Implementation and Evaluation of Depth-First IBEX in Fast Downward

Bachelor's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Remo Christen

Petr Sabovčik
petr.sabovcik@stud.unibas.ch
2021-062-310

22.07.2024

Acknowledgments

Much thanks to my supervisor Remo Christen, who made writing this thesis a fun and rewarding experience.

Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing core facility at University of Basel.

Abstract

Budgeted Tree Search (BTS) is a depth-first version of the search algorithm framework Iterative Budgeted Exponential Search (IBEX). It aims to improve the worst case run time of Iterative Deepening A* (IDA*), a widely used search algorithm when memory is an issue. BTS seeks to remedy IDA*'s shortcomings while maintaining the same space complexity. A weakness of IDA* is that under certain circumstances each iteration spends a considerable amount of effort exploring a minimal portion of the state space in addition to what it explored in earlier iterations. A main component of BTS is its addition of the exponential search procedure, which forces the search to expand exponentially more nodes with each iteration. We implement BTS and evaluate its performance in Fast Downward, a classical planning system, and compare it to IDA* with the use of the International Planning Competition (IPC) benchmark suite.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

2 Background **3**

2.1 State Space . . . 3

2.2 Black Box Interface . . . 3

2.3 Search Algorithms . . . 4

2.3.1 Depth-First Search . . . 4

2.3.2 Informed and Uninformed Search . . . 5

2.3.3 Heuristics . . . 5

2.3.4 g-value . . . 5

2.3.5 f-value . . . 5

2.3.6 A* . . . 6

2.3.7 Iterative Deepening A* . . . 6

2.4 Classical Planning . . . 7

2.4.1 PDDL . . . 8

2.4.2 Fast Downward . . . 9

3 Depth-First IBEX **10**

3.1 Exponential Search . . . 11

3.2 Global Variables . . . 11

3.3 Algorithm Overview . . . 12

3.4 Algorithm Properties . . . 14

4 Implementation **15**

4.1 Unregistered States . . . 15

4.2 Estimate Caching . . . 16

4.3 Path Checking . . . 16

5 Evaluation **17**

5.1 Coverage . . . 18

5.2 Exponential Search . . . 19

5.3 Errors 19
5.3.1 Stack Overflows 20
5.3.2 Out of Memory and Out of Time Errors 21
5.4 Expansions until last jump 21
5.5 Iterations 21
5.6 Memory 23
5.7 Runtime 23

6 Conclusion 24

Bibliography 25

1 Introduction

Whether winning a game of chess or planning deliveries of millions of packages is at stake, search algorithms play a pivotal role in learning how to solve these tasks in an optimal way. In the field of artificial intelligence, search algorithms are used to navigate complex state spaces to find solutions to such problems. It is to be expected then, that this field is constantly innovating, seeking to find new and better strategies to solve important problems.

The two attributes of search algorithms that are most often the target of improvements are their time and space complexity. Iterative Budgeted Exponential Search (IBEX) is a novel search algorithm framework [9], the depth-first version of which aims to improve the worst case run time of an algorithm used when memory is an issue - Iterative Deepening A* (IDA*) [10]. Depth-first IBEX intends to remedy IDA*'s shortcomings, while maintaining the same space complexity [14].

Throughout this thesis, we will refer to the depth-first version of IBEX as Budgeted Tree Search (BTS). We consider the depth-first version of IBEX as it has a more straightforward implementation and is more suitable for the environment of Fast Downward [6]. Additionally, we have an already existing well-established search algorithm (IDA*) to compare it against. The paper *A Guide to Budgeted Tree Search* [14] will be the model for our implementation of BTS.

IDA*'s shortcomings are caused by its linear nature, which can lead to its iterations only exploring a very small portion of the state space (in addition to what it explored in earlier iterations) under certain circumstances. The way BTS seeks to bypass this issue is by employing exponential search, enforcing the search to expand exponentially more nodes with each iteration.

The goal of this thesis is to document the implementation and performance of BTS in Fast Downward, a classical planning system providing an environment that allows for the implementation and testing of search algorithms. We will examine BTS's performance using some well-known planning problems featured in past International Planning Competitions (IPC) and compare it to the performance of IDA* and A* [4], also examining the difference a simple path checking optimization makes on the performance of BTS and IDA*.

As Fast Downward has built-in duplicate checking mechanisms and favors non-iterative-deepening search algorithms, we will also delve into its inner workings, such as its state registry and heuristic estimate caching that can cause issues when trying to implement algorithms like BTS.

In the next chapter, we will discuss the concepts IBEX builds upon, the basics of classical planning, Fast Downward's main components, and the way it encodes planning tasks. For a complete context, we will also discuss the workings of IDA*. In later chapters, we will detail BTS's functioning, its implementation in Fast Downward, and the results of our performance analysis.

2
Background

First, we introduce the formal definitions of state spaces, the underlying search algorithms relevant to our evaluation of BTS, classical planning, the Planning Domain Definition Language (PDDL) format used by Fast Downward, and Fast Downward itself.

2.1 State Space

State spaces describe an environment in terms of states, actions, and transitions between states. A state represents a configuration of the environment, while actions define how states can be modified. Transitions explicitly describe how one state can be transformed into another.

A formal definition of a state space is a tuple $S = \langle S, A, cost, T, s_I, S_G \rangle$ where:

- S is a finite set of states.
- A is a finite set of actions.
- $cost$ is a cost function that assigns a non-negative cost to each action.
- T is a transition function that maps states and actions to successor states.
- s_I is the initial state.
- S_G is a set of goal states.

The state spaces we are interested in are solved by heuristic tree search with an admissible heuristic. What these properties entail is explored later in this chapter.

2.2 Black Box Interface

While search algorithms may differ in the ways they navigate the search space, the algorithms we are interested in share a common interface they use to find solutions to planning tasks. To be satisfactory for our purposes, the black box must be able to provide the following functions:

- **get_initial_state()**: Returns the initial state of the planning task.
- **is_goal_state(state)**: Returns whether the given state is a goal state.
- **get_successors(state)**: Returns a list of successor states of the given state along with the action that, upon application, results in the given successor state.
- **get_cost(operator)**: Returns the cost of the given operator.
- **get_evaluator_value(state)**: Returns the heuristic value of the given state.

Using these functions, search algorithms do not require an explicit representation of the entire state space, but can instead rely on this interface to provide them with the information needed to find plans. The Fast Downward planning system also provides a similar interface, as discussed later in this chapter.

