a goal state.

We formalize all the observations and show that they hold in general. We start with the second observation.

Theorem 10.6. *Let $\mathcal{T}$ be a state space topology with S and heuristic h. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Then GBFS expands at least one $s \in S$ in the search future of η with $h(s) = hwm(Generated(\eta[i]))$.*

Proof. With the definition of high-water mark (Definition 9.4 and 9.5), we know that all s-paths from generated states $s \in Generated(\eta[i])$ contain at least one state $s' \in S$ with $h(s') \geq hwm(Generated(\eta[i]))$. With Theorem 10.5, we know that GBFS never expands a state $s \in S$ with $h(s) > hwm(Generated(\eta[i]))$ in the search future. Consequently, GBFS must expand at least one state $s \in S$ with $h(s) = hwm(Generated(\eta[i]))$ in order to reach a goal state. □

Note, that this theorem implies that we can determine the hwm-value of an arbitrary state s or of a set of states S by starting GBFS with s or S and remembering the highest h-value that GBFS expands during its run.

We can use the theorem to prove a statement about states that GBFS necessarily expands based on a search history.

Theorem 10.7. *Let $\mathcal{T}$ be a state space topology with states S. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Then GBFS necessarily expands state $s \in S$ in the search future of η if $s \in Generated(\eta[i])$ and $h(s) < hwm(Generated(\eta[i]))$.*

Proof. With Theorem 10.6, we know that GBFS must expand at least one $s \in S$ in the search future with $h(s) = hwm(Generated(\eta[i]))$. GBFS first expands all states $s \in Generated(\eta[i])$ with $h(s) < hwm(Generated(\eta[i]))$ before being able to expand any state $s' \in S$ with $h(s') = hwm(Generated(\eta[i]))$ due to preferring states with low h-values. Therefore, GBFS necessarily expands states $s \in Generated(\eta[i])$ with $h(s) < hwm(Generated(\eta[i]))$. □

Finally, we formalize and prove our first observation from the beginning of this section about potentially expanded states.

Theorem 10.8. *Let $\mathcal{T}$ be a state space topology with states S and heuristic h. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Then GBFS potentially expands $s \in S$ in the search future of η if $s \in Generated(\eta[i])$ and $h(s) \leq hwm(Generated(\eta[i]))$.*

Proof. With Theorem 10.6, we know that GBFS must expand at least one $s \in S$ in the search future with $h(s) = hwm(Generated(\eta[i]))$. As states s from $Generated(\eta[i])$ are generated and GBFS needs to expand at least one state $s \in S$ where $h(s)$ equals $hwm(Generated(\eta[i]))$, GBFS potentially expands all states $s \in Generated(\eta[i])$ where $h(s)$ equals $hwm(Generated(\eta[i]))$. With Theorem 10.7, we know that GBFS necessarily expands states s from $Generated(\eta[i])$ where $h(s) = hwm(Generated(\eta[i]))$. As necessarily expanded states are also potentially expanded, we can conclude that GBFS potentially expands $s \in S$ in the search future if $s \in Generated(\eta[i])$ and $h(s) \leq hwm(Generated(\eta[i]))$. □

This theorem only says which of the currently generated states GBFS potentially expands, but makes no statement about other states that have not been generated yet. Nevertheless, it is an important result on our way towards the characterization of all the states that GBFS potentially expands under any tie-breaking policy.

In the last section we have already discussed that the lowest high-water mark value among states constitutes a bound for GBFS. Since GBFS potentially expands generated states with h-value equal or smaller that bound, we say that GBFS is searching on an *high-water mark level*.

Definition 10.2 (high-water mark level)**.** *Let $\mathcal{T}$ be a state space topology. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$.*

The high-water mark level *of $\eta[i]$ is defined as $level(\eta[i]) = hwm(Generated(\eta[i]))$.*

Example 10.1. *We consider the search history $\langle \mathrm{B}, \mathrm{E}, \mathrm{V}, \mathrm{C} \rangle$ of search run from Table 9.1 on the state space topology from Figure 9.2. The set $Generated(\langle \mathrm{B}, \mathrm{E}, \mathrm{V}, \mathrm{C} \rangle)$ is $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{G}, \mathrm{I}, \mathrm{N}, \mathrm{O}, \mathrm{V}\}$. The high-water mark of $Generated(\langle \mathrm{B}, \mathrm{E}, \mathrm{V}, \mathrm{C} \rangle)$ is 2. Therefore, $level(\langle \mathrm{B}, \mathrm{E}, \mathrm{V}, \mathrm{C} \rangle) = 2$.*

In the next chapter, we will learn more about the search behavior of GBFS on an hwm-level and among hwm-levels.

11. Search Progress and Benches

Many satisficing search algorithms based on GBFS consider it as a meaningful event when an algorithm generates a state with smaller heuristic value than that of any previously generated state. The boosted dual-queue search algorithm by Richter and Helmert (2009) adds additional priority to states generated by preferred operators, each time it expands a state with new all time low h-value. GBFS with local exploration introduced by Xie, Müller, and Holte (2014) starts local exploration after GBFS fails to find a new all time low heuristic value after a given number of expansions and stops local exploration when a state with a new all time low h-value is generated.

We define progress based on heuristic values, called *heuristic progress*, as

$$p^{\mathrm{h}}(\eta[i]) = - \min_{s \in Generated(\eta[i])} h(s).$$

We add the negation to the given term because progress is defined as a monotonically increasing function. This understanding of progress has some flaws. For example, GBFS expands state E in the second iteration during our example search run from Table 9.1 on the state space topology from Figure 9.2. The all time low h-value before the expansion of E is 4. GBFS generates V at the expansion of E drastically lowering the all time low h-value from 4 to 1. While GBFS seems to make considerable progress towards finding a goal, it has made this progress by generating dead end state V. Considering this event as search progress does not make sense because dead ends do not lead to goal states.

We now develop another notion of search progress. We observe that in some iterations GBFS generates states with lower hwm-value than that of any generated state before. For example GBFS has generated states $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{G}$ and V before its fourth iteration during the example search run. The lowest hwm-value among these states is 4. In the fourth iteration, GBFS generates state $\mathrm{B}, \mathrm{I}, \mathrm{N}$ and O. Among these states, state N currently has the lowest hwm-value of 2, which is lower than 4. Since we know that GBFS will not expand a state with h-value higher than 2 for the rest of its run, we can interpret this event as search progress.

We formalize our observation as a progress function.

Definition 11.1 (high-water mark progress)**.** *Let $\mathcal{T}$ be a state space topology with states S. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$.*

High-water mark progress *is defined as*

$$p^{hwm}(\eta[i]) = -level(\eta[i]).$$

As hwm-progress is based on high-water mark, which is based on s-paths of generated states s, it can only be used in hindsight or when adopting an omniscient view of the state space topology.

We show that high-water mark progress is a monotonic increasing function.

Theorem 11.1. *Let $\mathcal{T}$ be a state space topology with states S. Then p^{hwm} monotonically increases along a search run η, i.e., $p^{hwm}(\eta[i-1]) \leq p^{hwm}(\eta[i])$ for $i > 0$.*

Proof. We know that $p^{\text{hwm}}(\eta[i]) = -level(\eta[i])$ by definition of high-water mark progress (Definition 11.1). Moreover, we know that $level(\eta[i]) = hwm(Generated(\eta[i]))$ by definition of hwm-level (Definition 10.2)

(i) We know that GBFS expands a state s from $Generated(\eta[i-1])$ in iteration i and that the expansion of s generates states from $succ(s)$.

(ii) Since $succ(s)$ can contain states that are not already in $Generated(\eta[i-1])$, we know that $Generated(\eta[i-1]) \subseteq Generated(\eta[i])$.

(iii) With definition of high-water mark of states (Definition 9.5), we know that hwm minimizes the hwm-values among states from a given set of states.

Items (ii) and (iii) imply $hwm(Generated(\eta[i-1])) \geq hwm(Generated(\eta[i]))$ because each additional state can only decrease the hwm-value. Negating both terms result in $-hwm(Generated(\eta[i-1])) \leq -hwm(Generated(\eta[i]))$, which is $p^{\text{hwm}}(\eta[i-1]) \leq p^{\text{hwm}}(\eta[i])$. □

Search progress of GBFS is reflected more accurately by high-water mark progress than by heuristic progress. Whenever GBFS makes progress by reaching a lower hwm-level, it increases the set of states that it never expands in the search future. We will discuss this statement in more detail during the next sections.

11.1. Progress States and Bench States

High-water mark progress happens at the expansion of a state which can be characterized locally and independently of a search history.

Theorem 11.2. *Let $\mathcal{T}$ be a state space topology with states S. GBFS makes progress iff it expands a state $s \in S$ with $h(s) > hwm(succ(s))$.*

Proof. According to the definition of progress (Definition 5.2), GBFS makes progress iff $p^{\text{hwm}}(\eta[i-1]) < p^{\text{hwm}}(\eta[i])$. This is sufficiently represented by $hwm(Generated(\eta[i-1])) > hwm(Generated(\eta[i]))$ (the proof of Theorem 11.1 shows more details).

(i) The definition of high-water mark of set of states (Definition 9.5) and the condition $hwm(Generated(\eta[i-1])) > hwm(Generated(\eta[i]))$ imply that GBFS expands in iteration i a state s from $Generated(\eta[i-1])$ and generates a state $s' \in succ(s) \setminus Generated(\eta[i-1])$ with $hwm(s') < hwm(Generated(\eta[i-1]))$.

(ii) With $s' \in succ(s)$, the definition of high-water mark of states (Definition 9.5) and (i), we know that $hwm(succ(s)) \leq hwm(s') < hwm(Generated(\eta[i-1]))$.

(iii) With Theorem 10.8, we know that GBFS being able to expand state s implies $h(s) \leq hwm(Generated(\eta[i-1]))$.

(iv) With the definition of high-water mark of states (Definition 9.5), we know that $hwm(s) \geq hwm(Generated(\eta[i-1]))$.

(v) With the functional equation of high-water mark (Definition 9.6), we know that $hwm(s)$ is the maximum value between $h(s)$ and $hwm(succ(s))$. We can rule out case $hwm(s) = hwm(succ(s))$ because it results with (iv) and (ii) in inequation $hwm(succ(s)) = hwm(s) \geq hwm(Generated(\eta[i-1])) > hwm(succ(s))$. Therefore, case $hwm(s) = h(s)$ holds.

(vi) Items (iii), (iv) and (v) imply $h(s) \leq hwm(Generated(\eta[i-1])) \leq hwm(s) = h(s)$, which results in $h(s) = hwm(s) = hwm(Generated(\eta[i-1]))$.

Finally, (ii) and (vi) imply $hwm(succ(s)) < hwm(Generated(\eta[i-1])) = h(s)$, which can be rewritten as $h(s) > hwm(succ(s))$. □

We call a state whose expansion results in high-water mark progress *progress state*. For simplifying our further analysis, we count the initial state and goal states as progress states as well. While we explicitly define goal states as progress states, we implicitly define initial state s_{init} as progress states by requiring $h(s_{\text{init}}) > hwm(succ(s_{\text{init}}))$. We can do this simplification without loss of generality because the initial state is always the first expanded state and goal states are always the last expanded states in a search run.

Definition 11.2 (progress state)**.** *Let $\mathcal{T}$ be a state space topology $\langle\langle S, s_{init}, S_{goal}, succ\rangle, h\rangle$. A state $s \in S$ is a* progress state *iff $h(s) > hwm(succ(s))$ or $s \in S_{goal}$.*

Example 11.1. *State* B *is a progress state because $h(\mathrm{B}) = 5$ is larger than $hwm(succ(\mathrm{B}))$ with value 4. State* C *is an progress state because $h(\mathrm{C}) = 4$ is larger than $hwm(succ(\mathrm{C}))$ with value 2. State* Z *is a progress state because it is a goal state*

Figure 11.1 shows progress states painted in red.

Let $Open$ be a set of open states. The expansion of progress state $s \in Open$ implies that $h(s') \geq h(s)$ for all states $s' \in Open$. Since s satisfies $h(s) > hwm(succ(s))$, there is at least one $s' \in succ(s)$ with s'-path strictly below $h(s)$. Therefore, GBFS will never consider a state that has been in $Open$ before expansion of s for the rest of its run.

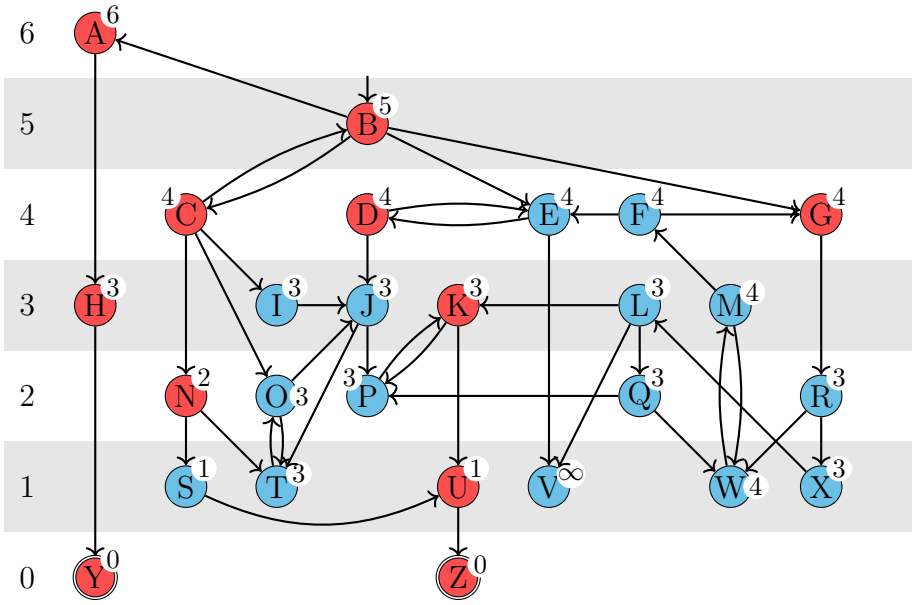


Figure 11.1: Example state space topology where progress states are painted in red, bench states are painted in blue and all states are annotated with their hwm-values.

This leads to the remarkable conclusion that upon expansion of a progress state s GBFS behaves like it starts a search from s, except that it will not expand previously expanded states (Proposition 9.1), which we ignore for the moment.

Since GBFS behaves like it starts a local GBFS search, a search run can be understood as a sequence of episodes. An episode starts with the expansion of a progress and ends right before the expansion of a next progress state. During an episode that starts with progress state s, GBFS searches on an hwm-level that is determined by $hwm(succ(s))$.

Corollary 11.1. *Let $\mathcal{T}$ be a state space topology. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Let s be the last expanded progress state in $\eta[i]$. Then $level(\eta[i]) = hwm(succ(s))$ if $s \notin S_{goal}$ and $level(\eta[i]) = hwm(s)$, otherwise.*

Proof. Case $s \in S_{\text{goal}}$ is trivial because s is the last state in a search run and always has the smallest hwm-value among all generated states due to the definition of high-water mark.

We now consider case $s \notin S_{\text{goal}}$. Assume s is expanded in iteration $j \leq i$ and is the most recently expanded progress state in $\eta[i]$. The proof of Theorem 11.2 shows that GBFS makes progress when it expands s because $hwm(succ(s)) < h(s)$ equals $hwm(Generated(\eta[j-1]))$. This leads to $hwm(Generated(\eta[j-1]) \cup succ(s)) = hwm(Generated(\eta[j]) = hwm(succ(s))$ because of the definition of high-water mark of states (Definition 9.5). With the definition of hwm-level (Definition 10.2), we get $level(\eta[j]) = hwm(succ(s))$. State s being the most recent expanded progress states implies $level(\eta[i]) = level(\eta[j]) = hwm(succ(s))$ because only a next progress state can

further decrease the hwm-level. □

We call states that GBFS expands within episodes and that are not progress states *bench states*. As they are complementary to progress states, they can be characterized locally and independently of a search history as well.

Definition 11.3 (bench state)**.** *Let $\mathcal{T}$ be a state space topology with states S, successor function $succ$ and heuristic h.*

A state $s \in S$ is a bench state *iff $h(s) \leq hwm(succ(s))$ and $s \notin S_{goal}$.*

Example 11.2. *State E is a bench state because $h(\mathrm{E}) = 4$ is equal to $hwm(succ(\mathrm{E})) = 4$. State O is a bench state because $h(\mathrm{O}) = 2$ is smaller than $hwm(succ(\mathrm{O})) = 3$.*

Figure 11.1 shows bench states painted in blue.

We have discovered that GBFS searches in episodes where each episode is a local search that starts with the expansion of a progress state and ends right before the expansion of a next progress states. Therefore, to understand which states a GBFS potentially expands under any tie-breaking policy, we have to understand (A) how GBFS searches from one progress state to a next progress state, and (B) in which order GBFS reaches different progress states. We will answer (A) in the next section and (B) in Section 11.3.

11.2. Bench

In this section we ask which states GBFS potentially expands after it expands a progress state and before it expands a next progress state.

We observe that after GBFS expands a progress state, it searches locally in a region of the state space that is reachable from the progress state but bounded by the value of the current hwm-level and framed by hwm-progress states. For example, suppose GBFS expands progress state C and enters hwm-level 2. On this level GBFS can reach states N, O and T, all having an h-value smaller or equal 2. GBFS will not expand J during the rest of its run due to high-water mark pruning. The only possibility to reach a lower hwm-level is by expanding progress state N. Therefore, after GBFS expands C it potentially expands O and T before exiting the current hwm-level by expanding progress state N. It is easy to see that it forms a state space with initial state C and goal state N.

We formalize this observation by introducing benches, which capture the possible behaviors of GBFS after it expands a progress state and until it expands a next progress state.

Definition 11.4 (bench)**.** *Let $\mathcal{T}$ be a state space topology $\langle\langle S, s_{init}, S_{goal}, succ\rangle, h\rangle$. Let $s \in S$ be a progress state.*

- *The* level *value $level(s)$ of s is $hwm(succ(s))$ if $s \notin S_{goal}$ and $-\infty$, otherwise.*
- *The set of* bench *states $Bench(s)$ of s contains each bench state s'' that is reachable from s on a path on which all states $s' \neq s$ (including s'' itself) are bench states and satisfy $h(s') \leq level(s)$.*
- *The set of* progress *states $Progress(s)$ of s contains each progress state s' with $h(s') = level(s)$ that is a successor of s or a successor of a bench state from $Bench(s)$.*

The bench $\mathcal{B}(s)$ induced by s is a state space topology $\langle\langle S', s'_{init}, S'_{goal}, succ'\rangle, h\rangle$, with

- $S' = \{s\} \cup Bench(s) \cup Progress(s)$,
- $s'_{init} = s$,
- $S'_{goal} = Progress(s)$,
- $succ'$ *is* $succ$ *restricted to transitions between states from S', and*
- h' *is* h *restricted to states from S'.*

The bench level $level(s)$ of progress state s is the hwm-level on which GBFS searches right after expansion of s. In order to encode the fact that GBFS stops when it expands a goal state $s \in S_{\text{goal}}$, we define $level(s) = -\infty$. It results in an empty set of bench states and an empty set of progress states. We call a bench induced by an initial state *initial bench* and a bench induced by a goal state *goal bench*. A non-goal bench $\mathcal{B}(s)$ always consists of a solvable states space, i.e., $Progress(s)$ is not empty. We call a bench $\mathcal{B}(s)$ with $Bench(s) = \emptyset$ *empty* bench. Let $\mathcal{B}(s)$ be a bench and s' be a progress state from $Progress(s)$. Then we generally assume that GBFS only *selects* s' on $\mathcal{B}(s)$ and expands s' on $\mathcal{B}(s')$.

