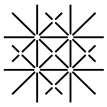

University of Basel

Depth-Bound Heuristics and Iterative-Deepening Search Algorithms in Classical Planning

Bachelor 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: Silvan Sievers

Florian Spiess
f.spiess@unibas.ch
14-058-994

09.06.2017

Acknowledgments

To pretend this thesis was the product of myself alone could not be farther from the truth. This thesis would not even have been possible without the help of so many other people. Unfortunately I will not be able to mention everyone who has helped me come this far and write this thesis by name, but be assured I am grateful nonetheless. I would like to thank Prof. Dr. Malte Helmert for giving me the opportunity to write this thesis. I would especially like to thank my supervisor Silvan Sievers for his great advice, support and his patience throughout the entire process of this thesis. Finally, I would also like to thank my friends and family who have motivated me to work hard and do my best even when I did not immediately see a solution to a problem.

Abstract

Heuristics play an important role in classical planning. Using heuristics during state space search often reduces the time required to find a solution, but constructing heuristics and using them to calculate heuristic values takes time, reducing this benefit. Constructing heuristics and calculating heuristic values as quickly as possible is very important to the effectiveness of a heuristic.

In this thesis we introduce methods to bound the construction of merge-and-shrink to reduce its construction time and increase its accuracy for small problems and to bound the heuristic calculation of landmark cut to reduce heuristic value calculation time. To evaluate the performance of these depth-bound heuristics we have implemented them in the Fast Downward planning system together with three iterative-deepening heuristic search algorithms: iterative-deepening A^* search, a new breadth-first iterative-deepening version of A^* search and iterative-deepening breadth-first heuristic search.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 Goal 2

2 Background **3**
2.1 Classical Planning 3
2.2 Heuristics 4
2.2.1 Merge-and-Shrink Heuristic 4
2.2.2 Landmark Cut Heuristic 5
2.3 Search Algorithms 6
2.3.1 A* 7
2.3.2 Iterative-Deepening A* 7
2.3.3 Breadth-First Heuristic Search (BFHS) 8

3 Implementation **11**
3.1 Depth-Bound Merge and Shrink 11
3.2 Depth-Bound Landmark Cut 12
3.3 Iterative-Deepening A* 12
3.4 Iterative-Deepening Breadth-First A* 13
3.5 Iterative-Deepening Breadth-First Heuristic Search 13

4 Evaluation **14**
4.1 Summary Results 14
4.2 IDA* 16
4.2.1 Merge-and-Shrink 16
4.2.2 Landmark Cut 18
4.3 IDBFA* 18
4.4 IDBFHS 19
4.5 Discussion 20

5 Conclusion **21**

5.1 Results Overview 21
5.2 Future Work 21

Bibliography **23**

Declaration on Scientific Integrity **24**

1 Introduction

1.1 Motivation

In our lives, we, as humans, encounter many different problems. These problems often vary greatly in nature of the desired goal and abilities that can be applied to arrive at such a goal. Most of these problems would be easily solved if only we had a plan, a series of actions, that would take us from our current situation to the desired destination. This is exactly the scenario for which classical planning was developed.

In this thesis, we concern ourselves only with classical planning problems. Classical planning is a type of automated planning of which the main goal is to find any, but often specifically an optimal solution for a search problem with a finite state space in a generic representation. More specifically, classical planning deals only with general problem solving methods, which can be applied to any deterministic, fully observable problem in such a representation.

Classical planning problems can be described by an agent attempting to reach one of many possible goal states from an initial state by transitioning through other states via a set of allowed transitions. Depending on the allowed transitions, a state space can be represented as an undirected graph, a directed graph or a tree. The goal of planning is to find an ordered sequence of these transitions or actions, which leads from the initial state to a goal state.

Although planners that are run on computers can find optimal plans to problems a lot faster than humans could, the speed and efficiency of planning algorithms is still an area of active development. This is, because as problems increase in size and the number of possible actions in each situation increases, the number of possibilities that need to be considered increases drastically. To alleviate this problem, a variety of techniques has been developed. One of the most widely used approaches is heuristic search.

Heuristics are functions that attempt to approximate the cost of an optimal plan from the current state to a goal state. Heuristics can be specific to a certain problem or task, but there are also those heuristics which can be applied to a more general problem representation and can therefore be used for any problem in such a representation. While specific and general heuristics each have their applications, since we cannot assume any problem specific features from the general representation of a problem, only general heuristics are of interest in classical planning. Heuristics allow search algorithms to judge how promising a state is and, using this information, choose to expand those states which are most likely on the optimal

path. Search algorithms can also use this cost estimate to disregard states completely if their projected cost is higher than a certain threshold. This is called search space pruning and is of central importance to many search algorithms. Ignoring these costly states can yield a substantial increase in performance, because large amounts of suboptimal or even dead end paths would otherwise be explored. On the other hand, depending on the heuristic used and the pruning strategy implemented, it is possible that pruning removes a state on the optimal path, in which case either a suboptimal solution is found or none at all.
Heuristics often create an abstract representation of the state space of the problem. This internal representation is usually a highly simplified version of the actual state space, because problem state spaces are usually too large to work on directly. By considering a simplified version of the problem state space, heuristics can approximate the cost of paths to a goal state without having to directly work with the much larger actual state space. However, since simplifying the state space too much will result in inaccurate heuristic values, many heuristics are designed to retain only aspects of a state space considered to be important for the optimal solution.

1.2 Goal

In this thesis, we modify two heuristics, the merge-and-shrink [6] as well as the landmark cut [5] heuristic, to only use internal representations of the state space up to a given depth bound. This should increase the speed at which the abstraction is constructed or the heuristic value is generated, but make these heuristics useless for estimating heuristic values for states only reachable by greater cost than the given bound. To avoid running into states with undefined heuristic values and to complement the depth-bound nature of these heuristics, we also implement three search algorithms designed for iterative deepening searches: iterative-deepening depth-first A* search (IDA*) [7], iterative-deepening breadth-first A* search (IDBFA*) and iterative-deepening breadth-first heuristic search (IDBFHS) [12].
We use these heuristics to explore the effectiveness of bounding the depth of construction of heuristic state space representations in increasing heuristic accuracy, reducing heuristic construction time and decreasing the overall time required to solve problems. We specifically analyze the viability of this approach in conjunction with the mentioned iterative-deepening, depth-bound search algorithms, to ensure states pruned from the internal representation of the heuristics are also pruned from the respective iterations of the search.

2
Background

In this section we will introduce the concepts used in our implementation.

2.1 Classical Planning

Classical planning is concerned with finding a path through a state space. State spaces are defined as follows:

Definition 2.1.1. A state space is a 6-tuple $\mathcal{S} = \langle S, A, cost, T, s_0, S_\star \rangle$ where S is a finite set of states, A is a finite set of actions, $cost \in A \to \mathbb{R}_0^+$ maps actions to their costs, $T \subseteq S \times A \times S$ is the set of allowed state transitions, s_0 is the initial state and $S_\star \subseteq S$ is a set of goal states.