2.3 Search Algorithms

Search algorithms navigate state spaces, with the aim of finding plans. Each algorithm has its own strategy on how to explore this space, and each has its own strengths and weaknesses. Their shared purpose is to find a plan from the initial state to a goal state.

In the following section, we will discuss heuristics and how search algorithms use them to guide their exploration of the state space, also examining search algorithms that do not use heuristics.

2.3.1 Depth-First Search

Depth-First Search is a search algorithm that forgoes the use of heuristics and instead simply recursively explores the state space reaching deeper and deeper before backtracking. As can be expected, this algorithm is not optimal and might not even terminate if it gets stuck in a cycle. It has a time complexity of $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the state space. Typically, it is implemented as a tree search algorithm.

Depth-First search can be implemented using the following pseudocode:

Algorithm 1 Depth-First Search (DFS)

```
1: if is_goal_state(state)
2:    return SOLVED
3: successors ← get_successors(state)
4: for ⟨successor, operator⟩ in successors
5:    solutionPath.push(operator)
6:    if DFS(successor) == SOLVED
7:       return SOLVED
8:    solutionPath.pop()
```

Although it has certain applications, it is generally inadvisable to use in larger state spaces.

2.3.2 Informed and Uninformed Search

Search algorithms can be split into informed and uninformed search algorithms. Uninformed search algorithms, such as Depth-First search, simply explore the state space without any additional information. They do not prefer any states over others and do not take into account the cost to reach states. As can be expected, this runs the risk of exploring the state space inefficiently and finding suboptimal solutions.

In contrast, informed search algorithms use heuristics to guide their exploration of the state space. Heuristics, as explained in the following, help the search algorithm prioritize states that are more likely to lead to the goal state with a lower cost. Provided the heuristic has certain properties, the plans found by informed search algorithms that are relevant to this thesis are guaranteed to be optimal.

2.3.3 Heuristics

Heuristics are functions that estimate the cost to reach a goal state from a given state. They are used by informed search algorithms to guide the search process by prioritizing states that are likely to lead to the goal. While depth-first search does not utilize heuristics, both BTS and IDA* do, making them informed search algorithms

The heuristics we are interested in are admissible heuristics. These are heuristics that never overestimate the cost to reach the goal state, in formal terms $h(s) \leq h^*(s)$, where $h^*(s)$ is the perfect heuristic. The perfect heuristic $h^*(s)$ is the least-cost path from the state s to the closest goal state. If no such path exists, then $h^*(s) = \infty$. It should also be noted that a heuristic h always outputs non-negative values or infinity.

Admissible heuristics are essential for informed search algorithms, as the plans found by A*, IDA*, and BTS are guaranteed to be optimal using admissible heuristics .

2.3.4 g-value

Each state has a g-value, which denotes the cost to reach the state from the initial state. It does not need to reflect the true least-cost path to the state, but instead shows the least-cost path found so far by the search algorithm. Due to their iterative-deepening nature, both IDA* and BTS recalculate the g-value of states in each iteration.

2.3.5 f-value

The f-value of a state is some combination of the cost to reach the state from the initial state and the heuristic value of the state. This is important to note as both IBEX and IDA* use the f-value to determine which states to explore next. For our purposes it is sufficient to assume that $f(s) = g(s) + h(s)$, where $g(s)$ is the cost to reach the state from the initial state and $h(s)$ is the heuristic value of the state.

2.3.6 A*

A* is a simple algorithm that uses heuristics to guide its navigation of the state space [4]. It orders states by their f-value, expanding those with the lowest f-value first. In doing so, it targets states that are likely to lead to the goal state with the lowest cost. A* is optimal and complete, provided it uses an admissible heuristic.

It is a very simple example of an informed search algorithm. Its main drawback is its space complexity, which is $O(b^d)$, where b is the branching factor and d is the depth of the state space. IDA* is an extension of A* that is more space efficient, which we will discuss in the following section.

2.3.7 Iterative Deepening A*

It may seem counterintuitive, but it can be preferable to iteratively explore the state space rather than diving deep into it. The idea is to avoid considering the whole state space at once, which is very taxing on memory, and instead to explore the state space in layers. Ideally considering the smallest subset of the state space needed to find a solution. Iterative Deepening A* (IDA*), as its name suggests, iteratively deepens the search space, increasing the maximal f-value to be considered in each iteration [10].

Unlike DFS, IDA* is an informed search algorithm and is optimal and semi-complete. These qualities make it a popular choice for solving planning tasks. IDA*, compared to simply A*, is much more space efficient with a worst-case space complexity of $O(d)$ compared to A*'s $O(b^d)$.

IDA* can be implemented as follows:

Algorithm 2 Iterative Deepening A* (IDA*)

```
1: threshold ← h(state)
2: while true
3:    result ← search(state, 0, threshold)
4:    if result = FOUND
5:       return solutionPath
6:    if result = ∞
7:       return NO_SOLUTION
8:    threshold ← result
```

The searches called by IDA* are f-bounded, meaning that a state s is only expanded if $f(s) \leq threshold$, where $f(s)$ is the f-value of the state s. When a state s with $f(s) > threshold$ is encountered, the search function returns the f-value of the state. This way, IDA* can iteratively explore the state space, increasing the threshold in each iteration.

Algorithm 3 Search function for IDA*

```
1:  if is_goal_state(state)
2:    return SOLVED
3:  successors ← get_successors(state)
4:  for ⟨successor, operator⟩ in successors
5:    solutionPath.push(operator)
6:    result ← search(successor, g + get_cost(operator), threshold)
7:    if result == SOLVED
8:      return SOLVED
9:    else if result < threshold
10:     threshold ← result
11:   solutionPath.pop()
12: return threshold
```

2.4 Classical Planning

We call the problems mentioned in the introduction that we are trying to solve classical planning tasks. These involve finding a plan that leads from some given initial state to some desired goal state. To traverse between states, we use actions defined for the specific planning task. These actions have preconditions that must be met in order to be applicable, and effects that modify the state when the action is executed. Additionally, actions also have costs, meaning that plans can have differing overall costs depending on the sum of the costs of the actions the plan is made up of.

The specific way of formalizing planning task Fast Downward uses are called SAS^+ planning tasks [3]. Formally, these tasks are defined as a tuple $\Pi = \langle V, O, I, \gamma \rangle$ where:

- V is a finite set of variables with a finite set of possible values.
- O is a finite set of actions.
- I is the initial state.
- γ is a goal state, which may or may not be reachable.

