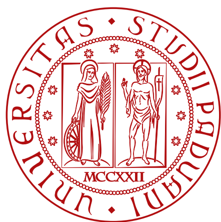

Department of Information Engineering

Master's Degree in Computer Engineering

Design and Evaluation of a Novel Heuristic for Delete-Free AI Planning

Supervisor
Prof. Domenico Salvagnin

Candidate
Andrea Stocco
2108885

Academic Year 2024-2025

Graduation date 07/10/2025

To my family.

Abstract

In this thesis, we present a novel heuristic for solving *delete-free AI planning* problems. Delete-free planning, where actions do not remove facts from the world state, offers a simplified yet meaningful subset of classical planning. Our proposed heuristic estimates, for each action, the cost of the shortest path required to reach a goal fact. This is complemented by a pruning strategy that identifies and excludes actions which do not contribute to achieving the goal.

We evaluate our method against established primal heuristics on a comprehensive set of benchmark problems. The results demonstrate that our method achieves superior performance, solving the most instances and producing higher-quality plans with greater stability across random seeds. While iterative refinement techniques showed limited gains, the core heuristic proves to be a highly effective constructive method for delete-free planning.

Contents

1 Introduction **1**

2 Methodology **5**

3 Proposed Approaches **7**

3.1 Random 8

3.2 Greedy 9

3.3 Revised $h^{\max}$ Heuristic 10

3.4 $h^{\max}$ Heuristic + Lookahead 12

3.5 Shortest Path Heuristic 14

4 Results **17**

4.1 Random vs Greedy vs Greedy + Pruning vs Shortest Path 18

4.2 Greedy + Pruning vs $h^{\max}$ + Lookahead 19

4.3 Different Versions of Backward Propagation 20

4.4 Randomization Analysis 21

4.5 Iterative Refinement via Shortest Path Reapplication 23

4.6 Iterative Refinement via Uniform Cost Search 24

5 Conclusions **31**

Bibliography **33**

List of Figures

3.1 Running Example 8
3.2 Running example evaluated with the $h^{\max}$ heuristic (without pruning) . . . 12
3.3 Running example evaluated with the $h^{\max}$ heuristic with pruning 13
3.4 Running example evaluated with the Shortest Path heuristic 15

4.1 Cumulative distribution plot for Random, Greedy, Greedy + Pruning, and Shortest Path strategies. 18
4.2 Cumulative distribution plot for Greedy + Pruning, and $h^{\max}$ + Lookahead strategies. 20
4.3 Cumulative distribution plot for Greedy + Pruning, $h^{\max}$ + Lookahead, and Shortest Path strategies, considering only the instances solved by all of them. 21
4.4 Cumulative distribution plot for different backpropagation strategies 22
4.5 Effect of random seed on the outcome variability of the Random approach. 23
4.6 Effect of random seed on the outcome variability of the Greedy approach. . 24
4.7 Effect of random seed on the outcome variability of the Greedy + Pruning approach. 25
4.8 Effect of random seed on the outcome variability of the Shortest Path approach. 27
4.9 Comparison of cumulative distributions for Shortest Path reapplied to different plan segments. 28
4.10 Cumulative distributions for UCS refinement applied to different 10% segments of the initial solution (first five intervals) 28
4.11 Cumulative distributions for UCS refinement applied to different 10% segments of the initial solution (last five intervals) 29
4.12 Cumulative distributions for UCS refinement applied to larger plan segments: first 30%, middle 40%, and last 30% of the initial solution 29

List of Tables

4.1 Number of unsolved runs within the time limit (out of 31,040 total runs). . 17

List of Algorithms

1 Apply . . . 7
2 Random . . . 9
3 Greedy . . . 9
4 $h^{\max}$ heuristic . . . 11
5 Greedy search with $h^{\max}$ heuristic . . . 11
6 Lookahead . . . 13
7 $h^{\max}$ heuristic with lookahead . . . 14
8 Shortest Path . . . 16
9 Uniform Cost Search . . . 26

Chapter 1

Introduction

Automated Planning (or AI Planning) is a core subfield of Artificial Intelligence. It enables intelligent agents to reason about actions and outcomes to achieve long-term objectives. It plays a vital role in applications such as robotics, autonomous systems, logistics, and game AI, where efficient generation of high-quality plans is essential. These real-world scenarios demand algorithms that can generate high-quality plans efficiently under various constraints.

A planning problem, in general terms, can be defined as the task of finding a sequence of actions that leads from a given initial state to a desired goal state. While several formal models exist for defining planning problems, this thesis adopts the SAS^+ *formalism* defined in Definition 1.0.1

Definition 1.0.1 (SAS$^+$ Planning Task)**.** A SAS$^+$ planning task is a 4-tuple $\Pi = \langle V, s_0, s_*, A \rangle$ with the following components:

- V: finite set of state variables v, each with finite domain *dom(v)*
- s_0: variable assignment defining the initial state
- s_*: partial variable assignment defining the goal
- A: finite set of actions (or operators), where each action $a \in A$ has the following components:
  - Preconditions *pre(a)*: partial variable assignment
  - Effects *eff(a)*: partial variable assignment
  - Cost *cost(a)*: non-negative real number

For an action $a \in A$ to be applicable, all of its preconditions *pre(a)* must be satisfied in the current state. Once applied, its effects *eff(a)* are used to update the current state

accordingly. In this context, a *fact* is defined as the assignment of a value to a state variable, i.e., an expression of the form $v = d$ with $v \in V$ and $d \in dom(v)$. Thus, a state can be regarded as the set of facts that are satisfied simultaneously. The solutions to planning problems are called *plans*, and they are formally defined in Definition 1.0.2

Definition 1.0.2 (Plan)**.** A plan for a planning problem is a sequence of actions occurring as labels on a path from the initial state to a goal state. The cost of a plan $\langle a_1, a_2, \ldots, a_n \rangle$ is $\sum_{i=1}^{n} cost(a_i)$. A plan is optimal if it has minimal cost.

The class of problems considered in this thesis is a relaxation of the general SAS$^+$ model, referred to as *delete-free* planning. In the standard formalism, applying an action may overwrite existing facts: for instance, if a variable v currently satisfies $v = d$, an effect setting $v = d'$ implicitly deletes the previous assignment. In contrast, in the delete-free setting, actions never remove facts. Once a fact becomes true, it remains true for the rest of the plan, even if the same variable is later assigned a different value. As a consequence, states are monotonic and may contain multiple values for the same variable, distinguishing them from those of the original planning problem. This structural property simplifies reasoning about reachability and reduces the complexity of exploring the search space, while still retaining much of the problem's inherent difficulty.
Optimal planners, such as *Fast Downward*, are guaranteed to find the best possible solution to a planning task, but they can be computationally expensive. Such planners rely on a core search algorithm named A*. Specifically, A* selects actions by minimizing the function

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost required to reach node n, and $h(n)$ is a *heuristic function* that estimates the cost to reach the goal state from n. A heuristic is called *admissible* if it never overestimates the true cost to reach the goal. By expanding nodes in order of increasing $f(n)$, A* always explores the most promising states first, and when an admissible heuristic is used, it is guaranteed to return an optimal solution.
While heuristic functions serve as estimators within search algorithms like A*, this thesis instead focuses on *primal heuristics*—constructive strategies that directly generate feasible plans. These methods aim to efficiently produce solutions of good quality, even though they do not guarantee optimality. Designing effective primal heuristics for delete-free domains is a key area of study, especially when exact methods become impractical due to problem complexity.
After outlining the experimental methodology, the thesis presents several well-known

primal heuristics for delete-free planning, together with the implementation of a novel approach. All algorithms are evaluated in terms of solution quality across a range of benchmark instances. In addition, the study explores the impact of randomization on performance and investigates refinement strategies aimed at improving initial solutions.