In most realistic problems the state space in question is too large to be worked on directly as a graph or even to be entered into a computer by hand. To compactly describe planning problem domains, planning formalisms have been developed which represent state spaces not directly as states and edges, but as rules from which the original search space can be generated. We consider planning tasks in the SAS$^+$ formalism [1], borrowing the notation from Sievers et al. [11].

Definition 2.1.2. A planning task is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where $\mathcal{V}$ is a finite set of state variables, $\mathcal{O}$ is a finite state of operators, s_0 is the initial state and $s_\star$ is the goal. Each variable $v \in \mathcal{V}$ has a finite domain $\mathcal{D}(v)$. A partial state is a variable assignment on a subset of $\mathcal{V}$ denoted as $vars(s)$. We write $s[v]$ for the variable assignment to $v \in vars(s)$, which must satisfy the condition $s \in \mathcal{D}(v)$. A state s complies with partial state s' if $s[v] = s'[v]$ for all $v \in vars(s) \cap vars(s')$. A partial state s is a is a state if $vars(s) = \mathcal{V}$. Each operator $o \in \mathcal{O}$ has a precondition $pre(o)$ and effect $\mathit{eff}(o)$, which are partial states and a cost $c(o) \in \mathbb{R}_0^+$. An operator o is applicable in a state s if s complies with $pre(o)$, in which case o can be applied, resulting in the successor state s' that complies with $\mathit{eff}(o)$ and satisfies $s'[v] = s[v]$ for all $v \notin vars(\mathit{eff}(o))$. The initial state s_0 is a state; the goal $s_\star$ is a partial state.
A plan is a sequence $o_1, ..., o_n \in \mathcal{O}$ of operators which are applicable, in order, to the initial state, resulting in a state that complies with the goal. Such a plan is optimal if $\sum_{i=1}^{n} c(o)$ is

minimal among all plans. The objective of optimal planning is to find an optimal plan for a planning task or prove that no plan exists.

Planning formalisms are used to separate the task of defining a problem in such a way that a planner can understand it from the task of writing code to solve such a problem. A planning formalism often used in practice is the planning domain definition language (PDDL)[8]. For the purpose of testing the concepts investigated in this thesis, we use PDDL planning problems from the Fast Downward benchmark collection, which contains benchmark instances from International Planning Competitions.[1]

2.2 Heuristics

Heuristics are functions used in best-first search algorithms and even some breadth- and depth-first searches. Let S be the set of states of a search space. The heuristic function h is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$, which approximates the perfect heuristic h^* that returns the cost of an optimal plan from a state s to some goal state. Heuristics can have different properties, some of which are necessary for certain search algorithms to find correct solutions. The following properties are of particular interest to us:

Definition 2.2.1. A heuristic is *goal aware* if $h(s) = 0$ for all $s \in S_\star$. A heuristic can also be *safe* if $h^*(s) = \infty$ for all s where $h(s) = \infty$, admissible if $h(s) \leq h^*(s)$ for all $s \in S$ and consistent if $h(s) \leq cost(a) + h(s')$ for all $s \xrightarrow{a} s' \in T$. [10]

All search algorithms examined in this thesis rely on heuristics and require admissible heuristics to ensure they will find optimal plans. This is why all heuristics we modify in this thesis are admissible heuristics.

2.2.1 Merge-and-Shrink Heuristic

One of the heuristics we modify in this thesis is the merge-and-shrink heuristic originally developed by Dräger et al. [2] for model checking and adapted for classical planning by Helmert et al. [6]. While other heuristics directly compute a cost estimate from the evaluated state and the goal state description, the merge-and-shrink heuristic computes an abstraction of the entire state space and calculates the true cost of the states in the abstraction.
The merge-and-shrink heuristic makes use of the synchronized product, which combines the information of two abstract state spaces into one. The synchronized product $\otimes$ of two state spaces is defined as follows:

Definition 2.2.2. For $i \in \{1, 2\}$, let $\mathcal{S} = \langle S^i, A, cost, T^i, s_0^i, S_\star^i \rangle$ be state spaces with equal sets of actions and costs. The synchronized product of $\mathcal{S}^1$ and $\mathcal{S}^2$, denoted by $\mathcal{S}^1 \otimes \mathcal{S}^2$, is the state space $\mathcal{S}^\otimes = \langle S^\otimes, A, cost, T^\otimes, s_0^\otimes, S_\star^\otimes \rangle$ with $S^\otimes := S^1 \times S^2$, $T^\otimes := \{\langle \langle s_1, s_2 \rangle, a, \langle t_1, t_2 \rangle \rangle \mid \langle s_1, a, t_1 \rangle \in T^1 \wedge \langle s_2, a, t_2 \rangle \in T^2\}$, $s_0^\otimes := \langle s_0^1, s_0^2 \rangle$ and $S_\star^\otimes := S_\star^1 \times S_\star^2$.

Merge-and-shrink also uses abstractions which map one state space to another. A state space abstraction α is defined as follows:

[1] https://bitbucket.org/aibasel/downward-benchmarks

Definition 2.2.3. A state space abstraction α is a surjective function $\alpha : S \to S'$ which maps every state in the set of states of the original state space S to a state from the set of states of the resultant abstract state space S'. The abstract state space $\mathcal{S}^\alpha$ induced by the abstraction α is $\mathcal{S}^\alpha = \langle S', A, \mathit{cost}, T', s'_0, S'_\star \rangle$ with: $T' = \{\langle \alpha(s), a, \alpha(t) \rangle \mid \langle s, a, t \rangle \in T\}$, $s'_0 = \alpha(s_0)$ and $S'_\star = \{\alpha(s) \mid s \in S_\star\}$.

When a merge-and-shrink heuristic is computed the initial step is to calculate the projections of the original state space onto each state variable $v \in \mathcal{V}$. A projection onto a state variable v_i is an abstraction $\alpha(s) = \{v_i \mapsto d_i\}$ where d_i is the value of v_i in the state s, which preserves only information about a single state variable. The abstract state space used in the resultant merge-and-shrink heuristic is generated by merging and shrinking these state space projections until only one abstract state space remains. Two state spaces are merged using the synchronized product. The order in which abstract state spaces are merged is called the merge strategy and is a configurable parameter of the merge-and-shrink heuristic construction. If the resultant size of the merge of two abstract state spaces would be too large, a shrink operation is applied to them. A shrink operation is a state space abstraction aiming to reduce the size of the state space while keeping as much information about the state space as possible. The exact abstraction used in the shrink operation is called the shrink strategy and is another parameter of the merge-and-shrink heuristic construction. Another step during the construction is label reduction. Label reduction reduces the number of transition labels of an abstract search space by merging similar labels.
Additionally, the abstract state spaces are pruned between merge operations. During the merge-and-shrink heuristic construction, all states s which are not reachable from the initial state ($g(s) = \infty$ where $g : s \to \mathbb{R}_0^+ \cup \{\infty\}, s \in S$ returns the cost of the path to reach s) or are on no path to a goal state ($h(s) = \infty$) are removed in this step. This decreases the size of the abstract state space, while only removing states which cannot be on an optimal path from the initial state to a goal state. However, this means that a generated merge-and-shrink heuristic, while it can be used like an admissible heuristic for states reachable from the initial state, is not technically admissible, because states s with $h^*(s) \neq \infty$ but $g(s) = \infty$ will have $h(s) = \infty$.

