University of Basel

Efficient Implementation of h^2 in the Fast Downward Planning System

Master's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group

Examiner: Prof. Dr. Malte Helmert
Supervisor: Clemens Büchner

Elia Hänggi
elia.haenggi@stud.unibas.ch
2020-055-497

May 20, 2025

Acknowledgments

I would like to express my sincere gratitude to my examiner, Prof. Dr. Malte Helmert, for providing the opportunity to work on this thesis. I am especially grateful to my supervisor, Clemens Büchner, for his continuous support, insightful feedback, and weekly valuable discussions during all phases of this work.
The computational work presented in this thesis was performed at sciCORE, the scientific computing center at the University of Basel. I would like to thank sciCORE for providing the necessary infrastructure and resources. Furthermore, the experiments conducted in this thesis were managed using the Downward Lab framework, which provided essential tools for configuring and evaluating search experiments in Fast Downward (Seipp et al., 2017).
I also acknowledge the use of the Python package Machetli, which greatly facilitated the debugging of the implementation in Fast Downward.

Abstract

This thesis investigates different implementations of the critical path heuristic h^2 in the Fast Downward planning system. We identify inefficiencies in the existing implementation and propose several optimizations. These include moving calculations outside the search process and reusing intermediate results. The improved implementation performs significantly better in practice, with the vast majority of the solved benchmark problems being solved more than ten times faster.

We also explore the Π^m compilation method, which computes h^2 values using the $h^{\max}$ heuristic on a Π^2 compiled task. Although this method consumes more memory, it outperforms our optimized h^2 implementation. A crucial factor in achieving this performance is the simplification of the resulting operator set. Detecting duplicate and dominated operators is essential to reduce redundancy, though this process is only efficient for small precondition sets.

To enable the reuse of heuristic values, we apply STRIPS duality to simulate regression search. This leads to faster evaluations but results in weaker heuristic estimates. The reduced informativeness is mainly caused by how our SAS$^+$ to STRIPS transformation interacts with the duality mapping, making this approach suboptimal.

Due to the nature of progression search where each heuristic evaluation must estimate the cost from the current state to a set of atoms independently, it remains difficult to achieve performance comparable to other informed heuristics such as $h^{\max}$.

Table of Contents

1 Introduction

The objective of planning is to find suitable sequences of actions that achieve a desired goal in a given environment. A common strategy is to search through a state space. Since state spaces can be very large, it is often infeasible to explore them exhaustively. Therefore, a heuristic is used to guide the search toward a goal more efficiently. A heuristic h is a function that estimates the cost to reach a defined goal. If the heuristic is admissible, i.e. it never overestimates the true cost, using it with an admissible search algorithm such as A^* guarantees finding an optimal solution (Russell and Norvig, 2021).

When designing heuristics for planning tasks, there is always a trade-off between accuracy and computational cost. A heuristic that provides less accurate estimates may cause the search to explore many irrelevant states, while a highly accurate heuristic may be too expensive to compute. A commonly used approach to simplify heuristic computation is delete-relaxation, which removes the negative effects of actions. One well-known heuristic that uses this idea is $h^{\max}$, which further simplifies the task by considering only the cost of the most expensive single goal variable. However, this simplification often leads to strong underestimation (Bonet and Geffner, 2001). For example, if several goal variables can only be achieved by disjoint sets of actions, $h^{\max}$ might only reflect the cost of one of them.

The critical path heuristic h^m aims to improve on this limitation. Instead of considering just one variable, h^m approximates the cost of achieving the most difficult subgoal, where subgoals are subsets of variables of size at most m. Therefore, h^m is a generalization of $h^{\max}$ as it holds $h^{\max} = h^1$. Typically, larger m-values lead to more accurate estimates and thus smaller search spaces. However, the computational complexity of h^m grows exponentially with m. In this thesis, we focus on h^2 as a practical compromise between accuracy and efficiency. Since h^2 considers interactions between pairs of subgoals, we expect it to produce more informative estimates than $h^{\max}$, resulting in significantly smaller search spaces.

Fast Downward is a heuristic-based planning system that supports a wide range of heuristics and search algorithms (Helmert, 2006). While it provides implementations of both $h^{\max}$ and h^m, the current h^m implementation is inefficient. It relies on suboptimal data structures

with poor worst-case performance and repeatedly performs computations that either are redundant or could potentially be moved to a preprocessing phase.
The goal of this thesis is to implement h^2 more efficiently and compare its performance with the existing implementation using a variety of benchmark problems. Several promising techniques motivate this work. The current implementation uses a heuristic table that is iteratively updated by evaluating operators. Another interesting method is the Π^m compilation, which shifts the complexity of computing h^m into the planning problem itself. Applying $h^{\max}$ to this compiled task produces h^m values. Additionally, regression search is worth exploring. Fast Downward currently only supports forward search. Investigating a backwards search approach in combination with h^2 may lead to new insights.

This thesis is structured as follows. Chapter 2 provides the necessary mathematical background on classical planning, search strategies, and the h^m heuristic. Chapter 3 presents the existing implementation of h^2 in Fast Downward, followed by improvements and optimizations in Chapter 4. Chapter 5 introduces the theoretical foundation of the Π^m compilation, which is then implemented and evaluated in Chapter 6. Chapter 7 explores the use of regression search via STRIPS duality as an alternative to implementing regression search directly in Fast Downward. Finally, Chapter 8 concludes the thesis and outlines directions for future research.

2 Background

In the background section, we establish mathematical definitions and knowledge needed in order to state the definition of the critical path heuristic h^m. We begin with the definition of a planning task. Then we introduce the two main search approaches in a planning task: progression and regression. We discuss the concept of heuristics and delete relaxation. At the end we state the definition of the critical path heuristic h^m.

2.1 Planning Task

In order to fully define a state space, we need to come up with a representation for states, including the initial state, the goal states, and the transition system containing those states. Ideally, this representation should be as compact as possible to describe very complex systems concisely. To this end, we introduce two types of planning task representations: *STRIPS planning tasks* and *SAS$^+$ planning tasks*.

2.1.1 STRIPS Planning Tasks

The STRIPS planning formalism uses propositional state variables to represent states. A propositional state variable is a variable that can be assigned either $true$ or $false$. Let V denote the finite set of propositional state variables. A state s is a truth assignment $s : V \to \{true, false\}$. Therefore, n propositional state variables allow us to express 2^n different states. To simplify state representations, we denote a STRIPS state s as a set of variables: $s = \{x \in V \mid x = true\}$.
Transitions are defined using the concept of an *operator* $o = \langle pre(o), add(o), del(o), cost(o) \rangle$, where:

- $pre(o)$ is a set of variables that must be true to apply the operator,
- $add(o)$ contains variables set to true when the operator is applied,
- $del(o)$ contains variables set to false when the operator is applied, and
- $cost(o) \in \mathbb{R}_0^+$ is the cost of applying the operator.

Applying an operator o in a state s results in a state $s[o] = (s \setminus del(o)) \cup add(o)$. The goal G of the system is represented as a set of variables that must be true. A planning task based on propositional state variables is called STRIPS planning task (Fikes and Nilsson, 1971).

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

- *V is a finite set of propositional state variables,*
- *O is a finite set of operators, each with preconditions $pre(o) \subseteq V$, add effects $add(o) \subseteq V$, delete effects $del(o) \subseteq V$, and cost $cost(o) \in \mathcal{R}_0^+$,*
- *$I \subseteq V$ is the initial state, and*
- *$G \subseteq V$ is the set of goal states.*

A *plan* for a STRIPS planning task is a sequence of operators $\pi = \langle o_1, \ldots, o_n \rangle$, $o_i \in O$, such that $I[\pi]$ is a goal state. The cost of a plan is $cost(\pi) = \sum_{i=1}^{n} cost(o_i)$ where $n = |\pi|$. A plan is *optimal* if it has minimal cost.

2.1.2 SAS$^+$ Planning Tasks

To enable more compact representations of states, we generalize state variables to finite-domain variables. We allow a variable to be assigned not only *true* or $false$, but a value from a defined finite set.

Definition 2 (Finite-Domain State Variable)**.** *A finite-domain state variable is a variable v with an associated finite domain $dom(v)$. A state s over V is an assignment such that $s(v) \in dom(v)$ for all $v \in V$.*

With these state variables, we are able to define a SAS$^+$ planning task (Bäckström and Nebel, 1995).

Definition 3 (SAS$^+$ Planning Task)**.** *An SAS$^+$ planning task is a 4-tuple* $\Pi = \langle V, O, I, G \rangle$, *where:*

- *V is a finite set of finite-domain state variables,*
- *O is a finite set of operators, where each operator $o \in O$ has:*
  - *$pre(o)$: a partial assignment over V specifying the preconditions,*
  - *$eff(o)$: a partial assignment over V specifying how the state changes,*
  - *$cost(o) \in \mathcal{R}_0^+$: the cost of applying o,*
- *I is a full assignment over V specifying the initial state,*
- *G is a partial assignment over V describing the goal.*

The effects of an operator $\mathit{eff}(o)$ are a set of assignments, where each assignment sets a variable v to a value d from its domain, i.e. $v := d$ with $d \in \mathit{dom}(v)$. It holds that an operator cannot have two effects assigning the same variable two different values. Operators with this restriction are referred to as *conflict-free* operators. Every operator in a SAS$^+$ planning task has to be conflict-free. Fast Downward generally uses a task representation that is slightly different from SAS$^+$ as it also allows other more general forms of effects. However, the h^m implementation uses exactly this SAS$^+$ formalism which is why we limit ourselves to SAS$^+$ tasks in this thesis.

SAS$^+$ can be viewed as generalization of STRIPS planning tasks, as each propositional variable in STRIPS corresponds to a finite-domain variable in SAS$^+$ with domain $\{true, false\}$. However, any SAS$^+$ planning task can be translated into an equivalent STRIPS representation. The core idea behind this translation is to introduce one propositional variable for each possible domain value of a finite-domain variable. In the translated task it needs to be ensured that mutual exclusivity between domain values is preserved.
In the following, we introduce key concepts of heuristic search, including the critical path heuristic, using the STRIPS representation. This choice is motivated by the explicit distinction between add and delete effects in STRIPS, which allows for more concise and intuitive definitions. Thus, the critical path heuristic is typically formulated in STRIPS in foundational work. Nonetheless, the theoretical ideas and heuristics presented here are equally applicable to SAS$^+$ planning tasks. Equivalent SAS$^+$ formulations can be systematically derived using the translation approach described above. The detailed algorithmic translation between SAS$^+$ and STRIPS will be presented in Section 5.2.

2.1.3 Transition Systems for Planning Tasks

Independent from the specific representations, every planning task Π induces a transition system $\mathcal{T}(\Pi)$:

Definition 4 (Planning task as transition system)**.** *The planning task* $\Pi = \langle V, O, I, G\rangle$ *induces the transition system* $T(\Pi) = \langle S, L, T, s_0, S^\star, \mathit{cost}\rangle$*, where*

- S *is the set of all states over* V*,*
- L *is the set of operators* O*,*
- $T = \{\langle s, o, s'\rangle \mid s, s' \in S, s' = s[o]\}$ *is a set of labeled transitions,*
- $s_0 \in S$ *is the initial state,*
- $S^\star \subseteq S$ *is the set of goal states, and*
- $\mathit{cost} : L \to \mathcal{R}_0^+$ *is the cost function.*

In the following, we will use the terminology of transition systems also for planning tasks. Therefore, when using terms like transitions or graph edges in the context of a planning task, we refer to the transition system induced by the task.
We now establish different search approaches to find a plan in a planning task.

2.2 Progression and Regression Search

In this section, we examine two fundamental approaches to exploring the state space in planning: *progression search* and *regression search*.
The first approach, known as progression search or forward search, begins at the initial state and generates successor states by applying operators that are applicable in the current state. We formalize this process as STRIPS progression:

Definition 5 (STRIPS progression)**.** *The STRIPS progression $sprog(s, o)$ for a state s and an operator o is defined as:*

$$sprog(s, o) = \begin{cases} (s \setminus del(o)) \cup add(o) & \textit{if } pre(o) \subseteq s \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

The search continues until a goal state in G of the planning task is reached. This method is intuitive as it aligns with the natural flow of the planning problem from the initial state towards the goal.
In contrast, regression search or backward search takes a different perspective. Rather than reasoning about individual states, regression operates over sets of states. The search begins with the set of goal states and works backward by identifying sets of predecessor states that could lead to the goal. Each expansion step yields a new set of states. We define the regression operation in the STRIPS framework as follows:

Definition 6 (STRIPS regression)**.** *The STRIPS regression $sregr(A, o)$ for a set of states A and an operator o is defined as:*

$$sregr(A, o) = \begin{cases} (A \setminus add(o)) \cup pre(o) & \textit{if } del(o) \cap A = \emptyset \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

A valid plan is found in regression when the initial state is contained within one of the regressed sets.

2.3 Heuristic Search

In state space search, we distinguish between *uninformed* and *informed* search strategies (Russell and Norvig, 2021). Uninformed strategies explore the state space by expanding states in a fixed, problem-independent order until a goal is found. In contrast, informed strategies make use of knowledge about the problem to guide the search process, and in the case of A^* prioritizing states that are more likely to lead to an optimal solution. Heuristics are cost estimates that help in finding such states.

Definition 7 (Heuristic)**.** *A heuristic h for a planning task Π is defined as a function*

$$h : A \times B \to \mathbb{R}_0^+ \cup \{\infty\}$$

where $A, B \subseteq V$ are sets of variables.

The heuristic h estimates the cost of reaching the set of variables B starting from the set A. We also define the *perfect heuristic* h^*, which returns the actual optimal cost to reach

B from A, or ∞ if B is unreachable from A. This means h^* is the perfect heuristic since it perfectly estimates the distance between two sets of variables.
An important property of practical heuristics is *admissibility.* A heuristic h is admissible if, for all sets of variables A and B, it holds that $h(A, B) \leq h^*(A, B)$. In other words, admissible heuristics never overestimate the true cost. When an admissible heuristic is used in conjunction with an admissible search algorithm such as A^* (Hart et al., 1968), the resulting plan is guaranteed to be optimal (Russell and Norvig, 2021).

In the context of progression search, heuristics are often defined as the estimated cost from a specific state s to the goal G, denoted $h(s, G)$. However, the heuristic definition above is also applicable in regression search, where we search backward from the goal. In this case, the heuristic estimate for a set of states S is given by $h(I, S)$, where I is the initial state.

Heuristics in planning are usually derived exclusively from the structure of the planning task itself, without relying on external knowledge. This makes them broadly applicable across a wide range of problems. Nevertheless, there is always a trade-off between the accuracy of a heuristic and the computational effort required to compute it. While the perfect heuristic h^* provides the best possible guidance, its computation is NP-hard and thus impractical for most tasks.
To address this, one common approach is to simplify the planning task by applying relaxations that make heuristic computation more tractable while preserving informative guidance.

2.4 Delete Relaxation

Delete relaxation is a widely used technique for simplifying the computation of heuristics in planning tasks. The central idea is to ignore the delete effects of operators, thereby creating a relaxed version of the original problem that is easier to solve.
Let $o = \langle pre(o), add(o), del(o), cost(o) \rangle$ be a STRIPS operator in a planning task. The corresponding delete-relaxed operator is defined as $o^+ = \langle pre(o), add(o), \emptyset, cost(o) \rangle$, where the delete list is removed. For a planning task $\Pi = \langle V, O, I, G \rangle$, the delete-relaxed version is given by

$$\Pi^+ = \langle V, \{o^+ \mid o \in O\}, I, G \rangle.$$

In SAS$^+$ planning tasks, delete relaxation is applied by ignoring the effects of operators that unset variable assignments. Specifically, in the relaxed version of a SAS$^+$ operator, all effects that remove the value of a variable are omitted, and only value-setting effects are considered. As a result, once a variable is assigned a value in the relaxed task, it retains that value permanently.

We denote the perfect heuristic for the relaxed problem by h^+. Since removing delete effects makes the problem easier or at worst equally difficult, h^+ is an admissible heuristic. However, computing h^+ remains NP-hard (Bonet and Geffner, 2001), which limits its direct applicability in practice.

To obtain tractable approximations of h^+, one common approach is to estimate the cost of achieving each goal variable independently. Summing these individual costs yields the heuristic h^{add}, which often provides informative estimates. However, h^{add} is not admissible, as it can overestimate the true cost when operators achieve multiple goals simultaneously.[1]

A more conservative approximation is the h^{max} heuristic, which estimates the cost of achieving a set of goal variables as the maximum cost among the individual goal variables. This avoids overcounting and ensures admissibility.

Definition 8 (h^{max} Heuristic)**.** *Let $\Pi = \langle V, I, O, G\rangle$ be a STRIPS planning task. The h^{max} heuristic for a state s and a set of variables A is defined as the greatest fixed-point solution to the following:*

$$h^{max}(s, A) = \begin{cases} 0 & \text{if } A \subseteq s, \\ \min\limits_{\langle B,o\rangle \in R(A,O)} [cost(o) + h^{max}(s, B)] & \text{if } |A| \leq 1 \text{ and } A \not\subseteq s, \\ \max\limits_{v \in A} h^{max}(s, \{v\}) & \text{otherwise,} \end{cases}$$

where $R(A, O) = \{(B, o) \mid o \in O,\ B = sregr(A, o),\ del(o) \cap A = \emptyset\}$.

