UNIVERSITÄT BASEL

A Case Study on the Search Topology of Greedy Best-First Search

Master Thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Gabriele Röger

Lukas Beck
lukas.beck@unibas.ch

20. Mai 2014

UNI
BASEL

Acknowledgments

First, I thank the Universitätsrechenzentrum Basel for providing the cluster infrastructure to conduct my experiments. Additionally, I thank both Prof. Dr. Malte Helmert and Gabriele Röger for supervising my thesis, providing me insights and guiding me.
Furthermore, I thank both my mother and fiancée for proofreading my work and giving comments from another perspective. My special thanks goes to my fiancée for her support and understanding through this time. Also, I want to thank my fiancée, my parents and my whole family for supporting me throughout the entire time of my studies.

Abstract

Greedy best-first search (GBFS) is a prominent search algorithm for satisficing planning – finding good enough solutions to a planning task in reasonable time. GBFS selects the next node to consider based on the most promising node estimated by a heuristic function. However, this behaviour makes GBFS heavily depend on the quality of the heuristic estimator. Inaccurate heuristics can lead GBFS into regions far away from a goal. Additionally, if the heuristic ranks several nodes the same, GBFS has no information on which node it shall follow. *Diverse best-first search* (DBFS) is a new algorithm by Imai and Kishimoto [2011] which has a local search component to emphasis exploitation. To enable exploration, DBFS deploys probabilities to select the next node.
In two problem domains, we analyse GBFS' search behaviour and present theoretical results. We evaluate these results empirically and compare DBFS and GBFS on constructed as well as on provided problem instances.

Table of Contents

1

Introduction and Related Work

In this thesis, we discuss *greedy best-first search* (GBFS) – a widely used algorithm for *planning*. Planning is a branch of artificial intelligence which involves finding action sequences for known environments. This involves guiding agents in these environments from an initial state to one or more specified goal states. Our thesis works in the section of *classical planning*, that is the environment is static and known beforehand. It doesn't change while the agent is thinking. All actions which the agent executes are deterministic – their outcome is well defined and known to the agent in advance. Additionally, only one agent operates inside the environment.

Finding a *solution*, that is a valid sequence of actions from start to goal can be categorized into two scopes: satisficing and optimal planning. In optimal planning, the aim is to find the best possible solution in terms of costs for given actions, thus minimizing the sum of action costs of solutions. Our thesis works in the field of satisficing planning – finding "good enough" solutions[1], that is not necessarily optimal solutions while still considering the costs. One of the most prominent algorithms used in satisficing planning is *greedy best-first search*. GBFS belongs to the class of graph algorithms called best-first search. GBFS chooses the next action in the environment greedily, i.e. it selects the most promising node currently known. This ranking of nodes is done through goal estimators – heuristics. Heuristics try to estimate the costs from a state to the goal.

Consider the example problem depicted in Figure 1.1. Our goal is to reach s_* when starting at s_0. GBFS will always select the most promising node it currently knows measured by h. Thus, starting from s_0, GBFS will go to s_1 as the heuristic estimate of s_1 is lower than the

[1] The term *satisficing* goes back to Simon [1957]:

> "Administrative theory is peculiarly the theory of intended and bounded rationality – of the behaviour of human beings who *satisfice* because they have not the wits to *maximize*."

> "Whereas economic man supposedly maximizes – selects the best alternative from among all those available to him – his cousin, the administrator, satisfices – looks for a course of action that is satifactory or 'good enough'."

In order to reflect that the quality of a solution matters, the *International Planning Competition 2008* (IPC-2008) defined the score of a planner by the ratio Q^*/Q, that is the cost of the best known solution divided by the cost of the solution provided by the planner (Do et al. [2008]).

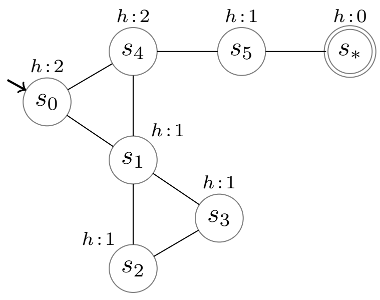


Figure 1.1: An example graph problem. The goal is to reach s_*, starting in s_0. Each node in the graph has been assigned goal distance estimate h. Actions include moving from one node to another – all actions have uniform costs of 1.

one of s_4. In s_1, it has two nodes with the best heuristic value, s_2 and s_3. GBFS will visit both nodes, but the ordering depends on how GBFS handles tie-breaking with the same heuristic estimates. Only after considering nodes s_1, s_2 and s_3, GBFS will visit nodes s_4, then s_5 and finally s_*.

The heuristic estimate of the example lead GBFS into a *local plateau* where it had to visit every node with $h = 1$. Only after visiting all nodes in the local plateau GBFS is directed to the goal node. This example shows that GBFS heavily depends on the quality of the used heuristic – if the heuristic estimates are inaccurate, GBFS is lead astray into unpromising nodes that may be far away from goal states. It also shows that GBFS comes into situations where multiple nodes have the same heuristic estimate and thus, GBFS has no information about which node is more promising.

To improve the search behaviour of GBFS, Imai and Kishimoto [2011] introduced the *diverse best-first search* (DBFS) algorithm – an algorithm that tries to tackle the problems of local plateaus and inaccurate heuristics when searching with GBFS. In short, DBFS visits both promising and unpromising nodes based on a probability parameter and narrows the search region down to local aspects.

The goal of this work is to locate and describe problems of GBFS theoretically, evaluating them empirically and compare the results to DBFS' search behaviour. We focus our analysis on the h^+ heuristic. h^+ estimates goal distances optimal for problems that are simplified by ignoring harmful effects of actions.

In the rest of this chapter we highlight related work to our thesis. The following chapter covers the background, including formal definitions of planning, planning tasks and the heuristic function. We also introduce planning domains considered in this thesis and go into detail about the relevant algorithms GBFS and DBFS. Following the background we analyse two problem domains, each in a separate chapter. This includes constructing artificial problems and deriving theoretical properties for them. Both chapters contain an empirical evaluation of the introduced theory as well as an empirical comparison of GBFS and DBFS on the artificial and provided problem instances. Finally, we conclude our thesis and point to possible future work.

The field of satisficing planning is highly relevant in practice. For many practical problems it is intractable or very expensive to find an optimal solution. Additionally, such problems often impose restrictions on time and costs of finding solutions. In these cases, satisficing planning may be able to find non-optimal but still useful solutions.

A great effort of research goes into improving GBFS and finding other algorithms for satisficing planning. Multiple enhancements for GBFS were introduced. *Preferred operators* are actions applied in a state that are identified to be more likely in a solution path. They were firstly introduced by Hoffmann and Nebel [2001] under the name of *helpful actions*. Helmert [2006] then adapted them to GBFS. Additionally, Helmert [2006] introduced two other techniques to improve the search: *deferred heuristic evaluation* and *multi-heuristic best-first search*. Deferred heuristic evaluation evaluates successors of a node in a lazy manner by inserting the successor with the parent's heuristic value into the open list. Multi-heuristic best-first search incorporates additional heuristics to identify the most promising node. This is especially useful when heuristics cover different aspects of the problem. Richter and Helmert [2009] evaluated the usefulness of preferred operators and deferred heuristic evaluation empirically. In addition, they employed a controlled, artificial experiment to analyse the benefit of preferred operators. Röger and Helmert [2010] revisited multi-heuristic best-first search and provided an extensive study on different methods to combine multiple heuristics in satisficing planning.

Valenzano et al. [2014] investigated the effects of these knowledge-based enhancements and compared them to knowledge-free techniques. In particular, they introduced a simple technique to GBFS called *ϵ-greedy node selection* which forces GBFS to expand a random node from the open list with a probability of ϵ. Similar to DBFS, *Roamer* by Lu et al. [2011] integrates stochastic elements into GBFS by doing random walks when detecting a local plateau. Xie et al. [2014a] also investigated local plateaus and introduced two local exploration strategies to exit these plateaus: *local GBFS* and *local random walk search*. Very similar to DBFS, Xie et al. [2014b] introduced a *type system* (used for heuristic search, by Lelis et al. [2013]). The resulting *type-GBFS* expands alternatively from the open list and the type data structure. A node in the type structure is selected uniformly out of possible pairs of heuristic value and path costs. Another way to enhance exploration of GBFS is *K-best-first search* (KBFS) by Felner et al. [2003]. KBFS(k) is a generalization of best-first search which expands the first k nodes from the open list at one time and then places the successors into the open list. López and Borrajo [2010] introduced EKBFS – an enhanced KBFS with a *goal agenda* (Koehler and Hoffmann [2000]) and helpful actions.

On delete relaxations and h^+, Betz and Helmert [2009] provided domain-dependent results on the complexity of calculating h^+ including Schedule. They did experiments to conclude that h^+ provides informative estimates compared to merge-and-shrink abstractions ($h^{\mathrm{m\&s}}$, Helmert et al. [2007]). Katz et al. [2013] introduced the *red-black relaxation*. This type of delete relaxation considers *some* delete effects by partitioning the variables of the planning task into two sets: *black state variables* and *red state variables*. Black variables take delete effects into account whereas red variables ignore them.

2 Background

This chapter introduces definitions that our work bases on. Here we define the notation used throughout the thesis and provide preliminaries to our work.
Firstly we describe planning including planning tasks, state spaces and heuristics. Secondly, the relevant algorithms are introduced and described. Lastly, we give an outline of the considered problem domains.

2.1 Planning

Planning is the task of solving tasks domain-independently, i.e. by working on a generalized model. In our thesis we work with the STRIPS planning task introduced by Fikes and Nilsson [1971] with a small extension.