Fast Downward takes PDDL files as input and then translates them into SAS^+ classical planning tasks, utilizing Finite Domain Representation (FDR) [7] within the program.

2.4.1 PDDL

Fast Downward uses the Planning Domain Definition Language (PDDL) as input. This format allows a compact and human-readable way to define planning tasks. It is composed of two parts: a domain file and a problem file [5]. In this section, we will examine the blocks world problem in the PDDL format. This problem involves stacking blocks on top of each other, with the goal being a certain configuration of blocks stacked atop each other

Domain File

The following is an excerpt of a domain file for the blocks world problem in the PDDL format:

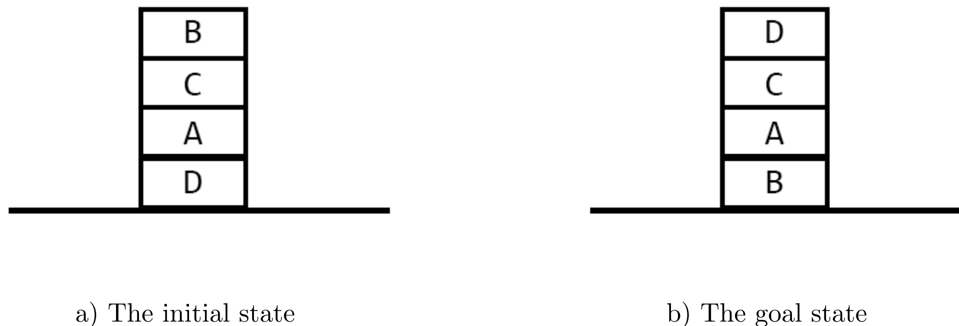
```
(define (domain BLOCKS)
        (:requirements :strips)
        (:predicates (on ?x ?y)
                (ontable ?x)
                (clear ?x)
                (handempty)
                (holding ?x)
                )

          (:action pick-up
                :parameters (?x)
                :precondition (and (clear ?x) (ontable ?x) (handempty))
                :effect
                (and (not (ontable ?x))
                        (not (clear ?x))
                        (not (handempty))
                        (holding ?x)))
                ...
```

In this excerpt we see the predicates involved in the blocks world problem, as well as the definition of the pick-up action. Predicates are used to describe properties of states in planning tasks. They can either be true or false. For example, we can see that a block can either be on another block or not, and the hand which moves the blocks around can either be empty or not.

Actions are used to describe how states can be changed. Actions have effects that modify the state upon the action's application, and preconditions that must be met for the action to be applied. For example, the pick-up action can only be applied if the block has no other blocks on top of it (*clear*), is on the table (*ontable*), and the hand that picks up the block is empty (*handempty*). The application of the pick-up action results in the block no longer being on the table, the block no longer being clear, and the hand not being empty anymore.

Problem File

Here we consider a problem file for the Blocks World problem in the PDDL format. First we consider the problem using an intuitive visual representation:



a) The initial state b) The goal state

In the actual PDDL format, these two states are represented as follows:

```
(define (problem BLOCKS-4-1)
(:domain BLOCKS)
(:objects A C D B )
(:init (clear B) (ontable D) (on B C) (on C A) (on A D) (handempty))
(:goal (and (on D C) (on C A) (on A B)))
)
```

The two problem and domain files together can be used as input for Fast Downward, which attempts to find a sequence of actions that lead from the initial configuration to the desired goal configuration. Its handling of PDDL files and its functioning is explained in the next section.

2.4.2 Fast Downward

Fast Downward is a classical planning system with a considerable degree of modularity [6]. It supports any given PDDL domain/problem pair, and also allows for the use of various search algorithms and heuristics, each with customizable options. It functions by translating the input PDDL file pair into a multi-valued planning task. It also handles heuristic value calculation and provides means for the search algorithm to access states within the state space as described by the planning task. All these steps Fast Downward takes to solve planning tasks are split into the following phases: translation, knowledge compilation, and search. For our implementation and evaluation of IBEX, we worked with Fast Downward's search algorithm classes, leaving its translation and knowledge compilation capabilities unaltered, hence why we will not be examining those components of Fast Downward.

Furthermore, it should be noted that iterative-deepening search algorithm approaches are not too easily implemented in Fast Downward, and our implementation of both IBEX and IDA* had to circumvent certain Fast Downward duplicate checking measures. Details on how we did this will be discussed in the implementation section.

3
Depth-First IBEX

In the past, there has been much effort to improve upon IDA* [2] [12], IBEX aims to outperform these past efforts and provide a more efficient search algorithm [9]. The depth-first version of IBEX, Budgeted Tree Search (BTS), is the focus of this thesis and in this chapter we will delve into its workings.

BTS is an instance of the IBEX search algorithm framework that improves upon IDA*'s design [14]. It aims to reduce the worst case time-complexity of IDA* while retaining its favorable space complexity. How it does this is best explained by using an example of where IDA* performs poorly. Consider the following example state space:

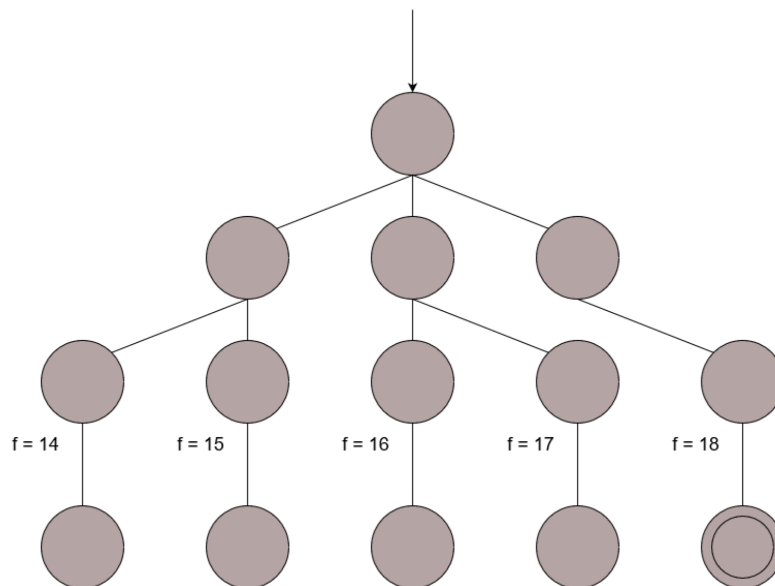


Figure 3.1: Poor performance scenario for IDA*

In this example, the goal state is located on the rightmost branch of the state space with the f-values of the leaf nodes increasing incrementally from left to right. IDA*'s expansion strategy of increasing the search bound using the highest f-value encountered may fail as each iteration could lead to only a single additional node being considered compared to the previous iteration.