If no operator satisfies the conditions such that $R(A, O)$ is defined, the minimum is taken to be ∞.
The following inequality holds for all sets of variables A and B:

$$h^{\text{max}}(A, B) \leq h^+(A, B) \leq h^*(A, B).$$

This inequality reflects the increasing informativeness and computational difficulty of the heuristics. The h^{max} heuristic simplifies the problem by assuming that achieving the most expensive subgoal implies that all other subgoals are also satisfied. Although this assumption is often too optimistic and can result in weak estimates, it leads to an admissible and efficient heuristic (Bonet and Geffner, 2001).
In the next section, we address the limitations of h^{max} by introducing the *critical path heuristic h^m*, which refines the relaxation further.

2.5 Critical Path Heuristic h^m

The critical path heuristic h^m generalizes the h^{max} heuristic by considering not just individual goal variables but subsets of goal variables of bounded size. Specifically, we define a parameter m to limit the maximum size of such subsets. The heuristic then estimates the cost of achieving a set of goals as the cost of the most expensive subset of at most m goal variables. For $m = 1$, this reduces to the h^{max} heuristic.

This generalization leads to a reinterpretation of planning as a shortest-path problem, where the goal is to find the shortest path from a source node to all other nodes in a graph (Ahuja

[1] There might be operators that add 2 or more goal variables simultaneously. In this case, h^{add} would count these operator costs multiple times and thus overestimating h^+.

et al., 1993). In this context, the source node corresponds to the initial state I, and the paths represent plans that achieve intermediate states from I. A heuristic that captures the true cost of these paths is called the *perfect regression heuristic* r^*.

Definition 9 (Perfect Regression Heuristic)**.** *The perfect regression heuristic* r^* *is defined as the greatest fixed-point solution of*

$$r^*(s,A) = \begin{cases} 0 & \text{if } A \subseteq s, \\ \min\limits_{(B,o)\in R(A,O)} [cost(o) + r^*(s,B)] & \text{otherwise,} \end{cases}$$

where $R(A,O) = \{(B,o) \mid o \in O,\ B = sregr(A,o),\ del(o) \cap A = \emptyset\}$*.*

Each pair $(B,o) \in R(A,O)$ represents a directed edge in the shortest-path graph. Intuitively, the operator o can achieve A if B was previously true.

The function r^* is equivalent to the perfect heuristic h^*, expressed in the form of a shortest-path problem. However, computing r^* is exponential in the number of variables (Haslum and Geffner, 2000). To obtain a tractable approximation, we apply a relaxation where subsets $B \subseteq A$ with $|B| > m$ are replaced with subsets $B' \subseteq B$ such that $|B'| \leq m$. This yields the *critical path heuristic* h^m.

Definition 10 (Critical Path Heuristic)**.** *The critical path heuristic* h^m *is defined as the greatest fixed-point solution of*

$$h^m(s,A) = \begin{cases} 0 & \text{if } A \subseteq s, \\ \min\limits_{(B,o)\in R(A,O)} [cost(o) + h^m(s,B)] & \text{if } |A| \leq m \text{ and } A \not\subseteq s, \\ \max\limits_{B\subseteq A, 1\leq |B| \leq m} h^m(s,B) & \text{otherwise,} \end{cases}$$

where $R(A,O) = \{(B,o) \mid o \in O,\ B = sregr(A,o),\ del(o) \cap A = \emptyset\}$*.*

The heuristic h^m approximates r^* by considering the cost of the most expensive subset $B \subseteq A$ of size at most m (i.e., the max-case). As m increases, the relaxation becomes less relevant and h^m approaches r^*. For sufficiently large m, h^m equals the perfect regression heuristic r^*. Since achieving a subset of goals can never be more costly than achieving the full set, h^m is admissible.
The computational complexity of evaluating h^m grows exponentially with m, as all subsets of size up to m must be considered. In this thesis, we focus on the case $m = 2$, for which h^2 can be computed in polynomial time (Haslum and Geffner, 2000).

The critical path heuristic h^m can be employed in both progression and regression search. In progression, the heuristic estimate is $h^m(s,G)$, and in regression, it is $h^m(I,S)$. In practice, it can be advantageous to precompute a look up table containing all values $h^m(s,A)$ for sets A with $|A| \leq m$. This allows efficient evaluation of the max-case. However, in progression search, the state s changes for each heuristic evaluation, requiring different tables. In regression search, a single precomputed table of $h^m(I,A)$ values suffices for all evaluations. Therefore, regression search is theoretically better suited for critical path heuristics with $m = 2$.

3 Existing Implementation

Fast Downward (Helmert, 2006) offers the critical path heuristic as part of its heuristic search algorithms. This chapter describes the design and structure of the existing implementation in the planning system, as well as important processes around it. We first introduce the general heuristic framework of Fast Downward. We describe how problems are processed and stored. Then we discuss the existing implementation of h^m. Finally, we conduct a runtime and space complexity analysis of the heuristic computation.

3.1 Heuristic Framework in Fast Downward

Fast Downward translates problem instances into SAS$^+$ representation (Definition 3). The central object used to store variables in this form is the `atom`. An atom consists of an integer, which is the variable identifier (`var`), and another integer representing the value a variable can take (`value`). All information about problems, such as operators, the initial state, or goal states, is stored in a task object that can be accessed via a `task_proxy`. The `task_proxy` is the interface for retrieving information about the task during the search process. The structure of a `task_proxy` is similar to our definition of tasks in SAS$^+$ representation (Definition 3).

If the specified search parameter uses a heuristic-based algorithm, states must be continuously evaluated. Each heuristic has an individual class where the implementation is contained. All specific heuristic classes extend a basic heuristic class, which provides the necessary framework to be used in search algorithms. The `compute_heuristic` function is the main method in each heuristic class. This function takes the current state as an argument and returns an integer that represents the heuristic value of the state. We now discuss the existing h^m implementation in Fast Downward.

3.2 Structure of the Implementation

The implementation of h^m is encapsulated in the `hmheuristic` class. The heuristic values are computed by iteratively propagating costs through the relaxed state space. These costs are maintained in a table that stores entries for all atom sets of size up to m. Since Fast

Downward performs progression search, the heuristic table is specific to each state and must be recomputed for every heuristic evaluation.
The algorithms in this chapter and in the ones following are presented in a code-oriented pseudo code style, closely resembling C++ conventions. This is intentional, as it aligns with the exact implementation in Fast Downward and helps in understanding the choice of data structures and the precise order of operations. These play a crucial role in the efficiency of the heuristic computation.

The primary components of the implementation are the initialization of the heuristic table, the iterative updates to table entries, and the evaluation of goal subsets (Algorithm 1). Those three components are sequentially called in the `compute_heuristic` method (lines 3 - 5). In the following, we take a closer look at each of these steps.

Algorithm 1 Compute h^m value for given state

```
Procedure: compute_heuristic
Input: State state, task information task_proxy
Output: Heuristic value of state
1 if is_goal_state(task_proxy, state) then
2     return 0
3 init_hm_table(state)
4 update_hm_table()
5 h ← eval(goals)
6 if $h = \infty$ then
7     return DEAD_END
8 return h
```

3.2.1 Heuristic Table Initialization

The heuristic table initialization begins with the generation of all possible subsets of atoms up to size m. These subsets are stored in a `std::vector` data structure. The heuristic table, called `hm_table`, is initialized with all combinations of atoms of size $\leq m$. This is achieved using a recursive function that first generates all sets of atoms of size 1. These sets are then recursively extended until size m is reached. The `hm_table` is implemented using a `std::map`, which takes atom sets as keys and stores the heuristic value of each set of atoms. It is therefore important that atom sets are always sorted. Otherwise, the same set could have different entries in the heuristic table. As soon as new atom sets are created to use them as table keys, they must be stored in the vector in sorted order.
Once all atom sets of size $\leq m$ are generated and initialized, the heuristic table is populated with initial integer values for the given state. A heuristic value of zero is assigned to sets of atoms fully contained in the state, while all other sets are initialized with infinity.[2] To find table entries contained in the state, the implementation checks if the entry is a subset of the state.

[2] In the implementation, infinity is represented by the largest possible integer value.

3.2.2 Iterative Updates to the Heuristic Table

The core computation of h^m is iteratively propagating heuristic values until no further improvements are possible (Algorithm 3). Propagation is performed by iterating over all operators and evaluating its preconditions to determine the current cost of applying the operator. Evaluating the preconditions of an operator o involves examining all partial preconditions $\{p \subseteq pre(o) \mid |p| \leq m\}$. This procedure is similar to $\max_{B \subseteq A, 1 \leq |B| \leq m} h^m(s, B)$ in our h^m definition for $h(s, A)$ (Definition 10). A sketch of this evaluation method is provided in Algorithm 2.

Algorithm 2 Evaluate maximum heuristic value of set of atoms

```
Procedure: eval
Input: Set of atoms atom_set, heuristic table hm_table
Output: Evaluation for all subsets of atom_set
1  partial_sets ← generate_all_partial_sets(atom_set)
2  max ← 0
3  foreach set in partial_sets do
4      h ← hm_table[set]
5      if h > max then
6          max ← h
7  return max
```

If there are no subsets of preconditions where the table entry is infinity, the operator is applicable. This means that all partial effects of an operator o can be achieved with cost eval($pre(o)$) + $cost(o)$. Entries of effects in the hm_table are only updated if the new value is smaller than the current value.

Algorithm 3 Update heuristic table until fixpoint is reached

```
Procedure: update_hm_table
Input: task_proxy with operators, integer m
Output: hm_table containing correct $h^m$ values
1  repeat
2      was_updated ← false
3      foreach o in O do
4          c1 ← eval(pre(o))
5          new_cost ← c1 + cost(o)
6          if c1 ≠ ∞ then
7              partial_effs ←generate_all_partial_sets(eff(o))
8              foreach partial_eff in partial_effs do
9                  if hm_table.at(partial_eff) > new_cost then
10                     hm_table[partial_eff] ← new_cost
11                     was_updated ← true
12                 if |partial_eff| < m then
13                     extend_entry(partial_eff, o)
14 until was_updated is false;
```

We define heuristic values in the table after evaluating the i-th operator as h_i. The procedure iterates over all operators until there are no updates possible. In this case, the was_updated

variable is not set to *true* (line 11) and the procedure terminates. This is exactly the case if it holds that $h_{i \times |O|} = h_{(i+1) \times |O|}$ for $i \in \mathbb{N}_0$. At this point, the fix-point is reached and $h_{i \times |O|} = h_{(i+1) \times |O|} = h^m$.
Algorithm 3 updates the heuristic table for all partial effects of operators of size up to m.

Definition 11 (Updates in `update_hm_table`)**.** *In Algorithm 3, heuristic values are updated for an operator $o \in O$, atom sets $X \subseteq \mathit{eff}(o)$ and $1 \leq |X| \leq m$ using the rule:*

$$h_{i+1}(X) = \min\left(h_i(X), \max_{A \subseteq pre(o), 1 \leq |A| \leq m} h_i(A) + \mathit{cost}(o)\right)$$

However, it is also necessary to update entries that only partially contain effects of an operator. This is enforced by the `extend_entry` function (Algorithm 4).

Algorithm 4 Extend entry and update heuristic table

```
Procedure: extend_entry
Input: Set of atoms atom_set and operator o
Output: Updates to all hm_table entries partially containing atom_set
foreach entry in hm_table do
    contradict ← false
    foreach atom in entry do
        if contradict_effect_of(o, atom) then
            contradict ← true break
    if not contradict and |entry| > |atom_set| and atom_set ⊂ entry then
        pre ← pre(o)
        foreach atom in entry do
            if atom not in atom_set and atom not in pre then
                pre.insert(atom)
        sort(pre)
        vars ← ∅
        is_valid ← true
        foreach atom in pre do
            if atom.var in vars then
                is_valid ← false break
            vars.insert(atom.var)
        if is_valid then
            c2 ← eval(pre)
            if hm_table.at(entry) > c2 + cost(o) then
                hm_table[entry] ← c2 + cost(o)
                was_updated ← true
```

We illustrate the necessity of `extend_entry` with a small example: Let there be a set of variables $V = \{a, b\}$ with domains $\mathit{dom}(a) = \mathit{dom}(b) = \{0, 1\}$ and an operator o where $\mathit{pre}(o) = \{a = 1\}$ and $\mathit{eff}(o) = \{b = 1\}$. [3] Furthermore, we assume that $m = 2$.
If o is applicable in the current state, the table entry for $[b = 1]$ might be updated when

[3] In Fast Downward, variable names are translated to integer values. For clarity, we use letters in this example.

applying Algorithm 3. We therefore need to consider a table update for all sets of atoms containing $b = 1$. However, to update these entries, we must ensure that the other elements in the atom set are also achievable. For example, when considering the entry $[a = 0, b = 1]$, we evaluate not only the preconditions of o but extend this set with the atoms $a = 0$ in the `eval` function.

Definition 12 (Updates in `extend_entry`)**.** *In Algorithm 4, heuristic values are updated for an operator $o \in O$ and an atom set X with $1 \leq |X| \leq m$ if there exists some subset $Y \subseteq X$ such that $Y \subseteq \mathit{eff}(o)$:*

$$h_{i+1}(X) = \min\left(h_i(X), \max_{B \subseteq (pre(o) \cup (X \setminus Y)), |B| \leq m} h_i(B) + cost(o)\right)$$

There are additional restrictions that must be applied, such as ensuring that the extended set of atoms does not contradict a precondition or an effect of the operator. In this context, a contradiction means that two different values are simultaneously assigned to the same variable. In our example, for the entry $[a = 0, b = 1]$ this would be the case as $pre(o)$ does not allow the atom $a = 0$.

3.2.3 Evaluating Goal Subsets

The heuristic value for the current state is computed by evaluating the maximum cost among all goal subsets. This step uses the `eval` method, which retrieves precomputed values from the heuristic table for all partial atom sets of the goal (Algorithm 2). Since this is done after the termination of Algorithm 3, the `hm_table` contains the final h^m values. If any of these entries still has a value of infinity, the current state is marked as a dead end and is pruned in the search process.

3.3 Runtime and Space Analysis

Analyzing the runtime and space usage of the existing h^m implementation in Fast Downward can be useful as it highlights potential bottlenecks and inefficiencies. Therefore, we conduct a worst-case runtime and space complexity analysis. The performance of the heuristic depends on several parameters, which we will introduce now:

- The total number of variables $|V|$,
- the largest domain size of all variables $d = \max_{v \in V} |dom(v)|$,
- the number of operators in the planning task $|O|$,
- and the maximum subset size of atoms m.

First, we utilize these variables to estimate other quantities. In the following, we will always indicate the worst-case values for these quantities. The number of different atoms is $|V| \times d$. This, in turn, means that we can use $(|V| \times d)^m$ as an upper-bound for the number of entries in `hm_table`. Actually, it holds that no entry contains the same variable $v \in V$ twice. This corresponds to the number of ways to choose m distinct elements without replacement from a set of size $|V|$ which is $\binom{|V|}{m}$. A more precise limit for the number of entries is therefore $\binom{|V|}{m} \times d^m$. However, since we are only carrying out a worst-case estimate, we will not use the binomial coefficient and instead consider $(|V| \times d)^m$ as the worst-case number of table entries.

We also need to assess the worst-case number of operator evaluations in Algorithm 3 which we call i_{max}. We know that all operators are evaluated subsequently. This means that all entries whose optimal distance from the initial state is exactly one operator are set to the correct heuristic value after $h_{|O|}$. Since this must apply to at least one entry, we can guarantee that one entry is set to the correct value after $|O|$ operator evaluations. Therefore, in the worst case, we need to evaluate $i_{max} = (|V| \times d)^m \times |O|$ operators.
Other variables cannot be further quantified, as they depend entirely on the planning task or, in the case of m, on the heuristic.

3.3.1 Runtime Complexity

We divided the implementation into three sequential parts: Table initialization, updating the heuristic table, and goal state evaluation. Initializing the heuristic table involves generating all table entries and populating them with the initial heuristic value. The recursive function generating all atom sets has a complexity of are $\mathcal{O}(|V|^m \times d^m)$. The initial value for an entry is a simple subset check of all state atoms which is $\mathcal{O}(|V|)$. Therefore, the total complexity of the table initialization is $\mathcal{O}(|V|^{m+1}d^m)$.

Now we examine the evaluation of the goal atom set (Algorithm 2). In there, all preconditions are divided into subsets $\leq m$ and looked up in the table. Preconditions can have a maximum length of $|V|$. This results in $|V|^m$ partial preconditions. The `hm_table` is a map that is implemented as a binary tree. Look ups are possible in logarithmic runtime. This implies a complexity of $\mathcal{O}(|V|^m) \times \mathcal{O}(\log(|V|^m \times d^m)) = \mathcal{O}(|V|^m \log(|V| \times d))$.