2.2.2 Landmark Cut Heuristic

Disjunctive action landmarks [5] in a planning task are sets of actions for which every solution must contain at least one action from each. The cost of each landmark is the cost of the cheapest action in the set. Landmarks of a given state space can be computed by building a justification graph.

Definition 2.2.4. A STRIPS planning task in normal form is a tuple $\Pi = \langle V, I, G, A \rangle$ where V is a finite set of propositions, $I = \{i\}$ is the initial state containing only a single proposition $i \in V$, $G = \{g\}$ is the goal state containing only a single proposition $g \in V$ and A is a finite set of actions a with cost $\mathit{cost}(a) \in \mathbb{N}_0$, preconditions $\mathit{pre}(a) \subseteq V$, add effects $\mathit{add}(a) \subseteq V$ and delete effects $\mathit{del}(a) \subseteq V$. The delete-free version of a STRIPS planning task $\Pi = \langle V, I, G, A \rangle$ is the task $\Pi^+ = \langle V, I, G, A \rangle$ where $\mathit{del}(a) = \emptyset$ for all actions $a \in A$.

A precondition choice function $P : A \mapsto V$ is a function that maps every action $a \in A$ to one of its preconditions $v \in add(a)$. The justification graph induced by the precondition choice function P is a labeled, directed graph where the vertices are the variables $v \in V$ and the edges are $P(a) \xrightarrow{a} e$ for every action $a \in A$ and effect $e \in add(a)$. Justification graphs only consider the delete-free representation of a planning task.

A landmark can then be computed by selecting all actions along a straight line or "cut" across all paths between the initial node and the goal node in a justification graph.
The landmark cut heuristic (LM-cut), developed by Helmert and Domshlak [5], is defined as follows:

Definition 2.2.5. Initialize $h^{\text{LM-cut}}(I) := 0$ of the initial state I

1. Compute $h^{\max}$ values for all variables. Stop if $h^{\max}(G) = 0$ of the goal state G.
2. Compute the justification graph for the precondition choice function that chooses preconditions with maximal $h^{\max}$ value.
3. Compute a cut for which $cost(L) > 0$ for the corresponding landmark L.
4. Increase $h^{\text{LM-cut}}(I)$ by $cost(L)$.
5. Decrease $cost(a)$ by $cost(L)$ for each action in the landmark $a \in L$.
6. Repeat from step 1.

$h^{\max}$ is an easily computed heuristic for relaxed planning graphs, the details of which will not be explored further as they are not important to understanding this thesis.

2.3 Search Algorithms

There exists a wide variety of different search algorithms for state space exploration and planning, many of which differ greatly from one another. However there are some structures and components that occur quite commonly in one way or another.
One such structure is the open list. Open lists contain the search nodes a search algorithm has yet to expand. In many cases, the open list also determines in what order these search nodes are retrieved from it and thereby specifies the expansion order of nodes. As an example, an open list might simply be a queue for breadth-first search algorithms or a stack for depth-first search algorithms.
Another structure often found in search algorithms is the closed list. A closed list contains a node for each state already expanded by the search. It is usually implemented as some form of hash map to allow direct access to each node. This fast access allows the closed list to not only store node information but also to be used for duplicate detection by checking if a state already exists as an expanded search node.
A commonly used concept in graph search algorithms is node reopening, which makes use of the previously discussed closed list. Algorithms supporting reopening not only check if a state has already been expanded as a node, but also if the new node has a lower path cost than the old node, in which case the old node is removed from the closed list and the new node is inserted into the open list.

2.3.1 A*

One of the most famous and most widely used search algorithms is A* search developed by Hart et al. [3]. It belongs to the category of best-first searches, meaning it relies on the calculated heuristic value to determine which node will most likely lead to a goal state the quickest and uses this to determine which node to expand next. To do this, A* search makes use of a priority queue as an open list. As priority value, A* uses the sum of the cost of the search node and the heuristic value of the associated state ($f(n) = g(n) + h(s)$ where $g : n \to \mathbb{R}_0^+$ returns the cost of the path on which the search node n was reached). As such, the open list of A* is ordered by an estimate of the cost of an optimal path from the initial state through the considered node to the closest goal node. This also means that A* will only expand nodes on the optimal path if the heuristic is perfect and f-value ties are broken by preferring lower h-values.
Hart et al. [3] define the A* algorithm as follows:

1. Mark s_0 as "open" and calculate $f(s_0)$.
2. Select the open node n whose f-value is smallest. Resolve ties favoring $n \in S_\star$.
3. If $n \in S_\star$, mark n "closed" and terminate the algorithm.
4. Otherwise, mark n closed and calculate the f-value of all successors. Mark all successor nodes "open" which aren't already marked closed. Mark any successor as open if it was previously marked as closed and if the new f-value is lower than the f-value of the node when it was closed. Go to step 2.

As shown by Hart et al. [3], A* is guaranteed to find an optimal plan or show that none exist either when a consistent and admissible heuristic is used, or when an admissible heuristic is used with node reopening. The authors have proven the former of these claims by proving that the f-values of nodes expanded by A* are non-decreasing when a consistent and admissible heuristic is used [3]. Because of this property, the order in which nodes are expanded by A* resembles layers of f-depths.

2.3.2 Iterative-Deepening A*

As a best-first search, A* may expand large amounts of nodes during a search, especially when the accuracy of the heuristic is low. Because A* stores all nodes previously encountered during the search, it can consume very large amounts of memory in unfavorable circumstances. The memory growth with solution depth d and problem branching factor b can be as bad as $\mathcal{O}(b^d)$ in case the heuristic is constant and the search degenerates into a pure breadth-first search [7].
A search algorithm that avoids this memory problem is iterative deepening depth-first search (IDDFS). IDDFS is a tree search that finds optimal plans for unit-cost problems by recursively searching a state space up to a certain depth and iteratively restarting the search with an increased maximum depth. Due to being a tree search however, IDDFS only has a spatial complexity of $\mathcal{O}(d)$, because only node information for the states on the currently investigated path are kept in memory [7]. Korf [7] has combined this approach with the idea behind A* search into a new search algorithm known as iterative-deepening A* search

(IDA*). IDA* search works in the same way as IDDFS, except that it uses f-value bounds, instead of depth bounds to restrict the individual iterative searches. The evaluation function of IDA* is the same as the one described for A*. As a result, every iteration the f-bound is deepened to the lowest f-value encountered during the previous iteration which is larger than the f-bound of that iteration [7].
Korf [7] shows that IDA* inherits the property of A* of always finding a cheapest solution path if an admissible heuristic is used, while also gaining the spatial efficiency of IDDFS. Furthermore, the author concludes that IDA* search expands the same number of nodes during its last iteration as A* search, however, with a far lower maximum memory usage.