Definition 1 (STRIPS Planning task)**.** A STRIPS planning task is a 4-tuple $\Pi = \langle V, O, I, G\rangle$ with

- V, a finite set of propositional state variables
- O, a finite set of operators, each operator $o \in O$ has preconditions $pre(o) \subseteq V$, add effects $add(o) \subseteq V$, delete effects $del(o) \subseteq V$ and costs $cost(o) \in \mathbb{N}_0$
- $I \subseteq V$, the initial state
- $G \subseteq V$, the set of goals

The original planning task by Fikes and Nilsson [1971] does not include a cost function for operators and assumed uniform action costs. We extended the original task by the cost function to allow non-uniform action costs. This type of planning task enables us to define problems in a simple yet effective manner. In the following we omit STRIPS if it is clear from context.
The goal of planning is to solve such planning tasks. Planning tasks are efficient representations of planning problems. However as the main goal of these tasks is to provide a complete but compact description of the problem definition, planning tasks are unsuitable for theoretical analysis. For this reason, we work conceptually with *state spaces*:

Definition 2 (State space)**.** A state space is a 6-tuple $\mathcal{S} = \langle S, A, cost, T, s_0, S^* \rangle$ with

- S, a finite set of states
- A, a finite set of actions
- $cost : A \rightarrow \mathbb{N}_0$, action costs
- $T \subseteq S \times A \times S$, the transition relation which is deterministic in the first two arguments, i.e. for all $s \in S, a \in A$, there is at most one s' with $\langle s, a, s' \rangle \in T$
- $s_0 \in S$, the initial state
- $S^* \subseteq S$, the set of goal states

We call an action *applicable* if it can be applied to state s, i.e. action a is applicable in s if there exists a $\langle s, a, s' \rangle$ for any state s' with $\langle s, a, s' \rangle \in T$. We write transitions as $s \xrightarrow{a} s'$ to denote that applying action a in s results in s'.
In order to solve a planning task, we need to define how such a task induces a state space.

Definition 3 (Π induced state space)**.** The state space $\mathcal{S}(\Pi) = \langle S, A, cost, T, s_0, S^* \rangle$ induced by the planning task $\Pi = \langle V, O, I, G \rangle$ is defined as:

- A set of states $S = \mathcal{P}(V)$. A state $s \subseteq V$ consists of a set of facts, representing all propositions that are true in s.
- A set of actions $A = O$.
- The cost function as defined in Π.
- The transition relation

  $$T = \{\langle s, a, s' \rangle \mid s, s' \in S, a \in A : pre(a) \subseteq s \text{ and } s' = (s \setminus del(a)) \cup add(a)\}.$$

  In words, a transition $s \xrightarrow{a} s'$ exists if the preconditions of a are met. The state s' results from applying the *add* and *delete effects* of a to s.
- The initial state $s_0 = I$.
- The set of goal states $S^* = \{s \mid s \in S : G \subseteq s\}$. A state is a goal if the set of goal propositions are true in s.

Along with state spaces, we need several other definitions. A *path* is a sequence of actions leading from one state to another:

Definition 4 (Path)**.** Let $\mathcal{S} = \langle S, A, cost, T, s_0, S^* \rangle$ be a state space. Let $s^{(0)}, \ldots, s^{(n)} \in S$ be states and $\pi_1, \ldots, \pi_n \in A$ actions such that $s^{(0)} \xrightarrow{\pi_1} s^{(1)}, \ldots, s^{(n-1)} \xrightarrow{\pi_n} s^{(n)}$.
The path from $s^{(0)}$ to $s^{(n)}$ is defined as

$$\pi = \langle \pi_1, \ldots, \pi_n \rangle \tag{2.1}$$

The length of a path π is given as $|\pi| = n$, the costs as $cost(\pi) = \sum_{i=1}^{n} cost(\pi_i)$. We call paths starting from s_0 to a goal state $s^* \in S^*$ *solutions*. When interested in the cheapest solution, we speak of optimal planning.
Solving state spaces without any further information is called *uninformed search* as no properties of the problem other than its formal definition are considered. Ideally, we want to differentiate between promising and unpromising states. For this reason, we introduce *heuristics*:

Definition 5 (Heuristic)**.** Let $\mathcal{S}$ be a state space with states S. A heuristic function for state space $\mathcal{S}$ is a function

$$h : S \to \mathbb{N}_0 \cup \{\infty\} \tag{2.2}$$

which assigns each state a non-negative number or infinity.

The intuition of $h(s)$ is to estimate the goal distance from s. This estimate is often based on the properties of the underlying state spaces and its actions. Heuristics enable us to rank states and prioritize states with lower estimates over states with higher ones.
The heuristic that estimates the best goal distance is called h^*:

Definition 6 (h^*)**.** Let $\mathcal{S}$ be a state space with states S. The perfect heuristic $h^*_{\mathcal{S}}$ assigns each state $s \in S$ the costs of the cheapest path from s to any goal state or ∞ if there exists no path to any goal state.

We write h^* for $h^*_{\mathcal{S}}$ when the state space is clear from context. Calculating $h^*(s)$ means finding the cost of the best solution from s – exactly the goal of optimal planning. Thus, h^* is a theoretical property that usually doesn't get calculated in practice.
One class of heuristic functions – heuristics based on *delete relaxations* – estimates its value by considering simplified planning tasks without problematic action effects. We can classify effects of operators of planning tasks into two categories: beneficial and harmful. As goals and preconditions are defined over a set of *true* propositions, add effects are always beneficial whereas delete effects are always harmful. Thus, the basic concept of *relaxed operators* is to ignore any delete effect:

Definition 7 (Relaxed operator)**.** Let Π be a planning task with operators O. The relaxation o^+ of an operator $o \in O$ is the operator with $pre(o^+) = pre(o)$, $add(o^+) = add(o)$, $cost(o^+) = cost(o)$ and $del(o^+) = \emptyset$.

A relaxed operator is an operator without its delete effects. When relaxing every operator of a planning task, we call this task a relaxed planning task:

Definition 8 (Relaxed planning task)**.** Let $\Pi = \langle V, O, I, G\rangle$ be a planning task. The relaxed planning task Π^+ of Π is defined as $\Pi^+ := \langle V, \{o^+ \mid o \in O\}, I, G\rangle$.

As the states are the same for both tasks, we can define heuristics on the relaxed problem for the original task. One such heuristic is h^+, the optimal heuristic of the relaxed task:

Definition 9 (h^+)**.** Let Π be a planning task with states S. The optimal relaxation heuristic function $h^+(s)$ for $\mathcal{S}(\Pi)$ for all $s \in S$ is defined as the perfect heuristic function of $\mathcal{S}(\Pi^+)$:

$$h^+(s) := h^*_{\mathcal{S}(\Pi^+)}(s). \tag{2.3}$$

h^+ is the optimal estimate for a simplified problem. As delete effects can only result in having to apply more actions but never less, it holds that h^+ never overestimates and thus $h^+(s) \leq h^*(s)$ for all $s \in S$.
It has been shown by Bylander [1994] that computing h^+ is a NP-hard problem and thus often intractable in practice. Thus, practical heuristics try to estimate h^+. Studied relaxation heuristics include h^{add} and $h^{\max}$ (Bonet and Geffner [2001]), h^{FF} (Hoffmann and Nebel [2001]), additive $h^{\max}$ (Coles et al. [2008], Haslum et al. [2005]), h^{pmax} (Mirkis and Domshlak [2007]), h^{sa} (Keyder and Geffner [2008]), h^{lst} (Keyder and Geffner [2009]) and $h^{\text{LM-cut}}$ (Helmert and Domshlak [2009]).
In our thesis we consider h^{FF} and $h^{\text{LM-cut}}$. h^{FF} overestimates h^+, whereas $h^{\text{LM-cut}}$ underestimates, i.e. $h^{\text{LM-cut}}(s) \leq h^+(s) \leq h^{\text{FF}}(s)$ for all $s \in S$.

2.2 Search Algorithms

In the following we present two algorithms that try to solve planning tasks in terms of satisficing planning. *Greedy best-first search* (GBFS) is a standard textbook algorithm (e.g. Russell and Norvig [2009]) whereas *diverse best-first search* (DBFS) introduced by Imai and Kishimoto [2011] is a new approach to improve GBFS with focus on avoiding local plateaus. Both algorithms make use of *search nodes* of which the search graph is made of. Algorithm 1 shows the definition of search nodes (line 1) as well as additional functions to simply the following algorithms. *make_node* constructs a node from the parent node as well as its state and the action that leads from the parent state to the node. *make_root_node* is a utility function to create the initial node without any parent or action. Lastly, *extract_path* constructs the path that leads from the initial node to this node by prepending all actions when traversing the graph to the root node.

2.2.1 Greedy Best-First Search

GBFS is a graph search algorithm that always expands nodes with minimal h value. Algorithm 2 shows the pseudo-code of GBFS. Like other best-first search algorithms, GBFS keeps track of possible nodes to expand in the open list and states which were already expanded in the *closed* set. GBFS takes the next element from the list (line 8), the node with the lowest h value, and expands it by inserting all successors into the open list (line 14 to 17). This process is repeated until GBFS either expands a node with a goal state (line 12 and 13) or the whole state space has been expanded and the open list is empty (line 18).
The shown algorithm is a graph search algorithm in contrary to tree search algorithms. This results in GBFS eliminating duplicates which is the usual behaviour of GBFS if not specified otherwise.

2.2.2 Diverse Best-First Search

DBFS tries to improve GBFS by avoiding local plateaus. The main goal of this algorithm is to consider multiple search directions. DBFS provides two main differences to GBFS: First,

```
SearchNode :: SearchNode(parent: SearchNode, state: State, $a$: Action, $g$: $\mathbb{N}_0$)
make_node(parent: SearchNode, $a$: Action, $s'$: State)
begin
    $n \leftarrow$ new SearchNode
    $n$.parent $\leftarrow$ parent
    $n$.state $\leftarrow s'$
    $n.a \leftarrow a$
    $n.g \leftarrow$ parent.$g + cost(a)$
    return n
make_root_node()
begin
    $n \leftarrow$ new SearchNode
    $n$.state $\leftarrow s_0$
    $n$.parent $\leftarrow$ nil
    $n.a \leftarrow$ nil
    $n.g \leftarrow 0$
    return n
extract_path(n: SearchNode)
begin
    list $\leftarrow$ empty list of Actions
    while $n.a \neq nil$ do
        list.prepend($n.a$)
        $n \leftarrow n$.parent
    return list
```

Algorithm 1: Definition of search nodes and utility functions to create new nodes as well as extract the path from the initial node to the given node.

DBFS has a probability to expand suboptimal nodes with respect to both heuristic value as well as path costs. Secondly, after fetching a node from the open list, DBFS limits its search locally.

The outline of DBFS is shown in Algorithm 3. DBFS uses a global open set and a global closed list. Each iteration starts with fetching a new node from the global open set (line 27). Starting from this node, DBFS searches locally by performing GBFS for a limited amount of expansions. When no solution is found, all locally found nodes are merged into the global open set (line 40) and a new iteration begins.