3.1 Exponential Search

Firstly, it should be addressed that the name of exponential search is somewhat confusing. The name refers to the whole process, but its first phase is often referred to as being the exponential search phase. For the sake of clarity, we will refer to its first phase as the exponential search phase, although it should be noted that in the most correct sense, "exponential search" should be reserved as the name for the whole process.

Binary search is a common search algorithm for searching for a value in a sorted list. However, for unbounded lists or lists with unknown lengths, binary search is not applicable. This is because we lack the information needed to determine the midpoint of the list.

In such cases, exponential search can be used. Exponential search works by checking elements at increasing powers of, in this implementation, 2 until the desired value or a greater value are found [1]. If the desired value is found, the search terminates. If a greater value is found, the binary search phase is employed to find the desired value, using the last index checked during the exponential search phase as the upper bound.

These two steps combined constitute exponential search and are instrumental for BTS, as it also desires to find the optimal solution cost with an unknown upper bound in the fewest iterations possible.

A disadvantage of this approach is that it may expand nodes with an f-value higher than the optimal solution cost. With an admissible and consistent heuristic, IDA* does not expand nodes with an f-value higher than the optimal solution cost [10]. However, BTS does not have this guarantee, which makes it prone to expand more nodes than IDA*. This issue is mitigated by a budget, which limits the number of nodes expanded in each iteration.

3.2 Global Variables

BTS keeps track of multiple variables shared between all three of its main functions. These variables are:

- **solutionPath**: The current path from the initial state to the goal state.
- **solutionCost**: The cost of the current solution path.
- **solutionLowerBound**: A copy of the lower bound of the i interval.
- **nodes**: The number of nodes expanded during the latest iteration.
- **budget**: The number of nodes expanded in the last iteration, also determines the growth rate of the search space.
- **i**: The search interval. This interval is set to $[h(intial_state), \infty]$ at the start of the algorithm and gets refined with each iteration in the search function. It contains a range of f-values that lead to exponentially more nodes being expanded with each iteration. Once the lower and upper bounds of the interval are equal, the algorithm returns the least costly solution.
- f_{below}: The maximum f-value expanded below the cost limit during the latest iteration.

- f_{above}: The next largest f-value encountered in the latest iteration.

Additionally, the algorithm takes c_1 and c_2 as input parameters. These parameters determine the growth rate of the search space and the maximum number of nodes expanded in each iteration. Specifically, c_1 and c_2 determine the minimum and maximum factors of the growth rate of the number of expanded nodes in each iteration, respectively. However, there are exceptions to this. If a solution is found, then the growth of nodes expanded may be less than c_1. If no f-bound exists to cause a growth rate lower than a factor of c_2, then the growth rate may exceed c_2.

3.3 Algorithm Overview

BTS addresses the issue of IDA* expanding a minimal number of nodes with each iteration when its node expansions grow linearly by employing a three-phase search strategy: regular IDA* iteration starting on line 9 (Algorithm 4), exponential search starting on line 14, and binary search starting on line 21. Each of these stages is shown in the pseudocode of BTS below:

Algorithm 4 BTS(c_1, c_2)

```
1:  solutionPath ← ∅
2:  solutionCost ← ∞
3:  budget ← 0                                          ▷ budget in node expansions
4:  i ← [h(get_initial_state()), ∞]                     ▷ initial f interval
5:  while solutionCost > i.lower
6:      solutionLowerBound ← i.lower
7:      i.upper ← ∞
8:                                                      ▷ 1. Regular IDA* iteration
9:      i ← i ∩ Search(i.lower, ∞)
10:     if nodes ≥ c1 · budget
11:         budget ← nodes
12:         continue
13:                                                     ▷ 2. Exponential Search phase
14:     Δ ← 0
15:     while (i.upper ≠ i.lower) ∧ (nodes < c1 · budget)
16:         nextCost ← i.lower + 2^Δ
17:         Δ ← Δ + 1
18:         solutionLowerBound ← i.lower
19:         i ← i ∩ Search(nextCost, c2 · budget)
20:                                                     ▷ 3. Binary Search phase
21:     while (i.upper ≠ i.lower) ∧ ¬(c1 · budget ≤ nodes < c2 · budget)
22:         nextCost ← (i.lower + i.upper) / 2
23:         solutionLowerBound ← i.lower
24:         i ← i ∩ Search(nextCost, c2 · budget)
25:     budget ← max(nodes, c1 · budget)
26:     if solutionCost = i.lower
27:         return
```

The algorithm is initialized with an interval i that represents the range of f-values to be explored, specifically using the heuristic value of the initial state as a lower bound and

infinity as the upper bound. The lower bound is chosen to be the heuristic value of the initial state because when using an admissible heuristic, the optimal solution cost is at least as high as the heuristic value of the initial state. Similarly, IDA* uses the heuristic value of the initial state as the initial search bound as well. It then performs a regular IDA* iteration and if less nodes have been expanded than desired, it moves on to the exponential search phase, increasing the upper search bound using powers of 2.

Once a solution is found or the budget is exceeded, the algorithm moves on to the binary search phase, narrowing down the search interval until the solution cost is found. Here, the search bound is halved with each iteration.

At any point, if the lower and upper bounds of the search interval are equal, the algorithm terminates. This is because once this condition is met, it is guaranteed that the solution cost is optimal. It may also occur that the regular IDA* iteration finds a solution cost lower than the lower bound of the search interval. In this case, the algorithm exits the while-loop and returns the optimal solution.

Algorithm 5 Search(*costLimit*, *nodeLimit*)

```
1: f_below ← 0                                              ▷ max f expanded below costLimit
2: f_above ← ∞                                              ▷ next largest f
3: nodes ← 0                                                ▷ nodes expanded
4: LimitedDFS(get_initial_state(), 0, costLimit, nodeLimit)
5: if nodes ≥ nodeLimit
6:     return [0, f_below]
7: else if f_below ≥ solutionCost
8:     return [solutionCost, solutionCost]
9: else
10:    return [f_above, ∞]
```

The search function functions as a wrapper for the LimitedDFS function. Depending on the number of nodes expanded during the search iteration and f_{below}, it adjusts the search interval.