The table update algorithm consists of looping over all operators until no further updates are possible (Algorithm 3). The complexity of this loop is $i_{max} = \mathcal{O}(V^m \times d^m \times |O|)$. In the inner loop, we iterate over all partial effects of an operator o of size $\leq m$. In the worst case, $\mathit{eff}(o)$ contains each variable once, which results in $\mathcal{O}(|V|^m)$. This leads to a complexity of $\mathcal{O}(|O||V|^{2m} \times d^m)$ for the the nested loops. Within the loop, partial effects are updated which is a table lookup and thus $\mathcal{O}(\log(|V| \times d))$. Afterward, the `extend_entry` function is called. Since it holds that `extend_entry` is only called on partial effects $< m$, the total complexity of the algorithm is $\mathcal{O}(|O||V|^{2m-1}d^m) \times \mathcal{O}(\texttt{extend\_entry})$.

The `extend_entry` method consists of a loop over the entire heuristic table. We have already determined the size of the table as $(|V| \times d)^m$. Inside the loop, there are many sequential operations. First, it is checked for each atom in the entry that it does not contradict an effect of the given operator (lines 3 - 6), which is $\mathcal{O}(|V|)$. Line 6 also checks whether the entry from the table is actually a superset of the specified `atom_set`. In terms of runtime complexity, this means that we no longer execute the following code for all table entries, but instead have a fix subset `atom_set`. Therefore, the complexity is $\mathcal{O}((|V| \times d)^{m-|\texttt{atom\_set}|})$. Because we have already made the worst-case assumption in algorithm 3 that $|\texttt{atom\_set}| = m - 1$, the resulting complexity is $\mathcal{O}((|V| \times d)^{m-(m-1)}) = \mathcal{O}(|V| \times d)$.

Afterwards, the preconditions are searched for contradictions with the current table entry (lines 8 - 10, 13 - 16) and sorted (line 11). Both operations do not exceed a complexity of $\mathcal{O}(|V| \log |V|)$ as the maximum size of preconditions is $|V|$. The `eval` function (line 18) with a complexity of $\mathcal{O}(|V|^m \log(|V| \times d))$ is the most expensive operation within these sequential steps. Therefore, it holds that $\mathcal{O}(\texttt{extend\_entry}) = \mathcal{O}(|V| \times d) \times \mathcal{O}(\texttt{eval})$.

By multiplying the complexity of Algorithm 2, 3 and 4, we can conclude for the runtime of $\mathcal{O}(\texttt{update\_hm\_table})$:

$$\begin{aligned}
\mathcal{O}(\texttt{update\_hm\_table}) &= \mathcal{O}(|O||V|^{2m-1}d^m) \times \mathcal{O}(\texttt{extend\_entry}) \\
&= \mathcal{O}(|O||V|^{2m-1}d^m) \times \mathcal{O}(|V| \times d) \times \mathcal{O}(\texttt{eval}) \\
&= \mathcal{O}(|O||V|^{2m-1}d^m) \times \mathcal{O}(|V| \times d) \times \mathcal{O}(|V|^m \log(|V| \times d)) \\
&= \mathcal{O}(|O||V|^{3m}d^{m+1} \log(|V| \times d))
\end{aligned}$$

It holds that updating the heuristic table is more expensive then initializing the table as well as evaluating the goal set of atoms. The term above therefore is the runtime complexity of the existing implementation in Fast Downward.
When substituting $m = 2$ into the complexity expression, it holds that

$$\mathcal{O}(\texttt{update\_hm\_table}) = \mathcal{O}(|O||V|^6 d^3 \log(|V| \times d))$$

The result is a complexity that depends on the sixth power of the number of variables $|V|$ and the domain size d to the power of three. It is obvious that this runtime is problematic, especially for larger problem instances with large sets of variables.

3.3.2 Space Complexity

The space complexity is dominated by the storage requirements of the heuristic table. The table contains one entry for each atom set of size $\leq m$. The total space of the map is the number of entries multiplied by the space required for a single entry. The `hm_table` contains $(|V| \times d)^m$ entries. Each entry requires space for the atom set itself and the associated heuristic integer value. A set of atoms is a vector of atoms, and since atoms consist of two integer values, the memory complexity of a atom is $\mathcal{O}(1)$. Vectors have a length $\leq m$, which results in $\mathcal{O}(m)$.
The total space complexity is therefore $\mathcal{O}((|V| \times d)^m) \times \mathcal{O}(m) = \mathcal{O}((|V| \times d)^m)$. With the existing implementation approach, saving the entire heuristic table is unavoidable. Since this is the only data structure that requires a large amount of space, the space usage of the existing implementation is difficult to improve.

4 Optimization of the Existing Implementation

As already seen in Section 3.2, the implementation existing in Fast Downward stores the heuristic value for each atom set of size $\leq m$ of a task. The final heuristic value can thus be calculated by looking up goal subsets in the table. The values in the table must be recalculated for each evaluation, as the original state has changed. Now, we optimize this approach by choosing more suitable data structures and saving reusable calculations. Then, we compare our potential improvements with the original heuristic. To do this, we conduct an experiment using all problems of the International Planning Competition (IPC) 1998 - 2023.

To better distinguish between the implementations, we call the heuristic computed by the existing one h^m, while we refer to our optimized version as h^2.

4.1 Improvements

In this work we are specifically interested in the case $m = 2$. This specification allows us to simplify some data structures. We know that all table entries consist of a maximum of two atoms. Therefore, instead of a `std::vector`, we can use a pair data structure to store atom pairs. Entries containing only a single atom are stored with an additional placeholder atom. For n different atom variables, integers from 0 to $n - 1$ are used as identifiers. Therefore, we assign all placeholder atoms the variable -1.

A further optimization involves the generation of all atom sets with size $\leq m$. In the existing implementation, recursive function calls are used to obtain all these sets (Section 3.2.1). In the case of $m = 2$, we can generate the pairs iteratively with two for-loops. It still is important that all generated pairs are correctly ordered so that they can be found in the `hm_table`.

These changes primarily help to make the implementation clearer. It is not expected to majorly improve runtime and space complexity of h^2. However, the clearer code allows for faster detection of possible bottlenecks and optimizations. Now, we introduce various ideas and concepts that speed up the existing implementation. First we introduce new data structures or modify existing ones, then we optimize specific algorithms.

4.1.1 Heuristic Table Data Structure

To improve the efficiency, we change the data structure of the heuristic table. In the existing implementation, a `std::map` is utilized. This data structure organizes its keys in a sorted manner and offers logarithmic access time for insertions, look ups and deletions. While the sorting feature of `std::map` is advantageous in scenarios requiring ordered traversal, it introduces unnecessary overhead when fast look ups are required.

In our context, the order of pairs is not as crucial as the ability to quickly find and access stored values. To this end, we replace `std::map` with `std::unordered_map`, a hash-based data structure. An `unordered_map` provides average constant-time complexity for insertions and look ups. Hash functions are used to determine the location of each element in the underlying hash table. When we iterate over entries in the table, the order is not deterministic. Therefore, switching the data structure means no longer having the option of making improvements by optimizing the traversal order of the table.

To efficiently use an `unordered_map`, a suitable hash function for pairs of atoms must be defined. A hash function maps a given input (in this case, a pair of atoms) to a numerical value. Each atom comprises a variable integer and a value integer. Our hash function for a pair of atoms combines the hash values of the individual atoms as follows:

$$\text{hash}(a_1, a_2) = (a_1.var \times p + a_1.value) \times p + (a_2.var \times p + a_2.value)$$

Here, p is a large prime number. This function ensures that the hash value of each pair depends on both its components. Furthermore, choosing a large prime number reduces hash collisions. The prime number p should be greater than the number of variables as well as the number of values in a domain.

Default hash functions for `unordered_map` use `size_t`, an unsigned integer type. If values exceed its limit, they wrap around and start again at zero. To prevent integer overflow, p should not be too large, as overflows could lead to hash collisions. In our implementation we choose $p = 100003$ which meets these requirements.

4.1.2 Operator Caches

To optimize operator handling, several auxiliary data structures, referred to as operator caches, are initialized and used throughout the implementation. These caches include the *precondition cache*, the *partial effect cache*, and the *effect conflict cache*. They are all created before the search process starts as they are independent of the state to evaluate. In the following, we show how these operator caches are storing and reusing relevant operator information. Conducted experiments show that all three caches significantly improve the runtime of h^2. Furthermore, we analyze the additional space complexity for each cache. Note that with these caches, we do not introduce new concepts to the implementation but rather speed up the retrieval of operator information and reduce sort operations on operator preconditions and effects.

Precondition Cache

The precondition cache stores the preconditions of each operator as a sorted vector of atoms. When evaluating an operator, preconditions must be retrieved from the `task_proxy`. By caching these preconditions in sorted order, redundant sorting operations are avoided.
Regarding space complexity, the maximum size of preconditions is the number of variables $|V|$. Since we store preconditions for all operators, the space complexity is $\mathcal{O}(|O||V|)$.

Partial Effect Cache

The partial effect cache is designed to handle all subsets of operator effects with sizes up to two. For each operator, all pairs of effects are precomputed, sorted, and stored in a vector. Single effects are saved in the list as well with an atom placeholder at the second position. In contrast to the preconditions, we directly store effects split up into pairs, as we only need them for updating the table directly after operator evaluations. This saves us having to generate effect subsets, but requires more space. Preconditions, on the other hand, are used as a list in various functions. It therefore makes less sense to store partial preconditions in a cache.
In the worst case, partial effects have a space complexity of $\mathcal{O}(|V|^2)$ for $m = 2$. We can conclude that the total space complexity is $\mathcal{O}(|O||V|^2)$.

Effect Conflict Cache

Finally, the effect conflict cache stores a set of all effect variables for each operator. The cache is used in the `extend_entry` method to detect conflicts between certain atoms and effects of an operator. For better understanding, we take a look at a quick example:
An operator o has an effect $v := d$ where $v \in V$ and $d \in \mathit{dom}(v)$. By definition, the `extend_entry` function tries to update entries containing partial effects of o. Since it holds that $m = 2$, those entries consist of an atom $\in \mathit{eff}(o)$ and an additional atom a. There are a few restrictions for a, where table updates can be ruled out:

- If atom a is $v = d$, it is already guaranteed to be covered when all partial effects of o are applied.
- If atom a is $v = b$, where $b \in (\mathit{dom}(v) \setminus \{d\})$, it directly conflicts with the effect $v := d$. This means no table updates are possible in this case.

In both cases, there are guaranteed no table updates when atom a contains v as variable. The effect conflict cache stores this type information for every operator.
On a programming level, the cache is a two-dimensional boolean vector, where the first index corresponds to the operator ID and the second index to the atom variable. It holds that the space complexity is $\mathcal{O}(|O||V|)$. The effect conflict cache directly replaces the search for contradictions in `extend_entry` with a constant-time look up (Algorithm 4, lines 2 - 5).

4.1.3 Table Initialization

As previously mentioned, the heuristic table is initialized and filled using two nested for-loops. In the h^m heuristic, each table entry requires checking whether the entry is contained in the state to evaluate. Originally, this involves repeated subset checks, which can be inefficient.
This overhead is avoided in h^2 by first storing the atoms of the state to evaluate into an `unordered_set`. After creating this set of state atoms, subset checks for each table entry can be performed in constant time. As a result, the initialization step is reduced to a simple iteration over the heuristic table, clearly improving the runtime complexity.

4.1.4 Operator Queue

In order to update the heuristic table, all operators are iterated over until no further updates are possible (Algorithm 3). As a result, operators are constantly being evaluated regardless if there may be an improvement or not. We introduce an operator queue for h^2 that ensures that operators that bring guaranteed no improvements to the `hm_table` are skipped in the updating process.
Instead of blindly iterating over all operators, we maintain an operator queue that dynamically tracks operators that might be relevant for updating the table. Whenever a precondition of an operator is updated in the heuristic table, the operator may be achievable with lower cost. We therefore insert this operator into the queue. In addition, if entries are updated that only partially contain preconditions of an operator, this operator is also added to the queue.
If the queue is empty we can guarantee that no operator leads to further improvements. In this case, the fix-point is reached and the correct h^2 values are present in the table.

We now establish a sequence of lemmas that characterize how heuristic values in the `hm_table` evolve throughout the evaluation of operators in an SAS$^+$ planning task. These results culminate in a condition that determines when evaluating an operator does not lead to any further updates of the heuristic table (Theorem 1).

Lemma 1. *Let $\Pi = \langle V, O, I, G\rangle$ be an SAS$^+$ planning task, and let $h(S)$ denote the heuristic values in `hm_table` for an atom set S with $1 \leq |S| \leq m$. For any two sets of atoms A, B such that $A \subseteq B$ with $1 \leq |A| \leq m$ and $1 \leq |B| \leq m$, it holds that $h(A) \leq h(B)$.*

Proof sketch. Since h computes the maximum cost over all subsets of size up to m, and since achieving atoms in A can always be done by reaching B, $h(A)$ cannot be higher then $h(B)$. Hence, $h(A) \leq h(B)$ for $A \subseteq B$. $\square$

This monotonicity property ensures that as atom sets grow, their heuristic estimates do not decrease. We now turn to the question of whether heuristic values for subsets of the precondition of an operator can change during its evaluation.

Lemma 2. *Let $\Pi = \langle V, O, I, G\rangle$ be an SAS$^+$ planning task, and let $h(S)$ denote the heuristic values in `hm_table` for atom sets S with $1 \leq |S| \leq m$. Let $o \in O$ be an operator, and let*

h_i and h_{i+1} denote the heuristic tables before and after evaluating o. Then it holds that for all sets $B \subseteq pre(o)$ with $1 \leq |B| \leq m$, we have
$h_{i+1}(B) = h_i(B)$.

Proof. We establish that $h_{i+1}(B) = h_i(B)$ by examining the specific methods that update heuristic values. There are exactly two instances where this occurs.

Procedure `new_update_hm_table` (Algorithm 5): Heuristic values are updated for atom sets $X \subseteq \mathit{eff}(o)$ and $1 \leq |X| \leq m$ using the rule:

$$h_{i+1}(X) = \min\left(h_i(X), \max_{A\subseteq pre(o), 1\leq|A|\leq m} h_i(A) + cost(o)\right)$$

In the case that $X \subseteq pre(o)$, we directly see that the left term in the minimum dominates, therefore:

$$h_{i+1}(B) = h_i(B).$$

Procedure `new_extend_entry` (Algorithm 6): Heuristic values are updated for atom sets X with $1 \leq |X| \leq m$ such that some subset $Y \subset X$ lies in $\mathit{eff}(o)$. Furthermore, note that h_i at this point may contain updated heuristic values by applying Algorithm 5. Therefore we call heuristic values during evaluation i containing these updates h'_i. The update is:

$$h_{i+1}(X) = \min\left(h'_i(X), \max_{A\subseteq (pre(o)\cup(X\setminus Y)), 1\leq|A|\leq m} h'_i(A) + cost(o)\right)$$

Since we are interested in the case $X \subseteq pre(o)$ and we know that Algorithm 5 did not update entries for $B \subseteq pre(o)$, it holds that $h'_i(B) = h_i(B)$.
For $X \subseteq pre(o)$, by Lemma 1 and $X \subseteq (pre(o) \cup (X \setminus Y))$, it holds that the left term in the minimum term dominates. We can conclude that:

$$h_{i+1}(B) = h_i(B).$$

For both update procedures, the values $h(B)$ with $B \subseteq pre(o)$ and $1 \leq |B| \leq m$ remain unchanged during the evaluation of o. □

We have now shown that heuristic values associated with preconditions of an operator remain stable during its evaluation. This insight allows us to reason about the effect of evaluating the same operator multiple times consecutively.

Lemma 3. *Let $\Pi = \langle V, O, I, G\rangle$ be an SAS$^+$ planning task, and let $h(S)$ denote the heuristic values in `hm_table` for atom sets S with $|S| \leq m$. Let $o \in O$ be an operator, and suppose that h_i is the heuristic table after evaluating o, and h_{i+1} is the heuristic table after a second, consecutive evaluation of the same operator o. Then it holds that $h_{i+1} = h_i$.*

Proof sketch. We show that a second application of operator o does not lead to further updates in the heuristic table.
From Lemma 2, it follows that the cost of reaching $pre(o)$ has not decreased from h_{i-1} to h_i. Therefore, re-evaluating o brings no further information to the heuristic table as the updating terms for `new_update_hm_table` and `new_extend_entry` remain the same for h_{i+1}. We conclude that $h_{i+1} = h_i$. □

Having shown that repeated application of the same operator has no effect on the heuristic values, we now establish a general monotonicity result. Heuristic values never increase across operator evaluations.

Lemma 4. *Let $\Pi = \langle V, O, I, G\rangle$ be a SAS$^+$ planning task, and let h_i denote the heuristic values after i operator evaluations in `hm_table`. Then, for any atom set S with $1 \leq |S| \leq m$, it holds that $h_{i+1}(S) \leq h_i(S)$.*

Proof. We establish that $h_{i+1}(S) \leq h_i(S)$ by examining the algorithm steps responsible for modifying heuristic values. We verify the statement for both cases.