Example 11.3. *The bench $\mathcal{B}(\mathrm{B})$ has initial state* B*, $Bench(\mathrm{B}) = \{\mathrm{E}, \mathrm{V}\}$, $Progress(\mathrm{B}) = \{\mathrm{C}, \mathrm{D}, \mathrm{G}\}$ and is the initial bench. The bench $\mathcal{B}(\mathrm{C})$ has initial state* C*, $Bench(\mathrm{C}) = \{\mathrm{O}, \mathrm{T}\}$ and $Progress(\mathrm{C}) = \{\mathrm{N}\}$. The bench $\mathcal{B}(\mathrm{K})$ has initial state* K*, $Bench(\mathrm{K}) = \emptyset$, $Progress(\mathrm{K}) = \{\mathrm{U}\}$ and is empty. The bench $\mathcal{B}(\mathrm{Z})$ has initial state* Z*, $Bench(\mathrm{Z}) = Progress(\mathrm{Z}) = \emptyset$ and is a goal bench.*

We formalize our observed behavior from the beginning of this section about which states GBFS potentially expands during an episode. An episode is a GBFS search on a bench that starts with the initial state of the bench and potentially expands all the bench states from the bench before it expands one of the goal states of the bench.

Theorem 11.3. *Let $\mathcal{T}$ be a state space topology with states S. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Let s be the last expanded state in $\eta[i]$ (in iteration i) and let s be a progress state. Then GBFS potentially expands $s' \in S$ in the search future of η if $s' \in Bench(s) \cup Progress(s)$.*

Proof. We use the definition of $Bench(s)$ and $Progress(s)$ (Definition 11.4) and Theorems 10.8 and 11.2 to prove this theorem.

(i) With the definition of $Bench(s)$ (Definition 11.4), we know that $Bench(s)$ contains each state s'' that is reachable from s on a path on which all states $s' \neq s$ are bench states and satisfy $h(s') \leq level(s)$.

(ii) With the definition of $Progress(s)$ (Definition 11.4), we know that $Progress(s)$ contains each progress state s' with $h(s') \leq level(s)$ that is a successor of s or a successor of a state from $Bench(s)$.

(iii) We know that $level(s) = level(\eta[i])$ represents the value of the current hwm-level on which GBFS searches. With Theorem 11.2, we also know that this value only changes with the expansion of a progress state from $Progress(s)$.

(iv) With Theorem 10.8, we know that GBFS potentially expands a generated state s' with $h(s')$ smaller or equal the value $level(\eta[i])$ of the current hwm-level.

When GBFS expands s, it generates the successor states of s. As s is a progress state, at least one successor state will be in $Bench(s)$ or $Progress(s)$. Due to (i) and (ii) we know that GBFS is able to generate all states from $Bench(s)$ and $Progress(s)$, which implies with (iii) and (iv) that GBFS potentially expands all states from $Bench(s)$ and $Progress(s)$. □

We characterized benches, which are subspaces of the whole state space, in which GBFS searches after the expansion of a progress state until the selection of a next progress state. The states in a bench are those states that GBFS potentially expands. Moreover, while searching in a bench GBFS only expands states from the bench. In the next section we will analyze the search behavior of GBFS among benches.

11.3. Bench Space

In this section we ask which states GBFS possibly expands among every possible search run.

We observe that when GBFS searches in a bench during an episode, it can leave the bench via one of many possible progress states. The expansion of a progress state leads to a next episode on another bench, where GBFS faces the same situation. Each time a GBFS search moves on to a next bench, it makes progress towards finding a goal because

subsequent benches have strictly decreasing levels. As there is more than one progress state on a bench, different search runs result in different sequences of visited benches. For example, in the search run from Table 9.1 on the state space topology from Figure 11.1 GBFS starts with searching on bench $\mathcal{B}(\mathrm{B})$. It leaves $\mathcal{B}(\mathrm{B})$ on level 4 by expanding progress state $\mathrm{C} \in \mathit{Progress}(\mathrm{B})$ and reaches bench $\mathcal{B}(\mathrm{C})$ on level 2. Afterwards, GBFS leaves $\mathcal{B}(\mathrm{C})$ via progress state N and leaves $\mathcal{B}(\mathrm{N})$ via U. It eventually selects goal state Z on bench $\mathcal{B}(\mathrm{U})$. Instead of leaving the first bench through C, GBFS could also have left it via progress states D or G.

We formalize our observation by defining a bench space that connects the benches.

Definition 11.5 (bench space)**.** *Let $\mathcal{T}$ be a state space topology with initial state s_{init} and goal states S_{goal}.*

The bench space $\mathbb{B}(\mathcal{T})$ *of $\mathcal{T}$ is a state space $\langle B, \mathcal{B}_{init}, B_{goal}, succ \rangle$, where*

- *B is a set of* benches*, which is inductively defined as follows: $\mathcal{B}(s') \in B$ iff*
  - *$s' = s_{init}$ or*
  - *$s' \in \mathit{Progress}(s)$ with $\mathcal{B}(s) \in B$,*
- *$\mathcal{B}_{init}$ is the* initial bench *$\mathcal{B}(s_{init})$,*
- *B_{goal} is the set of* goal benches *with $\mathcal{B}(s) \in B_{goal}$ if $s \in S_{goal}$ and $\mathcal{B}(s) \in B$, and*
- *$succ$ is the* successor function *with $\mathcal{B}(s') \in succ(\mathcal{B}(s))$ if $s' \in \mathit{Progress}(s)$ for $\mathcal{B}(s), \mathcal{B}(s') \in B$.*

Example 11.4. *Figure 11.2 shows the bench space $\mathbb{B}(\mathcal{T})$ induced by our example state space topology $\mathcal{T}$ from Figure 11.1. The rectangles are benches, the arrows between benches define the transitions between benches and the numbered layers indicate bench levels. The bench with the incoming arrow is the initial bench and the bench with the double lined rectangle is the goal bench.*

A bench space shows all the subspaces in which GBFS potentially searches. During a single search run GBFS visits the benches along a single path in the bench space. It provides us with a high-level view on the behavior of GBFS that entails the information about which bench can be reached from which bench but otherwise ignores the behavior within benches. Therefore, it is an abstract view on the behavior of GBFS that ignores details within benches, but that allows us to pick a bench and analyze the behavior of GBFS in more detail within the bench.

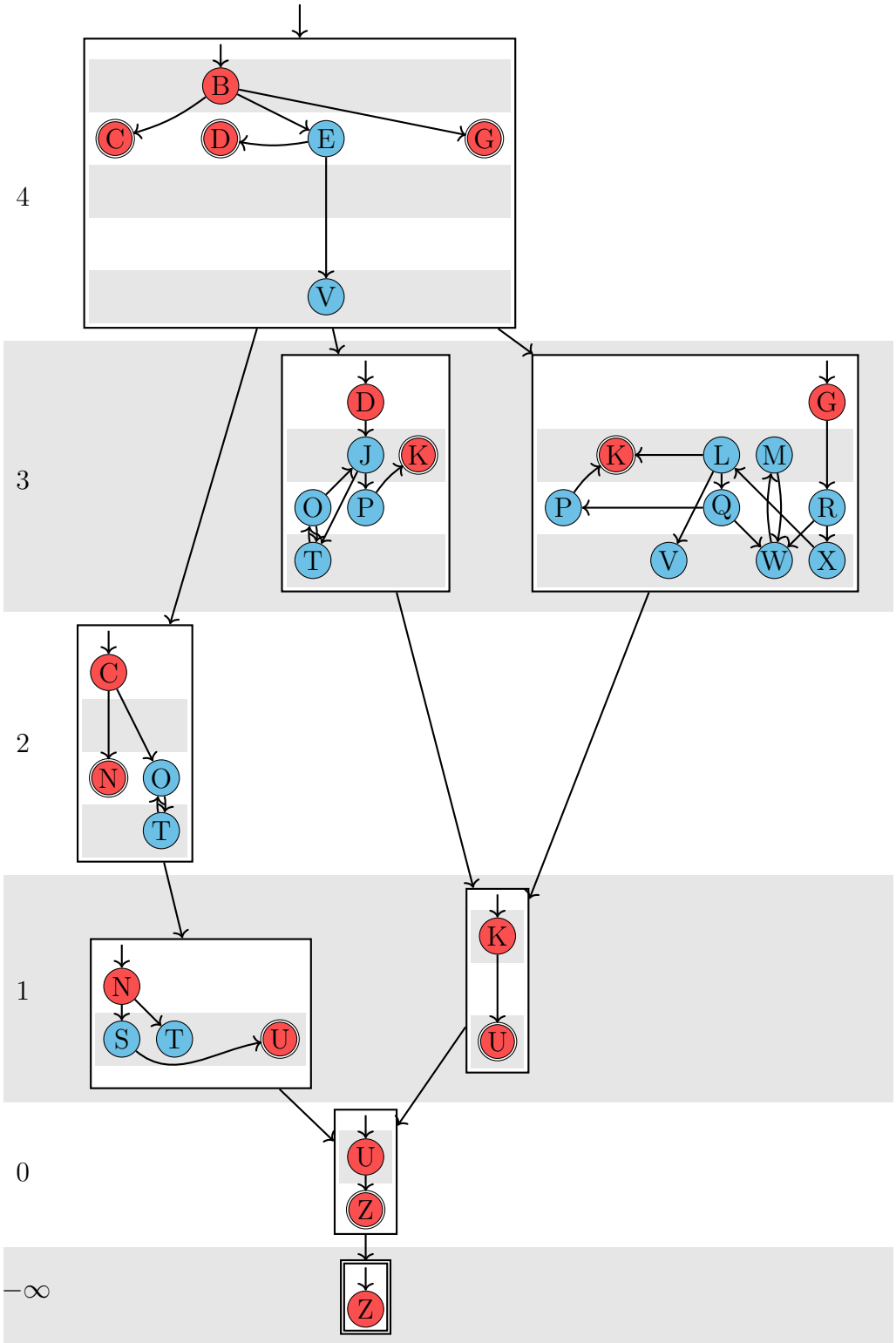


Figure 11.2: Bench space of our example state space topology from Figure 11.1.

Expanded States The bench space contains all the benches that GBFS visits under any tie-breaking policy, and a bench contains all the states that GBFS potentially expands within an episode. By combining this information we can precisely characterize which states GBFS potentially expands under any possible tie-breaking policy.

Theorem 11.4. *Let $\mathcal{T}$ be a state space topology with states S. Let $\mathbb{B}(\mathcal{T})$ be the bench space of $\mathcal{T}$ with set of benches B. Let $S' = \bigcup_{\mathcal{B}(s) \in B} \big(\{s\} \cup Bench(s) \cup Progress(s)\big)$.*

- *GBFS potentially expands $s' \in S$ if $s' \in S'$.*
- *GBFS never expands $s' \in S$ if $s' \notin S'$.*

Proof. With Theorem 11.3, we know that when GBFS searches on a bench $\mathcal{B}(s)$, it potentially expands all the states on the bench, i.e., state s and states from $Bench(s)$ and $Progress(s)$. GBFS potentially expanding each progress state s' from $Progress(s)$ implies that GBFS potentially expands each state from each subsequent bench $\mathcal{B}(s')$. We know that the bench space $\mathbb{B}(\mathcal{T})$ connects each bench with their subsequent benches, starting with the bench induced by the initial state of $\mathcal{T}$ and ending with a bench induced by a goal state of $\mathcal{T}$. Therefore, $\mathbb{B}(\mathcal{T})$ contains all benches that GBFS potentially reaches in at least one possible search run, respectively under at least one tie-breaking policy.

The set of states that GBFS never expands is complementary to the set of states that GBFS potentially expands. □

In other words, there exists a GBFS search run that expands a state s iff a bench from the bench space of a given state space topology contains s. Conversely, there exists no search run that expands a state s iff no bench from the bench state contains s. For example, GBFS never expands states $\mathrm{A}, \mathrm{H}, \mathrm{Y}, \mathrm{I}$ and F in any search run because they are absent from all the benches from the bench space from Figure 11.2. However, GBFS expands all other states in at least one possible search run.

The set of states that GBFS potentially expands can be computed in time and space polynomial in the number n of states of a given state space topology. We have already mentioned that the high-water mark is easy to compute. Given the high-water marks of states, we can construct the bench space according to the given definitions. The bench space consists of at most n benches, each consisting of at most n states. The union of the states of all benches results in the set of states that GBFS potentially expands.

Directed Acyclic Graph We observe that the bench space from Figure 11.2 is a directed acyclic graph, which turns out to be valid for all bench spaces.

Theorem 11.5. *Let $\mathcal{T}$ be a state space topology. The bench space $\mathbb{B}(\mathcal{T})$ of $\mathcal{T}$ is a directed acyclic graph.*

Proof. We show that the bench levels along all bench paths in $\mathbb{B}(\mathcal{T})$ are strictly decreasing, which implies that $\mathbb{B}(\mathcal{T})$ is a directed acyclic graph. It suffices to show that the levels of benches decrease in all transitions in $\mathbb{B}(\mathcal{T})$.

(i) $\mathcal{B}(s') \in succ(\mathcal{B}(s))$ implies that $s' \in Progress(s)$ by definition of bench space (Definition 11.5), which in return implies that $h(s') = level(s)$ and that s' is a progress state by definition of bench (Definition 11.4).

(ii) As s' is a progress state, we know that $h(s') > hwm(succ(s'))$ by definition of progress state (Definition 11.2).

(iii) Level $level(s')$ is $hwm(succ(s'))$ if s' is not a goal state from $\mathcal{T}$ and $-\infty$, otherwise, by definition of bench level (Definition 11.4).

- If $level(s') = hwm(succ(s'))$, we know with steps (i) and (ii) that $level(s) = h(s') > hwm(succ(s')) = level(s)$.
- If $level(s') = -\infty$, we know with step (i) that $level(s) = h(s') > -\infty = level(s')$.

Step (iii) shows that the bench level decreases in each transition from one bench to succeeding bench in $\mathbb{B}(\mathcal{T})$. □

Overlapping Benches Benches can overlap, i.e., the same state can appear on multiple benches. This is because a bench represents the subspace in which GBFS searches locally during an episode. Benches can overlap in two different ways: within an hwm-level or between hwm-levels.

Overlapping benches within an hwm-level do not interfere each other because GBFS enters an hwm-level at most once during a search run. The overlap stems from the fact that different search runs can enter the same hwm-level through different progress states, and that the progress state from which an hwm-level is entered induces the subspace on an hwm-level. For example, benches $\mathcal{B}(\mathrm{D})$ and $\mathcal{B}(\mathrm{G})$ overlap in state P and K in our example bench space from Figure 11.2. A search run either enters $\mathcal{B}(\mathrm{D})$ or $\mathcal{B}(\mathrm{G})$ but never both.

Overlapping benches between different hwm-levels can interfere each other in a search run, which needs to be considered when analyzing explicit search runs. When tracing a search run, one have to consider that GBFS expands each state at most once (Proposition 9.1). Consequently, benches are not independent from each other when considering actual search runs. When two benches on different levels share a state s, then they overlap at least in the subspace that is reachable from s on the lower bench level. For example, benches $\mathcal{B}(\mathrm{C}), \mathcal{B}(\mathrm{D})$ and $\mathcal{B}(\mathrm{N})$ overlap in state T. As bench $\mathcal{B}(\mathrm{N})$ is a successor of $\mathcal{B}(\mathrm{C})$, we have to consider that state T can either be expanded on $\mathcal{B}(\mathrm{C})$ or on $\mathcal{B}(\mathrm{N})$ but not on both during a search run. We will discuss the reason for this kind of overlapping benches in more detail in Section 12.3.

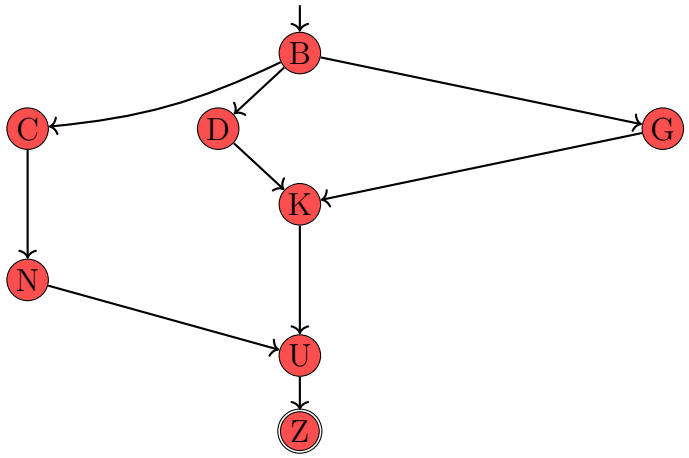


Figure 11.3: Progress state space of our example state space topology from Figure 9.1.

11.4. Progress State Space

Bench spaces show the connections between the local subspaces in which GBFS searches. Moreover, the transitions between benches show where GBFS makes progress. Consequently, it shows which progress state can be reached from which other progress state. To make this view more explicit we introduce a *progress state space* that is a bench space where all benches are replaced with their defining progress states.

Definition 11.6 (progress state space)**.** *Let $\mathcal{T}$ be a state space topology. Let $\mathbb{B}(\mathcal{T})$ be the bench space of $\mathcal{T}$.*

A progress state space *$\mathcal{P}(\mathcal{T})$ of $\mathcal{T}$ is $\mathbb{B}(\mathcal{T})$ where each bench $\mathcal{B}(s)$ is replaced with s.*

Example 11.5. *Figure 11.3 shows the progress state space $\mathcal{P}(\mathcal{T})$ induced by our example state space topology $\mathcal{T}$ from Figure 11.1.*

In a progress state space, we can track a search run by only considering the progress states of the search run. A single progress state solution path from the progress state space represents a set of many possible search runs. Such a path can be considered as an abstraction of search runs. If desired, one can still analyze the local behavior of GBFS on a bench induced by progress state s by accessing all the information through functions $\mathcal{B}(s)$, $level(s)$ and $Bench(s)$ based on the underlying state space topology. Note that $succ(s)$ of the progress state space equals $Progress(s)$.

12. Craters

Up to here, we have learned that GBFS searches in episodes where each episode is a GBFS search on another bench. Moreover, we have discussed how benches are connected to each other. From here on, we are also interested in the inner structure of benches.

A phenomenon of GBFS that has drawn plenty of interest in the heuristic search and planning community (e.g. Hoffmann, 2005; Nakhost and Müller, 2009; Lipovetzky and Geffner, 2011; Xie, Müller, and Holte, 2014) is what is often called an uninformative heuristic region, local minimum or heuristic plateau. Even though all of these differ slightly in definition, they have in common that they are causing serious problems for GBFS as they represent potentially large parts of the search space where the heuristic does not offer guidance or, in the worst case, even misguides the search. While benches already refine what is typically understood as a heuristic plateau, our theory also allows us to provide a more accurate definition of local minima in the form of craters.

12.1. Crater and Surface States

In this section, we characterize two types of states. One type is part of what is commonly understood as local minimum. States of this type are necessarily expanded in some situations. The other type characterizes states that are critical for tie-breaking and affect the main course of a search run.

We observe that sometimes when GBFS searches on a bench $\mathcal{B}(s)$ it is forced to expand a state with h-value smaller than $level(s)$. This is because GBFS prefers states with low heuristic values. However, in order to leave a bench and to make progress towards a goal state, GBFS has to expand a next progress state whose h-value equals $level(s)$. For example, GBFS generates state R with $h(\mathrm{R}) = 2$ on bench $\mathcal{B}(\mathrm{G})$ at the expansion of state G. Since GBFS can only leave $\mathcal{B}(\mathrm{G})$ via progress state K with $h(\mathrm{K}) = level(\mathrm{G}) = 3$ and since GBFS prefers states with low h-values, it is forced to expand R before being able to generate and expand the next progress state K. We already have formalized this observation in Theorem 10.7, since it was elementary for showing which states GBFS potentially expands.

We call a state from a bench with h-value smaller than the level value of the bench *crater state*.

Definition 12.1 (crater state)**.** *Let $\mathcal{B}(s)$ be a bench.*

A state s' is a crater *state on $\mathcal{B}(s)$ iff $s' \in Bench(s)$ and $h(s') < level(s)$.*

Example 12.1. *State* R *is a crater state on* $\mathcal{B}(\mathrm{G})$ *because* $h(\mathrm{R}) = 2$ *is smaller than* $level(\mathrm{G}) = 3$.

Note that only bench states can be crater states because a progress state s' has $h(s') = level(s)$. Now we can say, when GBFS searches on a bench and generates a crater state s, then it necessarily expands s.