Case 1 (Algorithm 5, line 5): If the search iteration exceeds the node limit, the search interval is set to $[0, f_{below}]$, meaning that assumptions about the least costly solution are disregarded and the upper bound is set to the highest f-value encountered below the cost limit.

Case 2 (Algorithm 5, line 7): If the maximum f-value below the cost limit encountered is greater than or equal to the solution cost, the search interval is set to $[solutionCost, solutionCost]$, indicating that the optimal solution has been found.

Case 3 (Algorithm 5, line 9): If neither of the above conditions are met, the search interval is set to $[f_{above}, \infty]$, where f_{above} is the next largest f-value encountered during the search iteration.

Algorithm 6 LimitedDFS(*currState*, *pathCost*, *costLimit*, *nodeLimit*)

```
1: currF ← pathCost + h(currState)
2: if solutionCost = solutionLowerBound
3:     return
4: else if currF > costLimit
5:     f_above ← min(f_above, currF)
6:     return
7: else if currF ≥ solutionCost
8:     f_below ← solutionCost
9:     return
10: else
11:     f_below ← max(currF, f_below)
12: if nodes ≥ nodeLimit
13:     return
14: if is_goal_state(state)
15:     solutionPath ← currentPath
16:     solutionCost ← currF
17:     return
18: successors ← get_successors(state)
19: for ⟨successor, operator⟩ in successors
20:     currentPath.push(operator)
21:     LimitedDFS(successor, pathCost + get_cost(operator), costLimit, nodeLimit)
22:     currentPath.pop()
```

Lastly, the LimitedDFS function bears a similar structure to a depth-first search as used by IDA*. The main difference of note is its more sophisticated early termination conditions. Using information like the number of nodes expanded and the f-values of the nodes encountered during the search iteration, it can ensure that it does not behave worse than IDA*. The early termination conditions contribute to the algorithm's efficiency.

3.4 Algorithm Properties

In this section we will briefly examine the properties of BTS, such as its time and space complexity, completeness, and optimality.

- **Time Complexity:** BTS has the worst case time complexity $O(N \log(C^*/\epsilon))$ [14], where C^* is the cost of the optimal solution and ϵ is the granularity of action costs, meaning that $\epsilon = 1$ for integers. $\epsilon = 0.1$ for problems where action costs differ by 1 decimal digit, and so on. Action costs matter in the time complexity because they play a role in setting the search bound.
- **Space Complexity:** BTS retains the space complexity of IDA*, which is $O(bd)$, where b is the branching factor and d is the depth of the optimal solution.
- **Completeness:** BTS is complete, meaning that it will always find a solution if one exists and terminate if none exists.
- **Optimality:** BTS is optimal, meaning that it will always find the least costly solution.

4 Implementation

Since our goal was to implement IBEX in Fast Downward, we decided to use the BTS guide as a reference for our implementation. The guide provides a very clear explanation of how BTS works and how it can be implemented and serves as a gentle introduction to depth-first IBEX.

At the time of writing this thesis, the latest release of Fast Downward is *release-23.06.0*, which is the version that was used for our implementation and evaluation of BTS. Our implementation of BTS in Fast Downward can be accessed in the following repository: https://github.com/lalancz/downward.

As mentioned in the introduction chapter, a major issue we faced was that Fast Downward stores states in memory in a not all too transparent way, as required due to its duplicate detection mechanisms. Diminishing their memory saving benefits, when testing BTS and IDA*, we were more interested in the conditions under which BTS enters the exponential search phase and how it compares to IDA* in terms of the number of iterations required.

4.1 Unregistered States

In Fast Downward, states can be registered in the state registry to (not only) enable duplicate detection and caching of heuristic values. States are "packed", meaning that they are stored in the most memory-efficient form possible. They are also assigned a state ID, which is used to identify them. Since Fast Downward's design leads it to store many states in memory at the same time, this is a very important memory saving step.

Furthermore, states can be opened, which changes their status to "open" and registers some other state as their parent. Using the parent information, Fast Downward can reconstruct the path from the initial state to the goal state.

Our iterative-deepening approach causes trouble with all of these features. Reopening nodes with each iteration creates cycles in the state space and makes the built in path tracing loop infinitely. This is why instead of using the path tracing function, we chose to use a vector to keep track of the operators used to reach the goal state.

We sidestep the issue by making use of functions that fetch unregistered states and not

registering them at all. However, this alone is not enough.

4.2 Estimate Caching

Simply replacing all relevant functions with their unregistered counterparts will still produce an error. Heuristics by default make use of caching, which requires per-state information to be stored. This is not possible with unregistered states. Therefore, the option `cache_estimates` has to be set to false for every heuristic used with our implementation of BTS and IDA*.

4.3 Path Checking

As mentioned earlier, we also considered a simple optimization that avoids the expansion of nodes that are already on the path to the current state. This optimization is not present in the original BTS paper, but we decided to include it in our implementation to attempt to mitigate stack overflows that were occurring.

The optimization is simple: before expanding a node, we check if it is already on the path to the current state. If it is, we skip the expansion. For this we make use of a vector, which stores the visited states. To check if a state is visited, we have to iterate over the vector, which is a linear operation and is quite costly. We are unable to use a unique state ID, because unregistered states do not have one and states cannot be hashed as they are implemented in Fast Downward, which rules out the use of a hash set.

BTS with this path checking optimization does not constitute a graph search algorithm, since it entails only partial duplicate checking. The set of visited states changes throughout each iteration, meaning that the same state may be considered multiple times in each iteration, despite the path checking. We only discard states that are already part of the currently considered path. This does not mean that the same state cannot be considered again in the same iteration, just as a part of a different path.

5 Evaluation

In this chapter we will discuss the performance of our implementation of BTS in comparison to our implementation of IDA* and the built-in implementation of A* in Fast Downward. For the evaluation, we used the sciCore Center for Scientific Computing at the University of Basel. The specific sciCore partition we used for these results was `infai_2`, which uses Core Intel Xeon Silver 4114 2.2 GHz Processors. We chose 30 minutes as the time limit and 3584 MB as the memory limit. To run these experiments we used Fast Downward Lab [11].