2.3.3 Breadth-First Heuristic Search (BFHS)

During state space search, the further the closest goal state is to the initial state, the exponentially more search nodes not on the optimal path will be explored along the way. While this is dependent on the search algorithm used, it can cause a search to fail due to a lack of memory. Even widely used search algorithms such as A* suffer from this problem when searching large state spaces. During a search of A*, not only the search frontier in the open list needs to be stored, but also all search nodes previously explored during the search. This is necessary to be able to trace a path back to the initial node. To determine if a goal exists and at what depth it is not necessary to store all previously encountered search nodes. Ideally, only the nodes in the search frontier needs to be stored at all.
This is the consideration behind breadth-first heuristic search (BFHS) developed by Zhou and Hansen [12]. As the name suggests, BFHS is in essence a breadth-first search. This means it expands nodes simply in the order in which they are put into the open list. Because of this, the expansion of nodes by BFHS can be separated into layers of nodes at different distances from the initial node in the search graph. In short, each layer is the set of successor nodes of the previous layer. To reduce the peak memory usage during search, the authors show that it is possible to delete previously explored layers and still find a solution. However, most graph searches also store all previously expanded nodes for duplicate detection. If previously explored layers of nodes are removed from memory, it is impossible to tell if a node has already been visited during the search. To avoid the search frontier growing much faster in size than it normally would, Zhou and Hansen [12] show that due to the simple, layered expansion order of BFHS, it is sufficient for undirected graph search problems to store the previous search frontier layer as a closed list to prevent duplicate node expansion. This concept can be seen in Figure 2.1 which shows how under the described conditions the search frontier is separated from the already explored states in the interior of the graph by the boundary, which is the search frontier of the previous layer. However, due to these requirements and assumptions, BFHS, in its original form, only finds optimal plans for unit-cost problems and only completely avoids expanding duplicates for undirected graph state spaces.
Since it is impossible to trace back a path from the initial state to a found goal state if all intermediate search nodes are deleted from memory, Zhou and Hansen [12] describe a divide-and-conquer approach to solution reconstruction, which only requires one additional search

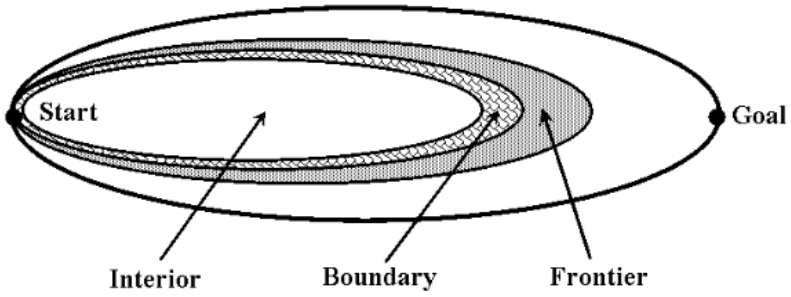

Figure 2.1: Visualization of breadth-first search by Zhou and Hansen [12]. The set of boundary nodes separates the search frontier from the interior of the search graph. All nodes inside the boundary are closed and all nodes outside the boundary are open while the boundary may contain both open and closed nodes.

node layer to be saved during the search. Using this approach BFHS can reconstruct the solution path through recursive subsearches once a goal node and some intermediate node on an optimal path to this goal node are known. The solution path found is reconstructed by performing two smaller searches for each successful search. One of these searches finds a path from the initial node to the intermediate node of the parent search, while the second search finds a path from the intermediate node to the goal node of the parent search. When the goal node is a direct successor of the initial node for any of these searches, the action required to transition from the initial to the goal node is returned. In this way, all actions required to reach the solution to the initial problem are individually found. Using this method of recursive subsearches, only the initial node, an intermediate layer, the search frontier and the search frontier of the previous layer need to be kept in memory at any one time. All other nodes no longer need to be stored. However, for the solution to be successfully reconstructed, all nodes expanded after the intermediate layer no longer need to contain a reference to their parent node, as this node will be deleted in the next layer anyways, but to the node in the intermediate layer on the path to this node. This allows the algorithm to determine which node in the intermediate layer is on the optimal path to the solution when a solution is found [12].

Because pure breadth-first search expands in all directions uniformly, which can lead to very large search frontiers with increasing length of the optimal solution, Zhou and Hansen [12] use pruning based on f-value in BFHS. BFHS uses the same function as A*. This method of pruning is reasonable, because it specifically disregards search nodes projected to be on a path costlier than the expected optimal solution. The authors have shown that, if BFHS is started with a pruning bound equal to the cost of the optimal path, BFHS does not expand more nodes before finding a goal than A* search on the same problem [12].

Since this method of pruning only allows solutions to be found up to the specified bound, pure BFHS cannot guarantee finding an optimal solution if one exists. Simply choosing a very large pruning bound does not solve this problem because the solution might still be at a higher cost and otherwise the bound might be set so much higher than the actual optimal path cost, that no nodes are pruned. To alleviate this problem, Zhou and Hansen [12] suggest a version of BFHS which incrementally increases the pruning bound when no solution is found. Due to its similarity to IDA* search, the authors call this iterative

deepening version of BFHS breadth-first iterative-deepening A* search [12].
In this thesis, we will call this iterative version iterative-deepening breadth-first heuristic search (IDBFHS) because we also implement a version of breadth-first A* modified to conduct iterative, f-value bound searches, which we call iterative-deepening breadth-first A* (IDBFA*) and which is more closely related to A* than IDBFHS.

3
Implementation

All search algorithms and heuristic modifications examined in this thesis were implemented in the Fast Downward planner[2] developed by Helmert [4].

3.1 Depth-Bound Merge and Shrink

We modified the merge-and-shrink construction to prune states with a higher cost than a given bound during construction of the merge-and-shrink heuristic. We achieved this by modifying the pruning step of the merge-and-shrink construction process. The merge-and-shrink construction uses Dijkstra's algorithm to determine the g- and h-value of each state in the abstract state spaces. Because the merge-and-shrink abstraction produces an admissible heuristic we know that the cost of any state in one of the abstracted state spaces is smaller or equal to the cost of the equivalent state in the actual state space. Therefore, if the cost of a state in any abstract state space is greater than the given bound, we can be sure that the cost of the state in the problem state space is greater, which is why we can remove it entirely from the abstraction.