Chapter 2

Methodology

This chapter outlines the methodological approach used to compare the selected algorithms. Several algorithms and variations were implemented in this thesis. Some were chosen solely as baselines for the experiments, such as Random and Greedy. As their names suggest, the former selects a random action at each iteration, while the latter always chooses the action with the minimum cost. The $h^{\max}$ heuristic [1] is one of the most important heuristics in delete-free planning. A revised version of this heuristic, incorporating a pruning logic, is proposed in this work. Additionally, a novel heuristic was developed, which assigns to each action the cost of the shortest path to reach the goal state and then selects the action that is closest to the goal.
The experiments were conducted on a testbed containing 3,104 domain-independent instances. Each algorithm was run on every instance with 10 different random seeds, resulting in 31,040 runs per algorithm. A time limit of 60 seconds was set for each run. All experiments presented in this thesis were conducted on a computing cluster maintained by the Operations Research group at the Department of Information Engineering (DEI), University of Padova. The cluster consists of 15 identical blades, each equipped with an Intel(R) Xeon(R) CPU E5-2623 v3 @ 3.00GHz, 16GB of RAM, and running Fedora 37.
The algorithms were compared in terms of solution cost, where a lower cost indicates a better solution. To evaluate the effectiveness of the algorithms, a *cumulative distribution plot* is used. The concept of the primal gap, originally defined in the context of Mixed Integer Programming (MIP) (see Definition 2.0.1) [2], is adapted here to quantify the quality of solutions across algorithms.

Definition 2.0.1 (Primal Gap)**.** Let $\tilde{x}$ be a solution for a problem, and $\tilde{x}_{opt}$ be an optimal (or best known) solution for that problem. We define the primal gap $\gamma \in [0, 1]$ of $\tilde{x}$ as

follows:

$$\gamma\left(\tilde{\mathrm{x}}\right) \coloneqq \begin{cases} 0, & \text{if } |c^T\tilde{x}_{opt}| = |c^T\tilde{x}| = 0, \\ 1, & \text{if } c^T\tilde{x}_{opt} \cdot c^T\tilde{x} < 0, \\ \frac{|c^T\tilde{x}_{opt} - c^T\tilde{x}|}{max\{|c^T\tilde{x}_{opt}|, |c^T\tilde{x}|\}}, & \text{else.} \end{cases}$$

An algorithm that finds the best solution for a given instance will exhibit a primal gap of zero. For all other algorithms, the primal gap approaches zero as the solution quality approaches the best known, and approaches one as it deviates further from it. For each algorithm, a cumulative distribution plot is generated by plotting the percentage of instances on which the algorithm has a primal gap below a given threshold. A more detailed explanation of this analysis will be provided in the following sections.

Chapter 3

Proposed Approaches

Heuristics are algorithms to determine a near-optimal solution to a problem. In certain scenarios, exact methods are unable to provide the optimal solution in a reasonable amount of time, or may fail entirely due to memory limits. In contrast, a heuristic is generally capable of offering a solution that is more or less close to the optimal one. While there is no assurance regarding the optimality of the provided solution, it may still be considered acceptable in some instances. This chapter focuses on explaining the algorithms developed, while the following one presents their evaluation.

Before presenting the algorithms, it is useful to introduce a fundamental function used by all of them. An action is applicable only if all its preconditions are satisfied in the current state. When applied, it produces a new state by adding its effects to the current one. The pseudocode for this function is shown in Algorithm 1.

Algorithm 1 Apply

Output: $newState$

1: **procedure** APPLY($currentState, actionToApply$)
2: $\quad newState \leftarrow$ COPY($currentState$)
3: $\quad$ **for all** $fact \in actionToApply.effects$ **do**
4: $\quad\quad$ **if** $fact \notin newState$ **then**
5: $\quad\quad\quad newState \leftarrow newState \cup \{fact\}$
6: $\quad\quad$ **end if**
7: $\quad$ **end for**
8: $\quad$ **return** $newState$
9: **end procedure**

To illustrate the behavior of the heuristics, a simple planning instance is introduced as a running example throughout this chapter.

Definition 3.0.1 (Running Example)**.** The instance consists of an initial state s_i, a goal state s_g, a set of intermediate states $s_1, \ldots, s_{13}$, and a set of actions $a_1, \ldots, a_{15}$, each with

uniform cost 1. Each state can take only two values, true or false, and each action has exactly one precondition and one effect. An action can be applied only if its precondition state is true; once applied, its effect state becomes true as well.

The structure of the instance is shown in Figure 3.1. It is represented as a graph, where each circle denotes a state and each edge corresponds to an action. Although the edges are drawn as undirected, each action conceptually represents a transition from its precondition to its effect. For example, action a_2 is applicable in s_i, and its effect is to reach s_2. Subsequent figures reuse this example to highlight the differences among heuristics, with each edge labeled by the corresponding heuristic cost.

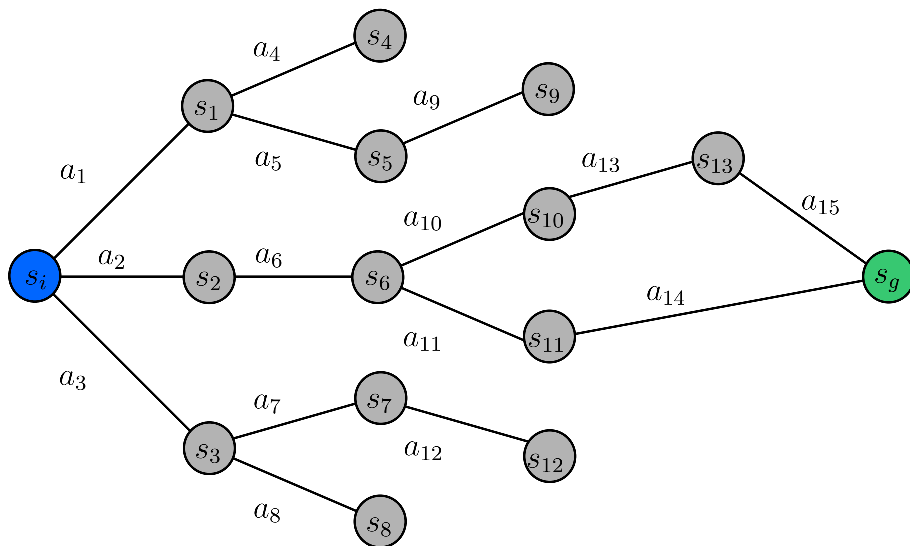

Figure 3.1: Running Example

3.1 Random

This is the simplest algorithm implemented, and it is used as a baseline for the evaluation. At each iteration, it applies a random action from the applicable ones. It is not a sophisticated or informed solution but this strategy can be useful in some cases because, by chance, a good plan may be found. Algorithm 2 shows the pseudocode for it.

It is important to notice that if *possibleActions* is empty, then a solution to the planning task cannot exist. As said in Chapter 1, the problems are delete-free—once a fact becomes true in the current state, it remains true until the end. Therefore, if there are no actions to apply, it does not mean that the algorithm has reached a dead end that could be avoided by applying different actions in a different order. Rather, it indicates that no sequence of actions can lead to the goal state, and therefore no plan exists. The natural evolution of

Algorithm 2 Random

Output: *solution*

1: **procedure** RANDOM($planningTask$)
2: $currentState \leftarrow planningTask.initialState$
3: $solution \leftarrow [\,]$ ▷ Initialize empty solution
4: **while** $planningTask.goalState \nsubseteq currentState$ **do**
5: $possibleActions \leftarrow \{a \in A \mid pre(a) \subseteq currentState\}$
6: **if** EMPTY($possibleActions$) **then**
7: **return** $failure$ ▷ Infeasible problem
8: **end if**
9: $actionToApply \leftarrow$ RANDOMCHOICE($possibleActions$)
10: $currentState \leftarrow$ APPLY($currentState, actionToApply$)
11: APPEND($solution$, $actionToApply$)
12: **end while**
13: **return** $solution$
14: **end procedure**

this approach is to avoid choosing randomly the next action to apply and, instead, use a strategy to select the action that is more likely to lead to a better plan.