Procedure `new_update_hm_table` (Algorithm 5): In Algorithm 5, heuristic updates are considered for an operator $o \in O$ and an atom set X with $1 \leq |X| \leq m$ if $X \subseteq \mathit{eff}(o)$ (line 10).

$$h_{i+1}(X) = \min\left(h_i(X), \max_{B \subseteq pre(o), 1 \leq |B| \leq m} h_i(B) + \mathit{cost}(o)\right)$$
$$h_{i+1}(X) \leq h_i(X)$$

Procedure `new_extend_entry` (Algorithm 6): In Algorithm 6, heuristic updates are considered for an operator $o \in O$ and an atom set X with $1 \leq |X| \leq m$ if there exists some subset $Y \subset X$ such that $Y \subseteq \mathit{eff}(o)$ (line 15). To make this distinction explicit, we define X as consisting of two parts: Y, which satisfies $Y \subseteq \mathit{eff}(o)$, and Z, which is given by $Z = X \setminus Y$ and contains the remaining elements of X.
Again, we call the table containing by Algorithm 5 updated heuristic values h'_i.

$$h_{i+1}(X) = \min\left(h'_i(X), \max_{B \subseteq (pre(o) \cup X), 1 \leq |B| \leq m} h'_i(B) + \mathit{cost}(o)\right)$$
$$h_{i+1}(X) \leq h'_i(X) \leq h_i(X)$$

Thus, for both procedures that update heuristic values, we conclude that $h_{i+1}(S) \leq h_i(S)$ for any atom set S with $1 \leq |S| \leq m$. □

With the above results, we are now in a position to precisely characterize when the evaluation of an operator does not lead to any updates in the heuristic table. This leads us to the central theorem of this section.

Theorem 1. *Let $\Pi = \langle V, O, I, G\rangle$ be an SAS$^+$ planning task, and let $h(S)$ denote the heuristic values in `hm_table` for an atom set S where $1 \leq |S| \leq m$. Define h_i as the heuristic values after the evaluation of an operator $o \in O$, let h_j be the heuristic values before evaluating o, with $i \leq j$, and denote by h_{j+1} the values after this evaluation. Then, it holds that $h_{j+1} = h_j$ if for all sets A with $1 \leq |A| \leq m$ and $A \cap pre(o) \neq \emptyset$, we have $h_j(A) = h_i(A)$.*

Proof. Lemma 3 already established that evaluating an operator $o \in O$ twice in succession does not bring any updates to the heuristic table. Hence, in the case of $i = j$, it holds that $h_{j+1} = h_j$.

Thus, we focus on the case $i < j$, where other operators may have been evaluated between evaluations of o.

Procedure `new_update_hm_table` (Algorithm 5): In Algorithm 5, heuristic updates are considered for an operator $o \in O$ and an atom set X with $1 \leq |X| \leq m$ if $X \subseteq \mathit{eff}(o)$.

$$h_{j+1}(X) = \min\left(h_j(X), \max_{B \subseteq pre(o), 1 \leq |B| \leq m} h_j(B) + cost(o)\right)$$

By assumption, we have $h_j(A) = h_i(A)$ for all $A \cap pre(o) \neq \emptyset$ with $1 \leq |A| \leq m$.

$$h_{j+1}(X) = \min\left(h_j(X), \max_{B \subseteq pre(o), 1 \leq |B| \leq m} h_i(B) + cost(o)\right)$$

It holds that $h_{i+1}(X) \leq \max_{B \subseteq pre(o), 1 \leq |B| \leq m} h_i(B) + cost(o)$. By Lemma 4, heuristic values cannot decrease after an operator evaluation. It follows that $h_j(X) \leq h_{i+1}(X) \leq \max_{B \subseteq pre(o), 1 \leq |B| \leq m} h_i(B) + cost(o)$.

$$h_{j+1}(X) = h_j(X)$$

Procedure `new_extend_entry` (Algorithm 6): In Algorithm 6, heuristic updates are considered for an operator $o \in O$ and an atom set X with $1 \leq |X| \leq m$ if there exists some subset $Y \subset X$ such that $Y \subseteq \mathit{eff}(o)$. It holds that $Z = X \setminus Y$. Since applying Algorithm 5 did not alter the heuristic table, $h'_j = h_j$.

$$h_{j+1}(X) = \min\left(h_j(X), \max_{B \subseteq (pre(o) \cup Z), 1 \leq |B| \leq m} h_j(B) + cost(o)\right).$$

We now consider two cases for the maximization term:

1. $B \subseteq Z$, containing only elements of Z,
2. $B \subseteq (pre(o) \cup Z)$ with $B \cap pre(o) \neq \emptyset$.

Thus, we rewrite the equation as:

$$h_{j+1}(X) = \min\left(h_j(X), \max\left(\max_{B \subseteq Z} h_j(B), \max_{B \subseteq (pre(o) \cup Z), B \cap pre(o) \neq \emptyset, 1 \leq |B| \leq m} h_j(B)\right) + cost(o)\right).$$

Since $|Z| < m$, there exists a set $C = Z \cup \{a\}$ for some $a \in pre(o)$ that is included in the second maximization term. Because $Z \subset C$, it follows that $h_j(Z) \leq h_j(C)$, implying that the second maximization term always dominates. Hence, we obtain:

$$h_{j+1}(X) = \min\left(h_j(X), \max_{B \subseteq (pre(o) \cup Z), B \cap pre(o) \neq \emptyset, 1 \leq |B| \leq m} h_j(B) + cost(o)\right).$$

By assumption, for all sets A with $1 \leq |A| \leq m$ and $A \cap pre(o) \neq \emptyset$, we have $h_j(A) = h_i(A)$, yielding:

$$h_{j+1}(X) = \min\left(h_j(X), \max_{B \subseteq (pre(o) \cup Z), B \cap pre(o) \neq \emptyset, 1 \leq |B| \leq m} h_i(B) + cost(o)\right).$$

Since $h_{i+1}(X) \leq max_{B \subseteq (pre(o) \cup Z), B \cap pre(o) \neq \emptyset, 1 \leq |B| \leq m} h_i(B) + cost(o)$, and $h_j(X) \leq h_{i+1}(X)$ (Lemma 1). We conclude that

$$h_{j+1}(X) = h_j(X).$$

Thus, for both procedures that modify heuristic values, it follows that $h_{j+1} = h_j$, completing the proof. □

We see that it is not only necessary for subsets of preconditions to remain unchanged but that each entry containing at least one precondition must have the same heuristic value to disregard the operator. An operator being reachable with the same costs as in the last evaluation does not exclude possible updates in the `extend_entry` method.

The operator queue is implemented using a `std::deque` data structure according to the first-in-first-out principle. We initialize the queue by inserting operators that are applicable in the initial state. Meanwhile, we maintain a list that tracks which operators have a given atom as a precondition. To achieve this, we use an `unordered_map` where atoms serve as keys, and each key maps to a list of operator IDs. We call this data structure *operator dictionary*. When updating a table entry, any operator that has one of the corresponding atoms in the entry as a precondition is inserted into the queue (Algorithm 5, line 15).

Algorithm 5 Update h^m Table

```
Procedure: new_update_hm_table
Input: task_proxy with operators, op_queue containing initially applicable operators
Output: hm_table containing correct $h^m$ values
1  while not op_queue.empty() do
2      o ← op_queue.pop()
3      op_queue_set.erase(o)
4      c1 ← eval(precondition_cache[o])
5      new_cost ← c1 + cost(o))
6      if $c1 =$ op_cost[o] then
7          if $c1 \neq \infty$ then
8              extend_changed_entry(o)
9          continue
10     changed_entries[o] ← {}
11     op_cost[o] ← c1
12     foreach partial_eff in partial_effect_cache[o] do
13         if hm_table.at(partial_eff) > new_cost then
14             hm_table[partial_eff] ← new_cost
15             add_op_to_queue(partial_eff)
16         if partial_eff.second.var $= -1$ then
17             new_extend_entry(partial_eff.first, o, c1)
```

We want to avoid an operator being in the queue twice at the same time. However, searching in the queue can only be done in linear time. As we have to possibly add operators for every table update, this results in a large overhead. Therefore, in addition to the queue itself, we maintain an `unordered_set` which contains the IDs of all operators in the queue. This guarantees constant checks as whether an operator is in the queue. Inserting and deleting elements of this set is always done simultaneously with the queue.

An interesting special case are operators without preconditions. These obviously are applicable in the initial state and are therefore added when the queue is initialized. Then, they would never be added to the queue again during the course of the algorithm, as they are not affected by any table update. However, there are scenarios where such operators have to be applied multiple times to ensure a correct algorithm. Let us consider the following example of a SAS$^+$ planning task $\Pi = \langle V, \{o_1, o_2\}, I, G\rangle$ with

$$\begin{aligned} V =& \{a, b\}, \quad \mathit{dom}(a) = \mathit{dom}(b) = \{0, 1\}, \\ o_1 =& \langle\{\}, \{a = 1\}, 1\rangle, \\ o_2 =& \langle\{a = 1\}, \{a = 0, b = 1\}, 1\rangle, \\ I =& \{a = 0, b = 0\}, \text{ and} \\ G =& \{a = 1, b = 1\}. \end{aligned}$$

The operator o_1 is initially added to the queue. After it has been applied, o_2 is also added to the queue and subsequently applied. In order to set the goal entry $a = 1, b = 1$ to a finite value, o_1 must be reconsidered again which does not occur under current rules. We solve this problem by adding all precondition-free operators by default to the operator dictionary entry of every atom. This does entail a certain overhead, however, precondition-free operators are generally very rare.

From a theoretical perspective, the operator queue approach does not guarantee an improvement in the worst case. If every operator depends on every atom, then all operators are enqueued and processed in each iteration, leading to a similar procedure as the naive approach. However, such worst-case scenarios are uncommon in practical planning tasks. By focusing only on operators that are actually affected by changes in the heuristic table, several redundant operator evaluations are avoided.

4.1.5 Extend Entry Function

The `extend_entry` function is responsible for updating the heuristic table by extending effects of operators. The existing implementation iterates over all entries in the heuristic table and checks if such an extendable entry is present (Algorithm 4). Since it holds that $m = 2$, we know that entries we are searching for consist of one atom that corresponds to an operator effect and another atom, which we call `extend_atom`. Instead of iterating over the whole table we directly loop over all possible `extend_atom` (Algorithm 6).
It is also essential to stop the `extend_entry` process as early as possible if it is clear that no improvements can be achieved with the current operator effect. There are two cases in particular that realize this in the optimized implementation:

- The `extend_atom` is unreachable in the current table. Thus, there will not be any updates of entries containing this atom (line 8).
- The current heuristic value of an entry is lower than the cost to achieve the given

operator o. Since the evaluation of the set of entries $pre(o) \cup$ `extend_atom` cannot be lower than the preconditions itself there are guaranteed no table updates (line 11).

In both cases, `extend_atom` can be discarded before evaluating any entries.

Algorithm 6 Extend entry and update heuristic table

```
Procedure: new_extend_entry
Input: Operator effect atom, operator $o$, precondition evaluation $eval$
Output: Updated $h^m$ table with extended fact pairs
1  pre ← precondition_cache[o.get_id()]
2  num_variables ← task_proxy.get_variables().size()
3  for i in V do
4      if effect_conflict_cache[o][i] then
5          continue
6      for j in dom(i) do
7          extend_atom ← atom(i, j)
8          if hm_table.at(extend_atom) = $\infty$ then
9              continue
10         hm_pair ← (atom, extend_atom)
11         if hm_table.at(hm_pair) $\leq$ eval then
12             continue
13         c2 ← extend_eval(extend_atom, pre, eval)
14         if hm_table.at(hm_pair) > c2 + cost(o) then
15             hm_table[hm_pair] ← c2 + cost(o)
16             add_op_to_queue(hm_pair)
```

The evaluation in `new_extend_entry` can also be optimized. Preconditions of passed operators are evaluated in the `new_update_hm_table` function (Algorithm 5, line 4). Instead of re-evaluating the preconditions $pre(o)$, we store the result of their first evaluation. This means that we only have to focus on all entries containing `extend_atom` and take the maximum of these entries and the stored evaluation of the preconditions. For better differentiation, we call this function `extend_eval` as opposed to the original `eval` function given in Algorithm 2. The pseudo code for `extend_eval` is given in Algorithm 7.

Once again, there are a few cases where we can abort the evaluation process in `extend_eval`. If `extend_atom` contradicts a precondition, we return infinity (line 6). If `extend_atom` is an element of the preconditions, we simply return the evaluation result of the preconditions (line 7).
The `extend_eval` method changes the theoretical runtime. In the worst case, the evaluation now takes place in linear time compared to the quadratic runtime of Algorithm 2. Another advantage involves the input of the `extend_eval` function. Instead of adding the `extend_atom` to the preconditions and sorting the resulting vector (Algorithm 4, lines 10 and 11), we pass both separately as arguments to the evaluation function. The sorting is done by preserving the correct order when creating the atom pair at line 8. This reduces processing overhead in the `new_extend_entry` method of h^2.

Algorithm 7 Compute evaluation for extended entries

```
Procedure: extend_eval
Input: Atom extend_atom, operator preconditions pre, precondition evaluation eval
Output: Evaluation result for extended entries
atom_eval ← hm_table[extend_atom]
max_eval ← max(eval, atom_eval)
foreach p in pre do
    if p.var = extend_atom.var then
        if p.value ≠ extend_atom.value then
            return ∞
        return eval
    key ← (extend_atom, p)
    h ← hm_table[key]
    if h > max_eval then
        if h == ∞ then
            return ∞
        max_eval ← h
return max_eval
```

4.1.6 Storage of Changed Entries

Although new_extend_entry improves the existing method, we still need to iterate over all possible extend_atom. Therefore, it might be interesting to reduce the number of atoms we need to consider.

Previously we stated that even if the costs of achieving operator o have not improved since the last evaluation of o, we cannot rule out updates. We take a closer look at which updates are possible in this scenario. Since the operation itself is not cheaper, there are no updates for its partial effects. However, the same does not hold for new_extend_entry

Consider a pair of atoms a, b where $a \in pre(o)$ and $b \notin pre(o)$ for an operator $o \in O$. If a different operator $o' \in O$ updates the table entry for $[a, b]$, it might lead to another update containing the atom b when reconsidering o, as the preconditions of o in combination with b might be cheaper to reach. Therefore, for each operator we store all atoms that were updated since its last evaluation. We call these *changed entries* of o.

Instead of iterating over all possible extend_atom, we only consider changed entries in the case that the cost to achieve an operator stayed the same. This requires besides the storage of the changed entries also saving previous operator costs for comparisons with the current evaluation. We call this data structure *operator cost*. Since in the worst case, all atoms could be stored as changed entries, the space complexity of changed entries is $\mathcal{O}(|O||V| \times d)$.

The concept of changed entries is implemented in the extend_changed_entry method. The algorithm is similar to new_extend_entry with the difference of iterating over all changed entries for a given operator as extend_atom. If the costs for an operator decreased since the last evaluation, we cannot make assumptions about changed entries as the new costs could potentially update all extend_atom. In this case we delete the changed entry list and set the new cost to reach the operator in the operator cost data structure (Algorithm 5).

4.2 Comparison

We now compare both implementations with each other. First, we calculate the theoretical runtime and space complexity of the optimized version. Then we make comparisons with the complexity of the existing implementation, calculated in Section 3.3. We perform an experiment where we solve different problem instances with both implementations. The quality of both implementations can be tracked using metrics such as the total search time, the memory used or whether a solution was found at all.

4.2.1 Theoretical Analysis

In the following, we derive the runtime and space complexity of h^2. We use the same variables and terminology as when analyzing the existing implementation (Section 3.3). For example, we continue to use $(|V| \times d)^2$ as upper bound for the number of table entries.

Runtime Complexity

Like the existing implementation, h^2 can be separated into different, sequential parts. The initialization of the table, the table updating and the evaluation of the goal state are still part of the implementation. An additional step is the initialization of various auxiliary data structures such as operator caches or the operator queue. As information about the operators does not change during the entire search, it is sufficient to set up operator caches once before the search starts. We therefore disregard their initialization in this analysis. What needs to be considered, however, is setting up the operator queue and thus, inserting the initially applicable operators into the queue. This requires iterating over all preconditions of each operator which results in a worst-case runtime of $\mathcal{O}(|O||V|)$.

The generation of table entries has been made a little clearer by using an iterative approach instead of recursive functions. For each entry, we make a constant look up if it is contained in the initial state. Therefore, the complexity is reduced to $\mathcal{O}((|V| \times d)^2)$.
The evaluation of the goal state now uses an `unordered_map` which provides constant look up time. The `eval` method itself does not change. It holds that the runtime complexity for goal subset look ups is $\mathcal{O}((|V| \times d)^2)$.