Algorithm 4 shows the procedure of extracting the next node from the global open set. DBFS assigns a probability to each pair $\langle h, g\rangle$ of possible heuristic and path cost values (lines 52 to 58). The parameter $T \in (0,1]$[2] controls the probability of selecting nodes with higher h values than h_{min} by raising T to the power of $h - h_{min}$ (line 57). Additionally, DBFS provides a second parameter G: G controls the probability by restricting the maximal path

[2] In contrast to DBFS introduced by Imai and Kishimoto [2011], we don't allow $T = 0$. The reason for this is to avoid cases where $p_{\text{total}} = 0$. This can happen when $T = 0$, $G < g_{\max}$ and all nodes with $h(n.\text{state}) = h_{\min}$ have path costs of $n.g = g_{\max}$. Then, the only way to increase p_{total} is by selecting nodes with $h_{\min}$ ($T^{(h-h_{\min})} = 0^0 = 1$). However, exactly these nodes are not considered as $G < g_{\max}$ and thus, $p_{\text{total}} = 0$.

costs of considered nodes (line 47 and 53).

2.3 Planning domains

In our thesis we considered two planning domains: Pathways and Schedule. Pathways originates from the *Fifth International Planning Competition* (IPC-5) (Gerevini et al. [2009]) whereas Schedule was used in the *Second International Planning Competition* (IPC-2) (Bacchus [2001]). Both domains have uniform action costs of 1. The domain definitions of Pathways and Schedule can be found in the appendix in Section A.3.1 and A.3.2.
We chose these two planning domains as the experiments conducted by Imai and Kishimoto [2011] show that DBFS is able to solve at least 3 times more problems than GBFS. For Pathways, DBFS solved all 30 instances. Additionally, both domains have a simple structure which simplifies a theoretical analysis.

2.3.1 Pathways

Pathways is a problem domain about chemical reactions. The basic goal is to make certain *complex molecules* available in the pool of present molecules as a result of chemical reactions. Complex molecules are molecules that can only be made from reacting reagents whereas *simple molecules* are the starting point. A limited amount of them can be *chosen*. Chosen molecules can then be made available indefinitely by *initializing* them.
The possible actions to create a new product are the following:

```
25 GBFS($\Pi$: planning task, $h$: heuristic)
26 begin
27     open ← priority queue of SearchNodes ordered by $h$
28     if $h(s_0) < \infty$ then
29         open.insert(make_root_node())
30     closed ← empty set of States
31     while open is not empty do
32         $n$ ← open.pop_min()
33         if $n.state \in closed$ then
34             continue
35         closed.insert($n$)
36         if $n.state \in S^*$ then
37             return extract_path($n$)
38         for each $\langle a, s' \rangle$ with $n.state \xrightarrow{a} s'$ do
39             if $h(s') < \infty$ then
40                 $n'$ ← make_node($n$, $a$, $s'$)
41                 open.insert($n'$)
42     return unsolvable
```

Algorithm 2: Pseudo-code of GBFS, a variant of best-first search.

- *associate* that consumes both reagents,
- *associate-with-catalyze* which consumes only one of the reagents, the other serves as a catalyst and remains available after the reaction
- and *synthesize* that creates the product from one reagent without consuming any molecule.

For all problems, the goal is to reach all sub-goals – a sub-goal consists of two molecules whereat at least one has to be available.
One interesting property of this domain is that the only possible dead-ends can result from choosing the wrong simple molecules. Accordingly, finding any solution solely depends on the set of chosen molecules and not on the delete effects of the reactions. This also means that as soon as the limit of chosen molecules is reached, h^+ immediately detects dead-ends and results in choosing a valid set of chosen molecules. Notice however, that this set can be too large.

```
DBFS($\Pi$: planning task, $h$: heuristic, $P$: $[0,1]$, $T$: $(0,1]$)
begin
    open ← empty set of SearchNodes
    closed ← empty set of States
    if $h(s_0) < \infty$ then
        open.insert(make_root_node())
    while open is not empty do
        local ← empty priority queue of SearchNodes ordered by $h$
        $n$ ← fetch_node(open, $P$, $T$)
        local.insert($n$)
        // Perform local GBFS rooted at n
        for each $i \in \{1, \ldots, \min(1, h(n.state))\}$ do
            if local is empty then
                break
            $m$ ← local.pop_min()
            closed.insert($m$.state)
            if $m.state \in S^*$ then
                return extract_path($m$)
            for each $\langle a, s' \rangle$ with $m.state \xrightarrow{a} s'$ do
                if $h(s') < \infty$ and $s' \notin closed$ then
                    $m'$ ← make_node($m$, $a$, $s'$)
                    local.insert($m'$)
        open.insert_all(local)
    return unsolvable
```

Algorithm 3: The outline of the algorithm introduced by Imai and Kishimoto [2011].

2.3.2 Schedule

The domain *Schedule* models the processing and machining of objects on different types of machines. The goal is to give certain objects the correct properties. Possible properties are shape, surface condition and paint. Additionally, objects have a temperature – either cold or hot. Each machine modifies different aspects of an object and can also remove other properties during the process:

- *Roller* shapes the object cylindrical, but makes the object hot and removes any surface and paint.
- Similar to rolling, the *lathe* makes objects cylindrical, but removes the paint and makes their surface rough.
- The *polisher* simply polishes the surface of the object. The polisher requires objects to be cold.
- *Grinding* an object results in a smooth surface without any paint.
- Lastly, the *immersion painter* and *spray painter* paint objects. Additionally, the spray painter removes the surface condition and requires cold objects.

Notice that in the original domain definition, two additional machines *punch* and *drill-press* are defined. As none of the machines are required for the defined problems (the additional prepositions don't appear in any goal condition), we omitted them to simplify our analysis.

```
66 DBFS :: fetch_node(open: set of SearchNodes, $P$: $[0,1]$, $T$: $(0,1]$)
67 begin
68     $\langle h_{min}, h_{max}\rangle \leftarrow$ minimum and maximum of $\{h(n.\text{state}) \mid n \in \text{open}\}$
69     $\langle g_{min}, g_{max}\rangle \leftarrow$ minimum and maximum of $\{n.g \mid n \in \text{open}\}$
70     if with probability of $P$ then
71         $G \leftarrow$ select uniformly from $\{g_{min}, \dots, g_{max}\}$
72     else
73         $G \leftarrow g_{max}$
74     $p_{total} \leftarrow 0$
75     $p \leftarrow$ empty map from $\langle h, g\rangle$ to $\mathbb{R}_0^+$
76     for each $h \in \{h_{min}, \dots, h_{max}\}$ do
77         for each $g \in \{g_{min}, \dots, G\}$ do
78             if there is no $n \in$ open with $h(n.state) = h$ and $n.g = g$ then
79                 $p(\langle h, g\rangle) \leftarrow 0$
80             else
81                 $p(\langle h, g\rangle) \leftarrow T^{h-h_{min}}$
82             $p_{total} \leftarrow p_{total} + p(\langle h, g\rangle)$
83     $\langle h, g\rangle \leftarrow$ select $\langle h, g\rangle$ with respect to probability $p(\langle h,g\rangle)/p_{total}$
84     $n \leftarrow$ select uniformly from $\{n \mid n \in \text{open} : h(n.\text{state}) = h, n.g = g\}$
85     return $n$
```

Algorithm 4: Details about the method to fetch the next node from the global open set used in DBFS.

Machining an object results in the machine being *busy* and the object being *scheduled*. In order to free machines and objects, we have to apply a *time-step*. The intention of time-step is to enable parallel processing of different objects on different machines.
Important to notice is that there is no action to cool objects down – thus, any hot object stays hot. This characteristic is the only way to result in a dead end: an object that has to be polished needs to be processed by the polisher – however, the polisher requires cold objects.

3

Pathways Analysis

In the following we begin by analysing the Pathways domain. The goal of this analysis is to locate specific problems of GBFS when searching with h^+. We show that in specific situations, GBFS has to search uninformed through local plateaus of exponential sizes.
One important aspect of the domain is that most of the reactions consume molecules. h^+ ignores the consumption in such reactions. In situations where two reactions consume the same molecule, h^+ underestimates the costs as it ignores the costs to build the consumed molecule. Figure 3.1 illustrates exactly this situation and shows that this property of h^+ can lead GBFS into a large local plateau. Notice that unlike the predefined problems of the Pathways domain our example requires *both* molecules V_1X and V_1Y to be available as a goal state. In order to construct the same situation in which only one of two molecules are needed for a goal, we can introduce another associate reaction that merges V_1X and V_1Y into the new goal molecule and then duplicate the graph. Then we require that only one of these goal molecules has to be available to reach the goal.

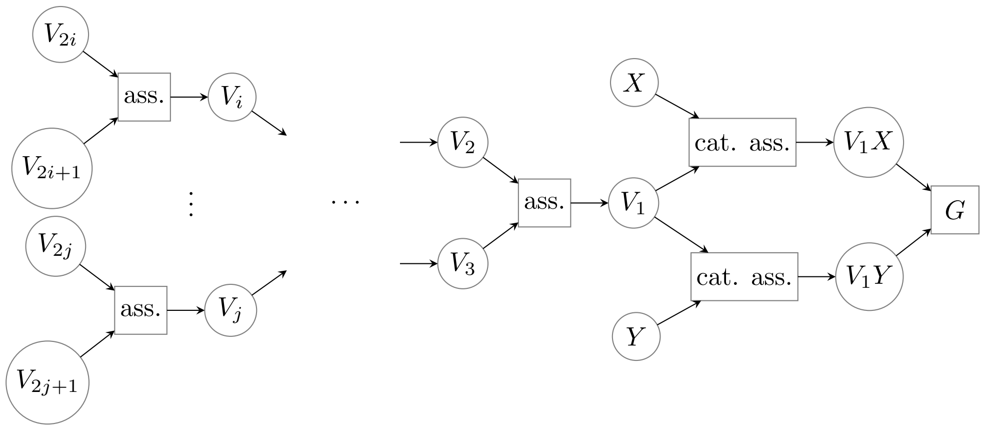


Figure 3.1: An exemplary Pathways problem. Actions choose and initialize are omitted for reasons of readability. Molecules X and Y serve as catalysts and don't get consumed when reacting with V_1.