We ran two experiments, one with the landmark cut heuristic ($h^{\text{LM-CUT}}$) [8] and one with the blind heuristic. $h^{\text{LM-CUT}}$ is a well-known admissible heuristic that has been implemented in Fast Downward. As its name implies, the blind heuristic simply returns a constant value for each state with the exception of the goal state, for which it returns 0.

We tested BTS and our own implementation of IDA* in Fast Downward, both with and without path checking. As in the BTS paper, we chose values $c_1 = 2$ and $c_2 = 8$ for BTS.

We present the results for $h^{\text{LM-CUT}}$ in table 5.1 and for the blind heuristic in table 5.2.
In the following sections, we will discuss the results in more detail.

Algorithm Name	A*	BTS	BTS path checking	IDA*	IDA* path checking
Coverage	966	559	596	556	591
Exponential search	—	13.82%	17.88%	—	—
Error out of time	858	1222	1235	1222	1241
Error out of memory	6	9	0	10	0
Expansions until last jump	—	978.25	1629.09	767.11	1091.47
# Iterations	—	1757	1988	1821	2188
Peak memory sum (MB)	1189.7	1149.4	1149.4	1149.3	1149.3
Search time (s)	0.09	0.77	0.54	0.70	0.49

Table 5.1: Results for $h^{\text{LM-CUT}}$ heuristic, the expansions until last jump and search time metrics are geometric means

Algorithm Name	A*	BTS	BTS path checking	IDA*	IDA* path checking
Coverage	718	257	287	246	270
Exponential search	—	19.21%	23.17%	—	—
Error out of time	0	1475	1545	1486	1562
Error out of memory	1112	21	0	15	0
Expansions until last jump	—	171745.17	121924.19	127070.28	54197.04
# Iterations	—	2158	2398	3075	3316
Peak memory sum (MB)	7144.4	4730.7	4730.4	4730.1	4730.1
Search time (s)	0.02	1.11	0.53	1.43	0.62

Table 5.2: Results for the blind heuristic, the expansions until last jump and search time metrics are geometric means

Domain	BTS	IDA*
mprime	**21**	20
nomystery-opt11-strips	11	**12**
organic-synthesis-split-opt18-strips	**14**	13
parcprinter-08-strips	**14**	13
parcprinter-opt11-strips	**9**	8

Table 5.3: Differences in coverage for the lmcut heuristic

Domain	BTS	IDA*
movie	**3**	2
organic-synthesis-split-opt18-strips	**10**	9
parcprinter-08-strips	**5**	3
parcprinter-opt11-strips	**2**	0
pegsol-08-strips	**26**	24
pegsol-opt11-strips	**16**	14
pipesworld-notankage	**6**	5

Table 5.4: Differences in coverage for the blind heuristic

5.1 Coverage

Coverage reflects the number of tasks that were able to be solved by each of the algorithms. Tables 5.3 and 5.4 show the differences in coverage of BTS and IDA* for the $h^{\text{LM-CUT}}$ and blind heuristics, respectively.

A* has the highest coverage for both heuristics, which could be explained by its emphasis on speed, avoiding the time limit. Given the chosen time and memory limits, it seems that A*'s sacrificing memory usage efficiency for speed is the best strategy. BTS has a higher coverage than IDA*, however only by a very small margin. The path checking option does not seem to have a significant impact on coverage.

Table 5.3 shows that there is one domain where IDA* has a higher coverage than BTS by one task. This is an anomaly caused by improper checking whether we have found a solution. Our implementation of BTS fails to report that it has found a solution, causing the task to be marked as failed. In this one case, the conditions for finding a solution are not met, even though an optimal solution has been found, and the search continues. With the search continuing, the conditions required for our implementation to report that it has found a

Domain	BTS Exponential Search %	BTS Search Time (s)	IDA* Search Time (s)
organic-synthesis-split-opt18-strips	32.50	2.30	2.16
transport-opt08-strips	25.93	0.51	0.58
snake-opt18-strips	25.00	7.77	7.72
blocks	13.56	0.23	0.22

Table 5.5: Search times for high percentage of iterations that entered the exponential search phase using $h^{\text{LM-CUT}}$

solution are not met. Rerunning the task with a modified version of our implementation, which checks if the solution path vector is empty before marking the task as failed and reporting that the task has been solved if it is not, resulted in the exact same coverage (559) as without this modification.

The BTS paper leaves some ambiguity as to under which conditions the algorithm should report the task as solved or failed, meaning that we had to make some assumptions when implementing BTS. Leaving the condition responsible for reporting the task as solved as strict as it currently is leaves less risk of suboptimal solutions being reported as optimal and potentially invalid plans being found.

5.2 Exponential Search

This metric shows the arithmetic mean of the percentage of iterations that entered the exponential search phase.

Tables 5.5 and 5.6 show the differences in search time for selected domains with a high percentage of iterations that entered the exponential search phase. They show that from a domain level perspective, it is difficult to predict what effect the percentage of iterations that enter the exponential search phase will have on the search time. A stark difference in search times can be seen for the movie domain with the blind heuristic, where BTS has a geometric mean search time of 355.66 seconds, while IDA* has a geometric mean search time of 535.95 seconds. However, the difference for the other domains is not as pronounced.

For domains where IDA* slightly outperforms BTS despite a high number of its iterations entering the exponential search phase, it may be that the number of nodes expanded by IDA* and BTS grows at just enough of a similar rate to trigger the exponential search phase in BTS, but the number of nodes expanded is so similar that the overhead required for the exponential search phase is not worth it.

5.3 Errors

In this section we will look at the errors that occurred during the experiments.

Domain	BTS Exponential Search %	BTS Search Time (s)	IDA* Search Time (s)
psr-small	19.58	0.10	0.12
movie	15.30	355.66	535.95
blocks	12.60	0.44	0.56
pegsol-08-strips	10.21	1.34	1.60

Table 5.6: Search times for high percentage of iterations that entered the exponential search phase using the blind heuristic

5.3.1 Stack Overflows

The recursive approaches of BTS and IDA* are prone to stack overflows, which can arise due to the depth-first nature of the search. Although they are less prone to stack overflows owing to their searches being f-bounded, stack overflows may still occur. They recurve too deeply into the search tree, causing the stack to overflow. This is especially prevalent in the Sokoban domain, where the search reaches very deep. As BTS may set the f-bound higher than whatever the optimal solution cost is, our implementation of BTS may end up in a situation where it is reaches deep enough into the state space to cause a stack overflow.