Now we look at updating the table entries. We start with the `new_update_hm_table` method (Algorithm 5). The most notable change there is the introduction of the operator queue. As already mentioned, this does not offer any improvements in runtime complexity. In the worst case, all operators in O are added to the queue and only after each operator has been considered, exactly one entry is set to its final value. For each operator, either the `new_extend_entry` or the `extend_changed_entry` function is called on at most $|V|$ single effects. In the worst case, the cost of reaching an operator becomes more favorable with each evaluation, which is why only `new_extend_entry` would be used. The extend entry method must therefore be called just as often as for h^m. We can conclude that $\mathcal{O}(\texttt{new\_update\_hm\_table}) = \mathcal{O}(|O||V|^3 \times d^2) \times \mathcal{O}(\texttt{new\_extend\_entry})$.

In `new_extend_entry`, we iterate over each `extend_atom` (Algorithm 6). Therefore, we call the `extend_eval` method $\mathcal{O}(|V| \times d)$-times. In Algorithm 7, we have reduced the quadratic complexity to linear by storing the evaluation of the operator preconditions. The efficient look up in the heuristic table eliminates the logarithmic time required to find entries. It holds that $\mathcal{O}(\texttt{extend\_eval}) = \mathcal{O}(|V|)$.

We now combine the results of the individual algorithms to compute `new_update_hm_table`:

$$\begin{aligned}
\mathcal{O}(\texttt{new\_update\_hm\_table}) &= \mathcal{O}(|O||V|^3 d^2) \times \mathcal{O}(\texttt{new\_extend\_entry}) \\
&= \mathcal{O}(|O||V|^3 \times d^2) \times \mathcal{O}(|V| \times d) \times \mathcal{O}(\texttt{extend\_eval}) \\
&= \mathcal{O}(|O||V|^3 \times d^2) \times \mathcal{O}(|V| \times d) \times \mathcal{O}(|V|) \\
&= \mathcal{O}(|O||V|^5 \times d^2)
\end{aligned}$$

The improved implementation has a runtime complexity of $|V|^5$, while for h^m, it is $|V|^6$. The maximum domain size is quadratic for h^2, whereas it is cubic for the existing implementation (Table 4.1). Conceptually, two main reasons contribute to the improved runtime complexity:

- The caching of the evaluation result in `new_update_hm_table` that can be re used in `extend_eval`.
- Using an `unordered_map` and a suitable hash function, for table look ups in constant time.

All other improvements also reduce runtime performance, but either do not affect the worst-case complexity or become negligible under strict worst-case assumptions, such as the operator queue. From a practical standpoint, all described improvements lead to noticeable execution speed ups.

Part of Implementation	h^m **(for** $m = 2$**)**	h^2
Table Initialization	$\|V\|^3 \times d^2$	$\|V\|^2 \times d^2$
Queue Initialization	-	$\|O\|\|V\|$
Table Updating	$\|O\|\|V\|^6 \times d^3 \log(\|V\| \times d)$	$\|O\|\|V\|^5 \times d^2$
Goal Evaluation	$\|V\|^2 \log(\|V\| \times d)$	$\|V\|^2$

Table 4.1: Worst case runtime complexity overview of sequential parts in h^m and h^2. Note that the initialization of operator caches is excluded as this is done in a preprocessing step before the search process.

Space Complexity

Several new data structures have been added for h^2 compared to the original implementation. The heuristic table remains unchanged with a space complexity of $\mathcal{O}(|V| \times d)^2$. The space complexity for the three operator caches has already been discussed. The partial effect cache requires the most memory with $\mathcal{O}(|O||V|^2)$. Additional structures include the operator queue and operator set, both of which store only integer IDs of operators, resulting

in a space complexity of $\mathcal{O}(|O|)$.
Other relevant structure are the operator dictionary and the changed entries data structure which both are implemented as `unordered_map`. The operator dictionary maps atoms to a vector of operators while changed entries maps operators to atoms. The worst-case space complexity of both data structures is $\mathcal{O}(|O||V| \times d)$.

It is difficult to determine definitively which structure requires the most space (Table 4.2). However, it can be concluded that the average space complexity of h^2 exceeds that of the original h^m implementation. The worst-case space complexity for h^2 is $\mathcal{O}(|V|^2 \times d^2 + |O||V|^2 + |O||V| \times d)$.

Data Structure	h^m	h^2
Heuristic Table	$\|V\|^2 \times d^2$	$\|V\|^2 \times d^2$
Precondition Cache	-	$\|O\|\|V\|$
Partial Effect Cache	-	$\|O\|\|V\|^2$
Effect Conflict Cache	-	$\|O\|\|V\|$
Operator Queue	-	$\|O\|$
Operator Set	-	$\|O\|$
Operator Dictionary	-	$\|O\|\|V\| \times d$
Operator Cost	-	$\|O\|$
Changed Entries	-	$\|O\|\|V\| \times d$

Table 4.2: Overview of worst case space complexities of data structures used in h^m and h^2.

4.2.2 Experiments

To validate our theoretical improvements, we conduct empirical evaluations using the Fast Downward planning system (Helmert, 2006). In Fast Downward, problem instances are passed as arguments in the *Planning Domain Definition Language* (PDDL). PDDL is a standardized language used to describe planning tasks (Ghallab et al., 1998). Fast Downward takes a problem instance as argument, along with an additional search parameter that specifies the search algorithm and the heuristic. If we want to use the A^* search algorithm (Hart et al., 1968) together with the critical path heuristic h^m, the corresponding search parameter is $astar(hm())$. When using the optimized implementation, the search parameter is $astar(h2())$. We carry out the experiment with *Downward Lab* (Seipp et al., 2017). This is a Python package that implements the package *Lab* specifically for Fast Downward. It takes care of assembling input calls and summarizing the results.

For the experiment we use benchmarks from the optimal track of the International Planning Competition (IPC).[4] In total, we utilize 65 problem domains that sum up to 1827 tasks. Each domain has various instances, which differ in their size. We set a time limit of 30 minutes for each problem and a memory limit of 3947 MB. The results are illustrated in Figure 4.1. The left scatter plot compares the total execution time for the original and

[4] https://github.com/aibasel/downward-benchmarks/

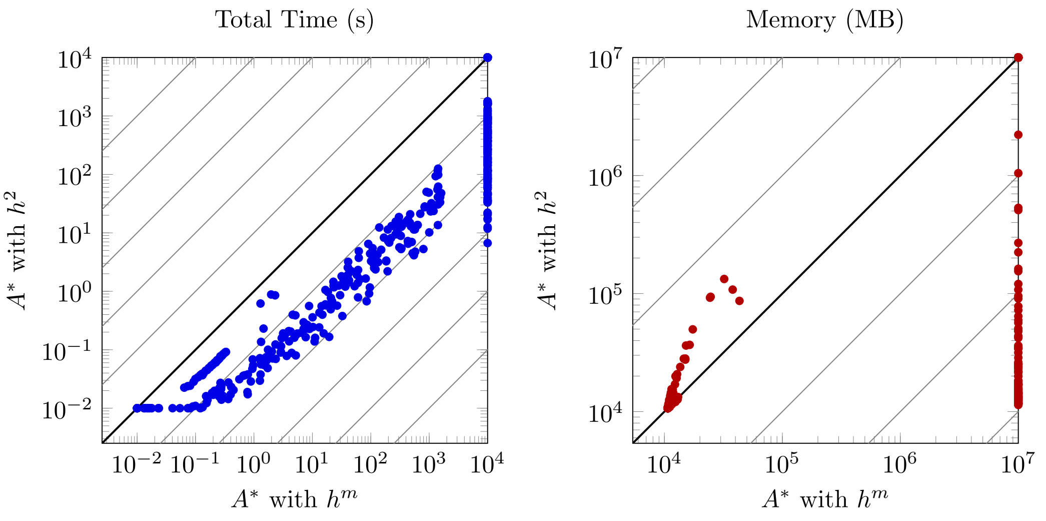


Figure 4.1: Total time and memory usage comparison between existing (A^* with h^m) and optimized implementation (A^* with h^2). Unsolved tasks are displayed on the edges of the plot, depending on which implementation solved the task.

optimized implementation in seconds. The right side represents memory consumption differences in mega-bytes. In both plots, diagonals are drawn as auxiliary lines to increase the comparability of the results. Points on the main diagonal, shown in black, indicate that both implementations yield the same result. Further diagonals in gray show a difference of an order of magnitude.

The resulting plot in Figure 4.1 confirms the runtime complexity analysis. As we can see, the optimized implementation solves problem instances faster than the existing one. This can be recognized by the fact that all data points in the left plot are located below the main diagonal. For very small problems, the difference is relatively small. This can be explained by the fact that for smaller problems the search takes up a smaller part of the entire planning process compared to i.e. task preprocessing steps. With larger instances, h^2 finds solutions faster by a factor of 10 or more. Problems that have only been solved by h^2 in the specified time limit can be found on the right-hand side of the plot. What is striking is that no task could be solved by h^m, but not by our optimized version.[5]

There are also significant differences between the two implementations in terms of memory. While the space for h^m remains much lower then for h^2. The memory consumption is only similar for individual, smaller instances. However, this difference in memory utilization does not result in the existing implementation being able to solve a problem, whereas h^2 is not. There were isolated errors due to memory consumption for h^2, but only for instances that exceeded the time limit for h^m. Therefore, although the memory space is large for h^2, this does not lead to practical disadvantages compared to the existing implementation.

[5] This is visible by the absence of data points at the top of the plot. Points in the top-right corner belong to tasks unsolved by both implementations.

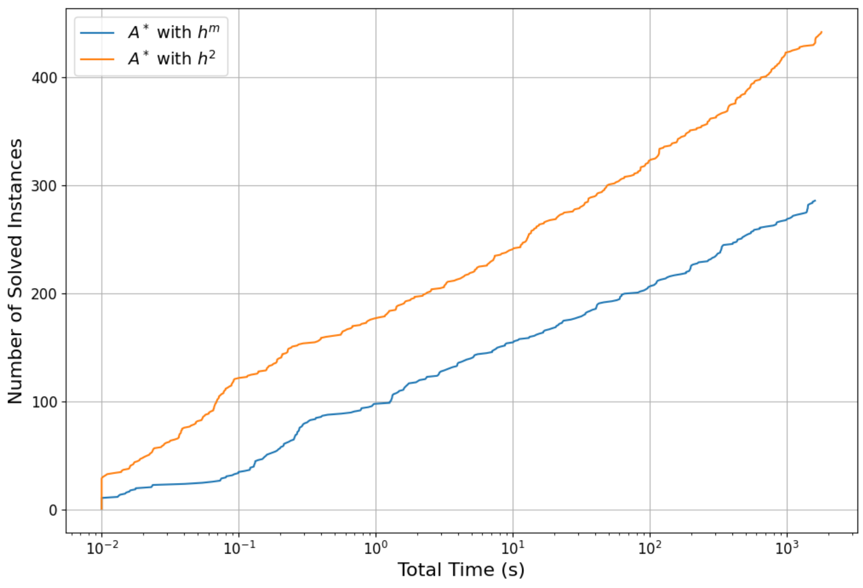


Figure 4.2: Number of solved instances after given amount of time for total of 1827 tasks. Existing implementation is labeled with A^* with h^m, optimized version as A^* with h^2.

To put these results more into the context of solved problems in general, we now look at the total number of tasks solved. To do this, we depict how many problems are solved after a given amount of time. The results are shown in Figure 4.2. Once again we see the improved results of h^2. However even with a time limit of 30 minutes, only just under 25% of problem instances are solved by h^2. This shows that although the runtime was reduced by a factor of more than ten, the optimized implementation is still unsuitable for solving very large planning tasks.

5 Theory of Π^{m} Compilation

In this chapter, we investigate an approach that shifts much of the computational effort involved in heuristic evaluation to a preprocessing phase before the actual search begins. Specifically, we focus on the Π^m compilation, a method that transforms the original planning task into a new form in where the computation complexity lies in the planning task itself, not the heuristic evaluation. The compiled task can be computed in the search initialization and then be re-used in each evaluation.

We state the theoretical foundations of the Π^m compilation, explaining its purpose and how it captures variable interactions in the original task through the introduction of meta-variables. Afterwards, we describe an algorithm that allows transforming a planning task in SAS$^+$ representation into STRIPS. This is necessary as the Π^m compilation is originally defined on STRIPS planning tasks (Haslum, 2009).

5.1 Definition of Π^m

In the critical path heuristic for $m = 1$, we only consider the costs of single atoms. This is equivalent to the $h^{\max}$ heuristic. Since the complexity of h^m increases exponentially with m, a heuristic evaluation with $h^{\max}$ is much faster than with h^2. Haslum (2009) introduced a method that constructs for a given STRIPS planning task Π a modified task Π^m, such that $h^{\max}(\Pi^m) = h^m(\Pi)$. This approach allows us to apply the $h^{\max}$ heuristic to a modified task in order to obtain h^m evaluations.

The core idea of the Π^m compilation is to capture interactions between variables in a STRIPS task by introducing *meta-variables*. A meta-variable $v_{\{a,b\}}$ represents the states in the original task Π where both propositional variables a and b are true. As we aim to capture interactions involving at most m variables, it holds for all meta-variables v_C in Π^m that $|C| \leq m$. The cost to reach variable v_C now corresponds to the cost of reaching atoms in C simultaneously. We define the Π^m compilation formally:

Definition 13 (Π^m Compilation)**.** *The problem* Π^m *of a STRIPS planning task* $\Pi = \langle V, O, I, G\rangle$ *is the STRIPS planning task* $\langle V^m, O^m, I^m, G^m\rangle$*, where:*

- $V^m = \{v_C \mid C \subseteq V, |C| \leq m\}$,
- $I^m = \{v_C \mid C \subseteq I, |C| \leq m\}$,
- $O^m = \{a_{o,S} \mid o \in O, S \subseteq V, |S| < m, S \cap (add(o) \cup del(o)) = \emptyset\}$ *with*
  $pre(a_{o,S}) = \{v_C \mid C \subseteq (pre(o) \cup S), |C| \leq m\}$,
  $add(a_{o,S}) = \{v_C \mid C \subseteq (add(o) \cup S), C \cap add(o) \neq \emptyset, |C| \leq m\}$,
  $del(a_{o,S}) = \emptyset$,
  $cost(a_{o,S}) = cost(o)$, *and*
- $G^m = \{v_C \mid C \subseteq G, |C| \leq m\}$.

We call operators in O^m *meta-operators*. Each meta-operator $a_{o,S}$ corresponds to the execution of operator o in Π while preserving all variables in S. Note that S can be the empty set which means that there are no variables preserved during the application of $a_{o,\emptyset}$.
Applying $h^{\max}$ on the compiled task Π^m yields $h^m(\Pi)$. Therefore, applying $h^{\max}$ on Π^m is admissible.

5.2 SAS$^+$ Task to STRIPS Translation

The definition of the Π^m compilation requires a STRIPS planning task. In Fast Downward, tasks are by default translated into tasks with finite-domain variables. We previously stated the idea of translating an SAS$^+$ task into STRIPS by introducing a propositional state variable for each element in the domain of a finite-domain state variable.
Another difference between both task definitions is that SAS$^+$ does not distinguish between positive and negative effects. Consider an SAS$^+$ planning task Π with a state s where $a = 0$, with $a \in V$ and $0 \in \text{dom}(a)$. Let us assume there exists an operator o that is applicable in s and it holds $a := 1 \in \mathit{eff}(o)$ with $1 \in \text{dom}(a)$. Therefore, in the state $s' = s[o]$, we have $a = 1$. Thus, the atomic effect $a := 1$ behaves in the same way as an add effect $a = 1$ and delete effects for every domain value in $\text{dom}(a)$ other than 1. These add and delete effects always occur together, otherwise the variable a would either have no value or multiple values in the new STRIPS state. With these concepts, we define a translation algorithm:

Definition 14 (SAS$^+$ Task into STRIPS form). *Let* $\Pi = \langle V, O, I, G\rangle$ *be a SAS$^+$ task. The task* $\Pi' = \langle V', O', I', G'\rangle$ *is the equivalent task in STRIPS form with*

- $V' = \{p_{v=d} \mid v \in V, d \in dom(v)\}$, *with* $dom(p_{v=d}) = \{true, false\}$,
- *Each operator* $o \in O$ *is mapped to a STRIPS operator* $o' = \langle pre(o'), add(o'), del(o'), cost(o')\rangle$, *where:*
  - $pre(o') = \{p_{v=d} \mid (v = d) \in pre(o)\}$,
  - $add(o') = \{p_{v=d} \mid (v := d) \in \mathit{eff}(o)\}$,
  - $del(o') = \{p_{v=d} \mid (v = d') \in add(o'), d \in (dom(v) \setminus \{d'\})\}$, *and*
  - $cost(o') = cost(o)$,
- $I' = \{p_{v=d} \mid (v = d) \in I\}$,

- $G' = \{p_{v=d} \mid (v = d) \in G\}$.