3.1 Preliminaries

Let $\mathcal{N} := \{V_1, \dots, V_n\}$ with $|\mathcal{N}| = n$ be the set of all molecules of the binary tree. As we use associate actions, every inner node needs to have two children and as such the number of molecules needs to be odd: $n = 2k + 1$ with $k \in \mathbb{N}_0$.
Let $\mathcal{M} := \mathcal{N} \cup \{X, Y, V_1X, V_1Y\}$ with $|\mathcal{M}| = n + 4$ be the set of all molecules, d_{M} be the depth of the layer containing molecule $M \in \mathcal{N}$ with $d_{V_1} = 0, \dots, d_{V_i} = \lfloor \log_2(i) \rfloor, \dots, d_{V_n} =: d$ and $\mathcal{N}_i$ be the set of molecules in depth i with

$$|\mathcal{N}_i| = \begin{cases} 2^i & \text{if } i < d \\ n - \displaystyle\sum_{j=0}^{d-1} |\mathcal{N}_j| = n - 2^d + 1 & \text{if } i = d. \end{cases} \tag{3.1}$$

Let $\mathcal{N}_s := \{V_i \mid V_i \in \mathcal{N} : 2i > n\} = \mathcal{N}_d \cup \{V_i \mid V_i \in \mathcal{N}_{d-1} : 2i > n\}$ be the set of all molecules of the binary tree which cannot be made available by any reaction. It contains all molecules $\mathcal{N}_d$ in the outermost depth d and all simple molecules in depth $d-1$. The size of this set is given as:

$$\begin{aligned} |\mathcal{N}_s| &= |\mathcal{N}_d| + |\mathcal{N}_{d-1}| - \frac{|\mathcal{N}_d|}{2} \\ &= \frac{1}{2}(|\mathcal{N}_d| + 2|\mathcal{N}_{d-1}|) \\ &\overset{(3.1)}{=} \frac{1}{2}(n - 2^d + 1 + 2 \cdot 2^{d-1}) \\ &= \frac{(n+1)}{2} \end{aligned} \tag{3.2}$$

To reach the goal in this example all molecules have to be made available at least once. Thus every simple molecule has to be chosen as well. For this reason, we assume that all simple molecules are already chosen in the initial state $s_0 = \{\text{chosen}(M) \mid M \in \mathcal{N}_s \cup \{X, Y\}\}$. When all simple molecules are already chosen, we can no longer apply any choose action and omit them in the following discussion as it is not part of our analysis. One side effect of this decision is that we can describe states as the set of available molecules $\tilde{s} \subseteq \mathcal{M}$:

$$s = s_0 \cup \{\text{available}(M) \mid M \in \tilde{s}\}. \tag{3.3}$$

For readability, we omit the tilde character and directly describe states $s \subseteq \mathcal{M}$ as sets of available molecules.

3.2 Heuristic Value

To calculate $h^+(s)$ and $h^*(s)$ values for any state $s \in S$ in this example problem, we define the costs $cost(M, s)$ to create molecule $M \in \mathcal{M}$ in state s.
The cost to build M is 0 if it is already included in s. For all simple molecules $\mathcal{N}_s$, X and Y we have costs of 1 to make them available if not already given. The costs of all other molecules V_i in the binary tree are the costs to build the reactants V_{2i} and V_{2i+1} and associating them to V_i. Similarly the costs to build the goal molecules V_1X and V_1Y are given by the costs to build their reactants V_1 and X/Y and associating them.

Formally, the costs $cost(M,s)$ to create $M \in \mathcal{M}$ in $s \in S$ are given by:

$$M \in \mathcal{N}_s \cup \{X,Y\} : cost(M,s) := \begin{cases} 0 & \text{if } M \in s \\ 1 & \text{otherwise} \end{cases}$$

$$M = V_i \in \mathcal{N} \setminus \mathcal{N}_s : cost(M,s) := \begin{cases} 0 & \text{if } M \in s \\ 1 + cost(V_{2i},s) & \text{otherwise} \\ \quad + cost(V_{2i+1},s) & \end{cases}$$

$$M = V_1Z, Z \in \{X,Y\} : cost(M,s) := \begin{cases} 0 & \text{if } M \in s \\ 1 + cost(V_1,s) + cost(Z,s) & \text{otherwise} \end{cases} \tag{3.4}$$

Now, $h^+(s)$ is given as the costs to build both goal molecules V_1X and V_1Y while counting the costs of V_1 only once:

$$h^+(s) = \begin{cases} cost(V_1X,s) + cost(V_1Y,s) - cost(V_1,s) & \text{if } V_1X \notin s \text{ and } V_1Y \notin s \\ cost(V_1X,s) + cost(V_1Y,s) & \text{otherwise} \end{cases} \tag{3.5}$$

3.3 Problem Analysis

The key molecule is molecule V_1. The problem arises as soon as GBFS expands the state $s := \{V1, X, Y\}$ with

$$\begin{aligned} h^+(s) &\overset{(3.5)}{=} cost(V_1X,s) + cost(V_1Y,s) - cost(V_1,s) \\ &= 1 + cost(V_1,s) + cost(X,s) + 1 + cost(V_1,s) + cost(Y,s) - cost(V_1,s) \\ &= 2 + cost(X,s) + cost(Y,s) + cost(V_1,s) = 2. \end{aligned} \tag{3.6}$$

h^+ leads GBFS directly to this state. State s has two successors which lead on a shortest path to the goal – combining V_1 with either X or Y: $s' := \{V_1X, X, Y\}$ and $s'' := \{V_1Y, X, Y\}$. Both states have the same heuristic estimate of

$$h^+(s') \overset{(3.5)}{=} cost(V_1Y,s') = 1 + cost(V_1,s') = 1 + cost(V_1,s'') = h^+(s''). \tag{3.7}$$

Notice that the estimate of h^+ is the cost of the optimal path to reach the goal, thus $h^+(s') = h^+(s'') = h^*(s') = h^*(s'')$.

Apart from these two successors s has $|\mathcal{N}_s|$ more successors $s_i := \{V_i, V_1, X, Y\}$ for all $V_i \in \mathcal{N}_s$, each making one of outermost molecules available. Important to notice is that the heuristic values stays the same for all s_i: $h^+(s_i) = h^+(s) = 2$. Going a layer further, all successors of s_i – with the exception of states that combined V_1 with either X or Y – also have a heuristic value of 2.

Generally the set of states $\mathcal{S}$ reached from s with heuristic estimate 2 is made of three sets:

- The set $\mathcal{S}_1 := \mathcal{P}(\{V_2, \ldots, V_n\}) \times s$, containing all combinations of molecules needed to build V_1 combined with s. The heuristic value $h^+(\hat{s})$ for all $\hat{s} \in \mathcal{S}_1$ is the same as $h^+(s)$ as making molecules available that are located before V_1 does not influence h^+ at all:

$$\begin{aligned} h^+(s) &\overset{(3.5)}{=} 2 + cost(X,s) + cost(Y,s) + cost(V_1,s) \\ &= 2 + \underbrace{cost(X,\hat{s})}_{0} + \underbrace{cost(Y,\hat{s})}_{0} + \underbrace{cost(V_1,\hat{s})}_{0} = h^+(\hat{s}). \end{aligned} \tag{3.8}$$

- The two sets

$$\mathcal{S}_2 := \mathcal{P}(\{V_4, \ldots, V_n\}) \times (\{V_2, V_3\} \cup s')$$
$$\mathcal{S}_3 := \mathcal{P}(\{V_4, \ldots, V_n\}) \times (\{V_2, V_3\} \cup s'')$$

  containing all combinations of molecules needed to build V_2 and V_3 when combined with V_2, V_3 and either s' or s''. The addition of V_2 and V_3 to s' and s'' changes their heuristic value to 2:

$$\begin{aligned} h^+(s' \cup \{V_2, V_3\}) &\overset{(3.5)}{=} cost(V_1Y, s' \cup \{V_2, V_3\}) \\ &= 1 + cost(V_1, s' \cup \{V_2, V_3\}) + cost(Y, s' \cup \{V_2, V_3\}) \\ &= 2 + cost(V_2, s' \cup \{V_2, V_3\}) + cost(V_3, s' \cup \{V_2, V_3\}) = 2 \end{aligned} \tag{3.9}$$

  Similar to $\mathcal{S}_1$, the addition of any molecules $\{V_4, \ldots, V_n\} \cup X$ to $\{V_2, V_3\} \cup s'$ does not change the heuristic value.

GBFS reaches a local plateau after expanding state s. The set of states that enable GBFS to leave the plateau is $\mathcal{S}_2 \cup \mathcal{S}_3$. For each state in this set – an *exit state* – there exists the action to associate V_2 with V_3 to V_1 and then reach the goal by associating V_1 with either X or Y:

$$\begin{tikzcd} \{V_2, V_3, V_1X, \ldots\} \\ & \{V_1, V_1X, \ldots\} \arrow[r, "\text{associate } V_1 \text{ and } Y"] & \{V_1X, V_1Y, \ldots\} \in S^* \\ \arrow[r, "\text{associate } V_2 \text{ and } V_3"] & \\ & \{V_1, V_1Y, \ldots\} \arrow[ruu, "\text{associate } V_1 \text{ and } X"'] \\ \{V_2, V_3, V_1Y, \ldots\} \end{tikzcd}$$

For these successors the h^+ value as well as the optimal costs are 1.