3.2 Greedy

A greedy algorithm is a heuristic that attempts to find an optimal solution by selecting the locally best possible choice at each iteration. For instance, in our case, the next action to apply will always be the one with the minimum cost. In the case of a tie, a random action among those with the same cost will be chosen. Algorithm 3 shows the pseudocode for this algorithm. The omitted parts are identical to those in Algorithm 2.

Algorithm 3 Greedy

Output: *solution*

1: **procedure** GREEDY($planningTask$)
2: ...
3: $possibleActions \leftarrow \{a \in A \mid pre(a) \subseteq currentState\}$
4: **if** EMPTY($possibleActions$) **then**
5: **return** $failure$
6: **end if**
7: $actionToApply \leftarrow$ MINIMUMCOSTACTION($possibleActions$)
8: ...
9: **end procedure**

Instead of randomly choosing the next action to apply, here the action with minimum cost is always selected. For this and all subsequent algorithms, if multiple eligible actions

share the same minimum (heuristic) cost, one of them is selected at random.

3.3 Revised $h^{\max}$ Heuristic

The $h^{\max}$ heuristic is one of the most well-known delete-relaxation heuristics in the literature. It is an *admissible* heuristic, meaning that it never overestimates the cost of reaching the goal. A formal definition is given in Equations 3.1 and 3.2.

$$h^{\max}(p;s) := \begin{cases} 0 & \text{if } p \in s \\ \min_{a \in O(p)} \left[h(a;s)\right] & \text{otherwise} \end{cases} \tag{3.1}$$

where $h^{\max}(p;s)$ stands for an estimate of the cost of achieving the fact p from the current state s, $O(p)$ is the set of actions $\{a \in A \mid p \in \mathit{eff(a)}\}$, and

$$h(a;s) := \mathit{cost}(a) + \max_{q \in \mathit{pre}(a)} \left[h^{\max}(q,s)\right] \tag{3.2}$$

stands for the cost of achieving the preconditions of an action a and applying it.
The idea behind the use of this heuristic is similar to that of the Greedy approach. While in the Greedy approach the action with the minimum actual cost is chosen, in this case, the action with the minimum heuristic cost is selected at each iteration. This thesis presents an implementation of the $h^{\max}$ heuristic that goes a step further. By definition, any action applicable in the current state has a heuristic cost equal to its own cost, since all its preconditions are satisfied—regardless of whether the action is actually useful for reaching the goal state or not. The implemented version performs a technique called *pruning*, which consists of assigning an infinite cost to useless actions—indicating that these actions are not helpful in finding a good plan. As a result, such actions are effectively excluded from consideration when selecting the next action to apply, while the remaining ones retain their correct heuristic cost. The pseudocode for this algorithm is presented in Algorithm 4 and 5.

Starting from each fact in the goal state, HMAX is invoked to perform backward cost propagation. Through this process, actions that do not contribute to achieving the goal will implicitly retain their initial cost of $+\infty$, effectively marking them as useless. When selecting the next action to apply, any action with infinite cost is ignored, and the one with the lowest heuristic cost among the remaining options is chosen.
Figure 3.2 illustrates the behaviour of the $h^{\max}$ heuristic without the pruning technique. When the heuristic is applied from the initial state, the actions a_1, a_2, and a_3 are all deemed applicable and assigned heuristic costs equal to their actual cost. However, since

Algorithm 4 $h^{\max}$ heuristic

Output: *minHcost*

```
 1: procedure HMAX(currentState, fact)
 2:     if fact ∈ currentState then
 3:         return 0                                        ▷ Base case
 4:     end if
 5:     actions ← {a ∈ A | fact ∈ eff(a)}   ▷ Get all the actions having fact as effect
 6:     if EMPTY(actions) then
 7:         return +∞                                       ▷ The fact is unreachable
 8:     end if
 9:     minHcost ← +∞
10:     for all action ∈ actions do
11:         maxCost ← 0
12:         for all pre ∈ action.preconditions do
13:             maxCost ← MAX(maxCost, HMAX(currentState, pre))
14:         end for
15:         action.hCost ← action.cost + maxCost
16:         minHcost ← MIN(minHcost, action.hCost)
17:     end for
18:     return minHcost
19: end procedure
```

Algorithm 5 Greedy search with $h^{\max}$ heuristic

Output: *solution*

```
 1: procedure GREEDYHMAX(planningTask)
 2:     ...
 3:     for all a ∈ A do
 4:         a.hCost ← +∞                    ▷ Initialize actions' heuristic cost
 5:     end for
 6:     estimatedCost ← 0
 7:     for all goal ∈ planningTask.goalState do
 8:         estimatedCost ← MAX(estimatedCost, HMAX(currentState, goal))
 9:     end for
10:     possibleActions ← {a ∈ A | pre(a) ⊆ currentState}
11:     if EMPTY(possibleActions) then
12:         return failure
13:     end if
14:     actionToApply ← MINIMUMHCOSTACTION(possibleActions)
15:     ...
16: end procedure
```

multiple actions have the same cost, the tie-breaking mechanism (e.g., random choice) may lead the planner to select a suboptimal branch—such as choosing a_1 or a_3— which does not contribute toward reaching the goal efficiently.

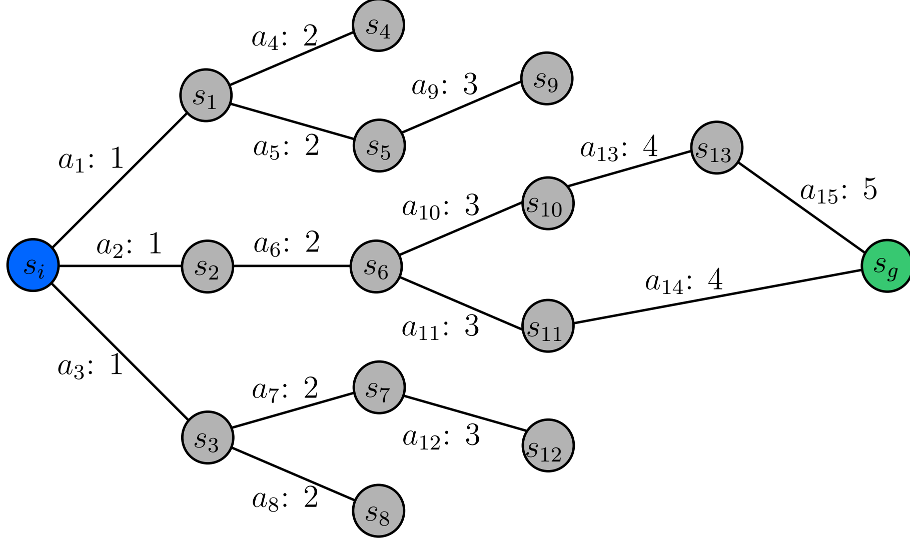

Figure 3.2: Running example evaluated with the $h^{\max}$ heuristic (without pruning)

Figure 3.3 shows the same planning instance, but this time using the $h^{\max}$ heuristic augmented with the pruning technique. Here, actions that do not contribute to achieving the goal are implicitly assigned a heuristic cost of $+\infty$. As a result, the planner deterministically selects a_2 from the initial state, as it is the only action with a finite cost. The heuristic is recomputed after each action, updating costs based on the new state. This guided progression ensures that only goal-relevant actions are considered. For this instance, a possible suboptimal plan that the pruning-enhanced heuristic might produce is $\langle a_2, a_6, a_{10}, a_{13}, a_{15} \rangle$. This occurs because, from state s_6, both a_{10} and a_{11} have the same heuristic cost of one, so the decision between them is made randomly.