The SAS$^+$ task Π and its translated STRIPS task Π' represent the same planning task because they encode the same reachable transition system. In SAS$^+$, a variable v can take multiple values from a finite domain, whereas in STRIPS, each possible value d of v is represented by a separate propositional variable $p_{v=d}$. This ensures that each state in Π has a corresponding state in Π', where exactly one proposition $p_{v=d}$ is `true` for each v. Operators in Π are translated such that applying an operator in Π corresponds intuitively to applying its counterpart in Π', producing analogous effects. Assigning $v := d$ in Π corresponds to adding $p_{v=d}$ in Π' while deleting all other propositions $p_{v=d'}$ for $d' \neq d$. The initial and goal states are transformed in the same way.
Since every valid sequence of operators in Π has an equivalent sequence in Π' and vice versa, the two tasks describe the same problem, just in different formal representations.

With the translation in Definition 14, we are able to use the Π^m compilation on SAS$^+$ planning tasks. For a SAS$^+$ task Π, we compile the translated STRIPS task Π'. The resulting heuristic values $h^{\max}((\Pi')^2)$ are equivalent to $h^2(\Pi)$.

6 Π^{m} Compilation in Fast Downward

In this section, we discuss an implementation of the h^2 heuristic using the Π^m compilation approach. First, we describe the details of the Π^m compilation in Fast Downward. Then, we examine the existing implementation of the $h^{\max}$ heuristic in Fast Downward. Finally, we present both a theoretical analysis and experimental results, comparing them to the h^2 implementation described in Chapter 4.

6.1 Implementation of the Π^m Compilation

Fast Downward already provides a framework for modifying a planning task before the search process starts. The `DelegatingTask` class offers an interface to modify the original task and is used, for instance, to perform domain abstractions. The core of our compilation is the `PiMCompiledTask` class, which implements `DelegatingTask`. This class is responsible for transforming a given planning task into Π^2 by introducing meta-variables and meta-operators.

We define a mapping from atom pairs a_1, a_2 to the corresponding meta-variable $v_{\{a_1,a_2\}}$ referred to as `meta_atom_map` (Figure 6.1). This allows us to bypass converting the task from SAS$^+$ to STRIPS by directly building the meta-variables from atoms in SAS$^+$. Each meta-variable has a domain of $\{0, 1\}$. A value of 1 represents the truth value `true` for a propositional state variable. The initial state and goal of the compiled task are built according to Definition 13.
Meta-operators are generated by iterating over all original operators $o \in O$. According to Definition 13, S is a set of STRIPS variables. Since our STRIPS variables are essentially atoms of Π and it holds that $m = 2$, S either consists of the empty set or one atom $v = d$, where $v \in V$ and $d \in dom(v)$. For each operator $o \in O$, we store precondition meta-variables $pre(a_{o,\emptyset})$ and add effect meta-variables $add(a_{o,\emptyset})$. Since it holds that $pre(a_{o,\emptyset})$ is a subset of $pre(a_{o,S})$, and likewise for the add effects, it is sufficient to extend these stored meta-variables by iterating over all allowed atoms in S.
Meta operators are stored in a separate struct `MetaOperators` and contain the preconditions, effects, cost of the new operator, the ID of the original operator and the corresponding

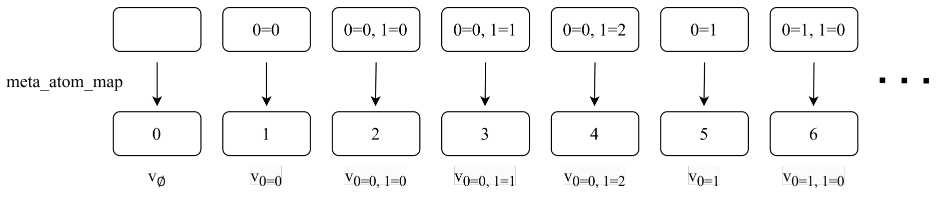


Figure 6.1: Illustration of `meta_atom_map`, showing atom sets mapped to meta-variable ids with annotated meta-variables. The example is limited to the first seven entries for clarity.

meta-variable in S.

The Π^m compilation is applied before the search begins, while the search uses $h^{\max}$. During the heuristic evaluation, however, the original state must be continuously translated into its compiled form. This is done using Algorithm 8:

Algorithm 8 Convert State into Compiled State

```
Procedure: compile_state
Input: State values from original task values
Output: Transformed values in $\Pi^2$ representation
1  new_values ← vector of size |meta_atom_map|, initialized to 0
2  new_values[0] ← 1
3  foreach i in indices of values do
4      atom ← (i, values[i])
5      meta_index ← meta_atom_map[atom]
6      new_values[meta_index] ← 1
7      foreach j in indices after i do
8          second_atom ← (j, values[j])
9          meta_index ← meta_atom_map[(atom, second_atom)]
10         new_values[meta_index] ← 1
11 values ← new_values
```

The algorithm iterates over all atom pairs in the original state and sets the meta-variables v_C to `true` if all atoms $a \in C$ are present in the state. The first meta-variable in `new_values` represents $v_\emptyset$ and is always set to *true* (line 2). Although this variable could be omitted without affecting correctness, it is included to to comply with the definition of Π^2.
With this, the compilation process is complete. We construct the Π^2 task and provide a method to convert states from the original representation to the compiled one.

6.2 $h^{\max}$ Heuristic in Fast Downward

The $h^{\max}$ heuristic is a relaxation heuristic that estimates the cost of achieving each goal atom in a planning task by considering the maximum cost among its contributing preconditions (Definition 8). We use this heuristic to estimate the goal distance in the compiled task representation.

The existing implementation is provided in the `HSPMaxHeuristic` class. This implementation assigns cost values to atoms and operators, representing the cost of reaching them or its effects respectively. A priority queue containing atoms sorted by this cost value is used to propagate their cost values. The heuristic computation follows three main steps:

- Initialization: The algorithm sets all atom costs to -1. This marks them as unvisited. Atoms of the initial state are inserted into the priority queue with a cost of zero. Effects of precondition-free operators are inserted with the operator cost.
- Exploration: The implementation processes elements from the queue one by one. When all preconditions of an operator have been reached, the operator can fire, and its effects are added to the queue. This process continues until the goal is reachable. The exploration step is shown in Algorithm 9.
- Heuristic evaluation: Once all reachable atoms have been assigned their final cost, the heuristic value is determined. It is the maximum cost among all goal atoms. If a goal atoms remains unvisited (cost -1), the problem is unsolvable in the relaxed space, and the heuristic returns infinity.

Algorithm 9 Relaxed graph Exploration

```
Procedure: relaxed_exploration
Input: queue with initial state atoms and precondition-free operator effects
Output: Goal atoms with final cost value
1  unsolved_goals ← $|G|$
2  while queue not empty do
3      (atom, distance) ← queue.pop()
4      if atom.cost < distance then
5          continue
6      if atom.is_goal then
7          unsolved_goals ← unsolved_goals − 1
8          if unsolved_goals = 0 then
9              return
10     foreach op in ops_with_atom_as_pre do
11         op.cost ← max(op.cost, op.base_cost + atom.cost)
12         op.unsatisfied_pre ← op.unsatisfied_pre −1
13         if op.unsatisfied_pre = 0 then
14             if op.effect.cost > op.cost then
15                 op.effect.cost ← op.cost
16                 queue.insert(op.effect, op.cost)
```

The implementation of $h^{\max}$ uses modified operators known as *unary operators*, each having only one effect. For each original operator o, one unary operator is generated per effect, i.e., $|eff(o)|$ unary operators in total.

This division into unary operators often results in duplicate or dominated operators. An operator o_1 dominates another operator o_2 if $eff(o_1) = eff(o_2)$ and $pre(o_1) \subseteq pre(o_2)$, and $cost(o_1) \leq cost(o_2)$. If both the preconditions and the costs relation are not strict, it is a duplicate. We are interested in removing both duplicates and dominated operators. The current Fast Downward implementation of $h^{\max}$ checks this for all operators o where $|pre(o)| \leq 5$ holds. This is a compromise since checking for dominated operators is not efficient. However, we will also experimentally test the heuristic where none and all duplicates or dominated operators are eliminated. We call this heuristic $h^{\max}_k(\Pi^2)$, where k indicates the maximum precondition size for which dominated operator pruning and duplicate detection were applied. If we omit the subscript in the notation, we mean the Fast Downward standard $k = 5$.

A distinction is made between base cost and total cost of a unary operator. The base cost (`op.base_cost`) corresponds to the original operator cost $cost(o)$. The total cost (`op.cost`) includes the base cost plus the costs of satisfying preconditions of o. The priority queue ensures that each atom is evaluated only when its cost has been fully determined. There is no possibility of a cost decrease as atoms are processed through the priority queue in ascending cost order. We can therefore be sure that as soon as the goal is achievable, the cost of the goal is final and cannot decrease further.

The priority queue is implemented as an adaptive data structure. Initially, it uses a bucket-based queue, where keys represent cost values and atoms are stored in corresponding buckets. This allows constant-time insertions and deletions. If the keys become large, the bucket-based approach becomes inefficient (Hopcroft et al., 1983). In this case, the queue automatically converts into a heap-based queue using a binary heap, which supports logarithmic-time operations.

6.3 Theoretical and Experimental Results

We now compare $h^{\max}(\Pi^2)$ with our h^2 implementation. To do so, we first derive the theoretical runtime and space complexity of $h^{\max}(\Pi^2)$. We then conduct an experiment using the same evaluation metrics as for the experiment in Section 4.2.2. A key difference to this experiment is the amount of preprocessing required before the actual search begins. While the h^m and h^2 heuristics only require moderate precomputation, the compilation into Π^2 is a fundamental part of the heuristic computation. This additional effort must be taken into account both in the runtime analysis and in the experimental evaluation.

6.3.1 Runtime and Space Complexity

We analyze the runtime and space complexity of $h^{\max}(\Pi^2)$. In this theoretical analysis, we use the same variables as in previous sections, such as the number of variables $|V|$ and the maximum domain size d. These quantities refer to the original task in SAS$^+$ form, not the STRIPS version or the compiled Π^2 task. To avoid confusion between the original task in SAS$^+$ and the compiled Π^2 task, we introduce subscripts instead of superscripts to denote the components of the compiled task. Thus, the notation for the compiled task is $\Pi^2 = \langle V_2, O_2, I_2, G_2 \rangle$.

Runtime Complexity

We separate the analysis into two parts: the compilation step and the actual search. We begin by analyzing the compilation process itself, as described in Section 6.1.

In the first step of the compilation, we iterate over all pairs of atoms to construct meta-variables. This leads to a worst-case runtime of $\mathcal{O}(|V|^2 \times d^2)$. Next, we generate meta-operators. For each operator $o \in O$, we determine which atoms (denoted as S in Definition 13) are applicable for o. Since $m = 2$, S consists of at most one of $|V| \times d$ atoms. As a result, the total runtime complexity of the Π^m compilation is $\mathcal{O}(|O||V| \times d + |V|^2 \times d^2)$.

During the search, $h^{\max}$ is applied to the compiled task. Therefore, we need to express the number of variables and operators in Π^2 in terms of the parameters of Π in order to perform runtime comparisons. Since each meta-variable is formed from a pair of atoms, we have $|V_2| = |V|^2 \times d^2$. For the number of operators, each operator $o \in O$ is combined with the corresponding S atom, resulting in $|O_2| = |O| \times |V| \times d$. The $h^{\max}$ implementation uses unary operators, which have only one effect. There are $|V|^2$ effects per operator in the worst-case. This is a consequence from our STRIPS translation, as each operator can assign only one domain value per SAS$^+$ variable. It means $|V|^2$ unary operators are generated in the worst-case for one meta-operator in O_2. For the number of unary Operators O_u, it holds

$$|O_u| = |V|^2 \times |O_2| = |O| \times |V|^3 \times d$$

We use these values to determine the runtime complexity of the $h^{\max}$ computation on the compiled task.

Before the $h^{\max}$ evaluation starts, the current state must be translated into the compiled state (Algorithm 8). In the algorithm, we iterate over all meta-variables, which leads to $\mathcal{O}(|V_2|) = \mathcal{O}(|V|^2 \times d^2)$.
The main part of the $h^{\max}$ implementation is the exploration with the queue (Algorithm 9). In the worst case, the goal can only be reached by exploring all possible V_2 meta variables. For each atom a from the queue, all unary operators o_u where $a \in pre(o_u)$ applies are iterated over. The effect of a newly applicable o_u is added to the queue (line 14). Since insertions can be done in logarithmic time we can conclude for the runtime complexity $\mathcal{O}(h^{\max})$:

$$\mathcal{O}(h^{\max}) = \mathcal{O}(|V_2| \times |O_u| \times log(|V_2|)) = \mathcal{O}(|O| \times |V|^5 \times d^3 \times log(|V| \times d))$$

This clearly exceeds the complexity of compiling the current state and is therefore also the final result for $h^{\max}(\Pi^2)$. Furthermore, from a theoretical perspective, the $h^{\max}$ heuristic evaluation on Π^2 is clearly more time consuming than compiling the task itself.

Space Complexity

The space consumption comes from two sources: the compiled, larger task and the priority queue used in $h^{\max}$. In the worst case, the queue holds all meta-variables, resulting in a

Heuristic	**Runtime**	**Space**
$h^m(\Pi)$	$\lvert O\rvert\lvert V\rvert^6 \times d^3 \log(\lvert V\rvert \times d)$	$\lvert V\rvert^2 \times d^2$
$h^2(\Pi)$	$\lvert O\rvert\lvert V\rvert^5 \times d^2$	$\lvert V\rvert^2 \times d^2 + \lvert O\rvert\lvert V\rvert^2 + \lvert O\rvert\lvert V\rvert \times d$
$h^{\max}(\Pi^2)$	$\lvert O\rvert\lvert V\rvert^5 \times d^3 \times log(\lvert V\rvert \times d))$	$\lvert O\rvert\lvert V\rvert^5 \times d$

Table 6.1: Overview of worst case runtime and space complexities for the analyzed heuristics so far.

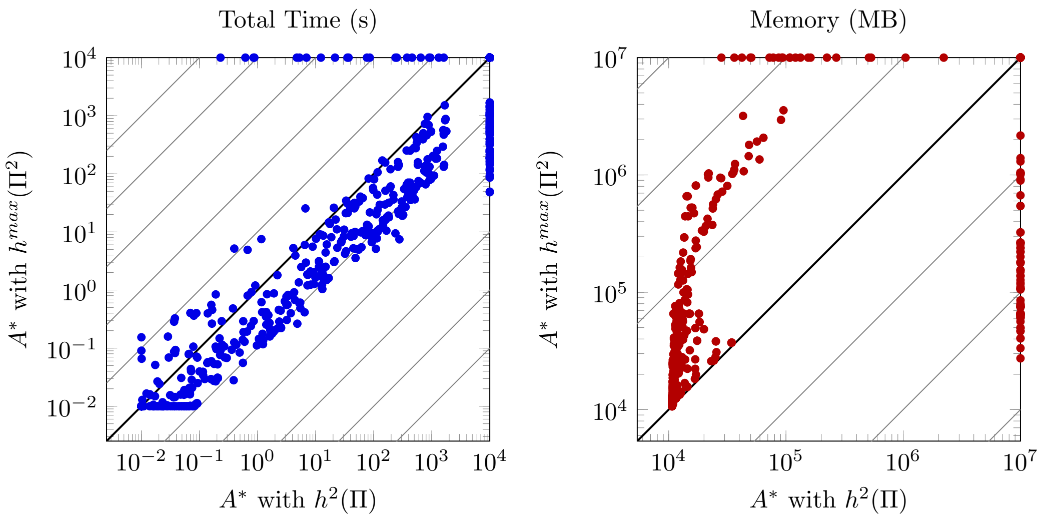


Figure 6.2: Total time and memory usage comparison between h^2 and $h^{\max}$ on Π^2 compiled task. Unsolved tasks are displayed on the edges of the plot, depending on which implementation solved the task.

complexity of $\mathcal{O}(|V_2|) = \mathcal{O}(|V|^2 \times d^2)$.

The more significant factor is the compiled task itself. In the $h^{\max}$ heuristic, operators in O_2 is transformed into $|O_u|$ unary operators. Each unary operator can require up to $\mathcal{O}(|V|^2)$ space, as it may store up to $|V|^2$ preconditions. We conclude for the space complexity that

$$\mathcal{O}(h^{\max}(\Pi^2)) = \mathcal{O}(|O_u| \times |V|^2) = \mathcal{O}(|O| \times |V|^5 \times d)$$

Table 6.1 summarizes the runtime and space complexities of the existing h^m heuristic, our optimized h^2 heuristic, and the $h^{\max}$ heuristic applied to the compiled task. The runtime of the Π^m compilation approach is slightly higher than that of the optimized h^2, but still lower than the original h^m implementation. Since the number of variables $|V|$ is typically much larger than the domain size d, the overall runtime complexity remains comparable to h^2. In terms of memory usage, $h^{\max}(\Pi^2)$ consumes significantly more space than the other two heuristics. This is mainly due to the increased size of the compiled task, particularly the creation of numerous unary operators.