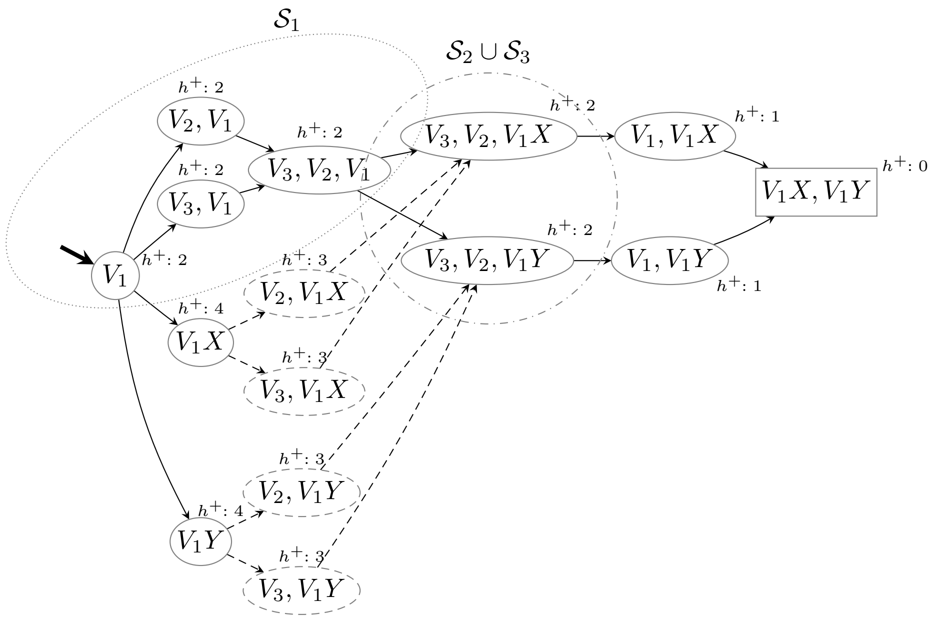


Figure 3.2: Simplified search graph with $n = 3$, starting from state $s = \{V_1, X, Y\}$. As molecules X and Y are in all states available, we omitted them for reasons of readability.
Dashed nodes are nodes that GBFS never creates when searching with h^+.

To illustrate the three sets $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$, Figure 3.2 shows a small search graph with $n = 3$. Notice that GBFS has to reach the exit states while searching uninformed. Following the higher valued states V_1X and V_1Y leads to an informed search.

3.4 Quantification of GBFS' Search Effort

How many states GBFS has to expand to reach an exit state heavily depends on the method that GBFS uses to handle tie-breaking h-values. When reaching the local plateau, GBFS has to search uninformed. If GBFS handles tie-breaking in a *first in, first out* (FIFO) manner, then GBFS searches the plateau accordingly to breadth-first search. If GBFS uses a stack for tie-breaking, then GBFS uses depth-first search.
We don't go into detail in the case of depth-first tie-breaking as the standard way to tie-break is in a FIFO manner. Notice however that GBFS' search behaviour depends on the action ordering. The optimal ordering relates to associating the most important molecules first. One possible ordering is associating goal molecules first, then associating complex V_i from lower to higher i and finally initializing simple molecules with V_i from lower to higher i. We assume that the number of expanded states scales exponentially even with an optimal action ordering.
When GBFS is tie-breaking with a FIFO queue, then GBFS will find the shortest path π from s to the nearest exit state. This is because breadth-first search is both complete and optimal when searching with uniform action costs. The costs of this path is given by costs of creating V_2 and V_3 and then reaching the exit state. As s does not contain any molecules $V_2, \ldots, V_n$, we can simply count the number of molecules required to create V_2 and V_3:

$$cost(\pi) = 1 + cost(V_2, s) + cost(V_3, s) = n. \tag{3.10}$$

It is obvious that at least all simple molecules $\mathcal{N}_s$ have to be made available at least once ($cost(\pi) = n > |\mathcal{N}_s|$ for $n > 1$). This results in breadth-first search having to expand at least all combinations $|\mathcal{P}(\mathcal{N}_s)|$ before reaching an exit state:

$$|\mathcal{P}(\mathcal{N}_s)| = 2^{|\mathcal{N}_s|} \overset{(3.2)}{=} 2^{(n+1)/2} = 2^{(n+4-3)/2} = 2^{(|\mathcal{M}|-3)/2} \tag{3.11}$$

Equation 3.11 gives a lower bound on the number of states that need to be expanded to reach an exit state in the local plateau. That means that the number of expanded states scales at least exponentially with $|\mathcal{M}|$ when GBFS is using a FIFO queue for tie-breaking h-values.

3.5 Evaluation

In order to evaluate the theoretical results, we created a test domain according to Figure 3.1 with problems having a number of molecules ranging from 5 to 67. On these problems we counted the number of expanded states by GBFS. We used the Fast Downward framework (Helmert [2006]) and its GBFS implementation *eager_greedy* without any enhancements. This and all following experiments were conducted single-threaded on Intel Xeon E5-2660 processors with 2.20 GHz and 64 GiB DDR3 1600 MHz memory. As we are mainly interested in the number of expanded states of GBFS, we used a time limit of 5 hours search time and a memory limit of 3.75 GiB.

Figure 3.3 shows the experimental results – comparing the empirical number of states compared to our lower bound in Equation 3.11. The experiment data can be found in the appendix in Table A.2.

An useful approximation of GBFS' scaling for higher problem sizes ($|\mathcal{M}| \geq 13$) is: $a \cdot 2^{b \cdot |\mathcal{M}|}$ with $a = \frac{3}{50}$ and $b = \frac{7}{8}$. Notice that GBFS scales almost with the same exponential factor of $2^{|\mathcal{M}|}$ which clearly indicates that GBFS searches uninformed through the local plateau and has to expand many combinations of molecules. It is also clear that our lower bound clearly holds but greatly underestimates the number of expansion in the local plateau. This is because our estimate only counts the combinations of simple molecules. However, GBFS needs to expand many combinations with complex molecules that Equation 3.11 ignores.

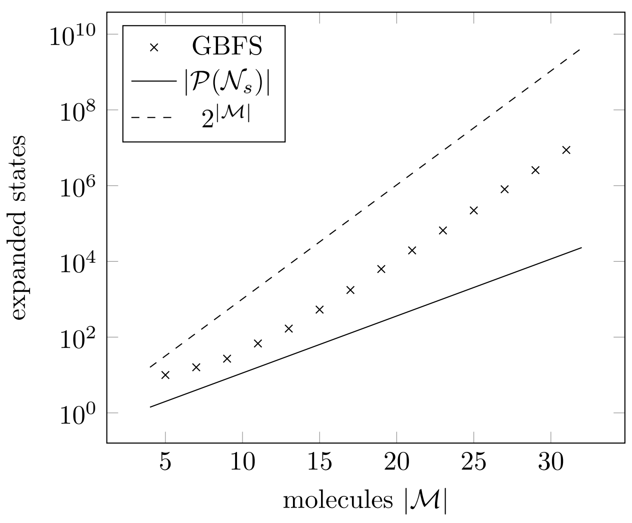


Figure 3.3: Plot comparing expanded states by GBFS to the number of combinations of simple molecules. Function $2^{|\mathcal{M}|}$ is shown for reference.

3.6 Comparison of GBFS and DBFS

In the following we compare GBFS and DBFS on the Pathways problem domain. First we show the performance of DBFS on our example problem and then compare the performance on the problems provided by IPC-5. For both figures, we run DBFS with three different random seeds[3], denoted by the markers ⅄, Y and ×. We fixed DBFS' parameters to $T = 0.5$ and $P = 0.1$[4]. These experiments had limits of 30 minutes and 2 GiB memory. For all experiments with DBFS, we used the implementation of Marxer [2013].

Figure 3.4 shows the number of expanded states of both algorithms on our example problem. We plotted for both algorithms approximations in the form of $a \cdot 2^{b \cdot |\mathcal{M}|}$ whereas for GBFS

[3] The exact seeds can be found in Table A.1 in the appendix.

[4] We chose the parameters accordingly to Imai and Kishimoto [2011]. They conducted most of their experiments with these parameters.

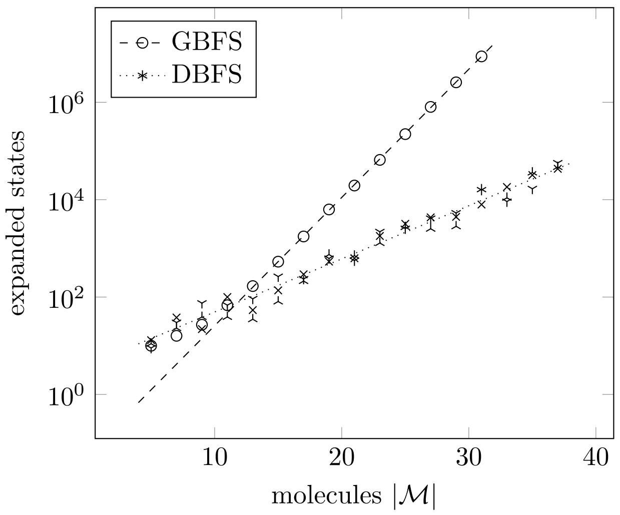


Figure 3.4: Comparison of the number of expanded states of GBFS and DBFS in our example problem with approximations for both algorithms. When using GBFS, both h^{FF} and $h^{\text{LM-cut}}$ result in the same number of expansions. For DBFS, we used h^{FF} as an approximation to h^+.

we used the approximation mentioned in Section 3.5. For DBFS, we set $a = 4$ and $b = \frac{29}{80}$. We can clearly see that DBFS is able to solve the problems with less expansions. We assume the reason is because DBFS conducts a local search with sometimes unpromising nodes. After arriving in the local plateau, expanding the states $s' = \{V_1X, X, Y\}$ and $s'' = \{V_1Y, X, Y\}$ enables DBFS to advance directly to the goal, depending on the quality of the h^+ estimates of the used heuristic. With h^+, DBFS would reach directly the goal as the h^+ estimates of all successors of s' and s'' are also the best goal distances h^*.

The number of expanded states of DBFS scales exponentially as well. We assume this is related to the structure of our local plateau. Notice that the local plateau contains the goal states itself. This means that GBFS does not have to expand the whole plateau to finally expand a higher estimate node that leads to the goal. Thus, DBFS does not benefit directly from expanding an unpromising node to avoid the plateau. Instead, the higher estimated nodes simply provide an informed search when ignoring the local plateau.

Additionally we conducted experiments for the standard problems of Pathways to compare the practical performance. Figure 3.5 shows the scatter plots of the results, comparing the number of expanded states for both h^{FF} and $h^{\text{LM-cut}}$. We can clearly see that DBFS dominates GBFS in terms of both number of expansions as well as number of solved instances. DBFS was able to solve between 16 and 19 instances, depending on the heuristic ($h^{\text{LM-cut}}$ has more solved instances) as well as the random seed, whereas GBFS was able to solve 9 with $h^{\text{LM-cut}}$ and 10 with h^{FF}.