Theorem 10.8 implies that there are states on a bench that GBFS is not forced to expand after their generation. The h-values of those states on a bench $\mathcal{B}(s)$ are equal to $level(s)$. Moreover, these states seem to be critical for the course of a GBFS. When GBFS expands such a state, then it may generate a state on a next bench resulting in search progress. In another case, GBFS may generate a crater state and be forced to expand it and possibly many more subsequent crater states. The expansion of such a state may also lead to the generation of further tie-breaking critical states. For example, GBFS generates states C, E and G on bench $\mathcal{B}(\mathrm{B})$ after expansion of B. The h-values of these states equal $level(\mathrm{B})$. GBFS has to break ties among them. The expansion of C or G would lead to progress. The expansion of E generates crater state V but also generates D. While V is necessarily expanded in this situation, state D becomes an additional candidate among C and G in the next tie-breaking situation.

We call a state from a bench with h-value equal to the $level$-value of the bench *surface state*.

Definition 12.2 (surface state)**.** *Let* $\mathcal{B}(s)$ *be a bench.*

A state s' *is a* surface *state on* $\mathcal{B}(s)$ *iff either* $s' = s$ *or* $s' \in Bench(s) \cup Progress(s)$ *with* $h(s') = level(s)$.

Example 12.2. *State* $\mathrm{G}, \mathrm{K}, \mathrm{L}$ *and* M *are surface states on* $\mathcal{B}(\mathrm{G})$. *State* G *is one because it is the initial state of the bench. The other states are surface states because their* h*-values equal* $level(\mathrm{G}) = 3$.

Our last observation made clear that the expansion of a surface state influences the course of a GBFS search run. We also observed that one kind of surface states triggers the expansion of crater states. We call such a surface state *trap* state.

Definition 12.3 (trap state)**.** *Let* $\mathcal{B}(s)$ *be a bench.*

A state s' *is a* trap *state on* $\mathcal{B}(s)$ *iff* s' *is a surface state on* $\mathcal{B}(s)$ *and at least one successor state of* s' *is a crater state on* $\mathcal{B}(s)$.

Note that states from $Progress(s)$ are not defined as trap states on $\mathcal{B}(s)$ because we assume that these states are not expanded on $\mathcal{B}(s)$. A state s' from $Progress(s)$ is then a trap state on bench $\mathcal{B}(s')$ if one of its successors is a crater state on $\mathcal{B}(s')$.

Example 12.3. *State* G, L *and* M *are trap states on* $\mathcal{B}(\mathrm{G})$ *because they are surface state on* $\mathcal{B}(\mathrm{G})$ *and the successor state* R *of* G *is a crater state, the successor state* Q *of* L *is a crater state and the successor state* W *of* M *is a crater state.*

We have discovered that benches contain different states that have individual effects when they are generated or expanded by GBFS. To better understand these effects we have to understand (A) how GBFS searches among crater states, and (B) how it searches among surface states. We will answer (A) in the next section and (B) in Section 12.3 and 12.4.

12.2. Crater

In this section, we formalize the subspace that GBFS necessarily expands after it expands a surface state. The boundary of that subspace consists of surface states which are critical for tie-breaking and the remaining course of a search run.

In the last section, we already observed the role of crater states, surface states and trap states. We further observe that when GBFS expands a surface state, then it can trigger a cascade of crater state expansions. The expansions of the surface states and crater states may generate new surface states. For example when GBFS expands trap state G on bench $\mathcal{B}(\mathrm{G})$, it generates crater state R. As R is necessarily expanded, GBFS expands R and generates successor states W and X. As these states are crater states as well, GBFS expands both under any tie-breaking decision and generates states L and M. These states are surface states that are critical for tie-breaking and, therefore, important for the remaining course of the search run. We formalize this observation by introducing craters, which capture the behavior of GBFS after it expands a surface state.

Definition 12.4 (crater)**.** *Let $\mathcal{T}$ be a state space topology $\langle\langle S, s_{init}, S_{goal}, succ\rangle, h\rangle$. Let $s \in S$ be a surface state.*

- *The level value $level(s)$ of s is $h(s)$ if s is a bench state, $hwm(succ(s))$ if s is a progress state with $s \notin S_{goal}$ and $-\infty$, otherwise.*
- *The set of* crater *states $Crater(s)$ of s contains each state s'' that is reachable from s on a path on which all states $s' \neq s$ (including s'' itself) satisfy $h(s') < level(s)$.*
- *The set of* surface *states $Surface(s)$ of s contains each state s' with $h(s') = level(s)$ that is a successor of s or a successor of a state from $Crater(s)$.*

The crater $\mathcal{C}(s)$ induced by s is a state space topology $\langle\langle S', s'_{init}, S'_{goal}, succ'\rangle, h'\rangle$ with

- $S' = \{s\} \cup Crater(s) \cup Surface(s)$,
- $s_{init} = s$,
- $S_{goal} = Surface(s)$,
- $succ'$ *is* $succ$ *restricted to transitions between states from S', and*
- h' *is* h *restricted to states from S'.*

We define craters independently of benches. This allows us to use craters in a more flexible way when analyzing search runs not only within benches but also between benches. This is no restriction since benches are defined as state spaces topologies as well. We use that $h(s') = level(s)$ holds by definition of surface state s' on bench $\mathcal{B}(s)$. Moreover, we use that if a surface state s is a progress state, then it is the first crater on a bench $\mathcal{B}(s)$ with level $level(s) = hwm(succ(s))$. We call a crater induced by a progress state *progress crater* and a crater induced by a bench state *bench crater*. The following statement holds in general for craters: a crater is induced by a trap state s iff $Crater(s)$ is not empty. We call a crater $\mathcal{C}(s)$ induced by a trap state s *real* crater. Otherwise, we call it *empty* crater. Moreover, a progress crater $\mathcal{C}(s)$ always consists of a solvable state space, while a bench crater $\mathcal{C}(s)$ can consist of an unsolvable state space because $Surface(s)$ can be empty for bench craters.

Example 12.4. *Crater* $\mathcal{C}(\mathrm{G})$ *has initial state* G, $level(\mathrm{G}) = 3$, $Crater(\mathrm{G}) = \{\mathrm{R}, \mathrm{W}, \mathrm{X}\}$, $Surface(\mathrm{G}) = \{\mathrm{L}\}$ *and is a progress crater that is real. Crater* $\mathcal{C}(\mathrm{L})$ *has initial state* L, $level(\mathrm{L}) = 3$, $Crater(\mathrm{L}) = \{\mathrm{P}, \mathrm{Q}, \mathrm{V}, \mathrm{W}\}$, $Surface(\mathrm{L}) = \{\mathrm{K}, \mathrm{M}\}$ *and is a bench crater that is real. Crater* $\mathcal{C}(\mathrm{S})$ *has initial state* S, $level(\mathrm{S}) = 1$, $Crater(\mathrm{S}) = \emptyset$, $Surface(\mathrm{S}) = \{\mathrm{U}\}$ *and is a bench crater that is empty. Crater* $\mathcal{C}(\mathrm{M})$ *has initial state* M, $Crater(\mathrm{M}) = \{\mathrm{W}\}$ *and* $Surface(\mathrm{M}) = \emptyset$. *It is a bench crater that is real and consists of an unsolvable state space.*

We formalize the observation that GBFS necessarily expands a set of crater states after the expansion of a trap state.

Theorem 12.1. *Let $\mathcal{T}$ be a state space topology with states S. Let $\eta[i]$ be a search history of GBFS on $\mathcal{T}$. Let s be the last expanded state in $\eta[i]$ (in iteration i) and let s be a progress state. Then GBFS necessarily expands $s' \in S$ in the search future of η if $s' \in Crater(s)$.*

Proof. We use the definition of $Crater(s)$ (Definition 12.4) and Theorem 10.7 to prove this theorem.

(i) With the definition of $Crater(s)$ (Definition 12.4), we know that all states s'' from $Crater(s)$ are reachable from s on a path on which all states $s' \neq s$ satisfy $h(s') < level(s)$.

(ii) We know that $level(s) = level(\eta[i])$ represents the value of the current hwm-level on which GBFS searches.

(iii) With Theorem 10.7, we know that GBFS necessarily expands a generated state s' with $h(s')$ smaller than the value $level(\eta[i])$ of the current hwm-level.

When GBFS expands s, then it generates the successor states of s. If $Crater(s)$ is not empty, then at least one of the successors of s is in $Crater(s)$. Due to (i) we know that GBFS is able to generate all states from $Crater(s)$, which implies with (iii) that GBFS necessarily expands all states from $Crater(s)$. □

A crater describes what is commonly understood as local minimum or (virtual) uninformative heuristic region (UHR). Xie, Müller, and Holte (2015) showed empirically that some state space topology from the planning community have many small UHRs within large virtual UHRs. Wilt and Ruml (2014) define a local minimum as a strongly connected component of particular states that are defined similarly to crater states. However, they only investigated undirected state spaces. In an undirected state space, a crater $\mathcal{C}(s)$ may consist of several of these local minima because each successor state of s may reach another strongly connected component of crater states. A more detailed analysis of the connectivity of states within craters is out of scope of this thesis. The advantage of craters is that they allow us to ignore their inner structure and to focus on the surface states that determine the main course of a search run.

12.3. Crater Space

In this section, we formalize the behavior of GBFS among craters. The resulting formalization allows us to ignore the search behavior within craters. Moreover, it provides us with a view on all the surface states and their connections to each other.

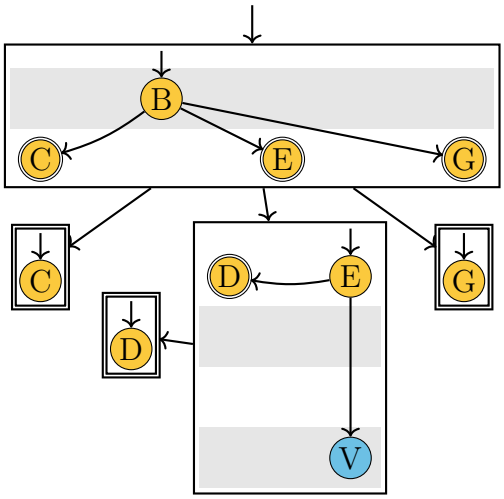
We observe that the expansions of surface states enable the generation of new surface states. Tie-breaking among surface states influences the time spent on a bench and determines on which next bench GBFS is going to search. By ignoring the behavior of GBFS within craters we can easily track search runs and analyze critical tie-breaking decisions. For example, when GBFS expands state G, then the crater $\mathcal{C}(\mathrm{G})$ allows us to immediately grasp the next surface states L and M and analyze the effect of their expansions without caring about the behavior of GBFS within the crater. At this point, we can analyze the effect of tie-breaking among surface states. The crater induced by state M has an empty set of surface states, which tells us that the expansion of M is quite useless in the process of reaching a goal state. The crater induced by state L has K and M in its set of surface states $\mathit{Surface}(\mathrm{L})$, which means that GBFS can reach these states after expansion of L. Consequently, expanding L is useful in reaching progress state K through which GBFS can leave the current bench.

We now define *crater spaces*, which connect craters. A crater space is induced by a state space topology.

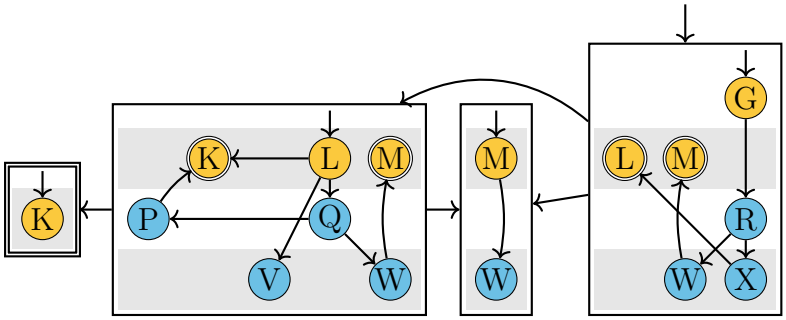
Definition 12.5 (crater space)**.** *Let $\mathcal{T}$ be a state space topology with initial state s_{init} and set of goal states S_{goal}.*

The crater space $\mathbb{C}(\mathcal{T})$ *of $\mathcal{T}$ is a state space $\langle C, \mathcal{C}_{init}, C_{goal}, succ\rangle$, where*

- *C is a set of* craters*, which is inductively defined as follows: $\mathcal{C}(s') \in C$ iff*
  - *$s' = s_{init}$ or*
  - *$s' \in \mathit{Surface}(s)$ with $\mathcal{C}(s) \in C$,*



(a) Crater space of bench $\mathcal{B}(\mathrm{B})$.



(b) Crater space of bench $\mathcal{B}(\mathrm{G})$.

Figure 12.1: Example crater spaces of benches from bench space of Figure 11.2.

- $\mathcal{C}_{init}$ *is the* initial crater $\mathcal{C}(s_{init})$,
- C_{goal} *is the set of* goal craters *with* $\mathcal{C}(s) \in C_{goal}$ *if* $s \in S_{goal}$ *and* $\mathcal{C}(s) \in C$, *and*
- $succ$ *is the* successor function *with* $\mathcal{C}(s') \in succ(\mathcal{C}(s))$ *if* $s' \in Surface(s)$ *for* $\mathcal{C}(s), \mathcal{C}(s') \in C$.

Example 12.5. *Figure 12.1 shows crater spaces* $\mathbb{C}(\mathcal{B}(\mathrm{B}))$ *and* $\mathbb{C}(\mathcal{B}(\mathrm{G}))$. *The rectangles are craters and the arrows between craters define the transitions between craters. The crater with the incoming arrow is the initial crater and the craters with double lined rectangles are the goal craters.*

We base the definition of the crater space on state space topologies because benches themselves are defined as state space topologies. Moreover, we will later see that it can be useful to consider crater spaces decoupled from benches in order to consider them straight away in the context of the whole state space topology.

A crater space gives us a high-level view on the behavior of GBFS that entails the information which crater can be reached from which crater. It is an abstract view on the behavior of GBFS that ignores details within craters. Nevertheless, one may pick a crater and analyze the behavior of GBFS within the crater isolated from all the other craters. One may also analyze how craters affect each other during a search run.

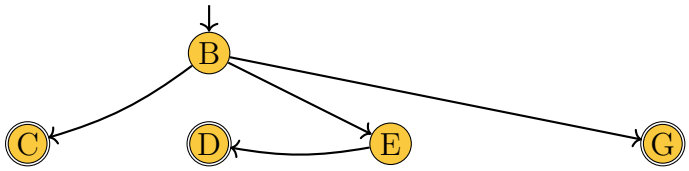
Overlapping Craters Craters do not partition a bench into distinct subspaces. Craters can highly overlap, i.e. a crater state can be shared among multiple craters. This is because craters capture the behavior of GBFS in a state space topology rather than representing topological structures that are independent of GBFS. When tracing a search run among craters, one has to consider that GBFS expands each state at most once (Proposition 9.1). Real craters are the reason for why benches can overlap along a bench path. In the following, we first discuss craters that overlap within a bench and then discuss craters that overlap among benches.

When craters on the same bench share a state s, then they overlap in the whole subspace that is reachable from s in the craters. Therefore, when GBFS has already expanded a crater state s in one crater $\mathcal{C}(t)$ and generates s in another crater $\mathcal{C}(t')$, then GBFS will not expand s again and will also not expand all the already expanded crater states in the subspace that is reachable from s in $\mathcal{C}(t')$. This holds because all the crater states in that subspace have already been necessarily expanded in $\mathcal{C}(t)$. For example, crater $\mathcal{C}(\mathrm{G})$ and $\mathcal{C}(\mathrm{L})$ overlap in state W on bench $\mathcal{B}(\mathrm{G})$. As GBFS explores $\mathcal{C}(\mathrm{G})$ before $\mathcal{C}(\mathrm{L})$ in a search run, it will expand M in $\mathcal{C}(\mathrm{G})$ and will not expand M in $\mathcal{C}(\mathrm{L})$. As GBFS does expand M only in $\mathcal{C}(\mathrm{G})$, it generates M only in $\mathcal{C}(\mathrm{G})$ and will not generate M in $\mathcal{C}(\mathrm{L})$.

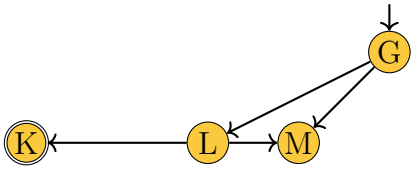
The behavior of GBFS is similar when craters overlap on benches of different levels. The difference is that crater states from a higher bench $\mathcal{B}(p)$ can become surface states on a lower bench $\mathcal{B}(p')$. When GBFS first expands s on $\mathcal{B}(p)$ as a crater state and then generates s again on $\mathcal{B}(p')$ as a surface state, then GBFS will not expand any state from the crater $\mathcal{C}(s)$ because states from $\mathcal{C}(s)$ have already been expanded in the crater on the higher bench. For example, craters $\mathcal{C}(\mathrm{O})$ from $\mathcal{B}(\mathrm{C})$ overlap with crater $\mathcal{C}(T)$ from $\mathcal{B}(\mathrm{N})$ in state T. While T is a crater state on bench $\mathcal{B}(\mathrm{C})$, it is a surface state on $\mathcal{B}(\mathrm{N})$. If GBFS expands T on $\mathcal{B}(\mathrm{C})$, then it will not expand $\mathcal{B}(\mathrm{C})$ on bench $\mathcal{B}(\mathrm{N})$. Another example of overlapping craters is $\mathcal{C}(\mathrm{E})$ from $\mathcal{B}(\mathrm{B})$ and $\mathcal{C}(L)$ from $\mathcal{B}(\mathrm{G})$, which overlap in state V. State V is a crater state in both craters.

12.4. Surface State Space

The advantage of crater spaces is that they show the connections between craters. However, we may be more interested in the connections between surface states that are induced by the craters. Visualizing these connections helps to observe the critical tie-breaking decisions of search runs. We introduce the *surface state space* that is nothing more than a crater space where all craters are replaced with their defining surface states.



(a) Surface state space of bench $\mathcal{B}(\mathrm{B})$.



(b) Surface state space of bench $\mathcal{B}(\mathrm{G})$.

Figure 12.2: Example surface state spaces of benches from bench space of Figure 11.2.

Definition 12.6 (surface state space)**.** *Let $\mathcal{T}$ be a state space topology. Let $\mathbb{C}(\mathcal{T})$ be the crater space of $\mathcal{T}$ with set of craters C.*

A surface state space *$\mathcal{F}(\mathcal{T})$ of $\mathcal{T}$ is $\mathbb{C}(\mathcal{T})$ where each crater $\mathcal{C}(s) \in C$ is replaced with s.*

Example 12.6. *Figure 12.2 show surface state spaces $\mathcal{F}(\mathcal{B}(\mathrm{B}))$ of and $\mathcal{F}(\mathcal{B}(\mathrm{G}))$.*

In a surface state space, we can follow different possible search runs among surface states. At the same time we can completely ignore the search behavior among crater states. When we need to analyze the possible search runs with respect to the structure of a crater, we can still access the required information with $\mathcal{C}(s)$, $level(s)$ and $Crater(s)$ based on the underlying bench or state space topology. Note that $succ(s)$ of a surface state space equals $Surface(s)$.

A surface state space $\mathcal{F}$ can be induced by a bench but also by the whole underlying state space topology. Figure 12.3 shows the surface state space of our example state space topology. In this space the benches are not explicit anymore. The surface state space contains all states that are surface states on at least one bench. It only contains transitions between surface states that appear together on the same bench. Transitions that only exist within craters and start or end in a crater state are not present in the surface state graph. For example, the transition $\mathrm{O} \rightarrow \mathrm{T}$ from crater $\mathcal{C}(\mathrm{O})$ starts in surface state O and ends in crater state T. As it only exists within a crater, it is not visible in the surface state space. Transitions between two states with different h-values only start from progress states. For example, E is not a progress state. Therefore, it has no transition to a state with lower h-value. State D is a progress state and has a transition to state J with lower h-value.