6.3.2 Experimental Results

We conduct an experiment comparing the performance of the Π^m compilation approach with the h^2 heuristic. The experimental setup is the same as described in Section 4.2.2. The time limit is 30 minutes and the memory limit is 3947 MB. Figure 6.2 shows the total time and peak memory consumption for Π^2 compilation and h^2. The main diagonal in black indicates same performance for both heuristics. The adjacent diagonals, shaded in gray, represent differences by successive powers of ten (10, 100, 1000, ...).

The Π^m compilation approach solves the majority of problem instances faster than h^2, particularly for larger tasks. However, there are a few smaller instances where h^2 achieves better performance. On average, $h^{\max}(\Pi^2)$ solves tasks approximately four times faster than $h^2(\Pi)$. This result may seem counterintuitive, given that the theoretical analysis predicted a runtime advantage for h^2. The difference arises because the worst-case assumption of having $|O_u|$ unary operators in total rarely holds in practice. As previously discussed, $h^{\max}$ performs internal optimizations, such as detecting and eliminating duplicate or dominated operators up to a certain size, which significantly reduces the actual complexity of heuristic evaluations.
The memory plot looks very different. The memory difference for small problem sizes is almost identical. For larger problems, $h^{\max}(\Pi^2)$ requires over 10 times more memory.
We now take a closer look at the total time metric. To do this, we split the total time into a preprocessing time and a search time. It holds that $t_{preprocessing} + t_{search} = t_{total}$. We can see the results of this in Figure 6.3.

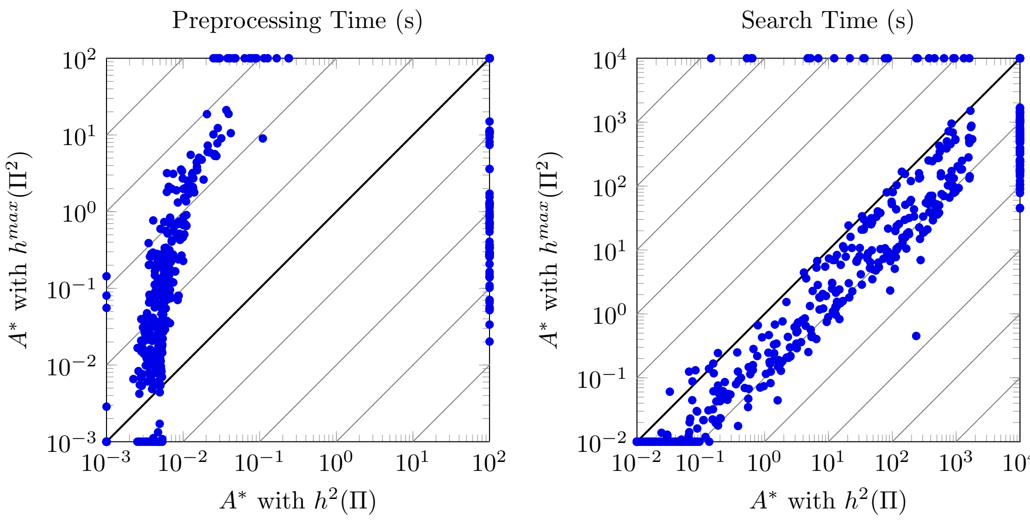


Figure 6.3: Preprocessing time and search time comparison between h^2 and $h^{\max}$ on Π^2 compiled task. Unsolved tasks are displayed on the edges of the plot, depending on which implementation solved the task.

The preprocessing time for the Π^2 compilation is much longer than for h^2. This is to be expected, as the main task compilation takes place there. Our h^2 implementation, on the other hand, only performs a few calculations concerning operator caches before the search. It should also be noted that the translation of the task from PDDL to SAS$^+$ also falls within the preprocessing time.
The results of the pure search time are very similar to the total time. In general, the search

Search Result	$h^2(\Pi)$	$h_0^{\max}(\Pi^2)$	$h_5^{\max}(\Pi^2)$	$h_\infty^{\max}(\Pi^2)$	$h_{\text{bin}}^2(\Pi)$
Solution found	431	404	493	371	461
Problem Unsolvable	11	7	7	4	7
Memory error	31	873	530	505	284
Time Error	1341	530	784	934	1062
Other Errors	13	13	13	13	13

Table 6.2: Search results for all 1827 tasks. $h^2(\Pi)$ and $h_{\text{bin}}^2(\Pi)$ refer to the standard and binary operator versions of h^2 heuristic. The $h_k^{\max}(\Pi^2)$ heuristics are based on $h^{\max}$ applied to the Π^2 compilation, where k indicates the maximum precondition size for which dominated operator pruning and duplicate detection were applied (0 = none, 5 = up to size 5, ∞ = all).

with the Π^2 compilation takes significantly less time. The main calculations of h^2 take place during the search and must be performed for each heuristic evaluation individually.
In general, preprocessing accounts for a much smaller part than the actual search. For this reason, better search results should be weighted more strongly than the preprocessing time.

Now, let us analyze the total number of solved instances. Table 6.2 presents the number of solved and unsolved instances for each heuristic. As already observed in the previous figure, $h^{\max}(\Pi^2)$ solves significantly more instances compared to h^2. The reasons for failing to solve instances are also noteworthy. The primary limitation for h^2 is the 30-minute time limit, while memory corruption is exceptionally rare. In contrast, with the Π^m compilation, both memory and runtime constraints pose similarly significant challenges. This shows that we have achieved a good trade-off between runtime and space complexity for the Π^m compilation.
Additionally, we see why Fast Downward limits to duplicate and dominated operator detection to precondition sets of a maximum of 5. The performance of $h_0^{\max}(\Pi^2)$ and $h_\infty^{\max}(\Pi^2)$ is clearly worse then for $h_5^{\max}(\Pi^2)$ and $h^2(\Pi)$. In general, it can be seen that the Π^2 compilation approach strongly depends on the extent to which the set of unary operators is simplified. Putting too much computational effort into this simplification also has a negative effect.

6.4 Connections between h^2 and Π^2 compiled $h^{\max}$

The results demonstrate that the Π^m compilation method outperforms the iterative calculation of the h^2 heuristic table. This improvement is due to the fact that the complexity of h^2 is effectively precomputed in the compilation step by considering atom pairs as basic components. The improvement becomes particularly clear when comparing meta-operators with the `extend_entry` function used in the h^2 implementation. A meta-operator $a_{o,S}$ consists of a base operator o from the original task and a set of atoms S that is preserved during its execution. For $m = 2$, it holds that $|S| \leq 1$. Such a meta-operator mirrors the behavior of `extend_entry` with operator o and an `extend_atom` that is contained in S. The key difference lies in when these combinations are processed. While the h^2 implementation checks potential `extend_atom` combinations continuously during search, the Π^2 compilation computes this information once during preprocessing. As a result, significantly more structural information is incorporated before the search begins, reducing the computational effort required during individual heuristic evaluations.

This idea of task compilation can also be applied to the h^2 implementation by introducing the concept of *binary operators*. A binary operator has at most two effects and is conceptually related to a unary operator. In a binary operator h^2 version, all original operators are decomposed into binary operators. Additionally, as in `extend_entry`, new binary operators are created by combining original effects with specific `extend_atom`. Technically, binary operators serve the same role as meta-operators in Π^2, but they operate over original atoms rather than compiled meta-variables.
The heuristic values can then be computed using the same algorithmic structure as $h^{\max}$, employing a priority queue. Instead of meta-variables, the queue contains atom pairs. In the end, both approaches are structurally equivalent.

Experimental results using binary operators h^2_{bin} show performance slightly below that of the Π^m compiled heuristic, but significantly better than the original h^2 implementation (Table 6.2). This outcome is expected, as our binary operator version omits duplicate elimination. Aside from this, the algorithmic procedure is essentially identical to the other Π^2 compiled approaches.

7 Regression Search and STRIPS Duality

We have observed that for all our methods implementing the h^2 heuristic, the majority of our benchmark problem instances remain unsolved in the defined time and memory constraints. A closer look reveals that we are approaching a fundamental limitation. Instances for which memory limits are exceeded and those where time limits are reached largely overlap. Consequently, adjusting the amount of preprocessing by increasing or decreasing tends to worsen the outcome, as we quickly encounter both memory and time constraints.

This motivates us to consider an alternative approach that differs from those previously explored. Earlier, we mentioned that the h^2 heuristic is particularly well-suited for regression search. This is because regression search evaluates heuristic values of the form $h^2(I, A)$, where I is the fixed initial state and A is the current state under evaluation. Since I remains constant throughout the search process, the heuristic table needs to be computed only once. In contrast, progression search typically requires the recomputation of heuristic estimates for atom sets in each state evaluation.
However, implementing regression search in practice presents significant challenges. In regression, states are represented as partial assignments, which correspond to sets of concrete states in progression. This leads to a larger transition system and increases the number of explored states. In addition, checking the applicability of measures is more complex due to the prevailing conditions and the need to maintain consistency with partial states. Efficient pruning requires reasoning about state subsumption, a technique where a partial state is discarded if it is subsumed by another previously expanded state (Alcázar et al., 2013). Since Fast Downward only supports progression-based search, implementing a regression framework from scratch is beyond the scope of this thesis.

What we aim for instead is a heuristic table that can be reused in a progression search without the need for repeated recomputation. One promising idea is to reverse the planning task itself, compute the heuristic table on this reversed task, and then use the resulting evaluations in the original progression search. This idea is based on the concept of a *duality task*.

7.1 STRIPS Duality Task

Suda (2013) showed that for every STRIPS planning task, there exists a corresponding dual task that effectively transforms progression search into regression search. This transformation is defined by a mapping that swaps the initial state and goal by taking their complements. Additionally, preconditions of operators are exchanged with their delete effects and vice versa.

Definition 15 (Dual task)**.** *The dual task* Π^d *of a STRIPS planning task* $\Pi = \langle V, O, I, G \rangle$ *is defined as*

$$\Pi^d = \langle V, \{o^d \mid o \in O\}, V \setminus G, V \setminus I \rangle$$

where for each operator $o \in O$*, its dual operator* o^d *is given by*

$$o^d = \langle \mathit{del}(o), \mathit{add}(o), \mathit{pre}(o), \mathit{cost}(o) \rangle.$$

We refer to the operators o^d as *dual operators.* For every planning task Π, the dual task Π^d has a solution if and only if Π does. Moreover, a sequence $\pi = \langle o_1, \dots, o_n \rangle$ is a plan for Π if and only if $\langle o_n^d, \dots, o_1^d \rangle$ is a plan for Π^d (Suda, 2013). Applying the dual transformation twice yields the original planning task again, i.e., $(\Pi^d)^d = \Pi$. This property is the reason the term duality is used in this context.
It is important to note that the duality definition applies to planning tasks in STRIPS form. To apply this transformation to SAS$^+$ tasks in Fast Downward, we first convert them into STRIPS tasks using the transformation described in Definition 14.

To construct the heuristic table, we apply our h^2 implementation to the dual task Π^d. Since we are still performing progression search, we evaluate the dual initial state as we want distance estimates to the original goal. Therefore, we compute h^2 values starting from the dual initial state $I^d = V \setminus G$.
During search, each encountered state s in the original SAS$^+$ task must be transformed into a corresponding dual state s^d to enable heuristic look ups. This transformation proceeds in two steps. First, the SAS$^+$ state s is translated into its equivalent STRIPS representation s', as defined by the translation in Definition 14. Then, we compute the dual state s^d by identifying the set of STRIPS variables not present in s'. Formally, this corresponds to $s^d = V \setminus s'$, where V denotes the full set of STRIPS variables resulting from the translation of Π. The look up is performed using the `eval` function (Algorithm 2).
Finally, the heuristic values for a SAS$^+$ state s are of the form $h^2(I^d, s^d)$. In Figure 7.1, we see an illustration of the transformation process to retrieve a state evaluation.

It is important to understand that the heuristic values obtained in this way are not equivalent to the actual h^2 values by Definition 10. Recall that h^2 evaluates the cost of reaching the most expensive subset of atoms of size ≤ 2 in a given state. In the dual approach, we evaluate the complement of the state, so the atom sets under consideration are inherently different. Thus, this is a method for deriving heuristic estimates using h^2 in the process, although the resulting values generally differ from the true h^2 values as defined in Definition 10.

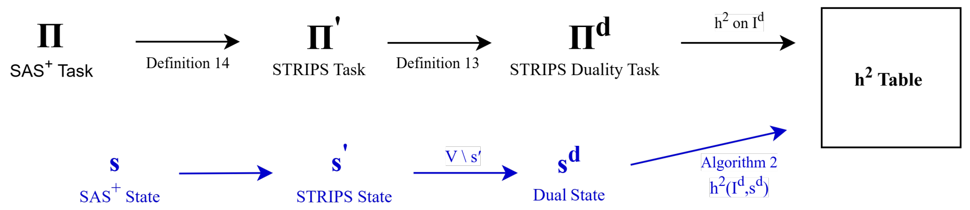


Figure 7.1: Overview of the transformation steps used to evaluate a state s in the original SAS$^+$ task Π. Preprocessing steps performed before the search are shown in black, while state-specific transformations applied during the search are shown in blue.

7.2 Comparison with Other Heuristics

We refer to the heuristic that simulates regression search using the dual task as $h^2(\Pi^d)$. First, we analyze this approach from a theoretical perspective. Afterwards, we evaluate $h^2(\Pi^d)$ experimentally by comparing it with other heuristics.

7.2.1 Theoretical Complexity

Since this approach transforms the original planning task, we need to redefine the relevant task parameters. We begin by translating the planning task $\Pi = \langle V, O, I, G \rangle$ into STRIPS form. This transformation has $|V| \times d$ new binary variables in the worst case. Next, we construct the dual task Π^d. Note that this transformation does not increase the size of the planning task. In particular, the number of operators remains unchanged as it holds $|O| = |O^d|$.

Runtime Complexity

For the runtime analysis, we distinguish between the preprocessing time and the time required for a single heuristic evaluation during the search.

Preprocessing Complexity: Before starting the search, we first transform the task into STRIPS form and then compute the heuristic table for Π^d. First, we iterate over all atoms to create STRIPS variables, which takes $\mathcal{O}(|V| \times d)$ time. The same holds for encoding the dual initial state and dual goal.
Next, we construct the dual operators. For efficiency reasons, the transformation to STRIPS operators and the subsequent translation to dual operators are performed in a single combined step. For each operator $o \in O$, we create a corresponding dual operator o^d as defined in Definition 14 and Definition 15. This process involves identifying and swapping the delete effects and preconditions for each operator in STRIPS form. The worst case size of delete effects for STRIPS operators is $|V| \times d$. As a result, constructing all dual operators has a runtime complexity of $\mathcal{O}(|O| \times |V| \times d)$.
After constructing the dual task, we compute the heuristic table using our optimized version of h^2. The complexity of computing the full h^2 table is $\mathcal{O}(|O^d| \times |V^d|^5)$, as we disregard the constant domain size of 2 (Section 4.2.1). When using the redefined parameter $|V^d| = |V| \times d$, we obtain a final complexity of $\mathcal{O}(|O| \times |V|^5 \times d^5)$. This dominates the complexity of the

earlier task transformation steps.

Evaluation Complexity: During the search, we evaluate individual states by first mapping the state to the dual task via negation of all STRIPS variables. We then obtain the heuristic value using the `eval` function (Algorithm 2). The runtime of this look up depends on the size of the state itself, which is $\mathcal{O}(|V^d|^2) = \mathcal{O}(|V|^2 \times d^2)$. As expected, this is significantly faster than the computation of the entire heuristic table.

Space Complexity

For the space complexity, we rely on the results from Section 4.2.1. However, these results hold for SAS$^+$ task. Note that in our STRIPS task we have $|V| \times d$ variables and a constant domain size of 2. The resulting space complexity becomes

$$\mathcal{O}(|V|^2 \times d^2 + |O| \times |V|^2 \times d^2 + |O| \times |V| \times d).$$

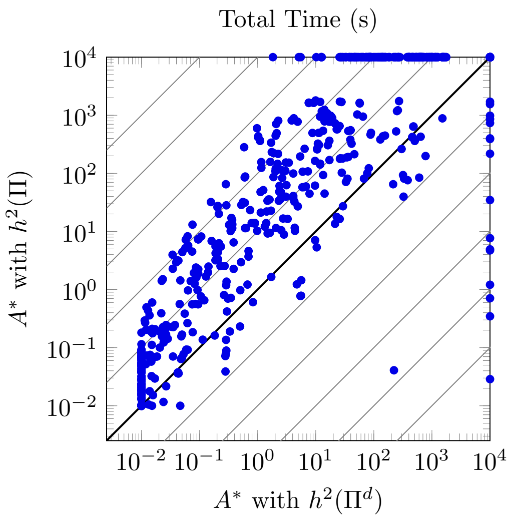
This accounts for the storage of heuristic values and further operator data structures used in the h^2 implementation.