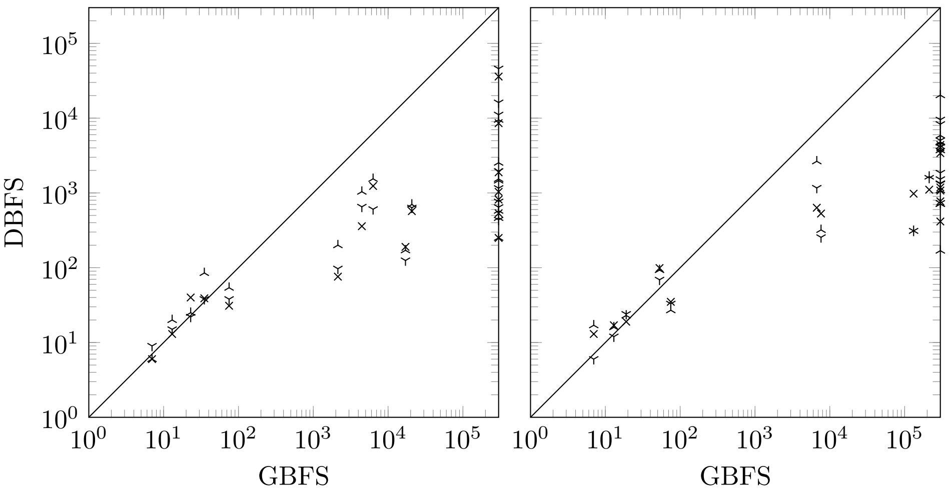
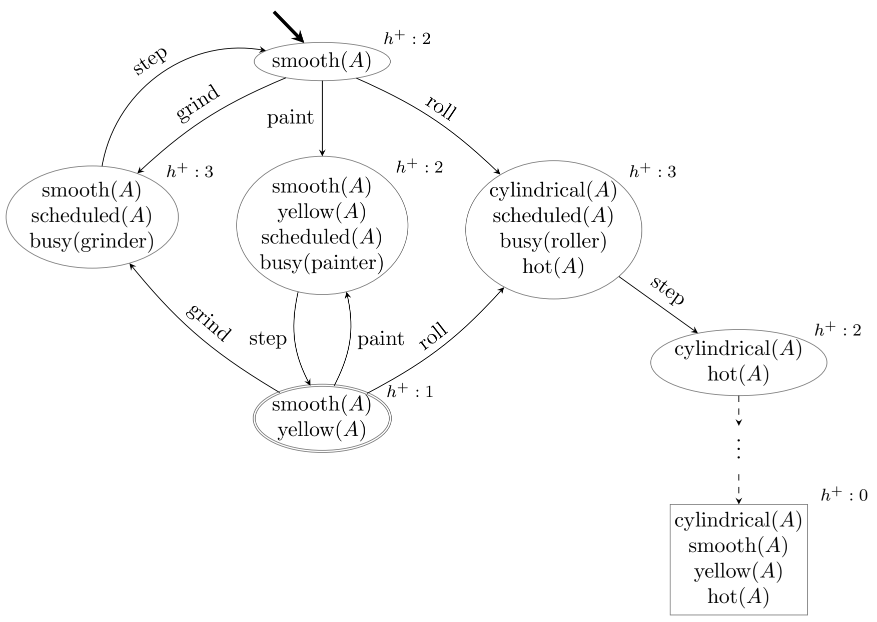


Figure 3.5: Scatter plots comparing the number of expansions of GBFS to DBFS. The left image shows the results using h^{FF}, the right using $h^{\mathrm{LM\text{-}cut}}$. Data points appearing on the right-most side of the plot indicate that GBFS was not able to solve the problem whereas DBFS was.

4

Schedule Analysis

After analysing the Pathways domain, we now take a closer look at Schedule: The Schedule domain has several machines that modify properties of a machined part. For example, the lathe makes an object cylindrical but removes any paint and surface finish. h^+ underestimates these costs as it ignores the removal of such properties. Additionally, if no object is scheduled, h^+ ignores the cost of processing an object by doing a *time-step*.



$h^+ : 2$
smooth(A)
step
grind
paint
roll
$h^+ : 3$
smooth(A) scheduled(A) busy(grinder)
$h^+ : 2$
smooth(A) yellow(A) scheduled(A) busy(painter)
$h^+ : 3$
cylindrical(A) scheduled(A) busy(roller) hot(A)
step
$h^+ : 2$
cylindrical(A) hot(A)
grind
step
paint
roll
$h^+ : 1$
smooth(A) yellow(A)
$h^+ : 0$
cylindrical(A) smooth(A) yellow(A) hot(A)

Figure 4.1: Simplified state space of an exemplary Schedule problem with a local minimum. Several details were simplified to improve readability of the graph: painting with the immersion-painter has been abbreviated to *paint*, the color of the paint as well as the three machines polisher, lathe and spray-painter were omitted in this example.

Figure 4.1 shows a simplified example of a Schedule problem. In the initial state object A is already smooth. The goal is that A is cylindrical, yellow and smooth. In this example the h^+ heuristic results in a local minimum: A already has the correct surface condition at the

start. h^+ estimates the costs as 2 – rolling and painting A. Rolling A makes it cylindrical but removes any paint and surface finish. This results in an increased heuristic estimate of 3: A is still missing two properties but is now scheduled on a machine and needs to be processed first. On the other hand, painting first does not increase the estimate as now a time-step and the roll process is needed. Having A both painted yellow and made smooth – state $s = \{\text{smooth}, \text{yellow}(A)\}$ – results in a local minimum.
In order to scale our example problem, we introduce additional objects $B_2, \dots, B_n$. With these additional parts, the local minimum transforms into a local plateau. Even though all objects B_i are irrelevant for the goal, the planner cannot ignore actions applied on these objects. This is because machines get occupied – *busy* – by processing these objects. Additionally, several machines have the temperature proposition in their preconditions and effects. Thus, the planner has also to consider the temperature state of each object B_i.
This means that we can schedule every object B_i onto every machine in state s without changing the h^+ value. Even when occupying the roller, we still have the possibility to use the lathe to make A cylindrical. The resulting local plateau has multiple layers, each containing a number of combinations of processing an object B_i on a machine. In the first layer reached from s, we can combine each machine with each part. This results in $\binom{m}{1}\binom{n-1}{1}$ combinations with m as the number of machines. In the second layer, we combine two machines with two parts, thus $\binom{m}{2}\binom{n-1}{2}$. The number of combinations of the kth layer is given by

$$|S_p^k| = \binom{m}{k}\binom{n-1}{k} \tag{4.1}$$

where $k \leq \min(m, n-1)$. The size of the whole plateau is given as the sum of all layers:

$$|S_p| = \sum_{k=0}^{\min(m,n-1)} |S_p^k| \tag{4.2}$$

For larger n $(n-1 > m)$, we can simplify the formula to:

$$|S_p| = \sum_{k=0}^{m} |S_p^k| \tag{4.3}$$

The size of the local plateau scales polynomially with $O((n-1)^m)$ or $O\big((n-1)^6\big)$ considering that the Schedule domain has a fixed number of machines $m = 6$. $|S_p|$ gives us a lower bound to the number of expanded states needed to reach the goal when using GBFS.

4.1 Evaluation

We created the example problem with different numbers of objects. Figure 4.2 shows the empirically measured number of expansions of GBFS in relation to our theoretical lower bound $|S_p|$. The plot shows that our lower bound is indeed correct, but it appears that the expansions scale faster than our estimate. We suspect that the problem lies within the machines lathe and roller. In our example problem, the local plateau exists because rolling removes two goal propositions. In terms of operator effects, the lathe is very similar to the roller. It makes objects cylindrical and removes the paint as well. Instead of removing the surface condition, lathing makes objects *rough*. In our example, this has the same effect as we still have to grind object A to make it smooth.

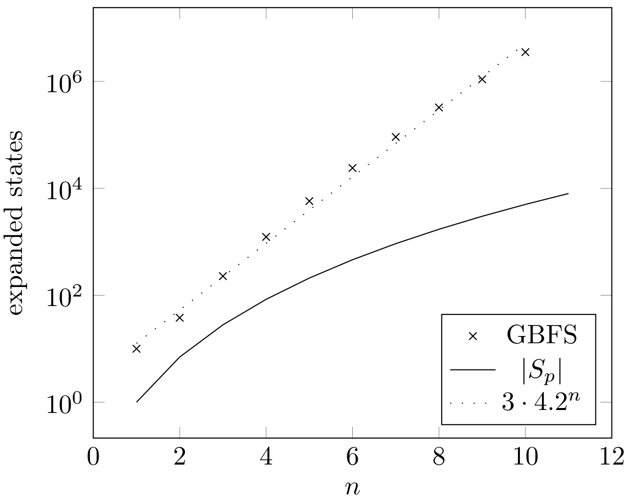


Figure 4.2: Plot comparing the number of expansions done by GBFS and our theoretical lower bound $|S_p|$.

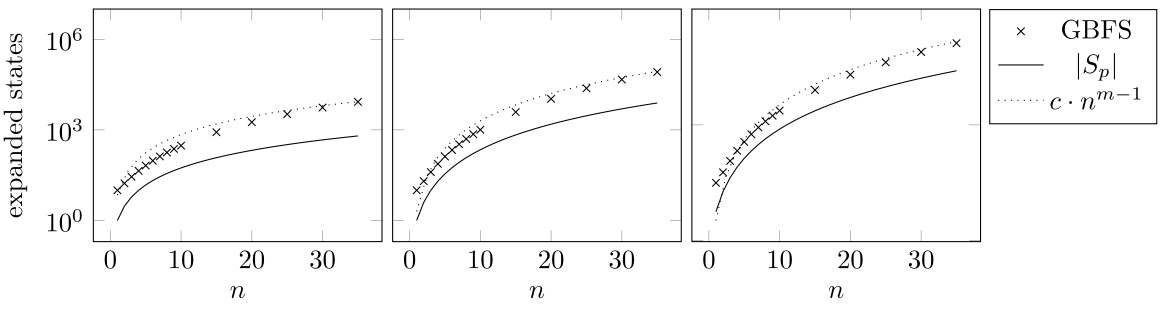


Figure 4.3: Three plots showing the number of expanded states by GBFS compared to the lower bound $|S_p|$ with modified domains. From right to left, in each sub-plot one more machine is disabled. Every plot is missing the lathe.
Identical to our example Figure 4.1, the left plot ($m = 3$) starts with the grinder, roller and immersion-painter. The polisher was appended to the middle one ($m = 4$) and the polisher and the spray-painter to the right ($m = 5$).
When excluding the lathe from the available machines, we can only combine $m - 1$ machines as rolling increases the heuristic value because of the costs of time-step. Thus, for these plots, $|S_p^k| = \binom{m-1}{k}\binom{n-1}{k}$ and $|S_p| = \sum_{k=0}^{\min(m-1,n-1)} |S_p^k|$.
The function $c \cdot n^{m-1}$ is shown as an approximation with $c_{\text{left}} = 7$, $c_{\text{middle}} = 2$ and $c_{\text{right}} = 0.5$.