As previously mentioned, any action that can be applied from the current state will always have a heuristic cost equal to its actual cost. Therefore, this algorithm can be seen as a Greedy + Pruning approach: it selects the next action based on the action's cost, while discarding useless actions.

3.4 $h^{\max}$ Heuristic + Lookahead

As mentioned in the previous section, the presented algorithm can be considered a Greedy + Pruning approach, since the full potential of the heuristic cannot be exploited—the heuristic cost of the applicable actions is always equal to the actions' actual cost. To evaluate its performance, a new version of the heuristic is proposed. Instead of selecting

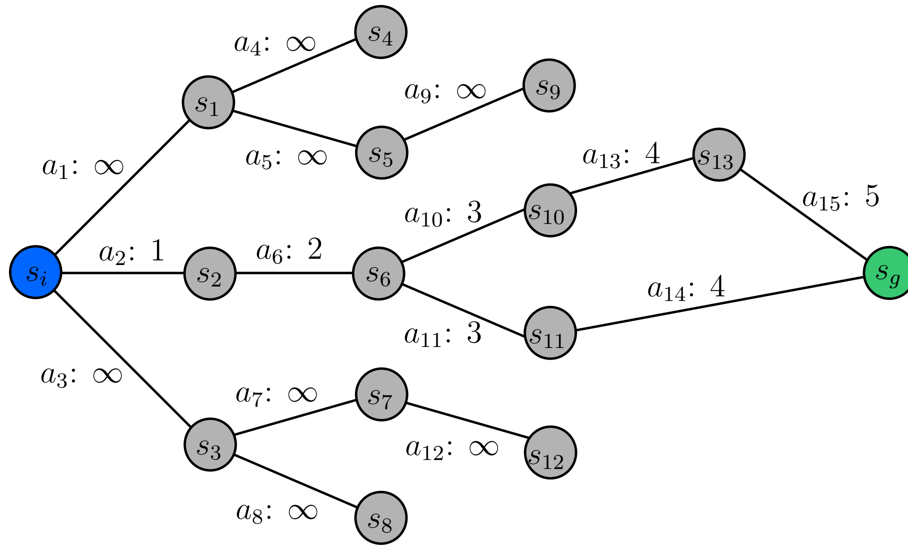

Figure 3.3: Running example evaluated with the $h^{\max}$ heuristic with pruning

the action with the minimum cost, the algorithm tries each possible action, recomputes the HMAX value for each resulting state, and finally applies the action that leads to the state with the lowest new heuristic value. The pseudocode for this approach is shown in Algorithm 6 and 7.

Algorithm 6 Lookahead

Output: *actionToApply*

```
1: procedure LOOKAHEAD(currentState, possibleActions)
2:     for all action ∈ possibleActions do
3:         newState ← APPLY(currentState, action)
4:         estimatedCost ← 0
5:         for all goal ∈ planningTask.goalState do
6:             estimatedCost ← MAX(estimatedCost, HMAX(newState, goal))
7:         end for
8:         action.hCost ← action.hCost + estimatedCost
9:     end for
10:    actionToApply ← MINIMUMHCOSTACTION(possibleActions)
11:    return actionToApply
12: end procedure
```

The algorithm ”looks ahead” at the heuristic cost of the state resulting from each possible action, and then applies only the action that appears to be the most convenient.

Algorithm 7 $h^{\max}$ heuristic with lookahead

Output: *solution*

```
 1: procedure HMAXLOOKAHEAD(planningTask)
 2:     ...
 3:     estimatedCost ← 0
 4:     for all goal ∈ planningTask.goalState do
 5:         estimatedCost ← MAX(estimatedCost, HMAX(currentState, goal))
 6:     end for
 7:     possibleActions ← {a ∈ A | pre(a) ⊆ currentState}
 8:     if EMPTY(possibleActions) then
 9:         return failure
10:     end if
11:     actionToApply ← LOOKAHEAD(currentState, possibleActions)
12:     ...
13: end procedure
```

3.5 Shortest Path Heuristic

In this section, a completely new approach is introduced. As discussed in Section 3.3, the modified version of the $h^{\max}$ heuristic ultimately behaves like a Greedy + Pruning algorithm, limiting its effectiveness. Here, we aim to use the same backward propagation mechanism, but with a different heuristic cost assigned to each action. The core idea is to assign each action a cost that estimates its actual distance to the goal. Specifically, the cost assigned to an action corresponds to the cost of the *shortest path* to reach a fact in the goal state. The mathematical formalism is presented in Equations 3.3 and 3.4:

$$h^{sp}(p; s_*) \coloneqq \begin{cases} 0 & \text{if } p \in s_* \\ \min_{a \in O(p)} \left[h(a; s_*)\right] & \text{otherwise} \end{cases} \tag{3.3}$$

Here, $h^{sp}(p; s_*)$ denotes the estimated cost of achieving the fact p from the goal state s_*. The set $O(p)$ represents the actions that produce p as an effect, that is, $O(p) = \{a \in A \mid p \in \mathit{eff}(a)\}$. The function $h(a; s_*)$ is defined as follows:

$$h(a; s_*) \coloneqq \mathit{cost}(a) + \min_{q \in \mathit{eff}(a)} \left[h^{sp}(q; s_*)\right] \tag{3.4}$$

This expression estimates the cost of achieving a fact in the goal state s_* by applying action a. The heuristic cost of an action is defined as its own cost plus the minimum estimated cost among its effects. In this way, each action is assigned a heuristic cost corresponding to the cost of the shortest path to reach a fact in the goal state. Figure 3.4 illustrates the same example introduced at the beginning of this chapter, but evaluated

with the newly presented heuristic.

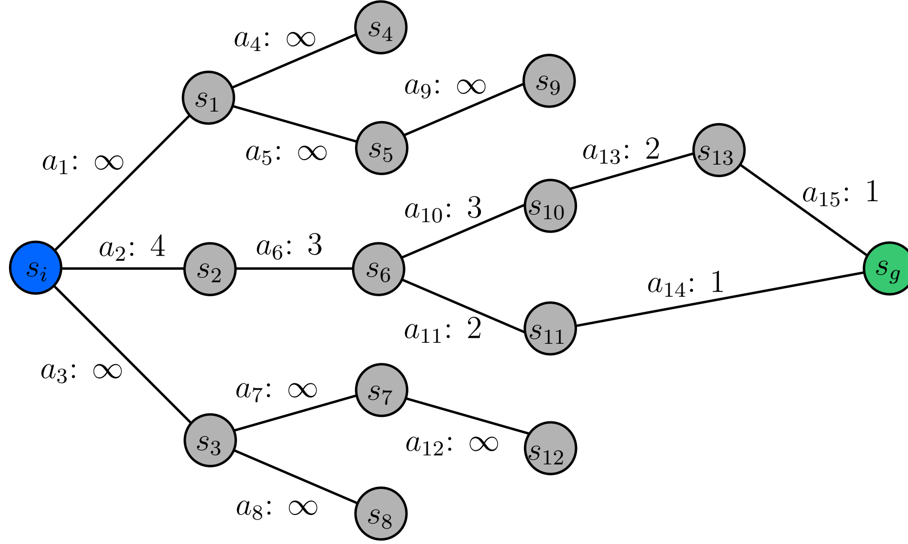

Figure 3.4: Running example evaluated with the Shortest Path heuristic

In this case, the heuristic cost of each action corresponds to the length of the shortest path required to reach a fact in the goal state. For example, if the current state is s_6, reached by sequentially applying actions a_2 and a_6, then the next selected action will be a_{11}, as it lies on the shortest path to the goal. In this case, the resulting plan is always the optimal one: $\langle a_2, a_6, a_{11}, a_{14} \rangle$. As with the previous method, a pruning strategy is applied: any action that does not contribute to reaching the goal state s_g is assigned a cost of $+\infty$ and is thus excluded from consideration. The actual implementation of this algorithm uses a *priority queue* to ensure an efficient and iterative computation. The pseudocode is presented in Algorithm 8.