To accomplish this, we modified the pruning step of merge-and-shrink to not only prune a state s if $h(s) = \infty$ or $g(s) = \infty$ but also when $h(s) + g(s) > f$-bound where f-bound is the bound to which the merge-and-shrink heuristic is generated. By pruning these states, the size of the individual abstract state spaces is reduced, which means fewer shrink operations are needed to stay within the merge-and-shrink abstraction size bound. Because of this, states up to the f-bound will be more accurately represented in the final merge-and-shrink abstraction since the pruned states do not take up any space. Another benefit to this approach are lower construction times, because states pruned are disregarded for the rest of the heuristic construction.

However, bounding a merge-and-shrink heuristic has the drawback that it can only be used for states with an f-value up to this f-bound since the heuristic value for all states with a higher f-value is undefined. Due to this limitation, depth-bound merge-and-shrink is best used with an iterative-deepening search algorithm. Many of these algorithms de-

[2] http://www.fast-downward.org

termine their next bound during the search by picking $\min_n(f(n))$ where n is any search node encountered and $f(n) > f$-bound. However, because all f-values for nodes with $g(n) + h^*(n) > f\text{-}bound$ have undefined $h(n)$ when using a depth-bound merge-and-shrink heuristic bounded to the f-bound, no new f-bound can be determined during the search. To make depth-bound merge-and-shrink compatible with iterative-deepening search algorithms, depth-bound merge-and-shrink stores $\min_s(f(s))$ for all pruned states s with $f(s) > f$-bound. Since a constructed depth-bound merge-and-shrink heuristic is only useful up to the construction f-bound, iterative-deepening searches need to be modified to construct a new depth-bound merge-and-shrink heuristic before every iteration.

3.2 Depth-Bound Landmark Cut

Because landmark cut does not generate and store the cost of all states in the state space during initialization, but rather calculates the heuristic value of a state only when it is needed by the search, we modified the heuristic calculation of landmark cut instead of the initialization. We implement an additional stop condition during the first step of landmark cut calculation. Instead of only stopping the heuristic calculation once $h^{\max}(G) = 0$ for the goal state G, the heuristic calculation is also stopped when $h^{\text{LM-cut}}(n) > f$-bound for the evaluated node n. When heuristic calculation is aborted due to violating the depth-bound, ∞ is returned instead of the calculated heuristic value.
Because of the way we bound the landmark cut heuristic, $g(n)$ of the node n and the f-bound need to be passed to the heuristic when evaluating $h^{\text{LM-cut}}(n)$. To make depth-bound landmark cut compatible with iterative-deepening search algorithms, we store $\min_s(f(s))$ for all states s where $f(s) > f$-bound in the heuristic object as the next possible f-bound.

3.3 Iterative-Deepening A*

We implemented IDA* as described by Korf [7]. An optimization we implemented, that was already described by Korf [7], is that we allow the initial bound to be determined by a heuristic of choice. Using an admissible heuristic to determine the initial bound of the search allows the search to skip all iterations up to this bound while still ensuring that any solution found is optimal. An optimization we have implemented specifically for the use with depth-bound merge-and-shrink is that we do not generate a new depth-bound merge-and-shrink heuristic if no states were pruned because $h(s) + g(s) > f$-bound during the last construction, because the resulting heuristic would be the same. Furthermore, to allow the use of our depth-bound heuristics, we modified IDA* so that if no goal was found during the iteration, a new iteration is started with the bound determined by the heuristic, if a depth-bound heuristic is used.
A major difference between our implementation of IDA* and the version described by Korf [7] is our use of a closed list, duplicate detection and reopening. Because of the technical challenges involved in generating successor states in Fast Downward without the use of a closed list, we decided to use a closed list in our IDA* implementation. While this means our IDA* implementation does not have a significant memory advantage over A*, we take

full advantage of the closed list by including duplicate detection and reopening, reducing the search time while still guaranteeing any solution found will be optimal.

3.4 Iterative-Deepening Breadth-First A*

As a comparison to IDA* we also implemented an iterative-deepening version of A* based on the A* implementation in Fast Downward. We call this new algorithm iterative-deepening breadth-first A* (IDBFA*).

In comparison to the classical A* described by Hart et al. [3] IDBFA* conducts iterative searches, of which each iteration is an A* search where all nodes with higher f-value than the f-bound of that iteration are not inserted into the open list. When depth-bound merge-and-shrink is used and no more states were pruned during generation of the heuristic, IDBFA* conducts an unbounded A* search.

3.5 Iterative-Deepening Breadth-First Heuristic Search

We implemented IDBFHS as described by Zhou and Hansen [12]. To be able to use IDBFHS like normal BFHS we also implemented an option to manually specify the bound and to turn off iterative search. Because the merge-and-shrink heuristic can only be used for the initial state and goal description it was generated with, we construct a new merge-and-shrink heuristic for each recursive solution-reconstruction search. Since we know the exact goal distance in these solution-reconstruction sub-problems, we construct the new depth-bound merge-and-shrink heuristics with the lowest possible depth-bound for best performance.

4 Evaluation

To be able to compare the implemented algorithms and heuristics fairly, we conducted our experiments on 57 planning domains containing 1667 planning tasks from the optimal sequential track of all International Planning Competitions up to 2014. IDBFHS was only evaluated on a subset of these problems containing 160 planning tasks from six unit-cost and undirected graph domains. Table 4.1 shows the 10 different configurations of algorithms and heuristics we tested. For every problem, each algorithm configuration was run with a time constraint of 30 minutes and a memory constraint of 2 GB. The experiments were performed on Intel Xeon E5-2660 CPUs running at 2.2 GHz.

Algorithm Name	Description
A* ms	Standard A* search with merge-and-shrink heuristic.
A* lmcut	Standard A* search with landmark cut heuristic.
IDA* ms	IDA* search with merge-and-shrink heuristic.
IDA* lmcut	IDA* search with landmark cut heuristic.
IDA* dbms	IDA* search with depth-bound merge-and-shrink heuristic.
IDA* dblmcut	IDA* search with depth-bound landmark cut heuristic.
IDBFA* dbms	IDBFA* search with depth-bound merge-and-shrink heuristic.
IDBFA* dblmcut	IDBFA* search with depth-bound landmark cut heuristic.
IDBFHS dbms	IDBFHS search with depth-bound merge-and-shrink heuristic.
IDBFHS dblmcut	IDBFHS search with depth-bound landmark cut heuristic.

Table 4.1: Evaluated algorithm configurations.