This fact leads to the conclusion that having two different machines setting the relevant goal proposition increases the local plateau substantially. This results in GBFS expanding more than a polynomial number of states in relation to the number of objects. To confirm this assumption, we concluded additional experiments by disabling the lathe and further adjusting the number of machines. Figure 4.3 shows the results of these experiments. The left plot shows exactly our example problem with the three machines roller, grinder and immersion-painter. The middle one contains additionally the polisher and to the right one, we appended both the polisher and the spray-painter. Thus, the three modified domains have 3, 4 and 5 machines. The data from Figure 4.2 can be found in the appendix in Table A.3. By removing the lathe, GBFS indeed scales similar to our theoretical lower bound $|S_p|$. This strengthens our assumption that the addition of the lathe is responsible for GBFS' super-polynomial scaling.

4.2 Comparison of GBFS and DBFS

The following section contains empirical comparisons of GBFS and DBFS. We used the same experiment setup as in Section 3.6.
Figure 4.4 compares the performance of GBFS and DBFS on our example problem. Notice that even for greater problem sizes, the number of expansions by DBFS is very low (less than 10^3). Additionally, DBFS was able to solve all provided instances. To show the scaling of DBFS, we plotted the linear function $3n$ to show the remarkable performance of DBFS.

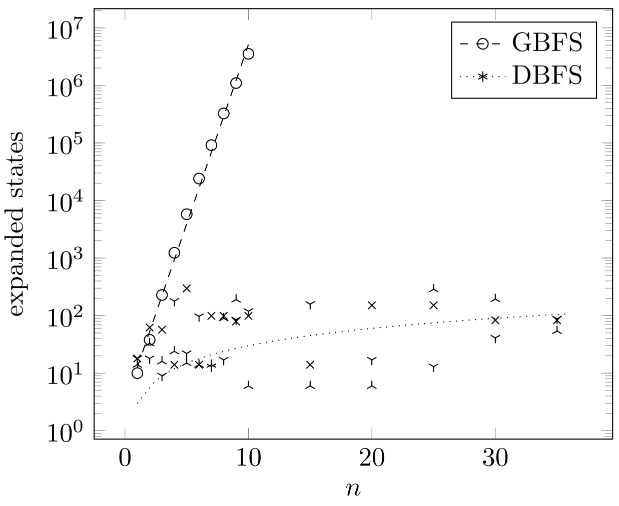


Figure 4.4: Comparison of GBFS and DBFS on our example Schedule problem. For both algorithms we plotted an approximation. For GBFS that is the function $3 \cdot 4.2^n$ used in Figure 4.2 – for DBFS we used $3n$.
As an approximation to h^+ we used h^{FF}. Notice that in the case of GBFS, the number of expansions is the same for h^{FF} and $h^{\text{LM-cut}}$.

There are several reasons why DBFS exits the local plateau so fast. One important aspect of the problem is that the possible heuristic values are very limited – the maximum possible value for h^+ is 4 – doing a time-step and achieving every property individually. This results in DBFS doing short local searches and fetching many nodes using Algorithm 4. Additionally, exit states ($h^+(s) = 3$) have a short distance to the plateau in terms of heuristic value. Thus, DBFS has a high chance to expand one of the unpromising exit states.
To investigate whether DBFS dominates GBFS on the provided Schedule problems as well,

we conducted experiments and plotted the results in Figure 4.5 as scatter plots. One can notice very clearly that DBFS solves more difficult problem instances with less expansions than GBFS. Additionally, DBFS was able to solve between 87% and 95% instances, whereas GBFS solved only 35%.

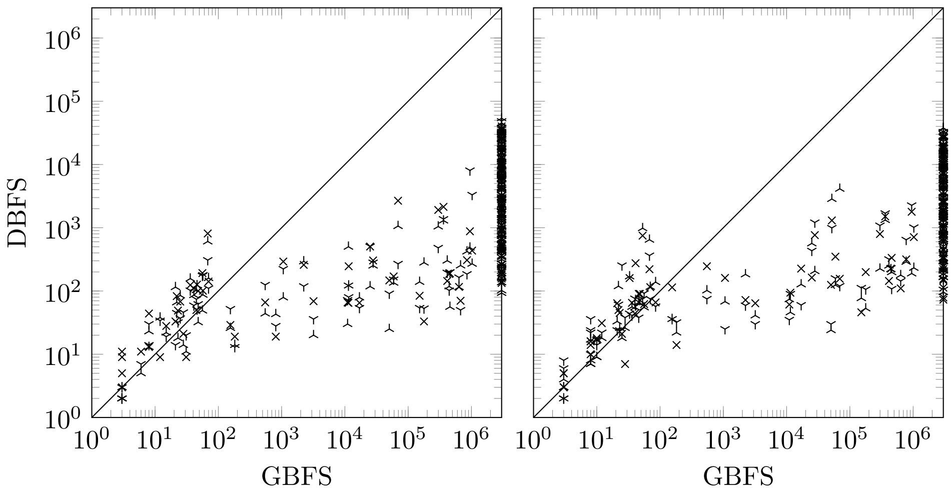


Figure 4.5: Scatter plot empirically comparing the number of expanded states of GBFS and DBFS on the Schedule problems of IPC-2. The left subfigure shows the values using h^{FF}, the right using $h^{\text{LM-cut}}$. All data points that appear on the right axis indicate that GBFS was not able to solve the instance.

5 Conclusion and Future Work

In this thesis we discussed the search algorithms *greedy* and *diverse best-first search*. We have given theoretical lower bounds on the scaling of GBFS with h^+ on artificial problems on the Pathways and Schedule domain and verified these bounds empirically with GBFS and DBFS. Additionally we evaluated GBFS and DBFS on the provided problems on Pathways and Schedule and concluded that DBFS outperforms GBFS on both our artificial and the provided problems.

Possible future work includes extending the theoretical analysis of the introduced artificial problems on DBFS and providing both lower bounds as well as expected values for DBFS. Further extending the theory, one could construct state space models suited for GBFS and DBFS. These models would enable the model-specific evaluation of expected costs, expansions and runtime of GBFS and DBFS. The main goal behind this theoretical analysis would be to find theoretical verified properties and reasons on why DBFS outperforms GBFS in terms of expansions and coverage.

Having such results at hand, one could try to extend DBFS and/or construct other algorithms that improve the performance of GBFS. Such work could also try to include additional algorithms in the studies, for example *enforced hill-climbing* (Hoffmann and Nebel [2001]) or *K-best-first search* (Felner et al. [2003]).

References

Fahiem Bacchus. AIPS'00 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *AI Magazine*, 22(3):47–56, 2001.

Christoph Betz and Malte Helmert. Planning with h^+ in theory and practice. In *Proceedings of the Second Workshop on Heuristics for Domain-independent Planning at ICAPS 2009*, 2009.

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

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

Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Additive-disjunctive heuristics for optimal planning. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 44–51. AAAI Press, 2008.

Minh Binh Do, Malte Helmert, and Ioannis Refanidis. Evaluation schema – satisficing tracks, 2008. URL `http://ipc.informatik.uni-freiburg.de/EvaluationSatisficing`. Accessed: 20. May 2014.

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

Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

Alfonso E. Gerevini, Derek Long, Patrik Haslum, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

Patrik Haslum, Blai Bonet, and Héctor Geffner. New admissible heuristics for domain-independent planning. In Manuela Veloso and Subbarao Kambhampati, editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1163–1168. AAAI Press, 2005.

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

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 162–169. AAAI Press, 2009.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 176–183. AAAI Press, 2007.

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

Tatsuya Imai and Akihiro Kishimoto. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pages 985–991. AAAI Press, 2011.

Michael Katz, Jörg Hoffmann, and Carmel Domshlak. Who said we need to relax *all* variables? In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, pages 126–134. AAAI Press, 2013.

Emil Keyder and Héctor Geffner. Heuristics for planning with action costs revisited. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence*, pages 588–592. IOS Press, 2008.

Emil Keyder and Héctor Geffner. Trees of shortest paths vs. steiner trees: Understanding and improving delete relaxation heuristics. In Craig Boutilier, editor, *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, pages 1734–1739. International Joint Conferences on Artificial Intelligence, 2009.

Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12: 339–386, 2000.

Levi H. S. Lelis, Sandra Zilles, and Robert C. Holte. Stratified tree search: A novel suboptimal heuristic search algorithm. In Takayuki Ito, Catholijn Jonker, Maria Gini, and Onn Shehory, editors, *Proceedings of the 2013 International Conference on Autonomous Agents & Multiagent Systems*, pages 555–562. IFAAMAS, 2013.

Carlos Linares López and Daniel Borrajo. Adding diversity to classical heuristic planning. In Ariel Felner and Nathan Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search*, pages 73–80. AAAI Press, 2010.

Qiang Lu, You Xu, Ruoyun Huang, and Yixin Chen. The Roamer planner. In *The 2011 International Planning Competition*, pages 73–76, 2011.

Claudio Marxer. An algorithm for avoiding plateaus in heuristic search. Bachelor thesis, Universität Basel, Departement Mathematik und Informatik, 2013.

Vitaly Mirkis and Carmel Domshlak. Cost-sharing approximations for h^+. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 240–247. AAAI Press, 2007.

Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 273–280. AAAI Press, 2009.

Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz, editors, *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pages 246–249. AAAI Press, 2010.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

Herbert A. Simon. *Administrative Behavior: a Study of Decision-Making Processes in Administrative Organization*. Macmillan, 2nd edition, 1957.

Richard Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, and Fan Xie. A comparison of knowledge-based GBFS enhancements and knowledge-free exploration. In Steve Chien, Alan Fern, Wheeler Ruml, and Minh Do, editors, *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*. AAAI Press, 2014. To appear.

Fan Xie, Martin Müller, and Robert Holte. Adding local exploration to greedy best-first search in satisficing planning. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence*. AAAI Press, 2014a. To appear.