The priority queue internally maintains the facts ordered by increasing cost. Each time a fact is removed from the queue, it is the one with the minimum heuristic cost. The heuristic cost of each action having the removed fact as an effect is then updated accordingly, and all of its preconditions are subsequently inserted into the queue. In this way, all useful actions will be assigned a cost equal to the shortest path to reach a fact in the goal state, while useless actions will retain an infinite cost. The usage of this algorithm is analogous to what was presented in Algorithm 5.

Algorithm 8 Shortest Path

Output:

```
1: procedure SHORTESTPATH(currentState, goalState)
2:     factCosts ← {+∞, . . . , +∞}                ▷ Initialize facts' cost to +∞
3:     pq ← {}                                      ▷ Empty priority queue
4:     for all goal ∈ goalState do
5:         factCosts[goal] ← 0
6:         PUSH(pq, goal, 0)
7:     end for
8:     while !EMPTY(pq) do
9:         fact ← POP(pq)                           ▷ Get the fact having minimum cost
10:        if fact ∈ currentState then
11:            continue                             ▷ Do not propagate the cost
12:        end if
13:        actions ← {a ∈ A | fact ∈ eff(a)}
14:        for all action ∈ actions do
15:            newCost ← action.cost + factCosts[fact]
16:            if newCost >= action.hCost then
17:                continue
18:            end if
19:            action.hCost = newCost
20:            for all pre ∈ action.preconditions do
21:                if newCost < factCosts[pre] then
22:                    factCosts[pre] ← newCost
23:                    if HAS(pq, pre) then
24:                        CHANGE(pq, pre, newCost)
25:                    else
26:                        PUSH(pq, pre, newCost)
27:                    end if
28:                end if
29:            end for
30:        end for
31:    end while
32: end procedure
```

Chapter 4

Results

This chapter presents an evaluation of the algorithms introduced in Chapter 3. To assess their performance, a cumulative distribution plot was used. This type of plot shows the percentage of instances for which an algorithm found a plan with a primal gap below the threshold indicated on the x-axis. The best algorithm is the one whose curve lies above all the others. The plots presented in the following sections will help illustrate this concept. As described in Chapter 2, each algorithm was run on every instance in the testbed 10 times, each with a different random seed. There are 3,104 instances, so each algorithm was executed a total of 31,040 times. To limit execution time, each run was given a time limit of 60 seconds. The instances come from different domains; some are very large in terms of number of actions, while others are relatively small. Additionally, since some algorithms are more efficient than others, not all the instances were solved by every algorithm. Table 4.1 shows, for each algorithm, the number of runs it was able to solve within the time limit.

Algorithm	**Unsolved within time limit**
Random	1363
Greedy	1328
Greedy + Pruning	1993
$h^{\max}$ + Lookahead	9903
Shortest Path	1042

Table 4.1: Number of unsolved runs within the time limit (out of 31,040 total runs).

By simply examining this table, we can identify the most effective algorithms. The lookahead process is computationally expensive, resulting in approximately one-third of the instances not being solved. In contrast, the efficient implementation of the Shortest Path heuristic appears to be the most capable, solving the highest number of instances.

4.1 Random vs Greedy vs Greedy + Pruning vs Shortest Path

This section presents an evaluation of four planning strategies, ranging from uninformed to increasingly informed approaches: Random, Greedy, Greedy + Pruning, and Shortest Path. As shown in Table 4.1, Random and Greedy exhibit similar efficiency, as both are able to solve a large number of runs. Greedy + Pruning is slightly slower but employs a strategy with the potential to yield higher-quality solutions. Ultimately, the Shortest Path approach achieves the best performance, solving the highest number of runs overall. The next step is to evaluate the quality of the solutions produced by each strategy. Figure 4.1 presents the cumulative distribution plot for the evaluated algorithms.

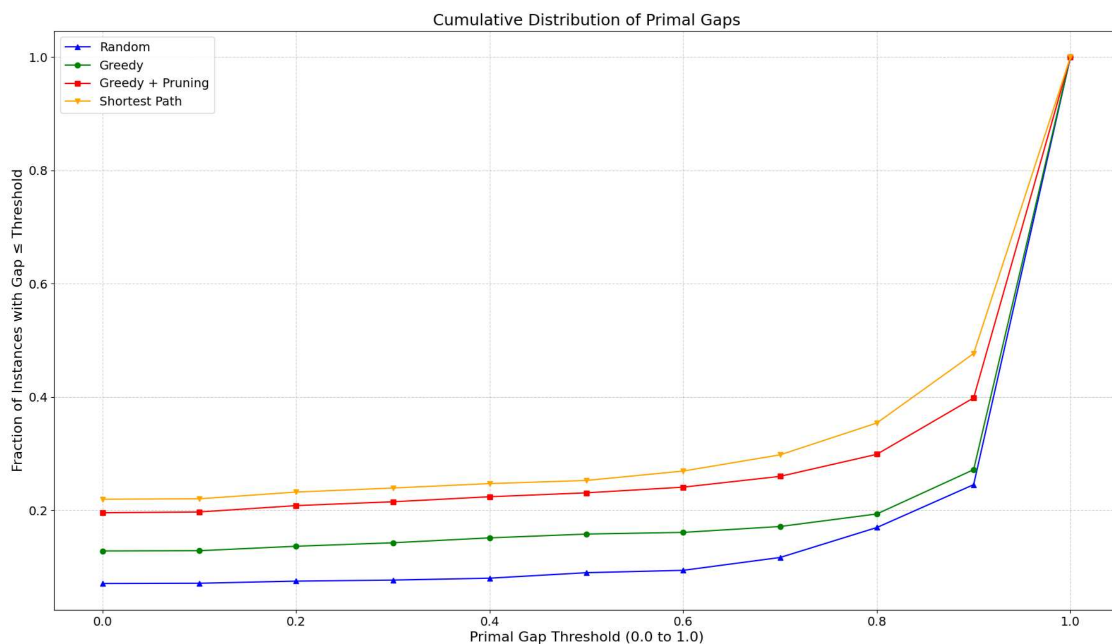

Figure 4.1: Cumulative distribution plot for Random, Greedy, Greedy + Pruning, and Shortest Path strategies.

For instance, to better interpret the plot, consider the Greedy strategy: the point on its curve corresponding to a primal gap threshold of 80% is close to 20%. This indicates that it achieves a solution with a primal gap less than or equal to 80% in approximately 20% of the problem instances. Conversely, the Shortest Path strategy found the best known solution (i.e., a primal gap of 0%) in approximately 22% of the instances. Now that the plot has been properly explained, it is possible to identify the most effective heuristic. The curve that remains above all the others represents the best-performing

strategy, as it indicates that the corresponding algorithm consistently found solutions within the primal gap thresholds for a larger number of instances. As expected, Random and Greedy strategies typically do not produce high-quality plans. Pruning the useless actions appears to be an effective strategy, as it allows the planner to discard actions that are known to be unhelpful whenever multiple options with the same cost are available. Shortest Path builds on pruning by also estimating the distance to the goal for each action, resulting in a more informed and effective selection of actions to include in the plan.