The parameters used for merge-and-shrink are the default Fast Downward parameters for all algorithms evaluated with merge-and-shrink heuristics. The shrink strategy used is shrink bisimulation [9]. The merge strategy used is DFP [11]. For label reduction, exact generalized label reduction was used. The maximum number of states was set to 50'000, always allowing perfect shrinking.

4.1 Summary Results

The summary of results collected is shown in Table 4.2. Bold values are the best in their category.

	A*		IDA*				IDBFA*	
	ms	**lmcut**	**ms**	**lmcut**	**dbms**	**dblmcut**	**dbms**	**dblmcut**
Coverage	745	**882**	725	848	721	833	728	840
Expansions	1822.21	**1301.20**	3939.90	3088.52	2587.65	3113.72	2389.86	3079.64
Memory	63368336	21006000	53595072	9802372	52926128	**9409960**	60730232	20403740
Search time	**0.13**	0.60	0.22	1.12	4.46	1.28	4.76	1.33
Total time	2.01	**0.65**	2.68	1.22	4.53	1.40	5.07	1.45

Table 4.2: Summary of experiment results.

The summary values for *coverage* and *memory* are calculated as the sum of the problem-wise results, while the values for *expansions*, *search time* and *total time* are calculated as the geometric mean over commonly solved tasks. *Coverage* is defined as the number of problems solved by an algorithm, where a problem is solved if a plan is found within the time and memory constraints. *Expansions* is the number of search nodes expanded during the search. For iterative-deepening search algorithms, this includes the number of expansions of all iterations, not only the final one. For IDBFHS this also includes the number of nodes expanded during the solution reconstruction searches. The value for *memory* is the peak memory allocated during the search. *Search time* is the time in seconds required to solve the problem after the initial heuristic generation but including the construction time of depth-bound heuristics, whereas *total time* is the time in seconds required for the entire run. The summary in Table 4.2 shows the results for the entire benchmark set, which is why results for IDBFHS are not shown.

The results show, that algorithm configurations using a landmark cut heuristic generally show better results than those using a merge-and-shrink heuristic. This was expected, since, as has been shown by other authors in previous work, landmark cut provides more accurate heuristic results and requires less *total time* for most problems. Peak memory usage also shows an expected advantage of landmark cut over merge-and-shrink, since merge-and-shrink stores an abstract state space representation while landmark cut does not. In *search time*, standard merge-and-shrink has a great advantage over landmark cut, since all heuristic values for merge-and-shrink are stored in the abstract state space representation, while landmark cut needs to calculate heuristic values during the search. The *total time* taken by landmark cut configurations is much lower than for merge-and-shrink searches, because landmark cut does not take as long to initialize. Unsurprisingly, the measured *search time* for depth-bound merge-and-shrink configurations is much larger than of the unbound equivalent, since the *search time* measured includes the time taken during the search to construct new depth-bound merge-and-shrink heuristics. Of the individual algorithms, A* search performed the best, which was also expected since A* search has been shown to find a solution faster than IDA* search in most cases and since it is not an iterative-deepening search, it expands far fewer search nodes during a search. As can be seen well from the *coverage* of the different configurations, IDBFA* search performed better than IDA* search when comparing only depth-bound heuristic configurations. This is most likely the case because A* search, even with the overhead of iterative deepening searches performs better than IDA* search in the final iteration, since the expansion order of A* search in the last f-layer can be controlled with a tie-breaking strategy, while IDA* search must expand nodes in the order of the tree search.

4.2 IDA*

An individual comparison of the results of IDA* with standard merge-and-shrink against depth-bound merge-and-shrink and standard landmark cut against depth-bound landmark cut can be seen in table 4.3. This table includes a new attribute: the geometric mean of the *real search time*. The *real search time* is the pure search time, disregarding time spent constructing new depth-bound merge-and-shrink heuristics. *Real search time* is smaller than *search time* even for configurations not using depth-bound merge-and-shrink, because *search time* includes the time required for the initialization of the search algorithm, part of which is the generation of the closed list, while *real search time* does not.

	Merge-and-Shrink			Landmark Cut		
	Standard	**Depth-bound**	**Difference**	**Standard**	**Depth-bound**	**Difference**
Coverage	725	721	-4	**848**	833	-15
Expansions	4252.10	**2790.90**	-1461.2	3259.94	3286.78	26.84
Memory	62302616	61688396	-614220	12920584	**12326636**	-593948
Real search time	0.05	**0.03**	-0.01	0.68	0.72	0.04
Search time	**0.24**	4.62	4.38	1.20	1.37	0.17
Total time	2.79	4.69	1.9	**1.30**	1.49	0.19

Table 4.3: Comparison of standard against depth-bound merge-and-shrink and standard against depth-bound landmark cut for IDA*.

4.2.1 Merge-and-Shrink

As can be seen in Table 4.3, IDA* with standard merge-and-shrink was able to solve 4 more problems than IDA* with depth-bound merge-and-shrink. This is most likely because IDA* with depth-bound merge-and-shrink takes more *total time* in the geometric mean and therefore more easily violates the time constraint. The results show that IDA* expanded about 34% fewer nodes in the geometric mean with depth-bound in comparison to standard merge-and-shrink. This is quite interesting, since it shows that depth-bound merge-and-shrink is in some cases more informative than standard merge-and-shrink. This confirms that there is an advantage to using depth-bound merge-and-shrink over standard merge-and-shrink. The difference in *real search time* also seems to support this theory, since the geometric mean of the *real search time* is lower for depth-bound merge-and-shrink. The difference in peak memory for depth-bound and standard merge-and-shrink in Table 4.3 is not particularly large. While the pruning of the merge-and-shrink abstraction might reduce the memory requirements for very low depth-bounds, the difference in peak memory required is probably this small because the space freed by pruning during the merge-and-shrink abstraction process is used to make the resulting abstraction more accurate. The geometric means of *search time* and *total time* are clearly better with standard merge-and-shrink instead of depth-bound merge-and-shrink with IDA* search, which can be explained by the overhead of constructing a new merge-and-shrink abstraction before every iteration. A detailed plot showing the comparison of *expansions* between depth-bound and standard merge-and-shrink can be seen in Figure 4.1. For the majority of problems the number of expanded nodes is very similar between IDA* with standard and depth-bound merge-and-shrink. This becomes especially clear for problems requiring large numbers of *expansions* to solve. However, while there are some problems for which IDA* with depth-bound merge-

and-shrink expanded more search nodes than IDA* with standard merge-and-shrink, for many problems that require few *expansions* to solve, the opposite is the case. What this shows is that depth-bound merge-and-shrink is only more informative for problems that do not require many *expansions* to solve. This is most probably the case because depth-bound merge-and-shrink construction only increases the accuracy of the resulting heuristic if states in the abstract state spaces are pruned. For larger problems, where no more states can be pruned from the depth-bound merge-and-shrink heuristic during construction, the resultant heuristic will be identical to an unbound merge-and-shrink heuristic and therefore expand the same number of states.