When analyzing possible search runs in $\mathcal{F}$, one still has to consider the semantic of progress states. It means that after expanding a progress state s in $\mathcal{F}$, GBFS continues its search with only expanding states with smaller h-value than that of s (GBFS transitions to a lower level) and that are reachable from s in $\mathcal{F}$ (GBFS searches in a bench). In general,

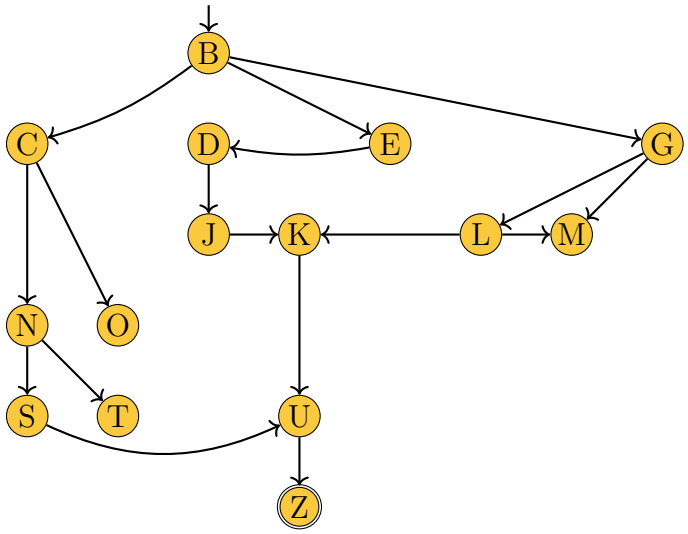


Figure 12.3: Surface state space of our example state space topology from Figure 9.1.

a search run does not induce a single path in the surface state space. However, a good tie-breaking policy will follow a single acyclic surface state solution path in $\mathcal{F}$. Such a path respects the property of progress states without the need of explicitly enforcing it.

13. Roles and Context of States

We have already observed that benches and craters can overlap within and between different hwm-levels because they define subspaces in a state space rather than a partitioning of the state space. In this section, we set the focus on single states and their roles in different context based on hwm-level, benches and craters. We determine the space complexity of explicitly associating each state with the possible context in which it appears. As a context is given by hwm-levels, benches and craters, the analysis also reveals the space complexity and runtime complexity of generating and storing bench spaces and crater spaces.

13.1. Alternative Criterion for State Expansion

The criteria which we have discovered in the last sections for whether GBFS never, potentially or necessarily expands a generated state s (from $Generated(\eta[i])$ of search history $\eta[i]$) are based on the relation between h-value of s and the hwm-level $level(\eta[i])$ (e.g., Theorems 10.5, 10.7, 10.8). These criteria imply that we need to find an explicit search history for deciding whether GBFS potentially expands state s. However, it suffices to regard a path from the initial state s_{init} to s that fulfills a condition based on high-water mark. We first define the set of generated states $Generated(\rho)$ of a state path ρ as

$$Generated(\rho) = \{s\} \cup \bigcup_{s \in \rho} succ(s).$$

We use $Generated(\langle s_{\text{init}}, \ldots, s \rangle)$ to determine whether s is potentially expanded by GBFS.

Theorem 13.1. *Let $\mathcal{T}$ be a state space topology with states S. Then GBFS potentially expands $s \in S$ iff there exists a path $\langle s_0, \ldots, s_n \rangle$ in $\mathcal{T}$ with $s_0 = s_{init}$ and $s_n = s$ and $h(s_i) \leq hwm(Generated(\langle s_0, \ldots, s_{i-1} \rangle)$ for all $i \in \{1, \ldots, n\}$.*

Proof. The initial state s_{init} is obviously expanded. Each expansion of a state s from a path $\langle s_0, \ldots, s_n \rangle$ generates a successor state $s' \in succ(s)$ that follows s on $\langle s_0, \ldots, s_n \rangle$. The rest can be shown with Theorem 10.8 by considering $\langle s_0, \ldots, s_n \rangle$ as a search run instead of a path and ensuring for each $i > 0$ that s_i is potentially expanded given $\langle s_0, \ldots, s_{i-1} \rangle$. □

This theorem relates to the sufficient and necessary condition for state expansions of Dechter and Pearl (1985) that we have discussed in Section 7.5. On the one hand, it is more general by considering arbitrary graphs instead of trees only. On the other hand, it is more restricted by only considering deterministic heuristics. In contrast to Theorems 10.8 and 11.4, the criterion of this theorem for potentially expanded states is independent of explicit search runs, benches and bench spaces. A path to a state s that fulfills the criterion determines the hwm-level, the crater and the bench on which state s appears.

13.2. Context of States

A path to a state that satisfies Theorem 13.1 determines the context of the state. The context is given by high-water mark levels, benches and craters.

High-Water Mark Levels In a search run, GBFS reaches a state s on a path and the path ends on a specific hwm-level. Different paths to s can end in different hwm-levels. Therefore, we associate s with different hwm-levels and consider s in context of hwm-levels. Explicitly associating each state with their hwm-levels results in space complexity $\mathcal{O}(n^2)$ because there can be at most n states in a state space topology, each associated with at most n levels. When we consider the path ρ on which GBFS reaches s, then we know that the hwm-level, which provides the context of s, is determined by $level(s')$ of the progress state s' that precedes s on ρ.

Benches Different paths to a state s can end on different benches. Therefore, we can associate s with different benches or the progress states that induce the benches. Explicitly associating each state s with the benches or progress states results in a space complexity of $\mathcal{O}(n^2)$ because each of n states from a state space topology can be on at most n benches, each induced by another progress state. The bench on which s appears is determined by s itself if s is a progress state or by the progress state that precedes s on a given path to s.

Craters Different paths to a state s can end in different craters. Therefore, we can associate s with different craters or the surface states that induce the craters. Explicitly associating each state s with the craters or surface states results in a space complexity of $\mathcal{O}(n^2)$ because each of n states from a state space topology can be in at most n craters, each induced by another surface state. The crater on which s appears is determined by s itself if s is a surface state or by the surface state that precedes s on a given path to s.

Note, that associating a state in context of craters and benches would result in a space complexity of $\mathcal{O}(n^3)$ because each of n states from a state space topology can be in at most n benches and on each of the benches s can be in at most n craters.

13.3. Roles of States

The roles of states are determined by the hwm-level on which they are reached.

Progress State A progress state always has the role of a progress state and is only associated with a single hwm-level l which is $h(s) = hwm(s) = l$ (Theorem 11.2). We have already seen that it can be determined locally. It is always a surface state. It additionally can be a trap state if $h(s') < hwm(succ(s))$ for at least one $s' \in succ(s)$.

Bench State While a bench state can be determined locally, it has many different roles that depend on the context of an hwm-level. Let us consider a state s in context of hwm-level with value l. Whether s appears in the role of a crater state or a surface state in this context depends on the heuristic value of s. State s is a crater state on level l if $h(s) < l$ and a surface state if $h(s) = l$. While we introduced these roles in context of benches, we clarify that they rather depend on an hwm-value than on particular a bench. A bench state that is a surface state on level l can have the additional role of being a trap state if $h(s') < l$ for at least one $s' \in succ(s)$.

We identify two other roles in context of hwm-levels: *alive* states and *dead* states. The roles determine whether GBFS is able to find an s-path of a state s in context of an hwm-level. The roles depend on an hwm-level l and on the high-water mark value of s. An *alive* state in context of hwm-level l is a state s with $hwm(s) = l$. A *dead* state in context of hwm-level l is a state s with $hwm(s) > l$. GBFS is able to find an s-path from a state s in the role of an alive state but not in the role of a dead state. A dead state can be regarded as a dead end on a bench.

14. Best-Case and Worst-Case Behavior

In this chapter, we characterize best-case and a worst-case search runs based on our knowledge about the search behavior of GBFS. We show that the problem of determining these search runs is NP-complete in general and identify some conditions under which the problem becomes tractable.

14.1. Best-Case Search Run

A best-case search run of GBFS is simply a shortest search run (Definition 5.3). Our knowledge about the search behavior of GBFS allows us to characterize the behavior of GBFS along a best-case search run. We know that the expansions of surface states determine the main course of search runs because surface states are critical for tie-breaking. Therefore, we can use the surface state space (Section 12.6) to characterize search runs. In its best-case behavior, GBFS only expands surface states along a single surface state path. It implies that in this case, GBFS does not expand surface states that are dead states (Section 13.3). A surface state path only reflects the behavior of GBFS among surface states, but we also have to consider crater states that are necessarily expanded with the expansion of surface states. Moreover, we have to consider that craters induced by surface states can overlap and that GBFS expands each state at most once.

We reformulate the optimization problem from Definition 5.3 such that it involves a surface state space and sets of crater states.

Definition 14.1 (best-case search run based on surface state space)**.** *Let $\mathcal{T}$ be a state space topology. Let $\mathcal{F}(\mathcal{T})$ be the surface state space of $\mathcal{T}$. Let P be the set of all acyclic solution paths in $\mathcal{F}(\mathcal{T})$.*

A best-case search run of GBFS is determined by

$$\arg\min_{\rho \in P} \big(\mathit{length}(\rho) + |\bigcup_{f \in \rho} \mathit{Crater}(f)| \big).$$

A solution of the optimization problem is a path of surface states $\langle f_1, f_2, \ldots, f_n \rangle$. It translates to a best-case search run $\eta = \langle f_1, \ldots, f_2, \ldots, f_n \rangle$ where each f_i with $1 \leq i \leq n$ is followed by state $c \in \mathit{Crater}(f_i)$ if c does not appear in η previously to f_i and the ordering among crater states is induced by an arbitrary tie-breaking policy of GBFS.

Example 14.1. *In the surface state space of Figure 12.3, the surface state path that minimizes the given optimization problem is* $\langle \mathrm{B}, \mathrm{C}, \mathrm{N}, \mathrm{S}, \mathrm{U}, \mathrm{Z} \rangle$ *and results in* 6*. It directly translates to the best-case search run* $\langle \mathrm{B}, \mathrm{C}, \mathrm{N}, \mathrm{S}, \mathrm{U}, \mathrm{Z} \rangle$ *of length* 6 *of GBFS that runs on the state space topology from Figure 9.1.*

14.2. Worst-Case Search Run

A worst-case search run is simply a longest search run (Definition 5.3). With our knowledge about the search behavior of GBFS we can characterize the behavior of GBFS along a worst-case search run. We know that when GBFS arrives on a bench, it potentially expands all the states on the bench. A worst-case search run will indeed expand all these states on a bench. We also know that a bench is induced by a progress state. Therefore, we can use progress state spaces (Definition 11.6) to characterize worst-case search runs. A path in a progress state space represents a set of possible search runs. We have to consider that benches induced by progress states can overlap and that GBFS expands each state at most once.

We reformulate the optimization problem from Definition 5.3 to be based on progress state spaces and sets of bench states.

Definition 14.2. *Let* $\mathcal{T}$ *be a state space topology. Let* $\mathcal{P}(\mathcal{T})$ *be the progress state space of* $\mathcal{T}$*. Let* P *be the set of all solution paths in* $\mathcal{P}$*.*

A worst-case search run of GBFS is determined by

$$\arg\max_{\rho \in P} \big(length(\rho) + |\bigcup_{p \in \rho} Bench(p)| \big).$$

A solution $\langle p_1, p_2, \ldots, p_n \rangle$ from the optimization problem translates to a worst-case search run $\eta = \langle p_1, \ldots, p_2, \ldots, p_n \rangle$ where each p_i with $1 \leq i \leq n$ is followed by state $b \in Bench(p_i)$ if b does not appear in η previously to p_i and the ordering among bench states is induced by an arbitrary tie-breaking policy of GBFS.

Example 14.2. *In the progress state space of Figure 11.3, the progress state path that maximizes the given optimization problem is* $\langle \mathrm{B}, \mathrm{G}, \mathrm{K}, \mathrm{U}, \mathrm{Z} \rangle$ *and results in* 14*. It translates to the worst-case search run* $\langle \mathrm{B}, \mathrm{E}, \mathrm{V}, \mathrm{G}, \mathrm{R}, \mathrm{W}, \mathrm{X}, \mathrm{L}, \mathrm{Q}, \mathrm{P}, \mathrm{M}, \mathrm{K}, \mathrm{U}, \mathrm{Z} \rangle$ *of length* 14 *of GBFS that runs on the state space topology from Figure 9.1.*

14.3. NP-Completeness Results

We study the *decision problems* that are related to the given optimization problems and show that they are already NP-complete. It follows that the optimization problems are NP-equivalent (Garey and Johnson, 1979) because the optimization problems can be solved based on the decision problems.

Definition 14.3 (GBFSBESTCASE decision problem)**.** *The* GBFSBESTCASE *decision problem is defined as follows: given a state space topology $\mathcal{T}$ and $K \in \mathbb{N}_0$, does there exist a GBFS search run on $\mathcal{T}$ with at most K expanded states?*

Definition 14.4 (GBFSWORSTCASE decision problem)**.** *The* GBFSWORSTCASE *decision problem is defined as follows: given a state space topology $\mathcal{T}$ and $K \in \mathbb{N}_0$, does there exist a GBFS run on $\mathcal{T}$ with at least K expanded states?*

We now show that both problems are NP-complete. To be clear, we assume that the state spaces in the input are explicitly represented as directed graphs, so the hardness is *not* related to the state explosion problem that makes state-space search in implicitly defined state spaces hard.

Theorem 14.1. GBFSBESTCASE *is NP-complete.*

Proof. For membership in NP, we guess a run for the given state space topology that satisfies the given bound and verify that it is a legal run.

For hardness, we polynomially reduce from the NP-complete VERTEXCOVER problem: given a graph $\langle V, E\rangle$ and number M, does there exist a subset $C \subseteq V$ with $|C| \leq M$ such that $C \cap e \neq \emptyset$ for all $e \in E$?

Given a VERTEXCOVER instance with vertices V, edges $E = \{e_1, \ldots, e_n\}$ and bound M, we produce a GBFSBESTCASE instance with bound $K = 2n + 2 + M$ and a state space topology with initial state s_1 with heuristic value 3; *edge branch states* $s_2, \ldots, s_{n+1}$ with heuristic value 2; for every edge $e_i = \{u, v\}$, two *edge decision states* s_i^u and s_i^v with heuristic value 2; for every vertex $v \in V$, a *vertex state* v with heuristic value 1; a goal state s^* with heuristic value 0; for every edge e_i and vertex $v \in e_i$, transitions $s_i \to s_i^v$, $s_i^v \to s_{i+1}$ and $s_i^v \to v$; and a transition $s_{n+1} \to s^*$.

The reduction is illustrated in Figure 14.1. Every GBFS run must expand the initial state (1 expansion), all edge branch states (n expansions), at least one edge decision state per edge (n expansions) and the goal state (1 expansion). If edge decision state e_i^u is expanded, vertex state u must also be expanded. It is easy to see that the expanded vertex states form a vertex cover, and that a run with at most $K = 2n + 2 + M$ expansions exists iff there exists a vertex cover of size at most M. □

The difficulty of minimizing the number of states in GBFS search runs stems from *overlapping craters*: expanding certain states (the edge decision states in the reduction) forces the expansion of adjacent *crater states* that form a heuristic depression (the vertex states in the reduction). The cost incurred by expansions of crater states cannot be quantified locally because craters can overlap and global optimization is hence required.

In contrast to the best case, the *worst case* of GBFS is easy to determine for a single bench: the worst case on a single bench is always to expand all bench states, and then any progress state. Still, analyzing the worst case of GBFS is hard as we prove next.

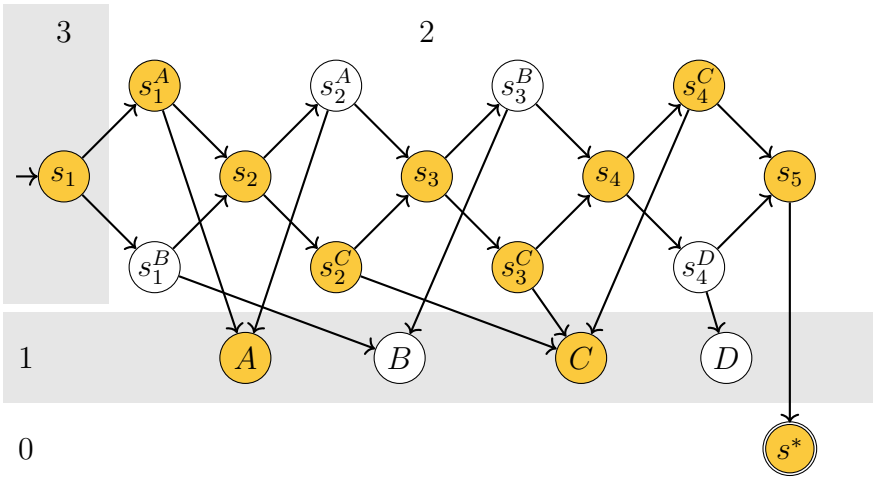


Figure 14.1: Illustration of the proof of Theorem 14.1, depicting a GBFS run that allows us to compute the vertex cover of a graph $\langle\{A, B, C, D\}, \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}\rangle$.

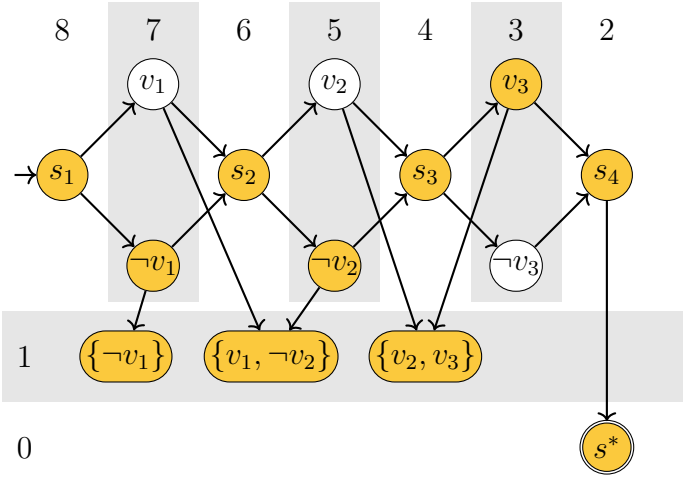


Figure 14.2: Illustration of the proof of Theorem 14.2, depicting a GBFS run that allows us to determine the satisfiability of the formula $\neg v_1 \wedge (v_1 \vee \neg v_2) \wedge (v_2 \vee v_3)$.

Theorem 14.2. GBFSWORSTCASE *is NP-complete.*

Proof. For membership in NP, we guess a run for the given state space topology that satisfies the given bound and verify that it is a legal run.

For hardness, we polynomially reduce from the NP-complete SAT problem: given a set of propositional variables V and a set of clauses C over V (represented as sets of literals), does there exist a truth assignment for V that satisfies all clauses?

Given a SAT instance with variables $V = \{v_1, \ldots, v_n\}$ and clauses C, we produce a GBFSWORSTCASE instance with bound $K = 2n + 2 + |C|$ and a state space topology with *variable branch states* $s_1, \ldots, s_{n+1}$ with heuristic values $h(s_i) = 2(n - i) + 4$ (s_1 is the initial state); for every variable v_i, two *literal states* v_i and $\neg v_i$ with heuristic value $2(n-i)+3$; for every clause $c \in C$, a *clause state* c with heuristic value 1; a goal state s^* with heuristic value 0; for every variable v_i and literal $\ell \in \{v_i, \neg v_i\}$, transitions $s_i \to \ell$, $\ell \to s_{i+1}$ and $\ell \to c$ for all clauses c with $\ell \in c$; and a transition $s_{n+1} \to s^*$.

The reduction is illustrated in Figure 14.2. It is very similar to the one from the previous proof. The main difference is that we define the heuristic values in such a way that GBFS is prevented from expanding both literal states belonging to the same variable in one run. Truth assignments correspond to paths through the variable branch and literal states. From this, it is easy to see that satisfying assignments correspond to runs where all clause vertices are expanded, which is equivalent to saying that at least $K = 2n + 2 + |C|$ states are expanded. □

The hardness stems from the difficulties caused by overlapping benches that are on different levels. These benches overlap because of craters. The difference to the result above is that now the overlap of craters within a bench does not contribute to the hardness.

14.4. Tractability Results

In this section, we discuss properties of state space topologies that can be determined in polynomial time and that allow us a polynomial time computation of GBFSBESTCASE and GBFSWORSTCASE despite the hardness of the general case.

Overlap-Free Benches and Craters The hardness of the given decision problems results from overlapping craters and benches. They can overlap in many different ways, but not all overlapping crater and benches affect each other in a search run. We present a criteria that characterize state space topologies in which none of the craters or benches interfere each other in any possible search run.