4.2 Greedy + Pruning vs $h^{\max}$ + Lookahead

As shown in the previous section, pruning proves to be an effective strategy. The initial idea was to combine pruning with the $h^{\max}$ heuristic. However, due to the nature of the heuristic's definition, this approach effectively reduces to a Greedy + Pruning strategy. To better evaluate its potential, a lookahead mechanism was implemented to assess whether combining pruning with the $h^{\max}$ heuristic could lead to a more promising solution. This section presents an evaluation of the performance of these two algorithms. The lookahead process is computationally expensive; as a result, the algorithm fails to find a plan for approximately one-third of the instances. Figure 4.2 presents the cumulative distribution plot comparing the performance of these two algorithms.

Unfortunately, this plot alone cannot definitively answer which of the two algorithms produces better-quality plans. The high number of unsolved instances by the $h^{\max}$ + Lookahead algorithm can skew the results: Greedy + Pruning may appear superior simply because it solves more instances, not necessarily because it generates better plans. There is no guarantee that the plans found by Greedy + Pruning are of higher quality than those returned by $h^{\max}$ + Lookahead. Figure 4.3 shows a comparison between the algorithms, considering only the instances that were successfully solved by all of them.

The curve for Shortest Path is included as a reference. Focusing only on the instances solved by all the presented algorithms is particularly valuable: in this case, the curves are not skewed by instances that some strategies failed to solve. This allows us to assess only the quality of the generated plans. From this comparison, it is evident that Shortest Path is the best-performing algorithm overall—both in terms of solution quality and the number of problems it can solve. Additionally, the performance of $h^{\max}$ + Lookahead, which surpasses that of Greedy + Pruning, confirms that combining pruning with the $h^{\max}$ heuristic can be an effective strategy when paired with a lookahead mechanism. Although $h^{\max}$ + Lookahead remains computationally slow, these results indicate that

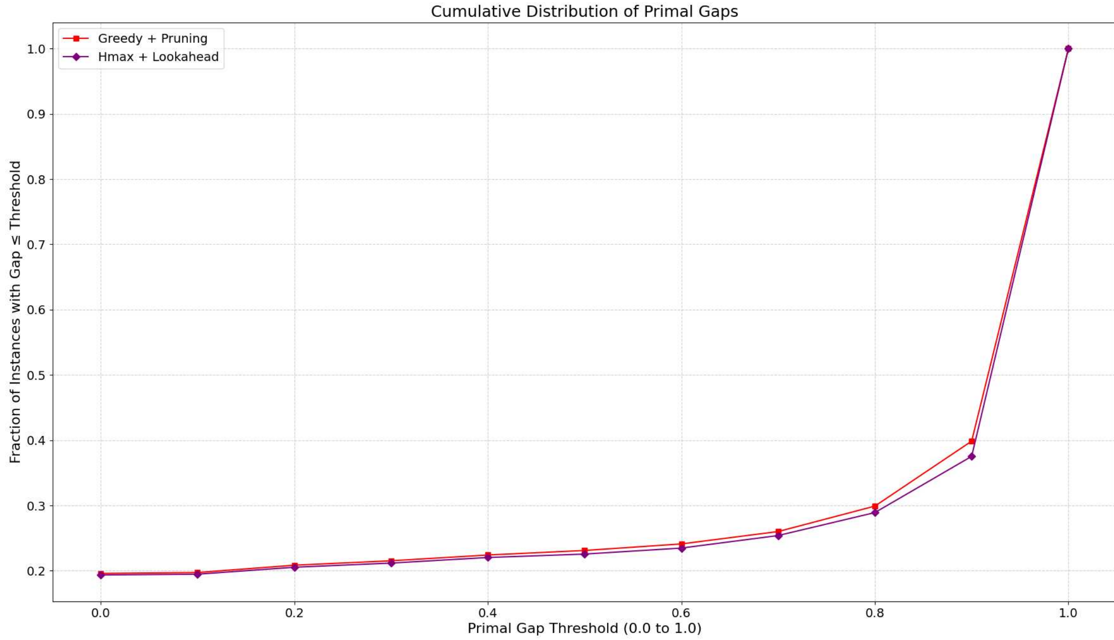

Figure 4.2: Cumulative distribution plot for Greedy + Pruning, and $h^{\max}$ + Lookahead strategies.

the pruning-enhanced $h^{\max}$ heuristic has the potential to yield high-quality solutions if embedded within a more powerful search algorithm.

4.3 Different Versions of Backward Propagation

As explained in Section 3.5, the core idea behind the algorithm is to back-propagate the heuristic cost of reaching the goal state from its constituent facts to those in the current state. Previously, an algorithm was introduced that propagates the minimum cost among an action's effects—effectively computing the shortest path to the goal. With minimal modification, alternative strategies can be explored, such as propagating the *maximum effect cost* or the *sum of all effect costs* to preceding actions. These variations are evaluated in Figure 4.4 to determine which propagation strategy yields better results.

Back-propagating the maximum cost among an action's effects can be interpreted as assigning the action a cost based on the worst-case path to the goal state. Conversely, back-propagating the sum of the effects' costs reflects a perspective where all possible outcomes of the action contribute to estimating the distance to the goal. Each approach offers a different trade-off between pessimism and comprehensiveness in cost estimation. The Shortest Path continues to deliver the best performance overall. In contrast, both

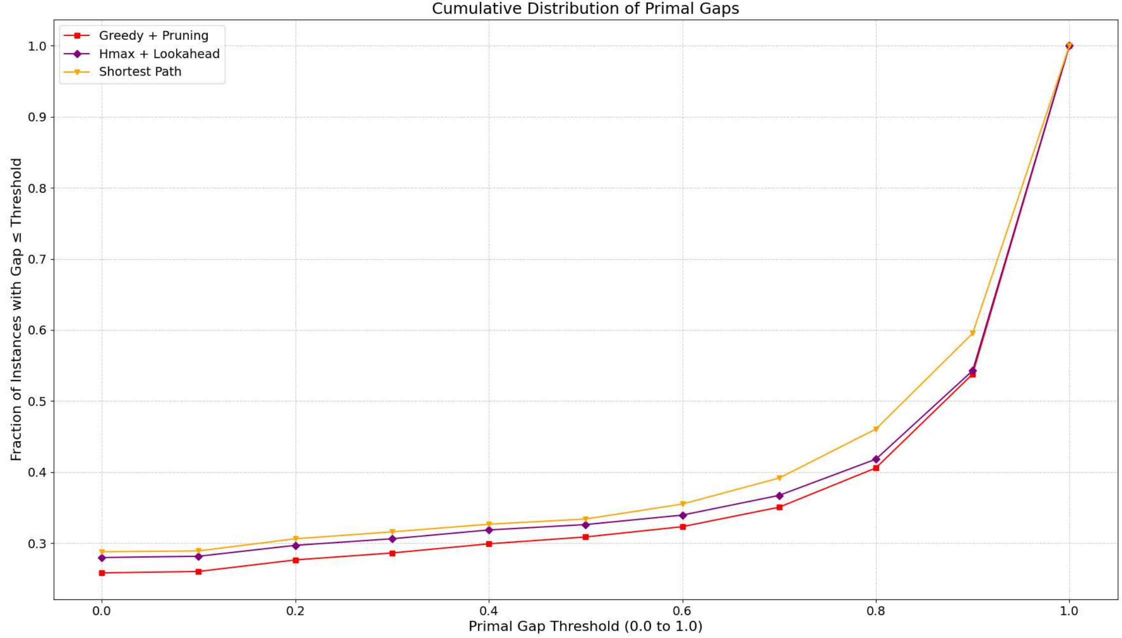

Figure 4.3: Cumulative distribution plot for Greedy + Pruning, $h^{\max}$ + Lookahead, and Shortest Path strategies, considering only the instances solved by all of them.

the max-cost and sum-cost backpropagation strategies exhibit similar behavior, with only marginal differences between them.