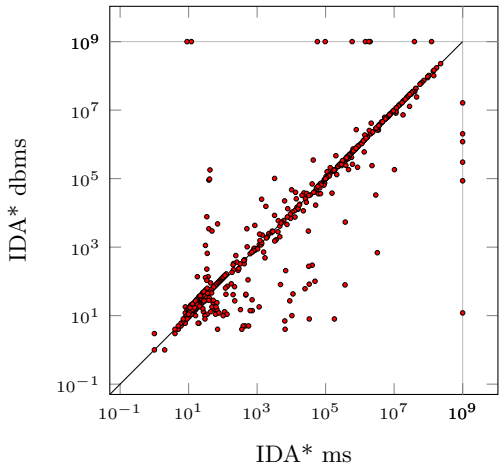

Figure 4.1: Comparison of expansions for IDA* search with standard against depth-bound merge-and-shrink

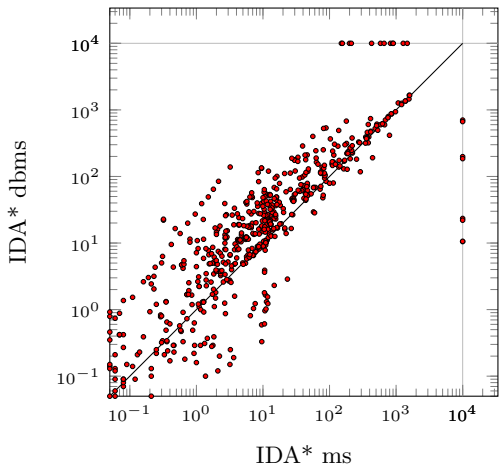

Figure 4.2: Comparison of total time for IDA* search with standard against depth-bound merge-and-shrink

A plot comparing the *total time* for depth-bound against standard merge-and-shrink with IDA* search can be seen in Figure 4.2. As expected, the *total time* taken for most problems is larger for IDA* with depth-bound merge-and-shrink than for IDA* with standard merge-

and-shrink. This is most probably due to the large time overhead of constructing a new merge-and-shrink heuristic before every iteration. What is interesting is that while there are only very few problems where IDA* was faster with depth-bound merge-and-shrink than with standard merge-and-shrink and IDA* with standard merge-and-shrink took longer than 10^2 seconds, there were quite a few problems where IDA* was faster with depth-bound merge-and-shrink than with standard merge-and-shrink and IDA* with standard merge-and-shrink took fewer than 10^2 seconds. This shows that while the overhead of creating a new merge-and-shrink abstraction every iteration results in a large overhead, for some quickly solved problems the increased accuracy and lower construction time make up enough time to overcome the overhead.

4.2.2 Landmark Cut

Table 4.3 shows that IDA* with standard landmark cut heuristic solved 15 more problems than depth-bound landmark cut. This is most likely due to the increased *search time* of depth-bound landmark cut. The difference in peak memory between depth-bound landmark cut configurations and their unbound equivalents is rather unexpected, since the depth-bound in landmark cut is only enforced by stopping a calculation, not by pruning an abstract state space. A possible explanation for this is that due to technical details of the implementation of depth-bound landmark cut, the heuristic values are not cached for searches with the depth-bound version, which would not only explain the lower peak memory, but also the minor increase in *search time* of depth-bound landmark cut in comparison to standard landmark cut. Unfortunately this means that the time gained by stopping calculation of the heuristic value early with landmark cut is not enough to negate this implementation specific overhead. If there is a time gain in aborting heuristic calculation when the calculated value is greater than a bound, it is not particularly significant.

4.3 IDBFA*

Table 4.4 shows comparison results of A* with merge-and-shrink and landmark cut heuristics against IDBFA* with depth-bound versions of the same heuristics.

	Merge-and-Shrink			Landmark Cut		
	A*	**IDBFA***	**Difference**	**A***	**IDBFA***	**Difference**
Coverage	745	728	-17.0	**882**	840	-42.0
Expansions	2572.05	3670.80	1098.75	**1566.76**	3715.71	2148.95
Memory	81405464	78403932	-3001532.0	21751332	**21101688**	-649644.0
Search time	**0.17**	5.67	5.5	0.74	1.66	0.92
Total time	2.36	6.03	3.67	**0.80**	1.80	0.99

Table 4.4: Comparison of A* search with standard merge-and-shrink and landmark cut against IDBFA* search with depth-bound merge-and-shrink and landmark cut.

The results show that IDBFA* with depth-bound merge-and-shrink or landmark cut solves fewer problems than A* with the equivalent unbounded heuristic. This is most likely because of the time overhead resulting from performing iterative A* searches, in comparison to a single one, which causes IDBFA* to violate the time constraint. The results also show that the geometric mean of *expansions* for standard heuristics with A* search is far lower than

the geometric mean of *expansions* for the equivalent depth-bound heuristic with IDBFA* search. This is expected, because *expansions* for iterative-deepening algorithms includes the expansions during all iterations. IDBFA* with depth-bound merge-and-shrink becomes an unbound A* search once no more states are pruned during construction of the merge-and-shrink abstraction, while IDBFA* with depth-bound landmark cut continues to iterate. Because of this, the relative difference in geometric mean of *expansions* of IDBFA* to A* with the equivalent unbound heuristic is greater for landmark cut than for merge-and-shrink. Furthermore, Figure 4.3 shows that the number of *expansions* for most tasks is greater for IDBFA* with depth-bound merge-and-shrink than A* with merge-and-shrink. However, there are some tasks that require relatively low numbers of *expansions* with IDBFA*, where IDBFA* requires fewer *expansions* than A* despite the *expansions* overhead due to iterative-deepening. This is a similar result as can be observed for IDA* with standard and depth-bound merge-and-shrink and is most likely due to the increased accuracy of depth-bound merge-and-shrink for relatively easy problems. The peak memory usage of IDBFA* search with depth-bound heuristics is slightly lower than that of A* search with the equivalent unbound heuristic, however not significantly. The *search time* and *total time* results for IDBFA* and A* show a trend similar to the results for *expansions*.

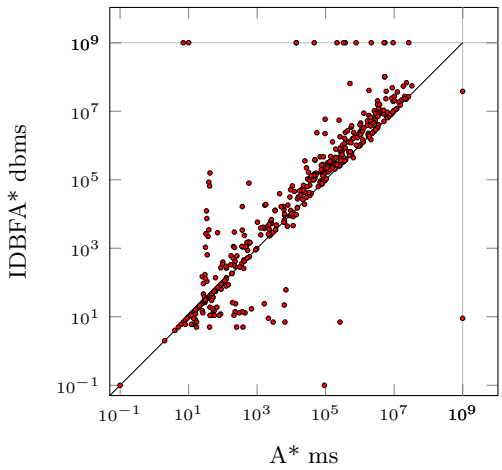

Figure 4.3: Comparison of expansions for A* search with standard merge-and-shrink against IDBFA* search with depth-bound merge-and-shrink

4.4 IDBFHS

The results for the experiments of IDBFHS in comparison to A* on problems from six unit-cost and undirected graph domains can be seen in Table 4.5.