Definition 14.5 (overlap-free surface state space)**.** *Let $\mathcal{T}$ be a state space topology. Let $\mathcal{F}(\mathcal{T})$ be the surface state space of $\mathcal{T}$ with surface states S.*

$\mathcal{F}$ is overlap-free *iff for each pair of distinct surface states f, $f' \in S$ following condition holds: if there exists a path between f and f' in $\mathcal{F}$, then $Crater(f) \cap Crater(f') = \emptyset$.*

Definition 14.6 (overlap-free progress state space)**.** *Let $\mathcal{T}$ be a state space topology. Let $\mathcal{P}(\mathcal{T})$ be the progress state space of $\mathcal{T}$ with progress states S.*

$\mathcal{P}(\mathcal{T})$ is overlap-free *iff for each pair of distinct progress states p, $p' \in S$ following condition holds: if there exists a path between p and p' in $\mathcal{P}(\mathcal{T})$, then $Bench(p) \cap Bench(p') = \emptyset$.*

Both properties can be determined in time and space that is polynomial in the number of states of a given state space topology. We now investigate the complexity of GBFS-BESTCASE and GBFSWORSTCASE given that an instance is overlap-free.

Theorem 14.3. GBFSBESTCASE *can be decided in polynomial time and space for a given state space topology $\mathcal{T}$ if the surface state space $\mathcal{F}(\mathcal{T})$ is overlap-free.*

Proof. We show membership of GBFSBESTCASE for overlap-free surface state spaces in P by reducing the problem to a shortest path problem.

As the given surface space $\mathcal{F}$ is overlap-free, there are no dependencies between craters in any search run. Let P be the set of all solution paths in $\mathcal{F}$, then following optimization problem determines a best-case search run:

$$\arg\min_{\rho \in P} \big(length(\rho) + \sum_{f \in \rho} |Crater(f)| \big).$$

We can reduce this optimization problem to a shortest path problem, which is known to be decidable in polynomial time and space. An instance of the shortest path problem consists of $\mathcal{F}$ and cost function

$$cost(f, f') = 1 + |Crater(f)|.$$

□

Not surprisingly the result for the worst case is analogue to the best case.

Theorem 14.4. GBFSWORSTCASE *can be decided in polynomial time and space for a given state space topology $\mathcal{T}$ if the progress space $\mathcal{P}(\mathcal{T})$ is overlap-free.*

Proof. We show membership of GBFSWORSTCASE for overlap-free progress spaces in P by reducing the problem to a longest path problem in a directed acyclic graph.

As the given progress space $\mathcal{P}$ is overlap-free, there are no dependencies between benches in any search run. Let P be the set of all solution paths in $\mathcal{F}$, then following optimization problem determines a worst-case search run:

$$\arg\max_{\rho \in P} \big(length(\rho) + \sum_{p \in \rho} |Bench(p)| \big).$$

As we know that $\mathcal{P}$ is a directed acyclic graph, we can reduce this optimization problem to a longest path problem in a directed acyclic graph. The longest path problem

is known to be decidable in polynomial time and space because it can be reduced to a shortest path problem by negating the given weights. The resulting shortest path problem is polynomial-time computable because there are no negative cost circles in a directed acyclic graph. An instance of the longest path problem consists of $\mathcal{P}$ and cost function

$$cost(p, p') = 1 + |Bench(p)|.$$

□

Undirected State Spaces Craters and benches can overlap each other in state space topologies with undirected state spaces, but only when they are next to each other. In other words, when two craters (or benches) share a crater state (bench state) then they are neighbors. Consequently, they affect each other in a search run but only locally. We now investigate these statements in more detail.

Theorem 14.5. GBFSBESTCASE *can be decided polynomial time and space for a given state space topology $\mathcal{T}$ if the state space of $\mathcal{T}$ is undirected.*

Proof. We show that only two consecutively visited craters can overlap each other along a shortest search run in an undirected state space topology. To do so, we use the surface state space $\mathcal{F}(\mathcal{T})$ and show that only craters induced by two consecutive surface states in a shortest surface state path can overlap.

We do a proof by contradiction and assume that there is a shortest surface state path ρ with three surface states f, f' and f'', and a crater state c with $c \in Crater(f) \cap Crater(f') \cap Crater(f'')$. We consider following cases:

- f, f' and f'' induce bench craters: Having an undirected state space and $c \in Crater(f) \cap Crater(f') \cap Crater(f'')$ imply that there is an undirected transition between each surface state. Since there is a transition from f to f'', we can remove f' from ρ, which results in a shorter surface state path.

- f and f' induce bench craters and f'' induces a progress crater: We use the same reasoning as above except that there are no transitions from f'' to f or to f' because f'' induces a progress crater.

- f induces a progress crater and f' and f'' induce bench craters: We use the same reasoning as above except that there are no transitions from f'' to f and from f' to f because f induces a progress crater.

- f and f'' induce bench or progress craters and f' induces a progress crater:
  (i) $c \in Crater(f)$ implies $hwm(c) \geq level(f)$.
  (ii) f' being a successor of f, and f' being a progress crater imply $level(f) > level(f')$.

(iii) f'' being a successor of f', and f'' being on a surface state solution path (surface state solution paths only consist of alive states) imply $level(f') = hwm(f'')$.

(iv) Having an undirected state space and $c \in Crater(f'')$ imply that there is a c-path over f'' in $\mathcal{C}(f'')$, and hence $hwm(c) \leq hwm(f'')$.

With this, we have $hwm(c) \overset{(i)}{\geq} level(f) \overset{(ii)}{>} level(f') \overset{(iii)}{=} hwm(f'') \overset{(iv)}{\geq} hwm(c)$.

All cases contradict our assumption. Consequently, dependencies can only arise between craters induced by two consecutive surface states along a shortest surface state path. Let P be the set of all solution paths in $\mathcal{F}(\mathcal{T})$, then following optimization problem determines a best-case search run:

$$\arg \min_{\langle f_1,\ldots,f_n\rangle \in P} \big(n + Crater(f_1) + \sum_{i=2}^{n} |Crater(f_i) \setminus Crater(f_{i-1})|\big).$$

We can reduce this optimization problem to a shortest path problem, which is known to be computable in polynomial time and space. An instance of the shortest path problem consists of $\mathcal{F}$ and cost function

$$cost(f, f') = 1 + |Crater(f') \setminus Crater(f)|.$$

□

Theorem 14.6. GBFSWORSTCASE *can be decided in polynomial time and space for a given state space topology $\mathcal{T}$ if the state space of $\mathcal{T}$ is undirected.*

Proof. We show that only consecutively visited benches can overlap along a search run in an undirected state space topology. To do so, we use the progress state space $\mathcal{P}(\mathcal{T})$ and show that bench states can only be shared between benches induced by two consecutive progress states along a progress state path.

We do a proof by contradiction and assume that there is a progress state path in $\mathcal{P}(\mathcal{T})$ with three progress states p', p'' and p''', and $b \in Bench(p) \cap Bench(p'')$.

(i) $b \in Bench(p)$ implies that $hwm(p) \geq level(p)$.

(ii) p' being a successor of p implies $level(p) > level(p')$.

(iii) p'' being a successor of p' implies $hwm(p'') = level(p')$.

(iv) Having an undirected state space and $b \in Bench(p'')$ implies that there is a b-path in $\mathcal{B}(p'')$ over p'', and hence $hwm(p) \leq hwm(p'')$.

With this, we have $hwm(p) \overset{(i)}{\geq} level(p) \overset{(ii)}{>} level(p') \overset{(iii)}{=} hwm(p'') \overset{(iv)}{\geq} hwm(p)$, which contradicts our assumption. Since dependencies can only arise between consecutive benches, following optimization problem determines a worst-case search run:

$$\arg \max_{\langle p_1,\ldots,p_n\rangle \in P} \big(n + Bench(p_1) + \sum_{i=2}^{n} |Bench(p_i) \setminus Bench(p_{i-1})|\big).$$

As we know that $\mathcal{P}$ is a directed acyclic graph, we can reduce this optimization problem to a longest path problem in a directed acyclic graph. The longest path problem is known to be decidable in polynomial time and space because it can be reduced to a shortest path problem by negating the given weights. The resulting shortest path problem is polynomial-time computable because there are no negative cost circles in a directed acyclic graph. An instance of the longest path problem consists of $\mathcal{P}$ and cost function

$$cost(p, p') = 1 + |Bench(p') \setminus Bench(p)|.$$

□

In this part of the thesis, we analyzed the search behavior of GBFS. We extracted important aspects of its behavior by asking a few basic questions about which states GBFS expands and when GBFS makes progress. We discovered that a search run can be decomposed into episodes and that each in each episode GBFS searches in another subspace of the state space that we call benches. Being aware of benches and the connection between benches allowed us to characterize the worst-case behavior of GBFS. On a more detailed level, we identified subspaces, called craters, which capture the behavior of GBFS when searching in local minima and which connect surface states that are critical for tie-breaking. Being aware of surface states, craters and the connection between surface states allowed us to characterize the best-case behavior of GBFS. With these characterizations we are able to extract best-case and worst-case search runs that would otherwise be infeasible.

Part III.

Algorithms and Experimental Results

All the theoretical results about the behavior of GBFS allow us to analyze GBFS on state spaces and with heuristic that are used in practice. The ability to extract the benches, craters, potentially expanded states and best-case or worst-case search runs would support in depth analyses of particular search instances. Therefore, we present algorithms that extract these information from search instances. Moreover, we conducted experiments on state spaces and a heuristic from the planning community in order to show that extracting these information is feasible in practice. We use the results to answer some first questions about the behavior of GBFS on the given instances. The implementation of the algorithms and the experimental data are published online[1].

[1] https://doi.org/10.5281/zenodo.3381388

15. Algorithms

In this chapter, we introduce the algorithms that allow us to analyze the search behavior of GBFS in state spaces from actual search problems and with heuristics that are used in practice. We show how the high-water mark of states can be computed. We introduce potential state spaces and explain how a potential state space can be extracted from a state space topology. The potential state space and the high-water marks are the main ingredients for generating the topological structures like benches, bench spaces and progress state spaces. After knowing how to generate surface state spaces and progress state spaces, we describe a procedure for testing if these spaces are overlap-free. Finally we present the algorithms for finding best-case and worst-case search runs.

15.1. High-Water Mark

We begin with presenting two straightforward approaches for determining the high-water mark of states. Afterwards, we consider the fact that for our applications, we only require the high-water mark values of potentially expanded states. We then combine the introduced approaches to determine the high-water marks in a more sophisticated way.

Dynamic Programming Approach We call the first approach *dynamic programming* approach (DP). It uses the functional equation of high-water mark (Definition 9.6) and the fact that high-water marks are monotonic (Theorem 9.2). It takes a state space and a heuristic as input and returns the hwm-values of states from the state space. With `hwm` we denote the function that is initially undefined for all states and that will eventually assign states with their hwm-values. We generally use the typewriter font to denote data structures that are defined specifically for these approaches.

DP consists of two phases: an *exploration* phase and a *dynamic programming* phase. In the exploration phase, DP exhaustively explores the given state space in order to get the set of all reachable states `S`, to build the predecessor function `pred` and to get the set of reachable goal states `G`.

In the dynamic programming phase, DP first initializes `hwm` and then starts a process that updates `hwm` until `hwm` assigns all states with their exact hwm-values. For each state $s \in \texttt{S}$, DP initializes `hwm` with $\texttt{hwm}(s) = h(s)$ if $s \in \texttt{G}$ and with $\texttt{hwm}(s) = \infty$, otherwise. We already know the hwm-values of goal states from `G` because a goal state determines its hwm-value with $h(s)$ (Definition 9.4). Then DP iteratively processes each unprocessed state preferring states s' that minimizes $\texttt{hwm}(s')$ among unprocessed states. When DP

processes a state s', then it updates $\texttt{hwm}(s)$ with $\max(h(s), \texttt{hwm}(s'))$ for each predecessor $s \in \texttt{pred}(s')$ if $\texttt{hwm}(s) = \infty$. This expresses the functional equation of high-water mark $\max(h(s), \min_{s' \in succ(s')} hwm(s))$ for state s. This expression is correct because $\texttt{hwm}(s)$ receives the smallest $\texttt{hwm}(s')$ among successor states s' from $succ(s)$ already at its first update. This is because DP processes states with low hwm-values first due to preferring states with low hwm-values. When DP has processed all states from $\texttt{S}$ then the hwm-values of all reachable states $s \in \texttt{S}$ are computed.

The worst-case runtime complexity of this approach is $\mathcal{O}(n \log n + m)$ in the number n of states and the number m of transitions from the given state space. The term $n \log n$ is due to the cost of inserting the states into a priority queue and m is due the need of processing all edges.

Search Run Approach We call the second approach *search run* approach (SR). It uses the fact that the hwm-value of a state s is the largest h-value of a state that GBFS starting in s expands during a search run if s is not a dead end and ∞, otherwise (Theorem 10.6). Therefore, running a GBFS that starts in state s suffices to determine $hwm(s)$. However, as this approach may expand all of the n states from a state space, the runtime complexity of determining the hwm-values for all of the n states is $\mathcal{O}(n \cdot (n \log n + m))$, with the number of transitions m from the given state space.

Pruning Both approaches can be improved in terms of fewer visited states when we are only interested in the high-water mark of potentially expanded states, and if knowing a lower bound on the hwm-values is sufficient for a desired application. The new approaches use an upper bound $\texttt{b} \in \mathbb{R}$ and prune states with h-values above this bound.

We denote the improved approach for DP with DP′. It applies high-water mark pruning of the initial state (Theorem 10.1). It is identical to DP except of the following differences. DP′ first determines the hwm-value of the initial state with SR. This value defines $\texttt{b}$. In the exploration phase, DP′ prunes all states with h-values above $\texttt{b}$ and adds the pruned states to $\texttt{G}$ in addition to the goal states. The dynamic programming phase is identical to that of DP. The hwm-values of pruned states from $\texttt{G}$ constitute *lower bounds* on their exact values. One can omit the insertion of pruned states to $\texttt{G}$ at the cost of weaker lower bounds.

We denote the improved approach for SR with SR′. Assume we are only interested in an hwm-value of a state if the hwm-value is not larger than a given bound $\texttt{b}$. The bound might be implied by the knowledge that a given state is considered in context of an hwm-level (Section 13.2). For example, a state s might be considered on hwm-level with value $\texttt{b}$. If $\texttt{b}$ is defined, then SR′ avoids to perform a whole GBFS run from s when the hwm-value of s is higher than $\texttt{b}$. SR′ is identical to SR except of the following differences. SR′ runs a GBFS search that prunes all states with h-values above $\texttt{b}$. When the search stops without finding a goal state, then SR′ returns the lowest h-value among all pruned states as a lower bound on the hwm-value of s, or ∞ if no states has been pruned.

Set of States Sometimes we might be interested in the high-water mark of a set of states. For example, we might want to compute $hwm(succ(s))$ in order to test if state s is a hwm-progress state. The hwm-value of a set of states is determined by a state with minimum hwm-value among the states from the set (Definition 9.5).

We add the ability to SR′ to compute hwm-values of set of states. By considering a set, we can save individual applications of SR′ to each of the states from the set. SR′ starts GBFS from a set of states, i.e., uses this set as initial open states without any other open or closed states. The rest is identical to SR.

Exploration Now we combine DP′ and SR′ to a new approach DPSR. For SR′ we wish to extract more information from a single GBFS search run than just the hwm-value of the given set of states. For DP′ we wish to only explore the part of a state space that only consists of states that GBFS potentially expands.

DPSR has an exploration and a dynamic programming phase which are similar to DP′. In the exploration phase, instead of a simple state-space search, DPSR starts a GBFS from the given set of states, prunes states with h-values above the given bound $\mathtt{b}$, and stops when GBFS selects a goal state or when the bounded space is fully explored. After GBFS stops, DPSR adds all pruned states, the possibly reached goal state and the open states to $\mathtt{G}$. The hwm-values of pruned or open states from $\mathtt{G}$ constitute *lower bounds* on their exact hwm-values. The dynamic programming phase is almost identical to that of DP′. It additionally maintains a set $\mathtt{E}$ that contains all the states s for which $\mathtt{hwm}(s)$ is the exact hwm-value of s. Initially $\mathtt{E}$ contains all the reached goal states from $\mathtt{G}$ because we know their exact hwm-values. During the dynamic programming phase, DPSR adds those states s to $\mathtt{E}$ for which $\mathtt{hwm}(s)$ turns out to be the exact hwm-value (what it means is described soon). DPSR prefers to process states s from $\mathtt{E}$ among all unprocessed states s' that minimize $\mathtt{hwm}(s')$ among all unprocessed states. When a state $s' \in \mathtt{E}$ is processed, then we know that $\mathtt{hwm}(s)$ of each predecessor state $s \in \mathtt{pred}(s')$ will be updated with the exact hwm-value of s because $\mathtt{hwm}(s')$ is the exact hwm-value of s'. State s is then added to $\mathtt{E}$ and will also be preferred in a next processing step.

We remark, that the preference of states with exact hwm-value is used as tie-breaking among states of same (and possibly inexact) hwm-values in order to recognize states that will be assigned with their exact hwm-values in a late iteration. Without this tie-breaking the algorithm would be correct but would fail to to recognize states that are assigned with their exact hwm-values.

We now can globally maintain a function $\mathtt{hwm}'$ that holds the best known lower bounds on hwm-values of states, and maintain a set $\mathtt{E}'$ that contains the states for which $\mathtt{hwm}'(s)$ is the exact hwm-value. The best known lower bound of each state s is initially $\mathtt{hwm}'(s) = h(s)$. Set $\mathtt{E}'$ initially contains all the goal states. Function $\mathtt{hwm}'$ and set $\mathtt{E}'$ are updated after each application of DPSR. More specifically, after each application of DPSR, which fills $\mathtt{hwm}$ and $\mathtt{E}$, $\mathtt{hwm}'(s)$ is updated with $\max(\mathtt{hwm}'(s), \mathtt{hwm}(s))$ and the states from $\mathtt{E}$ are added to $\mathtt{E}'$.

When we ask for hwm-values of states, then we might be lucky to find them already stored in $\texttt{hwm}'$. However, one must consider that $\texttt{hwm}'$ may contain lower bounds on the hwm-values of some states. Therefore, we can only use the values from $\texttt{hwm}'$ for computing the hwm-value of a set of states S' if $hwm(S')$ is determined by a state s from S' with $s \in \texttt{E}'$. Otherwise, DPSR must be applied to S' in order to get the correct hwm-value of S'

Reuse Information We now introduce a sophisticated variant of DPSR that we denote with DPSR$'$. It uses the information of the function $\texttt{hwm}'$ and the set $\texttt{E}'$ in order to shorten the GBFS runs in the exploration phase. It exploits insights that we got about the behavior of GBFS. Each application of DPSR$'$ will use and refine $\texttt{hwm}'$ and $\texttt{E}'$.

DPSR$'$ has an exploration phase that is similar to that of DPSR. Instead of stopping at a goal state, DPSR$'$ stops a GBFS run when GBFS *selects* a state s with $h(s) = \texttt{hwm}'(s)$ and $s \in \texttt{E}'$. Such a state is an alive surface state and has already been reached in this role in a previous application of DPSR$'$. Likewise to goal states, it is a state from which we can derive the exact hwm-values of other states in the dynamic programming phase.

Instead of just pruning states s with $h(s) > \texttt{b}$, DPSR$'$ additionally prunes states with $\texttt{hwm}'(s) > \texttt{b}$. This is safe because $\texttt{b}$ constitutes an upper bound on the hwm-level on which GBFS is currently searching and s is a dead state on the hwm-level in this case (Section 13.3). As s is a dead state on the hwm-level on which s appears in the current GBFS run, the expansion of s would not improve the lower bound $\texttt{hwm}'(s)$ in the current application of DPSR$'$.

If a state s with $s \in \texttt{E}'$ is generated during the GBFS run, then DPSR$'$ immediately updates $\texttt{b}$ with $\min(\texttt{b}, \texttt{hwm}'(s))$. Lowering the upper bound is safe, since $s \in \texttt{E}'$ implies that $\texttt{hwm}'(s)$ is the exact hwm-value of s, and as s is generated, GBFS is guaranteed to find a path without expanding any state s' with $h(s') > hwm(s)$ during the remainder of the GBFS run.