4.4 Randomization Analysis

All the presented algorithms share a common trait: when multiple actions have the same minimum heuristic cost, the next action is selected at random among them. This section analyzes the impact of random tie-breaking on the quality of the returned plans. As mentioned earlier, each algorithm was executed on every instance using 10 different random seeds. Varying the seed affects the algorithm's behavior only in cases where multiple actions share the same heuristic cost, influencing which action is selected among them. Figure 4.5 presents the analysis of the impact of randomization on the Random strategy.

It is a *logarithmic plot*: it uses a logarithmic scale on both axis to display data with a wide range of value. Each point represents one instance in the testbed, and its coordinates correspond to the minimum and maximum plan costs found by the algorithm across different random seeds. Some instances exhibit a large discrepancy between the minimum and maximum plan costs. In fact, certain points show a minimum cost of around 10 and a maximum close to 1000, indicating a fluctuation of two orders of magnitude. Selecting

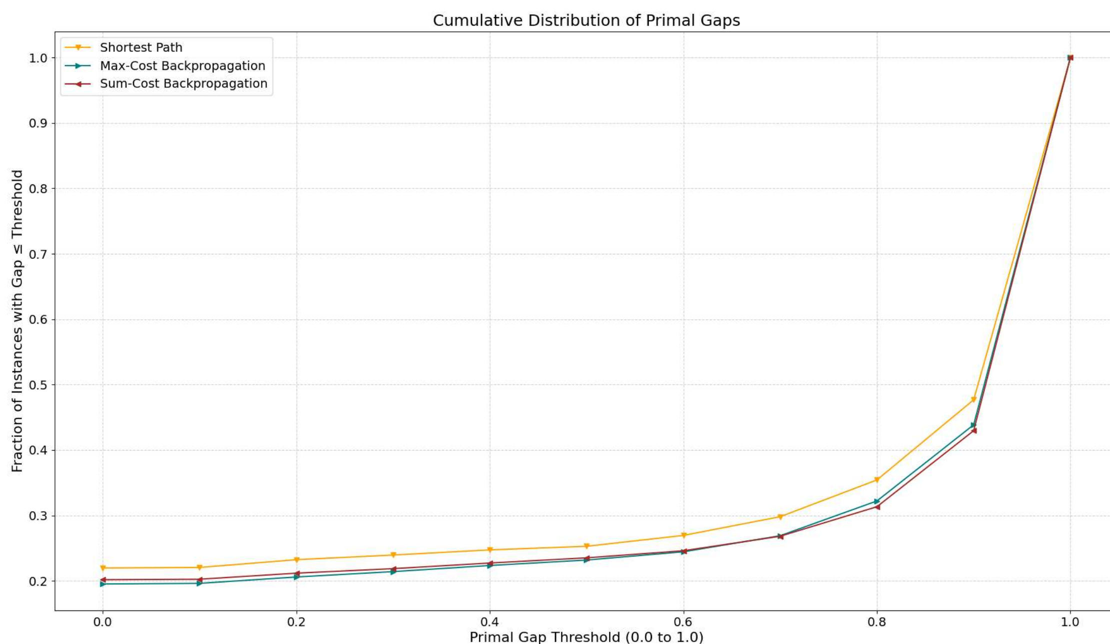

Figure 4.4: Cumulative distribution plot for different backpropagation strategies

the next action at random is highly dependent on the chosen random seed. Figure 4.6 shows the same plot for the Greedy approach.

In this case, the variability in the algorithm's output is more stable than in the previous one. While the Greedy strategy still exhibits instances with large discrepancies between minimum and maximum costs, both the frequency and the magnitude of these differences are slightly reduced. The overall trend emerging from this analysis is that the more informative the heuristic, the more stable its results become. This behavior is particularly evident in Figures 4.7 and 4.8.

The improvement is especially noticeable in the plot for the Shortest Path. Most points lie close to the red line, which represents the ideal case where the minimum and maximum plan costs are equal. This indicates a high degree of consistency across different random seeds. The Shortest Path approach stands out as the most informative among the heuristics presented, combining a pruning mechanism with an action-level estimate of the distance to the goal state.
In conclusion, the Shortest Path proves to be the most effective in all aspects: it solves the largest number of instances, yields the best performance in cumulative distribution plots, and demonstrates the highest stability with respect to random seed variation.

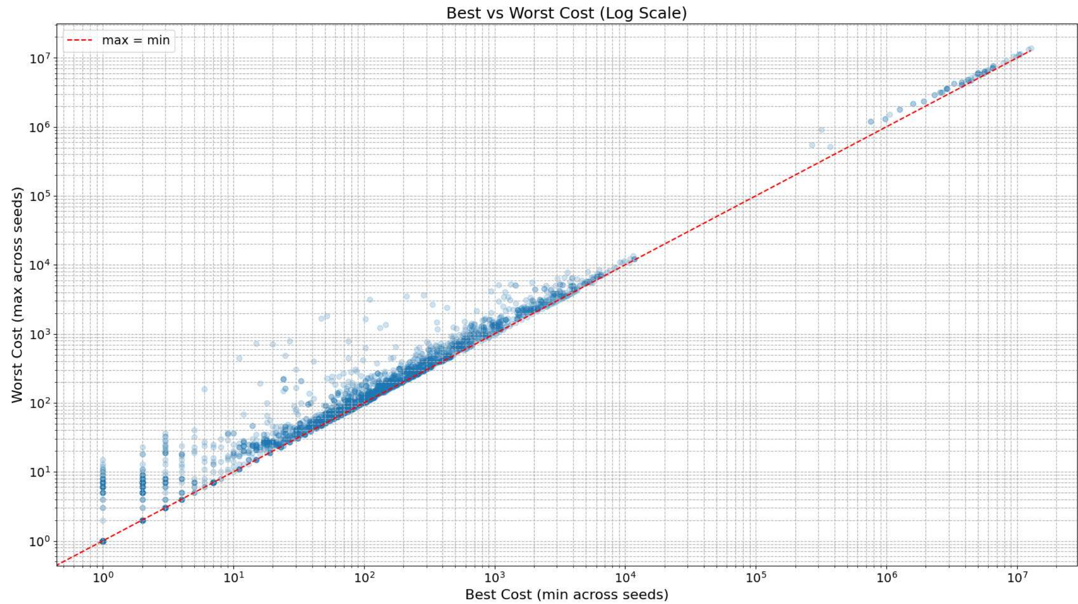

Figure 4.5: Effect of random seed on the outcome variability of the Random approach.

4.5 Iterative Refinement via Shortest Path Reapplication

In the context of Mixed Integer Programming (MIP), there is a technique known as *local branching* [3], which involves first finding a feasible solution using a fast constructive heuristic (or, in some applications, a metaheuristic), and then refining this solution by solving, to optimality, a subproblem of the original MIP using a solver as a "black box". This approach could also be applied to planning. However, at the time of writing, the most well-known optimal planner, Fast Downward, does not provide an API that would allow it to be used as a black-box component. To refine the solution found by the Shortest Path, the proposed idea is to reapply the algorithm to a subproblem and then replace the corresponding segment of the original solution with the new one. The hope is that solving a smaller, focused subproblem may yield a higher-quality solution than addressing the full problem in its entirety. Unlike the experiments in the previous sections, the refinement experiments were conducted using a single random seed. As shown in Section 4.4, the Shortest Path heuristic exhibits stable performance with respect to random seed variation. Therefore, repeating each run with multiple seeds is unnecessary in this context. Given an initial plan, the initial and goal states for the subproblem are derived from intermediate states along the original plan. In the experiment, these subproblem boundaries were