7.2.2 Experimental Comparison

We now present an experimental analysis of $h^2(\Pi^d)$. As a baseline, we compare it against our h^2 heuristic. From a theoretical standpoint, using the precomputed heuristic table of the dual task allows for significantly faster evaluations. However, unlike previous experiments, the use of different heuristic values results in different search spaces. This makes it interesting to compare not only the total time but also the number of expanded states.
Figure 7.2 presents two scatter plots depicting the total time for solving problem instances and the number of state expansions.

The first plot clearly shows that most instances are solved faster when using the duality task approach. Although there are a few problem instances where $h^2(\Pi)$ performs better, the majority of tasks massively benefit from using $h^2(\Pi^d)$ in terms of total runtime. However, the picture changes when considering the number of expanded states. The dual task heuristic typically requires significantly more expansions. On average, there are about ten times more than for $h^2(\Pi)$. This indicates that $h^2(\Pi^d)$ leads to larger search spaces compared to h^2. Nevertheless, due to the fast evaluation time of each state, the overall runtime is still lower in most cases.

Overall, $h^2(\Pi^d)$ is a heuristic that is computationally cheaper to evaluate but also significantly less informative. To further characterize its performance, we compare it with doing a blind search which does not incorporate any knowledge about the current state in the planning task. Together with A^*, this corresponds to a uniform-cost search. In Fast Downward, we can simulate a blind search by using the blind heuristic h_{blind}. This heuristic assigns a heuristic value of zero to all states that are a goal, i.e. $h_{\text{blind}}(s, G) = 0$ if $s \in G$. For other states it holds that $h_{\text{blind}}(s, G) = min_{o \in O} cost(o)$. The heuristic evaluation can be done



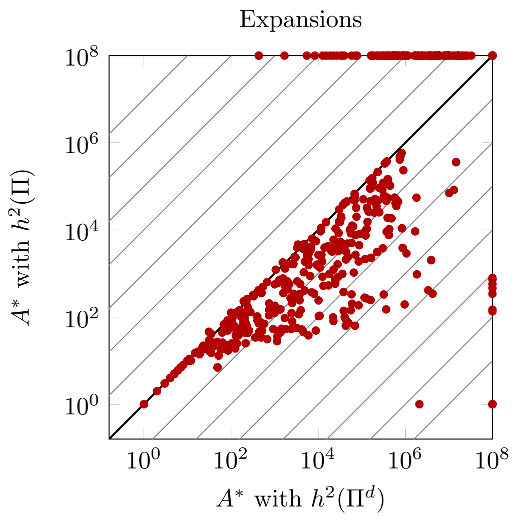


Figure 7.2: Total time and number of state evaluations for h^2 (A^* with $h^2(\Pi)$) and h^2 on the dual task (A^* with $h^2(\Pi^d)$). Unsolved tasks are shown at the plot boundaries, depending on which heuristic solved the task.

in constant time while lack of any heuristical accuracy usually results in very large search spaces.

It is important to note that we still use A^* as the search algorithm when employing h_{blind}. Thus, the search still prioritizes states that are closer to the initial state based on path cost. Figure 7.3 presents a comparison between $h^2(\Pi^d)$ and h_{blind}.
The results show that $h^2(\Pi^d)$ solves problems more slowly than $h_{\text{blind}}(\Pi)$. This can be attributed to the heuristic evaluation required for a single state which is significantly faster for $h_{\text{blind}}(\Pi)$. The second plot, which displays the total number of state expansions, provides additional insight. It shows that both heuristics are relatively similar in terms of the number of expanded states. While the number of expansions for dual h^2 is generally lower, the average difference is significantly less than one order of magnitude. This indicates that $h^2(\Pi^d)$ is only slightly more informative than h_{blind}. Given that the blind heuristic, by definition, provides almost no information about the planning task, this implies that the informational quality of our dual h^2 heuristic is very low.

More problem instances are solved with the dual version than with the original h^2 heuristic. However, this advantage is not due to the informativeness of the heuristic but rather due to the really fast computation of heuristic values. This is not necessarily desirable, especially in the case of h^2. The critical path heuristic h^m for $m \geq 2$ is specifically designed to be more informative as $h^{\max}$ for example. The goal of h^2 is to exploit more properties of the planning task to reduce the number of state expansions. In contrast, the dual h^2 heuristic exhibits the opposite behavior, massively increasing the number of expansions while reducing informativeness.

This raises the question of why $h^2(\Pi^d)$ contains significantly less information than the orig-

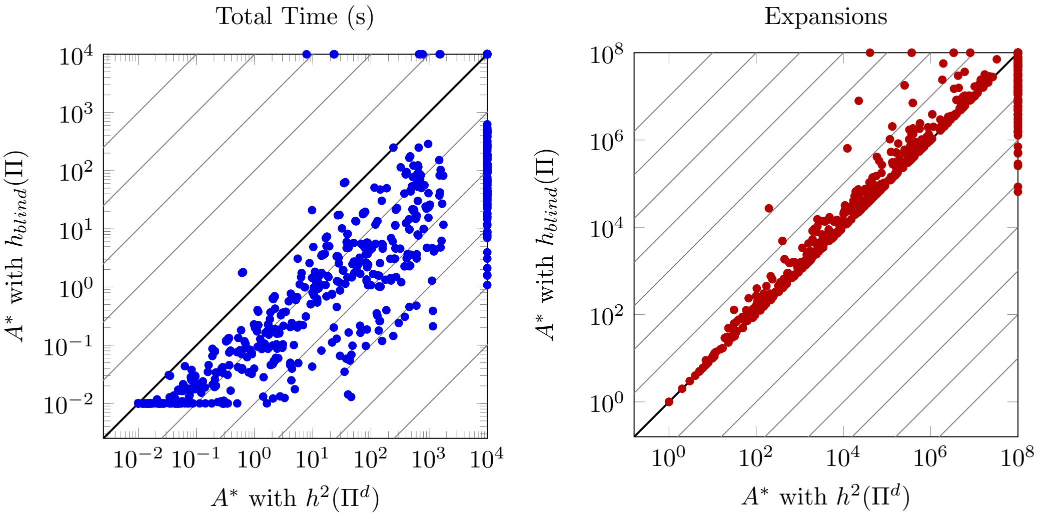


Figure 7.3: Total time and state evaluations of the dual h^2 (A^* with $h^2(\Pi^d)$) and h_{blind} (A^* with $h_{\text{blind}}(\Pi)$). Unsolved tasks are displayed on the edges of the plot, depending on which heuristic solved the task.

inal h^2 heuristic. A look at the heuristic table computed before the search process reveals consistently low heuristic values. The heuristic value of the initial state is typically much lower than the actual cost of the optimal solution.
For admissible heuristics, it holds that the heuristic value must be less than or equal to the actual distance to the goal. Higher heuristic values are generally more desirable as they offer better guidance during search.

One likely cause of the low heuristic values lies in the interaction between the STRIPS transformation and the dual translation. When converting from SAS$^+$ to STRIPS, many new variables are introduced, each representing a single atom in the SAS$^+$ task. Because each SAS$^+$ variable can only assume a single value, most of the STRIPS variables are set to `false`. Therefore, only a small subset of STRIPS variables must be `true` to reach the goal. In the dual task, the initial state is the complement of the original goal (Definition 15). As a result, many of the STRIPS variables are set to `true` in the dual initial state.
This can be problematic for h^2. The critical path heuristic for $m = 2$ evaluates the most expensive subset of atoms of size ≤ 2. If most variables are already present in the initial state, the heuristic values tend to be low as small subsets are easily reachable. This does not mean that the dual problem itself is easy to solve. According to the dual task definition, the optimal solution length remains the same. The challenge in the dual task often lies in reaching larger subsets of variables, a type of complexity that h^2 cannot capture effectively since it only considers subsets of size two or less.

In summary, our transformation from SAS$^+$ to STRIPS, combined with the duality transformation, results in a significantly less informative h^2 heuristic. Although the total time to solve problem instances is lower, $h^2(\Pi^d)$ does not fulfill the typical goals associated with the h^2 heuristic, namely providing informative estimates.

8 Conclusions

In this thesis, we explored different implementations of the critical path heuristic h^m for $m = 2$ in the context of the Fast Downward planning system. We began by analyzing the existing implementation of h^m in Fast Downward. In doing so, we identified several shortcomings, including the use of a map data structure with logarithmic-time access and redundant computations repeated across multiple heuristic evaluations. Furthermore, we showed that the runtime complexity of h^m is $\mathcal{O}(|O||V|^{3m}d^{m+1}\log(|V| \times d))$, while the space complexity remains low due to the fact that only the heuristic table is stored during computation.

We then proposed a series of optimizations that preserve the core idea of iterative table updates while significantly improving the efficiency of the implementation. Most of these optimizations either shift invariant computations to the preprocessing phase or cache intermediate results during the update steps to avoid redundant work. Additionally, we more precisely characterized when an operator can contribute to improving heuristic estimates and introduced an operator queue that excludes those with no effect.
While these optimizations offer limited theoretical improvement in the worst case, our empirical evaluation shows substantial gains. In particular, our optimized implementation of h^2 solves most of the solved problems more than ten times faster than the original. The increased space requirements proved unproblematic, as memory limits were rarely reached.

We then turned our attention to a Π^m compilation approach. In this method, the planning task is compiled to Π^2 such that computing the h^{max} heuristic on the compiled task yields h^2 values. Originally proposed by Haslum (2009) for STRIPS tasks, we adapted the method to SAS$^+$ tasks by implementing a translation from SAS$^+$ to STRIPS before applying the Π^2 compilation.
While this method performs similarly or slightly worse in theory, in practice it outperforms our optimized h^2 implementation. However, the compiled task requires significantly more memory. As a result, the number of instances where either memory or time limits are exceeded are similar. Additionally, we observed that the effectiveness of the compiled task approach heavily relies on identifying and removing duplicate or dominated operators. While checking for small precondition sets significantly increases performance, the results

decrease if the checks are performed on operators with larger preconditions sets.
We have also shown that the concept of encoding the complexity of h^2 in the task can be applied to our h^2 implementation by introducing binary operators.

Despite the performance improvements achieved, fewer than one-third of the benchmark problems were solved. The core issue lies in the inherent mismatch between h^m and progression search. Computing a single heuristic value requires computing costs for reaching many redundant atom sets of size ≤ 2. In contrast, regression search allows for the reuse of atom set costs across evaluations, since we estimate the distance to the constant initial state instead of the goal.

As Fast Downward currently supports only progression-based search algorithms, we utilized STRIPS duality to simulate regression. By transforming the original task into its dual, we enabled the reuse of heuristic table results across different state evaluations. Although the resulting heuristic values no longer match the original h^2 values, they can be computed significantly faster.
However, our experiments revealed that this approach results in extremely low heuristic values and thus yields a weakly informative heuristic. The root cause likely stems from the interaction between the SAS^+ to STRIPS translation and the dual task transformation, which leads to an oversimplification of the planning problem for h^2.

8.1 Future Work

This section outlines possible directions for future research that could build upon the work presented in this thesis. While some ideas further explore the critical path heuristic, others extend into related areas. We begin with topics most closely related to the goals of this thesis and then discuss broader extensions.

8.1.1 Regression Search in Fast Downward

As discussed earlier, Fast Downward currently only supports progression search algorithms. However, the critical path heuristic theoretically performs better in combination with regression search. A natural extension of this work would be to implement regression-based search algorithms in Fast Downward. Promising steps in this direction already exist. For instance, Thüring (2015) explored the challenges of regression search in Fast Downward, and Alcázar et al. (2013) introduced a regression planner based on the Fast Downward planning system. In that setting, heuristics such as h^{add} showed strong performance. Moreover, Haslum (2006) demonstrated that h^2 in a regression framework can achieve results comparable to state-of-the-art pattern database heuristics.
Structurally, a regression-based h^2 heuristic would resemble the duality task approach described in Chapter 7. Using our h^2 implementation, heuristic values for all atom sets of size ≤ 2 could be computed once in a preprocessing step. Our Π^2 compilation approach is not suitable for this purpose, as the $h^{\max}$ evaluation terminates as soon as all goal subsets

become reachable. This means the resulting atom costs contain accurate values for subsets related to the goal, but not for arbitrary sets that may be encountered during search.

8.1.2 Optimizations for h^m with $m > 2$

This thesis focused specifically on h^2, since increasing m leads to exponential growth in the size of the planning task. While this often makes h^m for $m > 2$ impractical (Haslum, 2006), Fast Downward does support arbitrary values of m in its existing implementation. Given that the original implementation of h^m already is rather inefficient for $m = 2$, and theoretically becomes increasingly inefficient for larger m, optimizing h^3 could be worthwhile.

One potential approach would be to generalize our Π^2 compilation to Π^m. This requires only changing the task compilation process, while $h^{\max}$ still performs the heuristic evaluation normally. Since the Π^2 compilation already yielded the best practical results in this thesis, generalizing to higher m values could offer similar advantages for h^3 in comparison to the h^m implementation.

8.1.3 Further Generalization of Π^m

In Chapter 5, we defined the Π^m compilation, which allows computing h^2 values using $h^{\max}$ on a Π^2-compiled task. However, the Π^m compilation itself does not preserve admissibility. This is because, by definition, a meta-operator $a_{o,S}$ in Π^m adds at most $m-1$ meta-variables, whereas in the original task, no such limit exists. As a result, multiple meta-operators in Π^m may be needed to simulate the effect of a single operator in Π. Consequently, using an admissible heuristic on Π^m does not guarantee optimality.

The Π^C compilation, introduced by Haslum (2012), provides an alternative to Π^m by allowing the selection of a custom set C of relevant conjunctions, which can be of any size. Each conjunction in C is represented by a new meta-variable indicating whether it holds in a state. This helps focus computational resources on the most informative atom combinations. Furthermore, each subset of meta-variables gets its own operator copy, ensuring that the Π^C compilation is an admissibility preserving transformation. However, the introduction of these operator copies let the size of the compiled task grow exponentially with $|C|$.
To address this, Keyder et al. (2014) proposed Π^C_{CE}, which introduces conditional effects for meta-operators. Conditional effects are effects that are triggered only if a specified condition holds. The introduction of conditional effects reduces the task size to grow linearly with $|C|$.
Like our dual h^2 variant, using $h^{\max}$ on Π^C or Π^C_{CE} does not lead to h^2 values. However, these compilations support other admissible heuristics, opening the door to further exploration by adjusting either the heuristic or the set C. The usage of conditional effects is no further challenge in Fast Downward, as they are already supported by the planner.

Bibliography

Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

Vidal Alcázar, Daniel Borrajo, Susana Fernandez, and Raquel Fuentetaja. Revisiting Regression in Planning. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 2254–2260, 2013.

Blai Bonet and Hector Geffner. Planning as Heuristic Search. *Artificial Intelligence*, 129: 5–33, 2001.

Christer Bäckström and Bernhard Nebel. Complexity Results for SAS$^+$ Planning. *Computational Intelligence*, 11:625–656, 1995.

Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3):189–208, 1971.

Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins, et al. PDDL — The Planning Domain Definition Language. *Technical Report CVC TR-98-003/DCS TR-1165. Yale Center for Computational Vision and Control.*, 1998.

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.

Patrik Haslum. *Admissible Heuristics for Automated Planning.* PhD thesis, Linköping University, 2006.

Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative Characterisations of the Generalisation From h^{max} to h^m. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 354–357, 2009.

Patrik Haslum. Incremental Lower Bounds for Additive Cost Planning Problems. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 74–82, 2012.

Patrik Haslum and Héctor Geffner. Admissible Heuristics for Optimal Planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning ans Scheduling*, pages 140–149, 2000.

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

John E. Hopcroft, Jeffrey D. Ullman, and Alfred V. Aho. *Data Structures and Algorithms*. Addison-Wesley, 1983.

Emil Keyder, Jörg Hoffmann, and Patrik Haslum. Improving Delete Relaxation Heuristics Through Explicitly Represented Conjunctions. *Journal of Artificial Intelligence Research*, 50:487–533, 2014.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Deutschland, 2021.

Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. Zenodo. https://doi.org/10.5281/zenodo.790461, 2017.

Martin Suda. Duality in STRIPS planning. *ArXiv*, abs/1304.0897, 2013.

Andreas Thüring. Evaluation of Regression Search and State Subsumption in Classical Planning. Bachelor's Thesis, University of Basel, 2015.

A Appendix

Use of Language Models

During the preparation of this thesis, I used large language models as a tool to support language refinement. Specifically, I employed the model to assist with rephrasing, grammar correction, and stylistic improvements of text that I had written myself.
At no point was any technical, conceptual, or scientific content generated by the language model. All ideas, results, formulations of arguments, and interpretations presented in this work are my own. The LLM was used solely as a writing aid and not as a source of original content.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

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

Title of Thesis:	Efficient Implementation of h^2 in the Fast Downward Planning System
Name Assessor:	Prof. Dr. Malte Helmert
Name Student:	Elia Hänggi
Matriculation No.:	2020-055-497

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: Brislach, 20.05.2025 Student: [illegible]

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

☐ 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: 20.05.2025

Place, Date: Brislach, 20.05.2025 Student: [illegible]

Place, Date: ______________ Assessor: ______________

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

September 2023