We define DPSR$'$ to also prune states s with $s \in \texttt{E}$ and $h(s) < \texttt{hwm}'(s)$. Since we already know the exact hwm-value of s, the expansion of s cannot improve $\texttt{hwm}'(s)$. Although the exact hwm-value of s is known, we cannot stop GBFS at this point and only use $\texttt{hwm}'(s)$ to derive the hwm-values of other states during the dynamic programming phase. This is because we do not exactly know on which hwm-level the current GBFS run has found s (we only know that $\texttt{hwm}'(s)$ is an upper bound $\texttt{b}$ on the current hwm-level). To be sure about the hwm-level, GBFS has to continue until it selects a state s with $h(s) = hwm'(s)$ and $s \in \texttt{E}'$. Nevertheless, pruning s is safe because DPSR$'$ updates the upper bound $\texttt{b}$ with $\min(\texttt{b}, \texttt{hwm}'(s))$. The update of $\texttt{b}$ avoids the expansion of states with h-values above $\texttt{hwm}'(s)$ of the pruned state s. Without this bound, GBFS of DPSR$'$ may expand some never expanded states.

After GBFS stops, the pruned states, the possible alive surface state (state s with $h(s) = hwm(s)$), and the open states are added to $\texttt{G}$.

The dynamic programming phase of DPSR$'$ is identical to that of DPSR, except that it

initializes $\texttt{hwm}$ with $\texttt{hwm}(s) = \texttt{hwm}'(s)$ for all $s \in \texttt{G}$ and $\texttt{hwm}(s) = \infty$, otherwise.

We remind that after the application of DPSR′, $\texttt{hwm}'$ and $\texttt{E}'$ are updated with the information gained by DPSR′.

The main advantage of this approach compared to the other approaches is that many states only need to be expanded in a single exploration phase of DPSR′. Nevertheless, the worst-case runtime complexity is still $\mathcal{O}(n \cdot (n \log n + m))$ in the number of states n and number of transitions m from a state space because states s with $h(s) < hwm(s)$ may be expanded in multiple applications of DPSR′, and DPSR′ may be applied to all of the n states from the state space.

15.2. Potential State Space

The potential state space is the part of a given state space that only consists of states that GBFS potentially expands and the transitions between these states. In other words, it contains all the states and transitions that occur in at least one bench from a bench space. The potential state space, the progress states, and $hwm(succ(s))$ of progress states s is all what we need in order to generate bench spaces and crater spaces according to their definitions.

The most straightforward way to compute the potential state space is to follow the definition of benches and bench spaces. We generate one bench after another starting from the initial bench that is induced by the initial state. Since we are only interested in the potential state space and not in the benches, we expand each of the potentially expanded states at most once among all the benches. In this way we start generating each bench but may not completely explore it because the remaining part of a bench might already have been explored in the generation of another bench. Nevertheless, we are able to collect all the potentially expanded states and all the transitions between them.

It is important to first generate benches on the higher hwm-levels before continuing with benches on lower levels because if we first expand a state s on a lower bench and prune s on a higher bench afterwards, then we may also completely prune the state space behind s on the higher bench.

Beside of a state space and a heuristic, the main ingredients for generating the benches are the progress states and $hwm(succ(s))$ of progress states. When we generate a bench $\mathcal{B}(s)$, then we know $level(s)$ of $\mathcal{B}(s)$. During the generation of $\mathcal{B}(s)$, we explore the state space that is reachable from s on a path on which all states have h-value smaller or equal $level(s)$ and test for each encountered state s' whether s' is a progress state. We assume that no information about progress states and high-water marks is precomputed in advance to the generation of a bench $\mathcal{B}(s)$ except of $hwm(succ(s))$ for $level(s)$. Nevertheless, we do not need to compute $hwm(succ(s'))$ for each state s'. Before we compute $hwm(succ(s'))$, we can first test if $h(s') = level(s)$ holds. If it holds then we test if $h(s'') < h(s)$ holds for at least one successor state $s'' \in succ(s')$. If both test were passed, then s' can be a trap state or a progress state. In order to determine whether s' is

a progress state, we must compute $hwm(succ(s'))$ with one of the approaches from Section 15.1. We can support an approach in the computation of hwm-values by reporting $level(s)$ as an upper bound on the hwm-value of s'.

The presented method is sufficient to determine the potentially expanded states and the potential state space. Depending on the approach for computing the high-water mark values, we may not have computed the high-water mark of all the potentially expanded states. Nevertheless, the missing high-water marks can be extracted by running the DP approach on the potential state space. The application of DP may not return the exact hwm-values with respect to the original state space topology, but returns values that are sufficient to determine alive and dead states.

15.3. Topological Structures

Benches (Definition 11.4), bench spaces (Definition 11.5), progress state spaces (Definition 11.6), craters (Definition 12.4), crater spaces (Definition 12.5) and surface state spaces (Definition 12.6) can be extracted according to their definitions. As the structures reflex the search behavior of GBFS they translate to local state-space searches.

The definition of benches simply translates to an exhaustive state-space search that starts in a progress state s, is upper bounded by $hwm(succ(s))$ and bounded by subsequent progress states. A bench space or progress state space can then be generated with an exhaustive state-space search that starts in initial state s_{init} and uses $Progress(s)$ as successor function.

The definition of craters simply translates to an exhaustive state-space search that starts in a surface state s, is strictly upper bounded by $hwm(succ(s))$ if s is a progress state or by $h(s)$ if s is a bench state, and bounded by subsequent surface states. A crater space or surface state space can then be generated with an exhaustive state-space search that starts in the initial state s_{init} and uses $Surface(s)$ as successor function. For generating the crater space or surface state space of a single bench $\mathcal{B}(s)$, one needs to start the exhaustive search in s and bound it with states from $Progress(s)$.

15.4. Properties

We describe the procedure for testing whether progress state spaces and surface state spaces are overlap-free according to Definitions 14.5 and 14.6.

A straightforward approach is to perform a search for each progress state s on a given progress state space and test whether the search reaches another progress state s' and whether $Bench(s)$ and $Bench(s')$ share states. If the benches share states, then the progress state space is not overlap-free. The approach is analogue for surface state spaces.

However, we can save many of these searches by first determining progress states that induce benches which potentially overlap with other benches. For example, when there

are no benches that share states then we do not need to perform any search. This can be tested by collecting for each bench state s the progress states that define the benches on which s appears, i.e., the different bench contexts of s (Section 13.2). Let $\texttt{context}(s)$ denote the set of progress states that a bench state s is associated with. If each bench state is associated with at most one progress state, i.e., $|\texttt{context}(s)| \leq 1$, then we can already say that the progress state space is overlap-free and do not need to perform any further search. Otherwise, we only need to search from progress states that share a bench state in order to determine whether one progress state is reachable from the other one. These progress states can be determined by collecting the all the progress states that appear together in the context set of a bench state $\{s' \in \texttt{context}(s) \mid s \text{ is bench state and } |\texttt{context}(s)| > 1\}$

We can further optimize the procedure. For each progress state s we can determine the progress states for which we want to test the reachability from s. Let us call them *target states*. We crate a set $\texttt{target}(s)$ for each progress state s that contains the target states of s. A progress state s' is in $\texttt{target}(s)$ if s' appears together with s in $\texttt{context}(s'')$ of a bench state s'' and if $h(s) > h(s')$. The latter condition exploits the knowledge that a progress state s can only reach another progress state s' that is on a lower hwm-level than s.

A progress state space is overlap free, when no progress state s reaches a target state from $\texttt{target}(s)$. A small optimization to shorten a search run is to determine the minimum h-value among target states from $\texttt{target}(s)$ and prune progress states with h-value smaller than the minimum value.

The optimizations are analogue for surface state spaces. A small difference is that a surface state can also reach another surface state on the same hwm-level.

15.5. Best-Case Search Run

In this section we present an algorithm that solves the optimization problem of finding a shortest search run from Definition 14.1. We define a meta search space that allows us to search for a best-case search run with uniform cost search. We first describe the basic algorithm and then discuss possible improvements.

Let $\mathcal{F}(\mathcal{T})$ be a surface state space with surface states S, initial state s_{init}, successor function $succ$ and set of goal states S_{goal}. Let $Crater(s)$ be the set of crater states that is given for each surface state $s \in S$.

Assume a surface state s is reached from s_{init} on a surface state path ρ. A node is a tuple $\langle s, D\rangle$ with s is the surface state and D is a set consisting of all crater states that appear in a crater $Crater(s')$ of a state s' on ρ. The initial node is $\langle s_{\text{init}}, Crater(s_{\text{init}})\rangle$. A successor node of $\langle s, D\rangle$ is defined for a state $s' \in succ(s)$ and is $\langle s', D \cup Crater(s')\rangle$. A goal node is a node $\langle s, D\rangle$ with $s \in S_{\text{goal}}$. The cost of a transition between two nodes is defined as $cost(\langle s, D\rangle, \langle s', D'\rangle) = 1 + |Crater(s') \setminus D|$. The set D ensures that crater states are counted at most once along a surface state path.

This instance spans a space of search nodes in which an uniform cost search can find a

cheapest surface state path. The node path represents a surface state path and the surface state path can then be translated to a search run as described in Section 14.1.

The size of the node space compared to the underlying surface state space depends on set D. The size and diversity of these sets among different nodes depend on how craters overlap. Overlapping craters that are situated far from each other on a surface state space make the search for a cheapest path on the surface state space an NP-complete problem (Section 14.3).

Special Cases There are search space topologies for which the best case is polynomial-time computable (Section 14.4). Those topologies are either undirected or induce overlap-free surface state spaces. As there are no global dependencies between craters in overlap-free surface state spaces, a best-case search run can be found with an uniform cost search that directly runs on the surface state space and uses cost function $cost(s, s') = 1 + |Crater(s') \setminus Crater(s)|$ for undirected instances and $cost(s, s') = 1 + |Crater(s')|$ for overlap-free instances.

Surface State Path An uniform cost search can save some of the node generations. A node $\langle s, D\rangle$ is based on the surface state path ρ on which s was reached. The uniform cost search ignores the fact that ρ is cycle-free for a shortest run (it only ensures that the node path is cycle-free). Therefore, we prune nodes that do not contribute to finding a shortest search run because of cycles on the surface state path.

Dead States A best-case search run can only be represented by a single surface state solution path from a surface state space. However, a surface state space can contain dead surface states which do not contribute to a best-case search run but, fortunately, can easily be detected based on high-water marks. Therefore, we prune nodes that contain a dead surface state, i.e., a state s with $h(s) < hwm(s)$.

Shared States Apart from the hardness of computing the best case, the current definition of search nodes might induce some accidental complexity that leads to an unnecessary blow up of the search node space. Currently, a node contains all the crater states that the uniform cost search has encountered along the path to the node. However, the set D of a node only needs to maintain states that can appear again in a crater that is reachable from the current node. Therefore, a node $\langle s, D\rangle$ only needs to maintain crater states with h-values smaller than $level(s)$. Consequently, each time a node $\langle s', D'\rangle$ is generated at the expansion of node $\langle s, D\rangle$ whereas s' is a progress state, we do not copy states s'' from D to D'' with $h(s'') \geq level(s')$ (s'' with $h(s'') > level(s')$ is never expanded and s'' with $h(s'') = level(s')$ is not in a crater in the search future).

We may be able to keep nodes free from many more states that are not shared among any craters along a surface state path. This requires either more advanced preprocessing of the surface state space in advance to uniform cost search or requires lookahead in the

surface state space during uniform cost search. However, the additional memory or time that is required to perform preprocessing or lookahead may not pay off. We leave the development and analysis of such advanced techniques to future work.

15.6. Worst-Case Search Run

The algorithm for computing a worst-case search run is in many ways similar to that of a best-case search run. We replace surface state space with progress state space, surface states with progress states, $Crater(s)$ with $Progress(s)$, and apply longest path search for directed acyclic graphs instead of uniform cost search.

The optimization for the best-case algorithm that are based on surface state paths do not apply to the worst-case algorithm because paths in the progress state space are acyclic. Having an acyclic graph is a necessary condition for the applicability of longest path search.

The optimization that reduces the number of states in D of a search node $\langle s, D \rangle$ does also apply to the worst-case algorithm. There is no need to maintain bench states s' with $h(s') > level(s)$ in D. The small difference to the best-case algorithm is that the worst case algorithm needs to keep bench states s' with $h(s') = level(s)$.

The longest path search starts with the initial node and uses $level(s)$ (or $h(s)$, which is also suitable) as a topological ordering of nodes $\langle s, D \rangle$, i.e. the search first processes nodes with high values before continuing with nodes of lower values. When the longest path search has processed all nodes, then it returns the most expensive node path, which translates to a progress state path, which in return translates to a worst-case search run as described in Section 14.1.

Special Cases In the special cases, the longest path search runs directly on the progress state space and uses the cost function $cost(s, s') = 1 + |Progress(s') \setminus Progress(s)|$ for undirected instances and $cost(s, s') = 1 + |Progress(s')|$ for overlap-free instances.

16. Experimental Results

In this chapter, we show that it is feasible to extract the information about the search behavior of GBFS from practically relevant state spaces that are evaluated with a state of the art heuristic from classical planning.

As a benchmark set we use planning tasks from the International Planning Competitions (IPC) 1998-2018 and the Fast Forward heuristic (Hoffmann and Nebel, 2001) with unit cost. The 3903 tasks from the benchmark set are formalized in PDDL (Fox and Long, 2003) and cover 56 different planning domains. For simplicity, we often call planning tasks, the resulting state space topologies and the other structures that result form the topologies instances.

We implemented our algorithms into the Fast Downward planning system (Helmert, 2006). Fast Downward translates a planning task into an initial state, a successor generator and a goal test function and evaluates states with the FF heuristic. We used the experiment framework Downward Lab (Seipp et al., 2017) to set up the experiments. The experiments ran on a cluster that consists of Intel Xeon Silver 4114 processors (2.2 GHz). We restricted the experiments to spend at most 3.5 GiB RAM. We used different time limits for the different steps in the process of computing the information about search behavior. We report the limits in the corresponding sections.

16.1. Evaluation of Tractability

We analyze the feasibility of computing the potential state space, the progress and surface state spaces, their properties and the best-case and worst-case search runs.

Potential State Spaces We start with an evaluation of the different approaches for computing the high-water marks from Section 15.1 that are necessary for the generation of potential state spaces. In particular, we applied DP$'$, SR$'$ and DPSR$'$.

DP$'$ first computes the high-water marks of all states with h-value less or equal the hwm-value of the initial state, and then the potential state space is generated. SR$'$ and DPSR$'$ compute the high-water marks of states during the generation process of a potential state space. Table 16.1 shows the number of instances for which the approaches were able to compute the potential state space of an instance within a 30 minute time limit. As expected DP$'$ covers fewer instances than the other approaches. This is because DP$'$ determines the hwm-values of many more states than necessary for the generation of the potential state space, while SR$'$ and DPSR$'$ only generate the potentially expanded

	DP′	SR′	DPSR′
coverage (2936)	832	1160	1320

Table 16.1: Coverage of different methods that compute the potential state space.

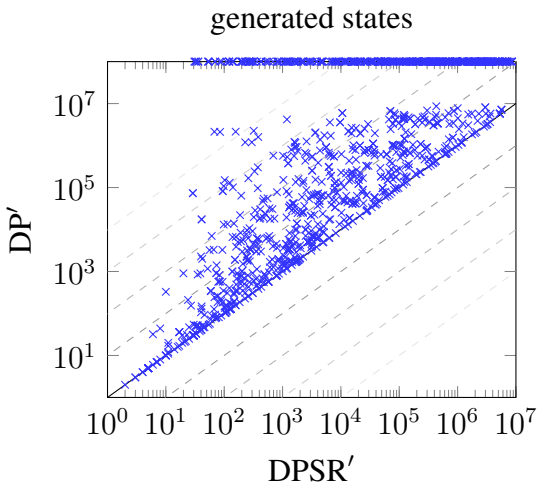


Figure 16.1: Comparison of number of states generated by DP′ versus the number of states generated by DPSR′.

states and their successor states. Figure 16.1 demonstrates that DP′ can generate up to several orders of magnitude more states than DPSR′. SR′ is not shown because it generates as many states as DPSR′. DPSR′ does not only saves space but also saves many time-intensive state evaluations of the heuristic. DP′ mainly fails because of out of memory (in 63% of the cases) and SR′ and DPSR′ mainly fail because of timeout (in 98%, respectively 73% of the cases). We also expected that DPSR′ covers more instances than SR′. However, the coverage only increased by 13%. The best approach, DPSR′, were able to determine the potential state space of 45% of the instances of which GBFS solved each one within 30 minutes. Can we further improve the coverage? Looking at Figure 16.2, the answer is rather negative because the potential state spaces often contain a number of states that is exponential in the number of states expanded in GBFS runs.

The covered 1320 instances provide the basis for computing the surface state spaces and progress state spaces. The potential state spaces consist of identifiers (integers) of states that replace the original states. The original states can be large due to possibly many state variables. We remove the original states in order to free memory for the computations that follow. Moreover, we make h-values of states explicit. Explicitly storing the h-values instead of recomputing them saves time because the computation of h-values can be a dominant factor. For the computations that follow, we set a time limit of 90 minutes for each instance. Within this time, the potential state space is computed and is expected to

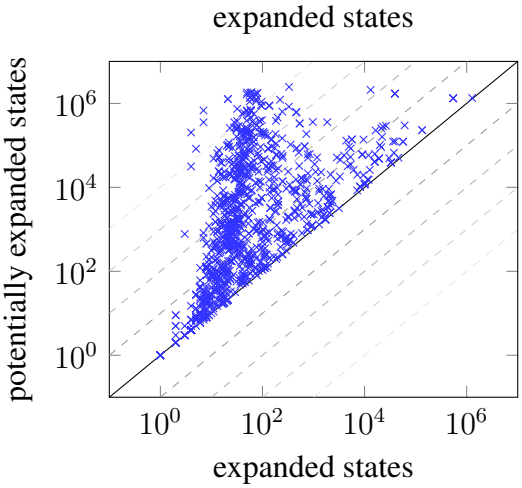


Figure 16.2: Number of states expanded by a GBFS run versus number of potentially expanded states.

require at most 30 minutes; leaving at least 60 minutes to the remaining computations which include the generation of the surface or progress state space, the extraction of properties, and the search for a best-case or worst-case search run.

Surface State Spaces and Progress State Spaces We now analyze the generation of surface state spaces and progress state spaces. Determining these spaces requires the generation of each crater, respectively bench. As each state can be on several craters or benches, each state may be expanded multiple times during the generation process. Table 16.2 shows that we were able to derive 1277 surface state spaces out of 1320 potential state spaces. The main reason for failing computations is shortage of memory (in 90% of the cases).

A plausible explanation for this phenomenon is that a surface state space may contain many more transitions than the potential state space. Assume a crater with a single crater state and n non-progress surface states. Moreover, it has undirected transitions between the crater state and the surface states but no transitions between surface states. It means that the crater has n transitions. Since the single crater state is connected with each surface state, the subspace in the resulting surface state space, that only consists of the surface states from the crater, will form a complete graph with $\frac{n(n-1)}{2}$ edges. Consequently, the worst-case space complexity is $\mathcal{O}(n^2)$. The reasons for failing computations of progress state spaces is similar to those of surface state spaces. Benches that overlap in a state cannot result in a complete graph in the progress state space. Nevertheless, we can infer a worst-case space complexity of $\mathcal{O}(n^2)$ by creating an example for progress state spaces that can be constructed analogously to the surface state space example from above.

The shortage of memory also affects the computation of progress state spaces but to a

		overlap-free	undirected	other	total
	total instances				3903
	instances solved by GBFS				2936
	potential state spaces				1320
best	surface state spaces				1277
	properties	689	97	460	1246
	solved	689	96	396	1181
worst	progress state spaces				1281
	properties	740	74	436	1250
	solved	733	70	399	1202

Table 16.2: Coverage of instances at different steps of the computation pipeline.

lesser extent than for surface state spaces (memory out in 70% of the cases). We were able to derive 1281 progress state spaces out of 1320 potential state spaces. Figure 16.3 confirms the possible quadratic increase of transitions when deriving a surface or progress state space from a potential state space. Crosses below the line are instances where the derived state space has more transitions than the potential state space. Not surprisingly, it is especially the case for undirected instances.