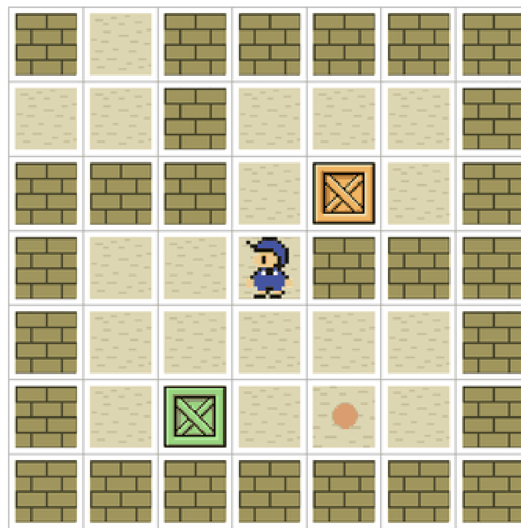


Figure 5.1: Visualization of a Sokoban problem [13]

Sokoban is a game packaged as a planning problem in the IPC benchmark suite. The objective of the game is to push boxes within a warehouse onto a target location. The game is played on a grid, where a warehouse worker can move in four directions, also being able to push boxes around.

The specific version of Sokoban used in the IPC benchmark suite assigns the cost of 0 to the action of moving the warehouse worker, meaning that the warehouse worker can move around the warehouse infinitely without incurring any cost. This causes the f-bound to not be met, allowing the search to go on indefinitely, ultimately ending in a stack overflow.

We also encountered stack overflow problems in the elevator domain for the blind heuristic, where passengers can enter and leave elevators with a cost of 0.

5.3.2 Out of Memory and Out of Time Errors

These metrics refer to the number of tasks that failed due to running out of memory or time.

Other than segmentation faults, there are also out of memory and out of time errors to consider. A* ends in a large number of out of memory errors with the blind heuristic, which can be explained by it being forced to explore much more of the state space than with $h^{\text{LM-CUT}}$.

To be expected due to their iterative-deepening approach, BTS and IDA* fail due to running out of time for about 50% more tasks than A*. This is due to the fact that they are slower than A* and are more likely to reach the time limit. Keeping in mind that the likeliest cause for a search algorithm to fail is either running out of time or memory, it is logical to see that due to their high memory efficiency, BTS and IDA* would have a higher proportion of their failures be due to running out of time.

5.4 Expansions until last jump

When examining the number of nodes expanded, we discard the number of nodes expanded in the last iteration. We refer to the number of nodes expanded excluding the last iteration as "expansions until last jump". We discard the number of nodes expanded in the last iteration because the number of nodes expanded in the last iteration is often cut short by the algorithm finding a solution and ceasing the search. Thus, the number of nodes expanded in the last iteration is the work of the chosen tie-breaking mechanism rather than the search algorithm. For this reason, the number of nodes expanded in the last iteration obscures the behavior of the search algorithms that we are actually interested in. To get this number of nodes expanded until last jump, we take the total number of nodes expanded and subtract the number of nodes expanded in the last iteration.

While BTS guarantees better worst case performance than IDA*, it can incur more node expansion than IDA*. This is because the f-bound can be set to be higher than the optimal solution cost during the exponential search phase. This can lead to the search expanding more nodes than IDA* would.

This metric also exists for A*, but it is calculated differently and thus we chose not to include it in the table.

5.5 Iterations

Plots 5.2 and 5.3 show the comparison between the number of iterations performed by BTS and IDA* for the $h^{\text{LM-CUT}}$ and blind heuristics, respectively. The metric in tables 5.1 and 5.2 show the total number of iterations performed by the search algorithms.

Note that multiple tasks may occupy the same points on the plot.
As the plots show, BTS never performs more iterations than IDA*. This is to be expected, as BTS never expands less nodes than IDA* per iteration.

The points on the edges signify that only one of the algorithms found a solution for the