Most of the results displayed in Table 4.5 are very similar to the results observed for IDBFA*. IDBFHS does not have as high a *coverage* as A*, expands more search nodes in the geometric mean due to iterations and solution reconstruction steps and takes longer both in *search time* and *total time*. What is surprising, is that IDBFHS has a higher peak memory than A*. Since IDBFHS is designed to use as little memory as possible during a search, this is

	A*		IDBFHS	
	Merge-and-Shrink	**Landmark Cut**	**Merge-and-Shrink**	**Landmark Cut**
Coverage	**88**	82	75	80
Expansions	2184.86	**2020.73**	23929.42	11388.37
Memory	6320500	**1032548**	9927308	1518924
Search time	**0.14**	0.50	4.39	1.79
Total time	1.30	**0.52**	4.43	1.81

Table 4.5: Experiment results of A* with merge-and-shrink and landmark cut heuristics and IDBFHS with depth-bound merge-and-shrink and landmark cut heuristics for problems from six unit-cost and undirected graph domains.

	Number	A*		IDBFHS	
Domain	of tasks	**Merge-and-Shrink**	**Landmark Cut**	**Merge-and-Shrink**	**Landmark Cut**
Blocks	24	1.95	**0.13**	8.46	0.30
Depot	6	10.45	**3.84**	59.51	10.09
Driverlog	12	1.50	**0.31**	4.16	1.09
Gripper	6	**0.02**	0.68	0.43	2.31
Logistics00	20	1.02	**0.48**	1.80	1.61
Logistics98	5	6.39	**0.37**	4.62	2.86
Total	73	1.30	**0.52**	4.43	1.81

Table 4.6: Total time results of A* with merge-and-shrink and landmark cut heuristics and IDBFHS with depth-bound merge-and-shrink and landmark cut heuristics by domain.

rather unfortunate and probably means that, due to the involved implementation of closed lists in Fast Downward, search node information is not correctly removed from memory in our implementation.

A more detailed summary of *total time* by domain for IDFHS and A* search can be seen in Table 4.6. Most domains show a clear advantage of A* search over BFHS. Curiously, there is one domain where this is not the case. For the planning domain Logistics98 IDBFHS with depth-bound merge-and-shrink achieves a lower total time than A* search with standard merge-and-shrink. Since only five tasks in this domain were solved by all four algorithm configurations this could simply be an outlier, but if it is not, this could show that the combination of IDBFHS with depth-bound merge-and-shrink can find solutions faster than A* search with merge-and-shrink for some domains.

4.5 Discussion

The experiments have shown that depth-bound heuristics tend to take longer during searches than their unbound equivalents. This is particularly apparent for depth-bound merge-and-shrink, where most of the time during a run is spent on generating new merge-and-shrink heuristics, as can be seen from the comparison of real search time to search time. The experiments also show that iterative-deepening searches with depth-bound merge-and-shrink can reduce the number of expansions necessary to find a solution. This observation can be attributed to the increased accuracy of depth-bound merge-and-shrink for problems where states are still pruned from the abstract state space due to the depth-bound during heuristic construction in the final iteration.

5
Conclusion

In this chapter we summarize our findings and reach a conclusion based on our results. We also suggest further areas of research related to the topic of this thesis which could be explored in future work.

5.1 Results Overview

We introduced two depth-bound heuristics based on the merge-and-shrink heuristic and the landmark cut heuristic. The strategy implemented to bound the merge-and-shrink heuristic is based on pruning by f-value, whereas the strategy to bound landmark cut stops heuristic calculation once an f-bound has been exceeded. We also introduced a breadth-first iterative-deepening version of A* search making use of the A* algorithm but also using concepts of IDA* search.

To test our depth-bound heuristics, we implemented three depth-bound iterative-deepening search algorithms: iterative-deepening A*, iterative-deepening breadth-first A* and iterative-deepening breadth-first heuristic search. We have shown that in general, depth-bound heuristics increase time required to solve a problem with iterative-deepening algorithms. However, we have also shown that depth-bound merge-and-shrink can provide more accurate heuristic values than unbound merge-and-shrink for some problems.

5.2 Future Work

The main drawback of using depth-bound merge-and-shrink to its unbound equivalent is the large time overhead of having to generate a new heuristic every iteration of iterative-deepening search, even for larger problems where the final heuristic might be equivalent to an unbound merge-and-shrink heuristic. This problem could be approached in various ways in future work. One potential solution would be to allow the algorithm to determine at the beginning of the search whether to use depth-bound merge-and-shrink or standard merge-and-shrink. Since depth-bound merge-and-shrink only shows an advantage over standard merge-and-shrink for problems with a low solution depth and search space complexity, the algorithm should only choose depth-bound merge-and-shrink in those cases. A second

approach to reducing the overhead of generating a new heuristic every iteration would be to only increase heuristic bounds in larger steps. By increasing the depth-bound of the heuristic by a multiple of the lowest action cost rather than just raising it to the next iteration depth the same heuristic could be used for several iterations, while only sacrificing some potential accuracy during some of the iterations. However, since each new f-bound layer could contain exponentially more search nodes than the previous, this might cause depth-bound merge-and-shrink to lose its size and accuracy advantages for easy problems. Both of these solution approaches could even be combined.

Bibliography

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

[2] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. *Directed Model Checking with Distance-Preserving Abstractions*, pages 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-33103-2. doi: 10.1007/11691617_2.

[3] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[4] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26: 191–246, 2006.

[5] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *19th International Conference on Automated Planning and Scheduling (ICAPS-09)*, 2009.

[6] Malte Helmert, Patrik Haslum, Jörg Hoffmann, et al. Flexible abstraction heuristics for optimal sequential planning. In *17th International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 176–183, 2007.

[7] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, September 1985.

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

[9] Raz Nissim, Jörg Hoffmann, and Malte Helmert. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 2011.

[10] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc.,Reading, MA, Jan 1984.

[11] Silvan Sievers, Martin Wehrle, and Malte Helmert. Generalized label reduction for merge-and-shrink heuristics. In *AAAI*, pages 2358–2366, 2014.

[12] Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. In *14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, June 2004.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Florian Spiess

Matriculation number — Matrikelnummer
14-058-994

Title of work — Titel der Arbeit
Depth-Bound Heuristics and Iterative-Deepening Search Algorithms in Classical Planning

Type of work — Typ der Arbeit
Bachelor Thesis

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

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

Basel, 09.06.2017

Signature — Unterschrift