Properties We now analyze the extraction of the properties. The extraction process includes the recognition of whether a potential state space is undirected and whether a surface state space or progress state space is overlap-free along paths. Table 16.2 shows that we were able to derive the properties of 1246 surface state spaces out of 1277. Again, the main reason for failing computations is shortage of memory (in 90% of the cases). Testing whether a potential state space is undirected does not affect memory and do not need to be discussed further. Determining whether a surface state space is overlap-free can consume a lot of memory. Our approach for determining the overlap requires the explicit generation of all the craters and associates each crater state with the trap state that entered the crater. This approach has a worst-case space complexity of $\mathcal{O}(n^2)$ because each crater state can be associated with at most all the n surface states from a surface state space. The result for the computation of properties of progress state spaces is similar to that of surface state spaces, except that the shortage of memory is as critical as the shortage of time.

We investigated the space and time requirements of the given instances by comparing the number of crater (bench) states to the number of their appearances on different craters (benches). Figure 16.4 visualizes the result for bench spaces and crater spaces. In both kind of spaces the blowup is clearly visible. The blowup is larger for instances that stem from undirected state spaces. A possible direction for reducing the memory and time overhead of crater spaces and bench spaces is to design special data structures and

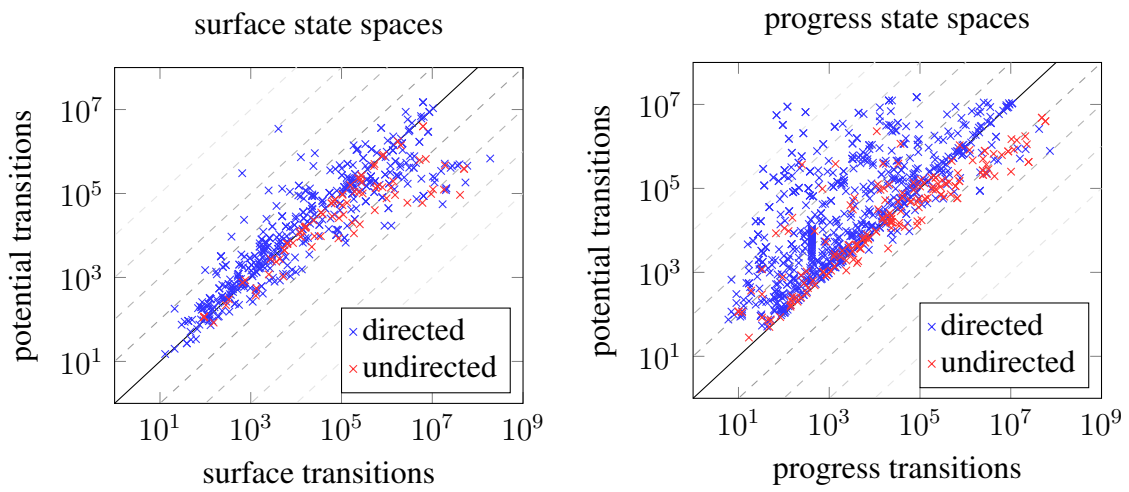


Figure 16.3: Comparison of number of transitions in potential state spaces versus the number of transitions in the surface state spaces (left) or progress state spaces (right).

algorithms for undirected state spaces by exploiting strongly connected components.

Best-Case and Worst-Case Tie-Breaking We now analyze the computation of the best-case and worst-case search runs. Even though the percentage of instances for which we could perform the search run analysis is high in general, one can see that the success rate is much higher for the polynomial special cases. We have results for the best case for all overlap- free instances and for 99% of the undirected instances. For the worst case, we have results for 99% of the overlap-free instances and 95% of the undirected instances.

The NP-hardness of the general problem can be observed for the numbers on the remaining instances. For the best case we have results for 86% and for the worst case we have results for 92% of the instances. The possibly exponential blow-up seems to be more present when computing the best case than when computing the worst case. To support this hypothesis, Figure 16.5 plots the number of generated surface states (best case) or progress states (worst case) in comparison to the number of generates search nodes for each instance that is neither undirected nor overlap-free. For the worst case, the blow-up is almost always significantly below an order of magnitude, while it is usually close to one and up to more than three orders of magnitude for the best case.

The main reason for failing computations is shortage of memory (in 80% of the instances for the best case and in 73% of the instances for the worst case). Since memory seems to be the main bottleneck, it could be worth to invest time into a sophisticated method for determining shared states. In the current implementations of the algorithms for the best case and worst case, the set of crater and bench states is bound by the hwm-

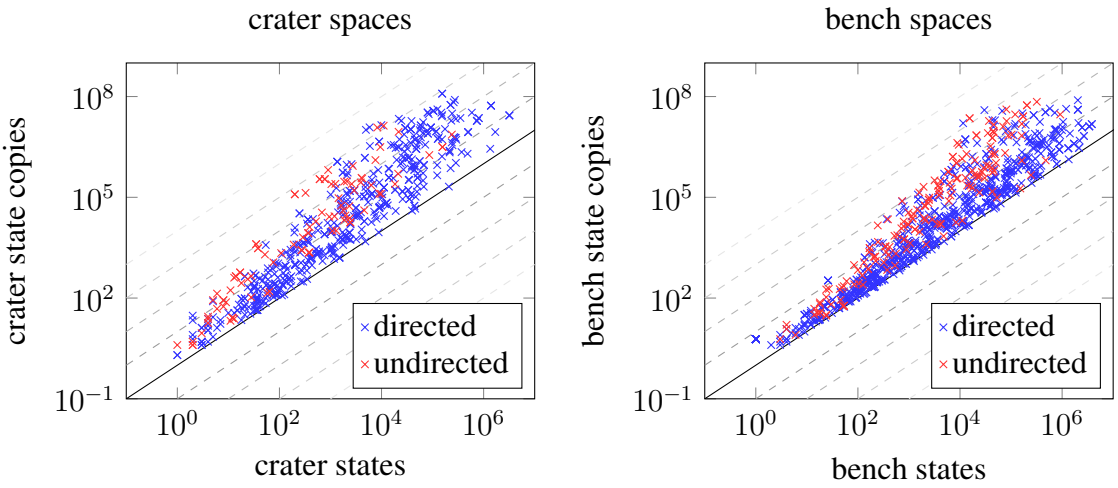


Figure 16.4: Comparison of number of states versus number of copies of the states on different crater spaces (left) and bench spaces (right).

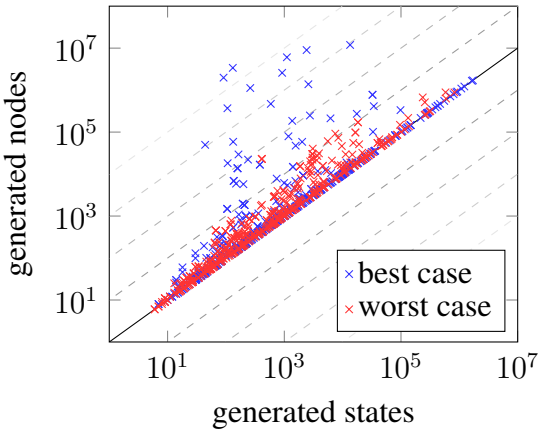


Figure 16.5: Number of states versus number of search nodes that were generated during a search for a best-case search run, respectively worst-case search run, for different instances.

level of the surface state, respectively progress state. However, many of the crater and bench states might never appear again in an upcoming crater or bench. We leave the development and analysis of such a method as future work.

From all 2936 instances that could be solved by GBFS we were able to extract the best case from 1182 (40%), the worst case from 1202 (41%) and both cases from 1118 (38%) instances. The main bottleneck seems to be the computation of potential state spaces, since we were able to extract these spaces from 45% of the instances. However, we demonstrated that this is due to the size of potential state spaces, which is exponential in the number of states that are usually expanded in a GBFS run.

16.2. Evaluation of Tie-Breaking Policies

We now compare the performance of well-known tie-breaking policies to the best case and worst case. The three policies are: the first-in-first-out policy π^{fifo}, the last-in-first-out policy π^{lifo} and the random policy π^{rand} as defined in Section 4.2. We compare the number of states that GBFS expands with each policy. For the random policy we take the average over 10 runs.

Figure 16.6 shows that the influence of tie-breaking is highly significant in practice because we see a large gap between the best cases and worst cases. Figure 16.7 (left) clarifies that the worst case can expand several orders of magnitude more states than the best case in an instance. Despite of the significance of tie-breaking in many instances, there are some instances with craters that require a lot of state expansions relatively independently of tie-breaking, i.e., the best case and worst case are similar for these instances. A possible reason for this behavior is the presence of huge craters that cannot be avoided by GBFS. We will discuss this hypothesis in more detail in Section 16.4.

The differences between π^{fifo}, π^{lifo} and π^{rand} are negligible when averaging over all instances. Therefore, we take π^{fifo} as a representative of the three algorithms. π^{fifo} is not far from the best case in crater-free state space topologies. Figure 16.7 (right) shows that GBFS with π^{fifo} is often close to the best case in these instances. We will discuss plausible reasons in Section 16.4. In state space topologies with craters, the picture is more diverse. Figure 16.7 (left) shows that GBFS with π^{fifo} occasionally expands up to three orders of magnitude more states than in its best case, but often expands one to ten times more states.

On the positive side, the results show that especially in instances with craters, there is potential for the development of better tie-breaking strategies or policies. There exist different strategies in literature (e.g. Asai and Fukunaga, 2017a; Lipovetzky and Geffner, 2017). We expect that averaged over all instances, they behave similarly to the standard tie-breaking policies because they are not explicitly tailored towards avoiding craters. The analysis of these strategies remains future work. On the negative side, the results indicate that for some instances, on which GBFS expands a huge number of states, tie-breaking has a marginal effect on the number of expanded states.

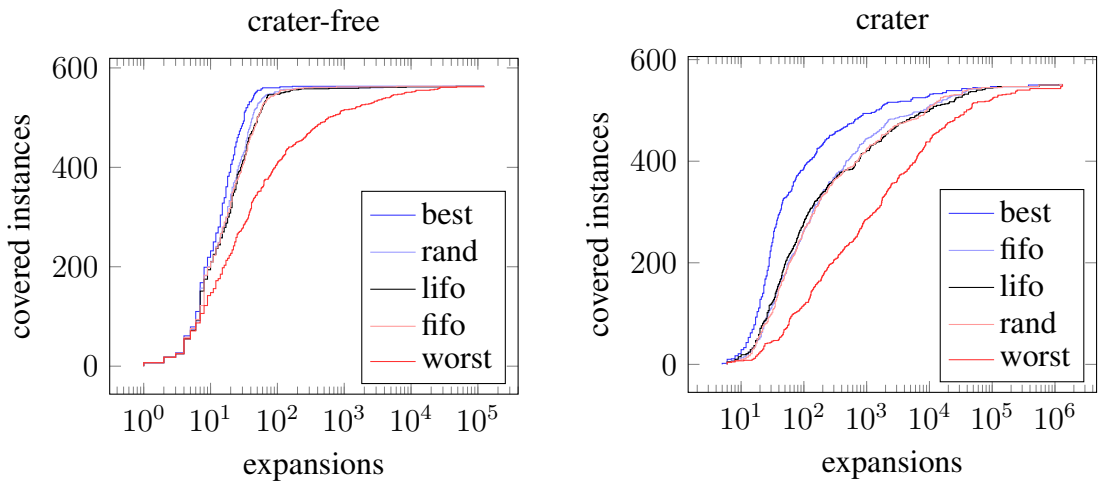


Figure 16.6: Coverage of instances where GBFS expands at most a given number of states for different tie-breaking policies. The results are shown separately for crater-free instances (left) and instances with craters (right).

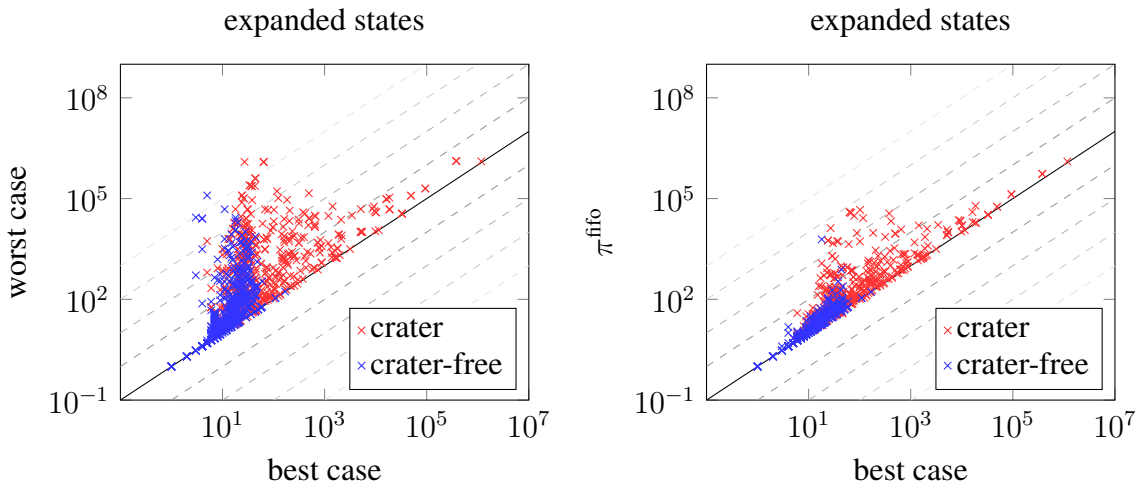


Figure 16.7: Number of expansions of the best-case tie-breaking policy versus number of expansions of the worst-case tie-breaking policy (left) and π^{fifo} tie-breaking policy (right).

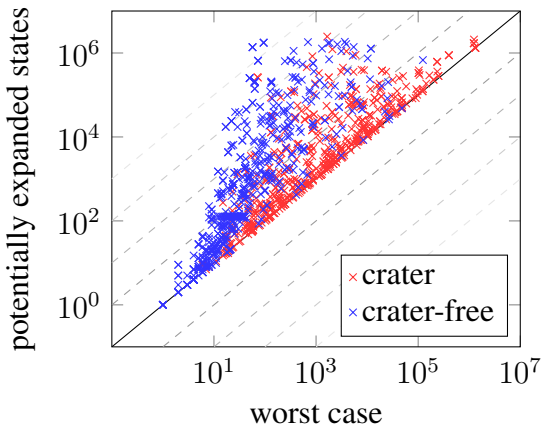


Figure 16.8: Number of states expanded by a worst-case tie-breaking policy vs. number of potentially expanded states.

16.3. Worst Case and Potentially Expanded States

In this section, we are interested in the relationship between the number of potentially expanded states and the number of states expanded under a worst-case search run, which is shown in Figure 16.8 for instances with or without craters. In the presence of craters, both numbers are often similar, i.e., almost all states that are expanded by some tie-breaking policy are expanded in the worst case. A possible explanation for this behavior is that the expansion of a large portion of potentially expanded states can often not be avoided by GBFS. The picture looks different for crater-free instances, where a worst-case tie-breaking typically expands a fraction of potentially expanded states. This is the behavior that we expect from a well informed heuristic. In the next section, we will be able to connect these results with the shapes of bench spaces and the presence of craters.

16.4. Analysis of State Space Topologies

The previous sections revealed different characteristics in the behavior of GBFS when searching in instances with craters or without craters. In this section, we make a first attempt to connect the characteristic behavior of GBFS to the structure of the underlying state space topology. We interpret different plots and discuss the shape of different spaces and the expected behavior of GBFS in these spaces.

Types of Potential States What is the relation between surface states and crater states in crater spaces and between progress states and bench states in bench spaces? Figure 16.9 visualizes these relations. It only shows data for instances that have crater, respectively

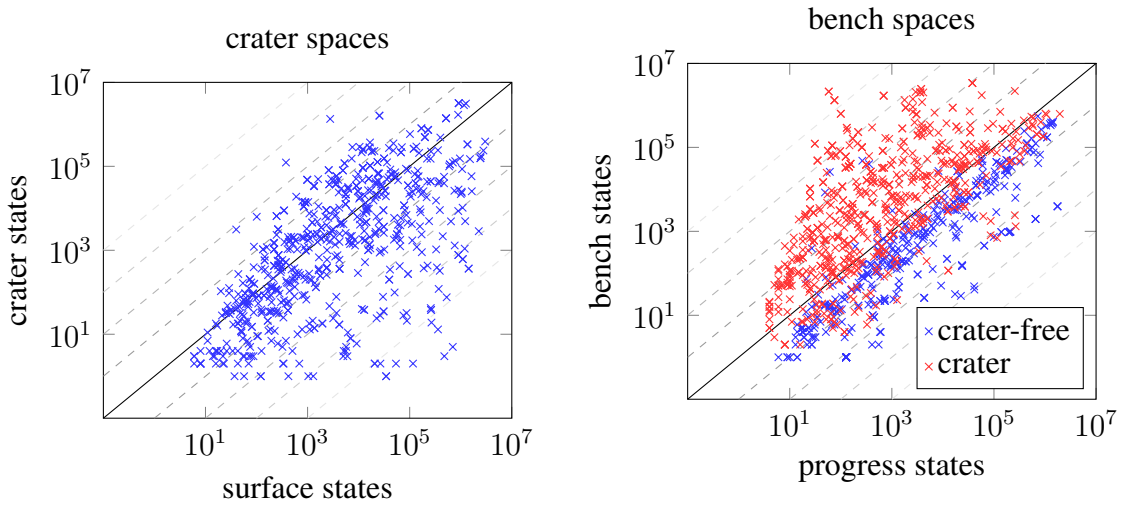


Figure 16.9: Comparison of number of surface states versus number of crater states of different crater spaces (left) and number of progress states versus number of bench states of different bench spaces (right).

benches. For crater spaces the picture looks mixed. It shows that the number of crater states and the number of surface states are similar in average. For bench spaces there are more bench states than progress states in average. When we separate crater-free instances from instances with craters, we observe that for crater-free instances there are typically one to ten times more progress states than bench states. Very few instances have significantly more bench states than progress states and vice versa. For instances with craters the result is again diverse, but typically there are significantly more bench states than progress states and there are rarely fewer bench states than crater states. We can see that in the presence of craters bench states are more dominant than progress state. A possible explanation is that most of the bench states are actually crater states.

Maximum Size of Craters and Benches Figure 16.9 only reveals the proportion of different types of states from instances. However, the shapes of the spaces from the instances remain unclear. To shed more light on the spaces we ask following question: What is the relation between the size of benches and the total number of bench states in bench spaces, and what is the relation between the size of craters and the total number of crater states in crater spaces? Figure 16.10 visualizes these relations. It shows that the largest crater in a crater space typically contains 10% to 100% of all the crater states. Although there are some exceptions, the picture is very similar when comparing the largest bench to the number of bench states in bench spaces that stem from instances with craters. This leads us to the question whether the size of the largest bench relates to the size of the largest crater in instances. Obviously, the number of states in a largest bench must be at least

the number of states from a largest crater. The picture shows that the largest bench is often considerably less than ten times larger than the largest crater. We now have a look at the bench spaces that are crater-free. In these spaces the largest bench often contains a fraction of all bench states. There are few exceptions where the largest bench contains almost all of the bench states. This pattern looks similar to that of Figure 16.8, where we compared the worst-case number of expansions against the number of potentially expanded states. This similarity strengthens our assumption that huge craters and benches are the reason for this observed behavior.

Types of Surface States We now have a clearer picture of the shape of spaces. The next question we ask is: How many surface states are trap states or progress states and how many progress states lead to non-empty benches. Figure 16.11 shows that often many of the surface states are trap states. Nevertheless, for some instances the portion of trap states can be rather small. The portion of progress states among surface states is mixed. It can as small as 0.001% but can also cover all the surface states. Surprisingly, in crater-free instances a huge amount of surface states are progress states. It is a strong indicator that the FF heuristic is well informed in crater-free instances. Now we compare the number of progress states that induce non-empty benches to the total number of progress states. Typically 2% to 100% of the progress states induce non-empty benches. For instances without craters the percentage is almost always below 30%. We can conclude that empty benches are more common in crater-free instances than in instances with craters.