Fan Xie, Martin Müller, Robert Holte, and Tatsuya Imai. Type-based exploration with multiple search queues for satisficing planning. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence*. AAAI Press, 2014b. To appear.

A Appendix

A.1 Random Seeds used by DBFS

marker	**seed**
⅄	19074890
Y	874709
×	12278773

Table A.1: The three random seeds we used for conducting our experiments with DBFS.

A.2 Experimental Results of Domain Analysis

A.2.1 Pathways

			expansions	
$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{N}_s\rvert$	$\lvert\mathcal{P}(\mathcal{N}_s)\rvert$	**GBFS**	∅ **DBFS**
5	1	2	10	11.00
7	2	4	16	30.33
9	3	8	27	45.33
11	4	16	68	69.00
13	5	32	168	60.33
15	6	64	555	160.67
17	7	128	1755	247.00
19	8	256	6286	633.67
21	7	512	19512	621.33
23	10	1024	67821	1758.00
25	11	2048	221754	2826.00
27	12	4096	808896	3655.67
29	13	8192	2567689	4198.33
31	14	16384	8749064	13354.67
33	15	32768	—	12761.67
35	16	65536	—	28287.67
37	17	131072	—	48900.67
39	18	262144	—	—

Table A.2: Measured number of expansions by GBFS and DBFS with our theoretical bound $|\mathcal{P}(\mathcal{N}_s)|$ on our example Pathways problem. DBFS' expansions are shown as the average over all three different random seeds. Figure 3.3 and 3.4 use this data.

A.2.2 Schedule

		expansions	
n	$\lvert S_p \rvert$	**GBFS**	**∅ DBFS**
1	1	10	16.66
2	7	38	37.66
3	28	228	27.33
4	84	1235	72.33
5	210	5772	111.33
6	462	24019	41.66
7	924	91507	42.00
8	1716	325527	68.66
9	3003	1096423	118.00
10	5005	3531655	74.66
15	38760	—	59.33
20	177100	—	58.00
25	593775	—	150.33
30	1623160	—	106.00
35	3838380	—	75.00

Table A.3: Measured number of expansions by GBFS and DBFS in relation to our theoretical lower bound $|S_p|$ of our our example Schedule problem. As in Table A.2, expansions by DBFS are shown as the average number over all three seeds. Figure 4.2 and 4.4 use this data.

A.3 PDDL Definitions

The following section contains the planning task definition in the *Planning Domain Definition Language* (PDDL). In practice, PDDL is used as the standard formalism to describe planning tasks. PDDL describes planning tasks more compact in comparison to STRIPS as it uses limited predicate instead of propositional logic.

Important to notice is that in contrast to STRIPS, PDDL allows negative preconditions. In order to transform such preconditions into a STRIPS equivalent, one needs to introduce a new predicate for all predicates that appear as negative in preconditions and goals. These predicates model the exact opposite of the original predicate. For example in Pathways, the action *choose* requires that the molecule is possible and not chosen. As chosen appears negative, we introduce another predicate not-chosen. In all preconditions and goals where not chosen appears, we replace it with not-chosen. All effects that change the predicate chosen need to change the inverse predicate not-chosen accordingly. Additionally, we need to extend the initial state of problems to explicitly list all molecules as not-chosen which are not already chosen.

We omitted this transformation for the Schedule domain to retain the readability of the PDDL definition.

A.3.1 Pathways

```
; IPC5 Domain: Pathways Propositional
; Authors: Yannis Dimopoulos, Alfonso Gerevini and Alessandro Saetti

(define (domain Pathways-Propositional)
(:requirements :typing :adl)

(:types
        level molecule - object
        simple complex - molecule)

(:predicates
        (association-reaction ?x1 ?x2 - molecule ?x3 - complex)
        (catalyzed-association-reaction ?x1 ?x2 - molecule ?x3 - complex)
        (synthesis-reaction ?x1 ?x2 - molecule)
        (possible ?x - molecule)
        (available ?x - molecule)
        (chosen ?s - simple)
        (not-chosen ?s - simple)
        (next ?l1 ?l2 - level)
        (num-subs ?l - level))

(:action choose
 :parameters (?x - simple ?l1 ?l2 - level)
 :precondition (and (possible ?x) (not-chosen ?x)
                    (num-subs ?l2) (next ?l1 ?l2))
 :effect (and (chosen ?x) (not (not-chosen ?x))
              (not (num-subs ?l2)) (num-subs ?l1)))

(:action initialize
  :parameters (?x - simple)
  :precondition (and (chosen ?x))
  :effect (and (available ?x)))

(:action associate
 :parameters (?x1 ?x2 - molecule ?x3 - complex)
 :precondition (and (association-reaction ?x1  ?x2  ?x3)
                    (available ?x1) (available ?x2))
 :effect (and (not (available ?x1)) (not (available ?x2)) (available ?x3)))

(:action associate-with-catalyze
 :parameters (?x1 ?x2 - molecule ?x3 - complex)
 :precondition (and (catalyzed-association-reaction ?x1 ?x2 ?x3)
                    (available ?x1) (available ?x2))
 :effect (and (not (available ?x1)) (available ?x3)))

(:action synthesize
 :parameters (?x1 ?x2 - molecule)
 :precondition (and (synthesis-reaction ?x1 ?x2) (available ?x1))
 :effect (and (available ?x2)))
)
```

Listing A.1: PDDL definition of Pathways

A.3.2 Schedule

```
;; Schedule World

(define (domain schedule)
  (:requirements :adl :typing)
  (:types temperature-type ashape surface machine part colour)
  (:constants cold hot - temperature-type
             cylindrical - ashape
             polisher roller lathe grinder spray-painter immersion-painter - machine
             polished rough smooth - surface)

  (:predicates (temperature ?obj - part ?temp - temperature-type)
               (available ?machine - machine)
               (busy ?machine - machine)
               (scheduled ?obj - part)
               (objscheduled)
               (surface-condition ?obj - part ?surface-cond - surface)
               (shape ?obj - part ?shape - ashape)
               (painted ?obj - part ?colour - colour)
               (has-paint ?machine - machine ?colour - colour))

  (:action do-polish
   :parameters (?x - part)
   :precondition (and (available polisher) (not (busy polisher))
                      (not (scheduled ?x)) (temperature ?x cold))
   :effect (and (busy polisher) (scheduled ?x)
                (surface-condition ?x polished)
                (when (not (objscheduled))
                    (objscheduled))
                (forall (?oldsurface - surface)
                    (when (and (surface-condition ?x ?oldsurface)
                               (not (= ?oldsurface polished)))
                        (not (surface-condition ?x ?oldsurface))))))

  (:action do-roll
   :parameters (?x - part)
   :precondition (and (available roller) (not (busy roller))
                      (not (scheduled ?x)))
   :effect (and (busy roller) (scheduled ?x)
                (temperature ?x hot) (not (temperature ?x cold))
                (shape ?x cylindrical)
                (when (not (objscheduled))
                    (objscheduled))
                (forall (?oldsurface - surface)
                    (not (surface-condition ?x ?oldsurface)))
                (forall (?oldpaint - colour)
                    (not (painted ?x ?oldpaint)))
                (forall (?oldshape - ashape)
                    (when (and (shape ?x ?oldshape)
                               (not (= ?oldshape cylindrical)))
                        (not (shape ?x ?oldshape)))))

  (:action do-lathe
   :parameters (?x - part)
   :precondition (and (available lathe) (not (busy lathe))
                      (not (scheduled ?x)))
   :effect (and (busy lathe) (scheduled ?x)
                (surface-condition ?x rough) (shape ?x cylindrical)
                (when (not (objscheduled))
                    (objscheduled))
                (forall (?oldshape - ashape)
                    (when (and (shape ?x ?oldshape)
                               (not (= ?oldshape cylindrical)))
                        (not (shape ?x ?oldshape))))
                (forall (?oldsurface - surface)
                    (when (and (surface-condition ?x ?oldsurface)
                               (not (= ?oldsurface rough)))
                        (not (surface-condition ?x ?oldsurface))))
                (forall (?oldpaint - colour)
                    (not (painted ?x ?oldpaint)))))

  (:action do-grind
   :parameters (?x - part)
   :precondition (and (available grinder) (not (busy grinder))
                      (not (scheduled ?x)))
   :effect (and (busy grinder) (scheduled ?x)
                (surface-condition ?x smooth)
                (when (not (objscheduled))
                    (objscheduled))
                (forall (?oldsurface - surface)
                    (when (and (surface-condition ?x ?oldsurface)
                               (not (= ?oldsurface smooth)))
                        (not (surface-condition ?x ?oldsurface))))
                (forall (?oldpaint - colour)
                    (not (painted ?x ?oldpaint)))))

  (:action do-spray-paint
   :parameters (?x - part ?newpaint - colour)
   :precondition (and (available spray-painter) (has-paint spray-painter ?newpaint)
                      (not (busy spray-painter)) (not (scheduled ?x))
                      (temperature ?x cold))
   :effect (and (busy spray-painter) (scheduled ?x)
                (painted ?x ?newpaint)
                (when (not (objscheduled))
                    (objscheduled))
                (forall (?oldsurface - surface)
                    (not (surface-condition ?x ?oldsurface)))
                (forall (?oldpaint - colour)
                    (when (and (painted ?x ?oldpaint)
                               (not (= ?oldpaint ?newpaint)))
                        (not (painted ?x ?oldpaint))))))

  (:action do-immersion-paint
   :parameters (?x - part ?newpaint - colour)
   :precondition (and (available immersion-painter) (has-paint immersion-painter ?newpaint)
                      (not (busy immersion-painter)) (not (scheduled ?x)))
   :effect (and (busy immersion-painter) (scheduled ?x)
                  (painted ?x ?newpaint)
                  (when (not (objscheduled))
                      (objscheduled))
                  (forall (?oldpaint - colour)
                      (when (and (painted ?x ?oldpaint)
                                 (not (= ?oldpaint ?newpaint)))
                          (not (painted ?x ?oldpaint))))))

    (:action do-time-step
     :parameters ()
     :precondition (objscheduled)
     :effect (and (forall (?x - part)
                      (not (scheduled ?x)))
                  (forall (?m - machine)
                      (not (busy ?m)))))

  )
```

Listing A.2: PDDL definition of Schedule