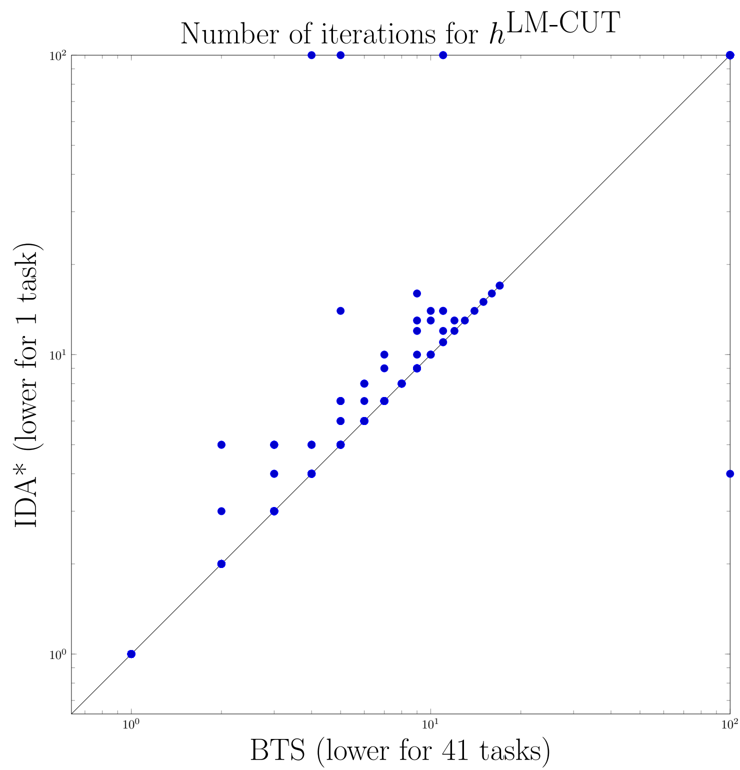


Figure 5.2: Number of iterations for BTS and IDA* using $h^{\text{LM-CUT}}$

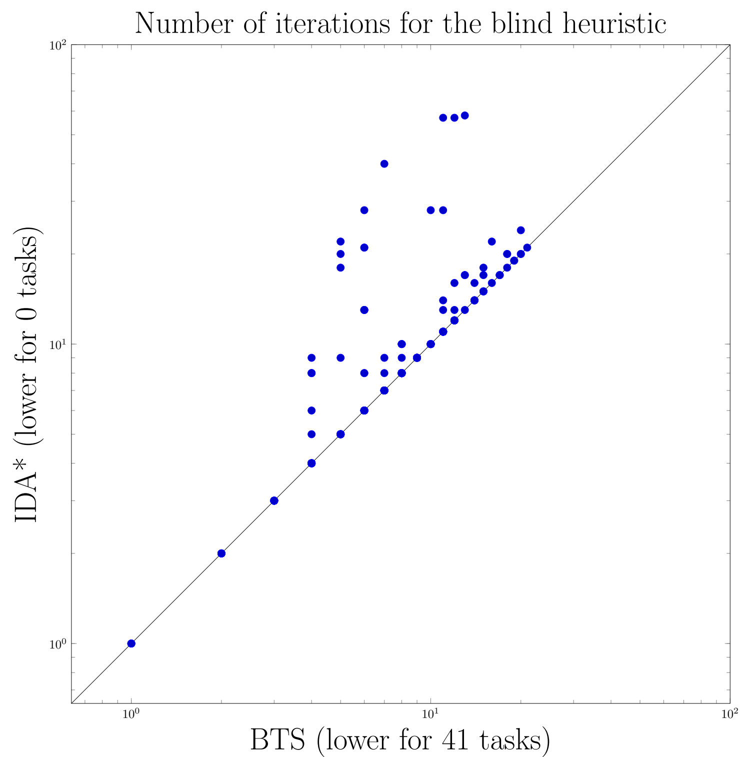


Figure 5.3: Number of iterations for BTS and IDA* using the blind heuristic

given problem. Points on the top edge signify that only BTS found a solution, while points on the right edge signify that only IDA* found a solution.

As can be seen in plot 5.2, there is 1 point on the right edge of the plot, signifying that IDA* found a solution that BTS did not. This is the only point in the plot where this is the case. It occurs for a task in the *nomystery-opt11-strips* domain, the reason for this is explained in subsection 5.1.

5.6 Memory

The metric shown in the tables refers specifically to the sum of the peak memory usage for all searches for a given algorithm. Each of the algorithms spikes in memory usage at the beginning of the search, but then stabilizes.

The tasks solved by both BTS and IDA* within the time allotted are small enough that A*'s memory usage is not a problem. This explains why the peak memory usage is so similar for all the algorithms with $h^{\text{LM-CUT}}$. However, with the blind heuristic, A* is not able use heuristic values to guide its search, and thus the difference in peak memory usage is much more pronounced. A* uses significantly more memory than the other algorithms with the blind heuristic, as shown in table 5.2.

A supporting factor for this explanation is that A* runs out of memory often with the blind heuristic, while the other algorithms instead run out of time. Minimal time is spent on calculating the heuristic value with the blind heuristic, which means that A* is allowed to expand a very high number of nodes, which in turn uses up a very high amount of memory.

5.7 Runtime

This metric shows the geometric mean of the search times for the algorithms in seconds. The geometric mean gives a better representation of the average runtime, as the runtime can vary significantly between tasks.

A* is the fastest considered algorithm by far, which can be expected by not using an iterative-deepening approach. Compared to IDA*, BTS has a better worst case guarantee, but for our tasks with $h^{\text{LM-CUT}}$ these guarantees do not outweigh the added overhead. However, for the blind heuristic, BTS is noticeably faster than IDA*. This could be explained by the blind heuristic causing IDA* to grow its f-bounds slowly, leading to IDA* having to perform more iterations. This can be seen in plot 5.3.

Optimizing using the path checking is a significant factor in the runtime of BTS and IDA*, more than halving the geometric mean of the search times when using the blind heuristic. Even for $h^{\text{LM-CUT}}$, the improvement in search times is significant. However, A* remains the fastest algorithm by far.

6 Conclusion

The goal of this thesis was to implement BTS in Fast Downward and evaluate its performance. We examined whether it truly does improve IDA*'s worst case runtime and found that it actually never requires more iterations than IDA* to find a solution. We also found that BTS is able to find solutions for some problems that IDA* cannot. However, we found that BTS often requires more node expansions than IDA*, as also stated in the BTS paper.

Our experiments show that BTS is a promising algorithm providing a clear improvement over IDA* in iterations required to solve a task. In implementing BTS, we made use of some niche features of Fast Downward, as Fast Downward can be quite unfriendly to iterative-deepening search algorithms. Along the way, we also encountered stack overflows caused by the recursive nature of BTS and IDA*. We attempted to mitigate this by avoiding the expanding of states that were already a part of the currently considered path, which helped avoid stack overflows, but did not raise the coverage by a large margin.

In the future, it would be interesting to see how a graph search instance of IBEX, Budgeted Graph Search, would perform and how it would fare against BTS. It would also be interesting to see how our implementation of BTS could be further optimized within Fast Downward.

Bibliography

[1] Jon L. Bentley and Andrew C.-C Yao. An Almost Optimal Algorithm for Unbounded Searching. *Information Processing Letters*, 5(3):82–87, 1976.

[2] Ethan Burns, Wheeler Ruml, and Minh Binh Do. Heuristic search when time matters. *Journal of Artificial Intelligence Research*, 47:697–740, 2013.

[3] Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11:625–656, 1995.

[4] 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.

[5] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An Introduction to the Planning Domain Definition Language, 2019.

[6] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[7] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5):503–535, 2009. Advances in Automated Plan Generation.

[8] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: what's the difference anyway? pages 162–169, 2009.

[9] Malte Helmert, Tor Lattimore, Levi H. S. Lelis, Laurent Orseau, and Nathan R. Sturtevant. Iterative Budgeted Exponential Search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence.*

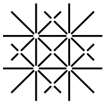
[10] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[11] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab, 2017.

[12] Guni Sharon, Ariel Felner, and Nathan R. Sturtevant. Exponential deepening A* for real-time agent-centered search. pages 871–877, 2014.

[13] Tom Silver and Rohan Chitnis. PDDLGym: Gym environments from PDDL problems. In *International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop*, 2020.

[14] Nathan R. Sturtevant and Malte Helmert. A Guide to Budgeted Tree Search. pages 75–81, 2020.



University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Implementation and Evaluation of Depth-First IBEX in Fast Downward

Name Assessor: Prof. Dr. Malte Helmert

Name Student: Petr Sabovčik

Matriculation No.: 2021-062-310

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used.
This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Basel, 22.07.2024 Student: Sabo

Will this work, or parts of it, be published?

- [x] No

- [ ] Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: ____________________

Place, Date: Basel, 22.07.2024 Student: Sabo

Place, Date: ____________________ Assessor: ____________________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.