Overall, we have learned that the shapes of different kind of spaces that result from planning tasks evaluated with the FF heuristic can be diverse. Nevertheless, we observed some patterns when splitting the instances into a set of crater-free instances and into a set of set of instances with craters. We have discovered that the FF heuristic is often almost perfectly informed when it does not induce craters in state spaces from planning tasks. When it does induce craters, then the craters can cover a large region of the potential state space. Then it is difficult for GBFS to avoid craters.

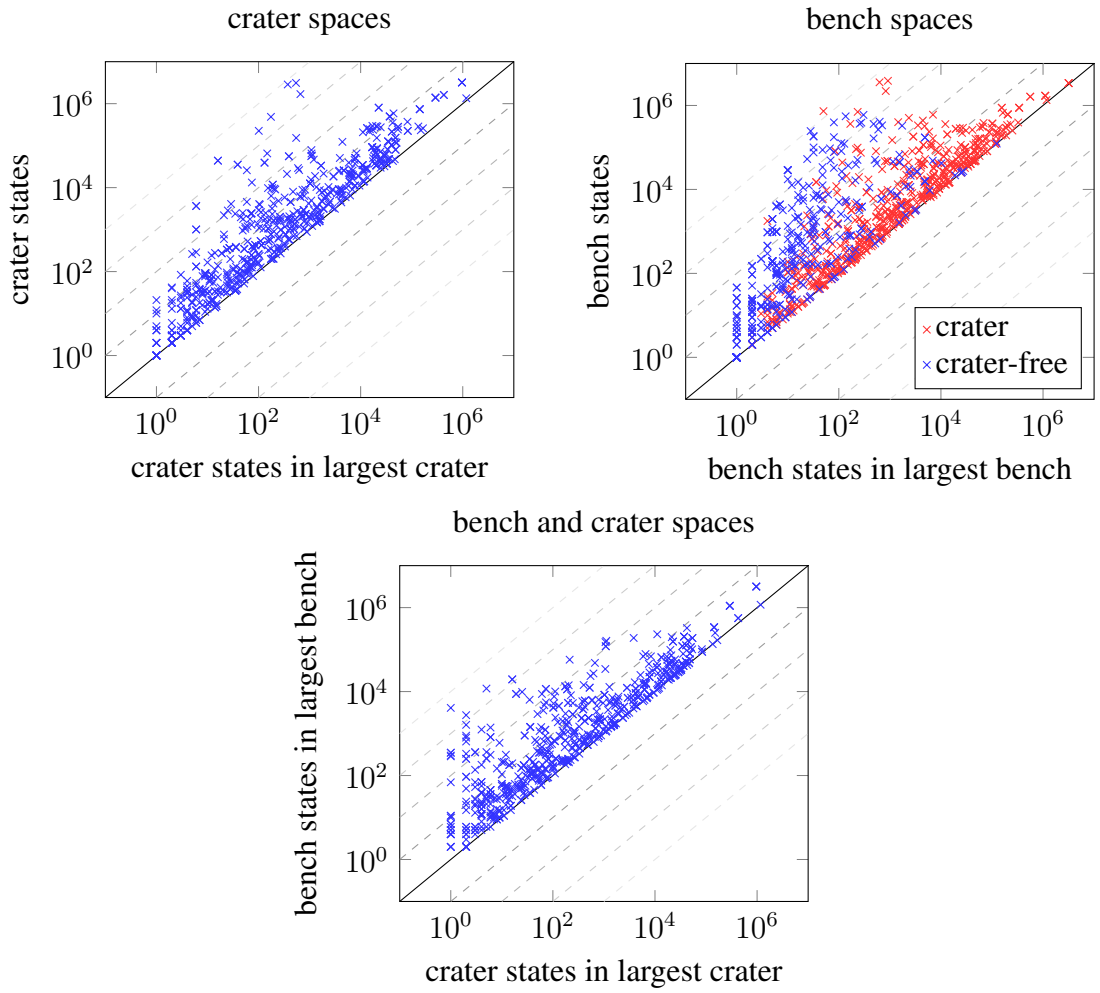


Figure 16.10: Comparison of maximum sizes of craters and benches relative to each other (lower) and relative to the number of total crater states (upper left) and bench states (upper right).

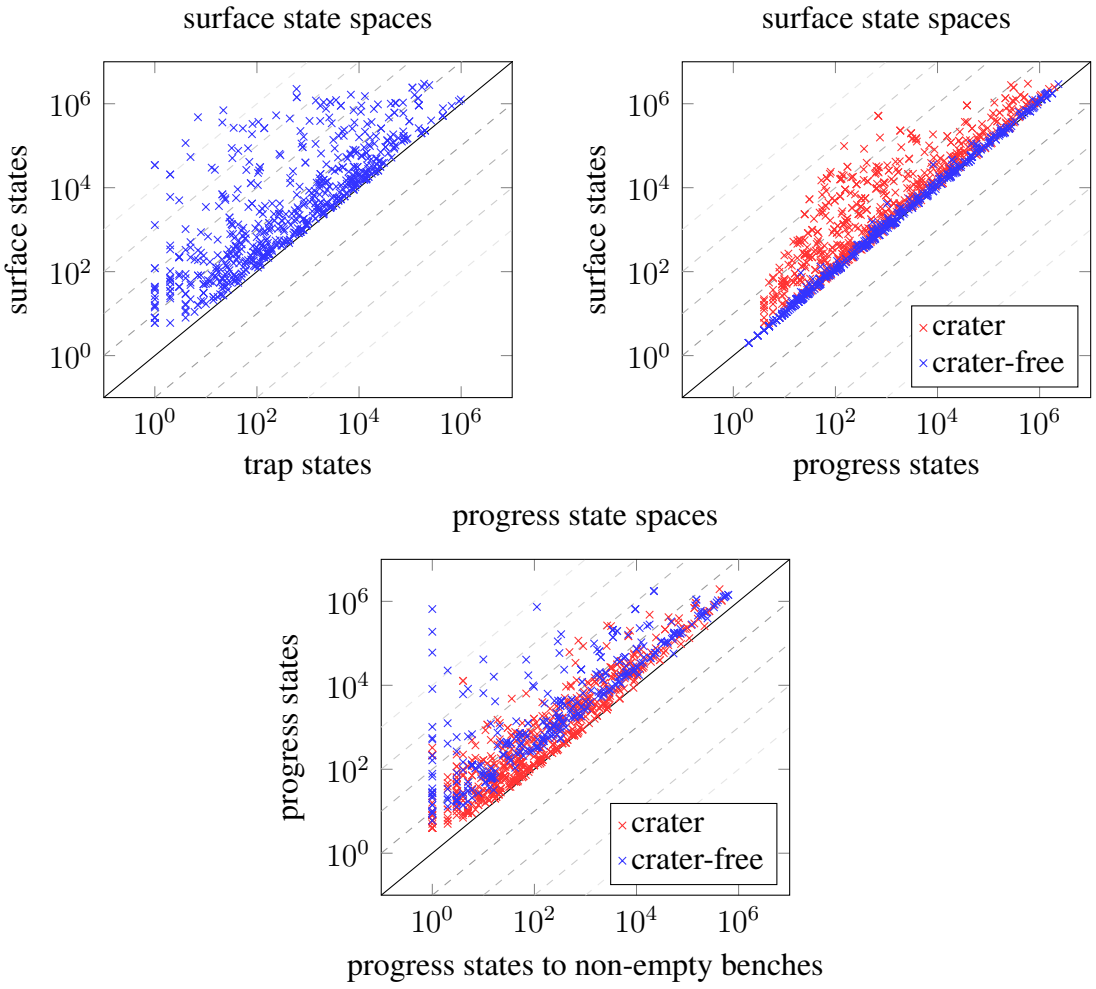


Figure 16.11: Frequencies of different states in surface state spaces and progress state spaces.

Part IV.

Conclusion

17. Future Work

We believe that this thesis provides opportunities for further theoretical analyses and practical applications. In the following, we present some interesting avenues for further research.

Uncovering State Spaces and Heuristics In the analysis of GBFS, we ignored the inner structure of states and the computation of heuristic values because considering this information as opaque is sufficient to explain the behavior of GBFS. A next step is to involve this information when analyzing craters and benches of specific search domains. In this way we can identify strength and weaknesses of particular heuristics in different search domains. The findings could then be used to improve heuristics for GBFS. Wilt and Ruml (2015) analyzed a family of heuristics and presented a technique to automatically build a good heuristic for GBFS. Fickert and Hoffmann (2017) applied a technique to GBFS that refines a heuristic during a search run. Our notion of benches and craters could help to further improve these techniques.

On the theoretical side, we could ask how complex a heuristic must be in order to be free of crater states or even free of bench states, or to guarantee some degree of informativeness, e.g., by bounding the size of craters or benches. It is analogous to Seipp et al. (2016) who asked about the complexity of features that a heuristic must consider in order to guide satisficing state-space search algorithms directly to the goal. Our theory allows us to replace "directly" with weaker conditions and to ask the same question under the weaker conditions.

On the side of tie-breaking, we could investigate the characteristics of best-case tie-breaking policies in a search domain like Corrêa, Pereira, and Ritt (2018) did it for optimal search. We could ask whether a best-case policy visits craters and whether there exists a policy that does not fall into craters. We could try to learn the characteristic of trap states and use the findings to avoid their expansion during a search. Similarly, we could extract the characteristic of progress states that can then serve as sub-goals during a search.

Another direction is to find lower and upper bounds on the expected cost of a solution path that GBFS with a given heuristic will find for a search problem by analyzing the connection between problem descriptions and the induced state space topologies. Moreover, it might be possible to derive run time bounds for GBFS. The goal is to automatically extract these bounds from a problem description. For example, Hoffmann (2005, 2011) found a property of search topologies that can be automatically extracted from a problem description and that relates to the effectiveness of enforced hill-climbing search on the

problem.

Understanding the Search Behavior of Bounded Suboptimal Search This thesis answers theoretical questions for GBFS that have already been fully understood for A* with consistent and admissible heuristic. Both algorithms are in the family of best-first search algorithms. As we now understand these questions for A* and GBFS, we believe that we have almost all knowledge at hand to answer these questions for the bounded suboptimal algorithms in this family as well, starting with Weighted A* (Pohl, 1970).

Analysis of GBFS-Style Algorithms A possible avenue is to use the concepts of our analysis to explicitly design state space topologies with varying structural properties, which could then help to develop a better understanding of the advantages and disadvantages of GBFS-variants that use different exploration and diversification techniques (e.g. Imai and Kishimoto, 2011; Xie et al., 2014; Xie, Müller, and Holte, 2014; Valenzano et al., 2014; Asai and Fukunaga, 2017a; Cohen and Beck, 2018).

Analysis of Tie-Breaking Strategies We analyzed standard tie-breaking policies with respect to the best-case and worst-case tie-breaking policies of GBFS under a given heuristic. There is a bunch of other tie-breaking strategies that are worth to be analyzed (e.g. Asai and Fukunaga, 2017a; Lipovetzky and Geffner, 2017).

Quality and Robustness of Heuristics The number of states that GBFS potentially expands and the numbers of states expansions in best-case and worst-case search runs of GBFS represent the character of heuristics in a given state space. A next step is to develop a metric that represents the quality and robustness of heuristics in context of GBFS. For example, higher admissible heuristic estimates are used to compare the dominance of heuristics in A*.

Exploiting the Episodic Behavior of GBFS We have discovered that GBFS searches in episodes and ignores all open states from the previous episodes. We also know that states above the current level will never be expanded for the rest of a GBFS run. Therefore, we can repeatedly erase a bunch of states and release memory during a GBFS search. In case progress states can be characterized based on a problem description, we could directly exploit these facts by removing the open states upon expansion of a progress state. Another way to exploit them is inspired by Iterative Deepening A* (Korf, 1985). Let us assume that the given state space topology has only craters with depth of at most d. Let l be the current all time low heuristic value. Then we continuously remove all states s with $h(s) > d + l$ (except the states along a possible solution path) during a GBFS run. If the assumption about the depth is true, then GBFS will reach a goal state. Otherwise, it will fail and we must restart GBFS assuming a larger maximum depth d.

18. Conclusion

We demonstrated that a better understanding of state-space search algorithms can be obtained by asking a few basic questions: which states does an algorithm never, necessarily or potentially expand, and when does it make progress? We first summarized the existing answers to these questions asked to A^* and emphasized their importance for many theoretical and practical results that followed up.

Some of the existing theoretical results for A^* and GBFS are based on a concept that is called high-water mark. We discovered that high-water mark is the key to a better understanding of GBFS. We identified the critical role of progress states in GBFS, which are states whose high-water mark exceeds the high-water mark of their successors. Whenever GBFS expands a progress state, all other open states become ineligible for expansion forever: the search effectively behaves as if the open list were cleared with the progress state as a new initial state. It follows that greedy best-first search is episodic in nature, with progress states separating different search episodes. High-water mark benches capture the behavior of one episode, and the bench space captures episode sequencing. Together, they allow us to exactly characterize the states expanded by GBFS under any tie-breaking policy. We also discovered the critical role of surface states for tie-breaking and the main course of a GBFS run. The expansion of a surface state can lead to a phase where GBFS is trapped in a crater or to a next episode. Crater spaces capture the connections between surface states and provide a high-level view on the behavior of GBFS under different tie-breaking policies. Craters allow us to characterize states that are necessary expanded under some conditions.

We showed that the size of bench spaces and crater spaces can be quadratic in the number of states from a state space because benches and craters can overlap. These spaces allow us to characterize the behavior of GBFS in its best case and worst case and to compute actual best-case and worst-case runs. We analyzed the problem of computing these cases and showed that the problem is NP-complete for the general case because of global dependencies between overlapping benches and craters. We showed that these dependencies are not present in overlap-free instances and only arise locally in undirected state spaces.

We implemented algorithms for extracting the different decompositions and properties of search instances and for computing best-case and worst-case runs. Despite the distracting complexity results, we demonstrated that it is feasible to analyze the behavior of GBFS in state spaces of interesting search domains under heuristics that are used in practice. For example, we discovered that the state-of-the-art heuristic for satisficing planning, FF, is well informed if it does not induce craters in state spaces. But when it

does, then craters can become huge and cover most of the area of benches. We were able to demonstrate that tie-breaking is especially important for instances with craters and that there is potential for the development of sophisticated tie-breaking strategies that pay attention to craters because the best-case tie-breaking requires up to 1000 times fewer state expansions than standard tie-breaking policies. Nevertheless, there also exist instances for which the performance of GBFS can only be improved by minimizing huge craters (e.g. by refining heuristics) because GBFS is not able to avoid these craters under any tie-breaking policy.

We believe that our theoretical results about the behavior of GBFS will help to improve existing heuristics for different search domains, will constitute the basis for providing run time guarantees for GBFS on given search instances, and will support the analysis of GBFS-style algorithms and best-first search algorithms in general.

Bibliography

Asai, M., and Fukunaga, A. 2017a. Exploration among and within plateaus in greedy best-first search. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 11–19. AAAI Press.

Asai, M., and Fukunaga, A. 2017b. Tie-breaking strategies for cost-optimal best first search. *Journal of Artificial Intelligence Research* 58:67–121.

Bagchi, A., and Mahanti, A. 1983. Search algorithms under different kinds of heuristics — a comparative study. *Journal of the ACM* 30:1–21.

Bellman, R. E. 1957. *Dynamic Programming*. Princeton University Press.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Cohen, E., and Beck, J. C. 2018. Local minima, heavy tails, and search effort for GBFS. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 4708–4714. IJCAI.

Corrêa, A. B.; Pereira, A. G.; and Ritt, M. 2018. Analyzing tie-breaking strategies for the A* algorithm. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 4715–4721. IJCAI.

Cserna, B.; Doyle, W. J.; Ramsdell, J. S.; and Ruml, W. 2018. Avoiding dead ends in real-time heuristic search. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 1306–1313. AAAI Press.

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

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

Doran, J. E., and Michie, D. 1966. Experiments with the graph traverser program. *Proceedings of the Royal Society A* 294:235–259.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In Kaelbling, L. P., and Saffiotti, A., eds., *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 103–108. Professional Book Center.

Felner, A. 2018. Position paper: Using early goal test in A*. In Bulitko, V., and Storandt, S., eds., *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*, 153–157. AAAI Press.

Fickert, M., and Hoffmann, J. 2017. Complete local search: Boosting hill-climbing through online relaxation refinement. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 107–115. AAAI Press.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman.

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

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 944–949. AAAI Press.

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

Heusner, M.; Keller, T.; and Helmert, M. 2017. Understanding the search behaviour of greedy best-first search. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 47–55. AAAI Press.

Heusner, M.; Keller, T.; and Helmert, M. 2018a. Best-case and worst-case behavior of greedy best-first search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 1463–1470. IJCAI.

Heusner, M.; Keller, T.; and Helmert, M. 2018b. Search progress and potentially expanded states in greedy best-first search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 5269–5273. IJCAI.

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

Hoffmann, J. 2005. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research* 24:685–758.

Hoffmann, J. 2011. Analyzing search topology without running any search: On the connection between causal graphs and h^+. *Journal of Artificial Intelligence Research* 41:155–229.

Holte, R. C. 2010. Common misconceptions concerning heuristic search. In Felner, A., and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 46–51. AAAI Press.

Imai, T., and Kishimoto, A. 2011. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, 985–991. AAAI Press.

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

Korf, R. E., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In Kautz, H., and Porter, B., eds., *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, 910–916. AAAI Press.

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

Lelis, L. H. S.; Zilles, S.; and Holte, R. C. 2012. Fast and accurate predictions of IDA*'s performance. In Hoffmann, J., and Selman, B., eds., *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2012)*, 514–520. AAAI Press.

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 154–161. AAAI Press.

Lipovetzky, N., and Geffner, H. 2017. Best-first width search: Exploration and exploitation in classical planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*. AAAI Press.

Lipovetzky, N.; Muise, C.; and Geffner, H. 2016. Traps, invariants, and dead-ends. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 211–215. AAAI Press.

Martelli, A. 1977. On the complexity of admissible search algorithms. *Artificial Intelligence* 8:1–13.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Mérő, L. 1984. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence* 23(1):13–27.

Nakhost, H., and Müller, M. 2009. Monte-Carlo exploration for deterministic planning. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1766–1771. AAAI Press.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1:193–204.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In Cesta, A., and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 174–182. AAAI Press.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280. AAAI Press.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Seipp, J.; Pommerening, F.; Röger, G.; and Helmert, M. 2016. Correlation complexity of classical planning domains. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 3242–3250. AAAI Press.

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

Steinmetz, M., and Hoffmann, J. 2017. State space search nogood learning: Online refinement of critical-path dead-end detectors in planning. *Artificial Intelligence* 245:1–37.

Thayer, J. T.; Stern, R.; and Lelis, L. H. 2012. Are we there yet? – estimating search progress. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 129–136. AAAI Press.

Valenzano, R.; Sturtevant, N. R.; Schaeffer, J.; and Xie, F. 2014. A comparison of knowledge-based GBFS enhancements and knowledge-free exploration. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 375–379. AAAI Press.

Wilt, C., and Ruml, W. 2012. When does weighted A* fail? In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 137–144. AAAI Press.

Wilt, C., and Ruml, W. 2014. Speedy versus greedy search. In Edelkamp, S., and Barták, R., eds., *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 184–192. AAAI Press.

Wilt, C., and Ruml, W. 2015. Building a heuristic for greedy search. In Lelis, L., and Stern, R., eds., *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SoCS 2015)*, 131–139. AAAI Press.

Xie, F.; Müller, M.; Holte, R. C.; and Imai, T. 2014. Type-based exploration with multiple search queues for satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2395–2401. AAAI Press.

Xie, F.; Müller, M.; and Holte, R. C. 2014. Adding local exploration to greedy best-first search in satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2388–2394. AAAI Press.

Xie, F.; Müller, M.; and Holte, R. C. 2015. Understanding and improving local exploration for GBFS. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 244–248. AAAI Press.

Zhang, Z.; Sturtevant, N. R.; Holte, R.; Schaeffer, J.; and Felner, A. 2009. A* search with inconsistent heuristics. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 634–639. AAAI Press.