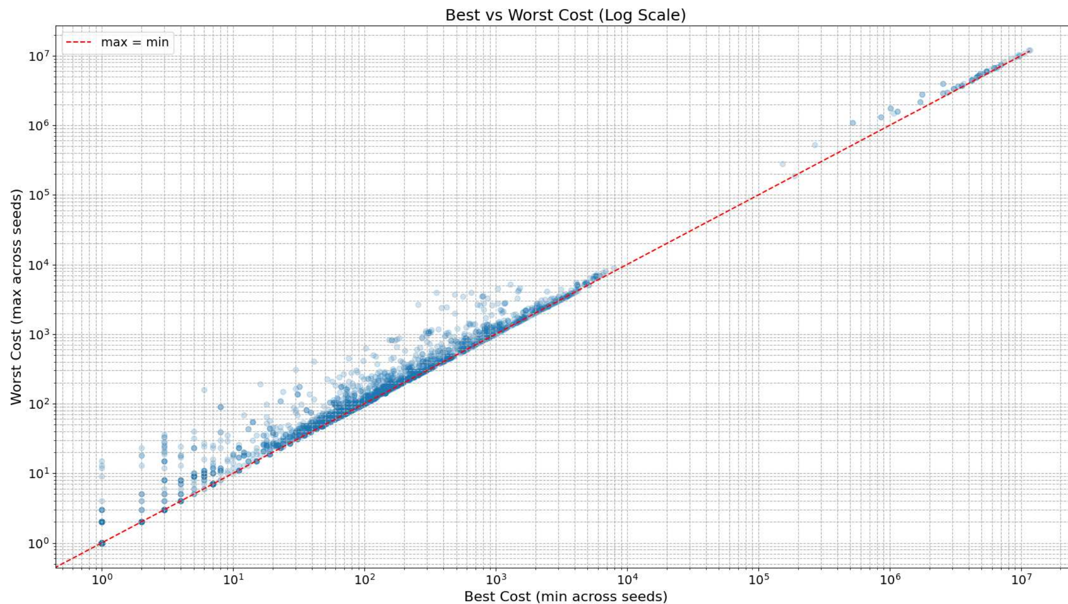

Figure 4.6: Effect of random seed on the outcome variability of the Greedy approach.

positioned at different segments of the original solution. For each instance, the Shortest Path algorithm was reapplied to three different segments of the original plan: the first 30% of actions, the central 40%, and the final 30%. The original segments of the solution were replaced with the newly generated ones, and the total plan cost was then recomputed. If the new cost is better than the initial one, the new solution is returned; otherwise, the initial solution is retained as the final plan. Figure 4.9 compares the cumulative distribution of the original Shortest Path strategy with the three variants where the algorithm was reapplied to different segments of the plan.

Unfortunately, the four curves overlap almost entirely, indicating that this strategy yields only minimal refinement over the initial solution. While the cost of the refined plans may occasionally differ from the original, the improvements are generally negligible. This suggests that reapplying the heuristic to subproblems does not significantly enhance solution quality and may not be an effective refinement approach in practice.

4.6 Iterative Refinement via Uniform Cost Search

In the previous section, an analysis was conducted on refining the solution provided by the Shortest Path by reapplying the same method to different segments of the original plan. Now, a different approach is explored by using *Uniform Cost Search (UCS)* [4] as

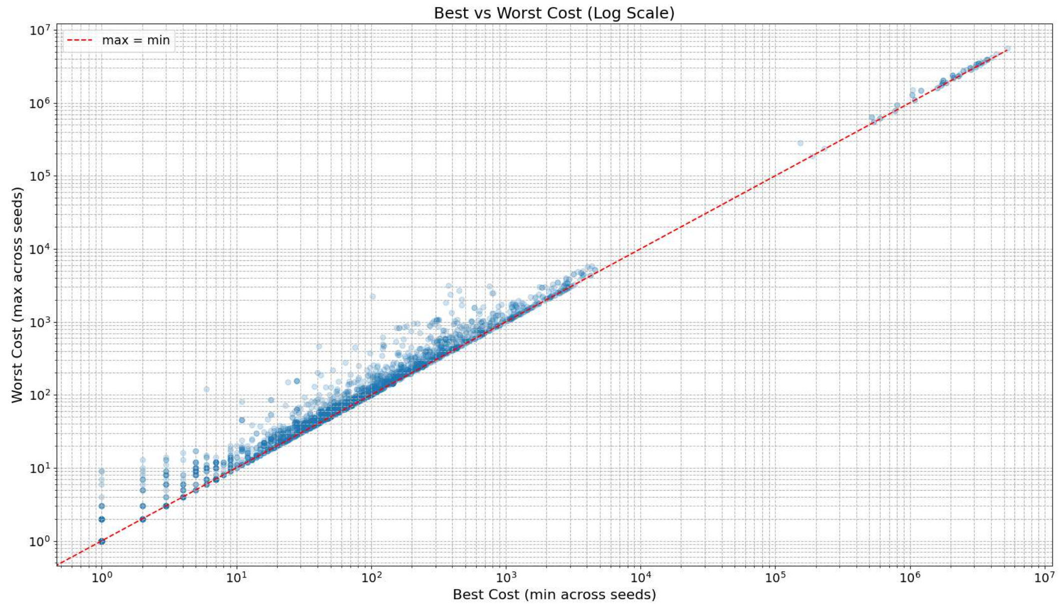

Figure 4.7: Effect of random seed on the outcome variability of the Greedy + Pruning approach.

the refinement algorithm. UCS is an exact algorithm that always finds the optimal path from a starting node to a goal node in a graph. It maintains a priority queue to track the frontier of nodes that have not yet been expanded. A node is considered expanded once all of its neighbors have been discovered. At each iteration, the node with the lowest cost—defined as the actual cost incurred to reach it from the starting node—is extracted from the queue. By always expanding the node with the minimum cost, UCS guarantees that the solution returned is optimal. Algorithm 9 presents the pseudocode for this strategy.

The experiments were conducted in a similar manner to those of the previous section. In the first run, a high-granularity approach was adopted: the solution provided by the Shortest Path was refined by reapplying UCS to consecutive portions of the plan, each covering 10% of its length. Specifically, UCS was applied to the intervals [0,0.1], [0.1,0.2], [0.2,0.3], and so on, up to the end of the plan. In this way, each run replaced 10% of the original solution with the UCS refinement. Figure 4.10 shows the cumulative distributions for the first five intervals, corresponding to the initial portions of the plan, while Figure 4.11 presents the results for the last five intervals, corresponding to the final portions.

In all cases, attempting to refine the solution does not appear to be a profitable approach: the curves almost completely overlap, and where differences exist, the gaps

Algorithm 9 Uniform Cost Search

Output:

```
1:  procedure UCS(planningTask)
2:      stateCosts ← {+∞, . . . , +∞}                         ▷ Initialize states' cost to +∞
3:      pq ← {}                                                ▷ Empty priority queue
4:      visited ← {}                                           ▷ Empty visited set
5:      stateCosts[planningTask.initialState] ← 0
6:      PUSH(pq, planningTask.initialState, 0)         ▷ Push the initial state with cost 0
7:      while !EMPTY(pq) do
8:          state ← POP(pq)                            ▷ Get the state having minimum cost
9:          if planningTask.goalState ⊆ state then
10:             return EXTRACTPLAN(state)
11:         end if
12:         if state ∈ visited then
13:             continue                                   ▷ Skip already expanded states
14:         end if
15:         visited ← visited ∪ {state}
16:         successors ← {a ∈ A | pre(a) ⊆ state}
17:         for all action ∈ successors do
18:             nextState ← APPLY(state, action)
19:             newCost ← action.cost + stateCosts[state]
20:             if newState ∉ visited and newCost < stateCosts[newState] then
21:                 stateCosts[newState] ← newCost
22:                 if HAS(pq, newState) then
23:                     CHANGE(pq, newState, newCost)
24:                 else
25:                     PUSH(pq, newState, newCost)
26:                 end if
27:             end if
28:         end for
29:     end while
30: end procedure
```

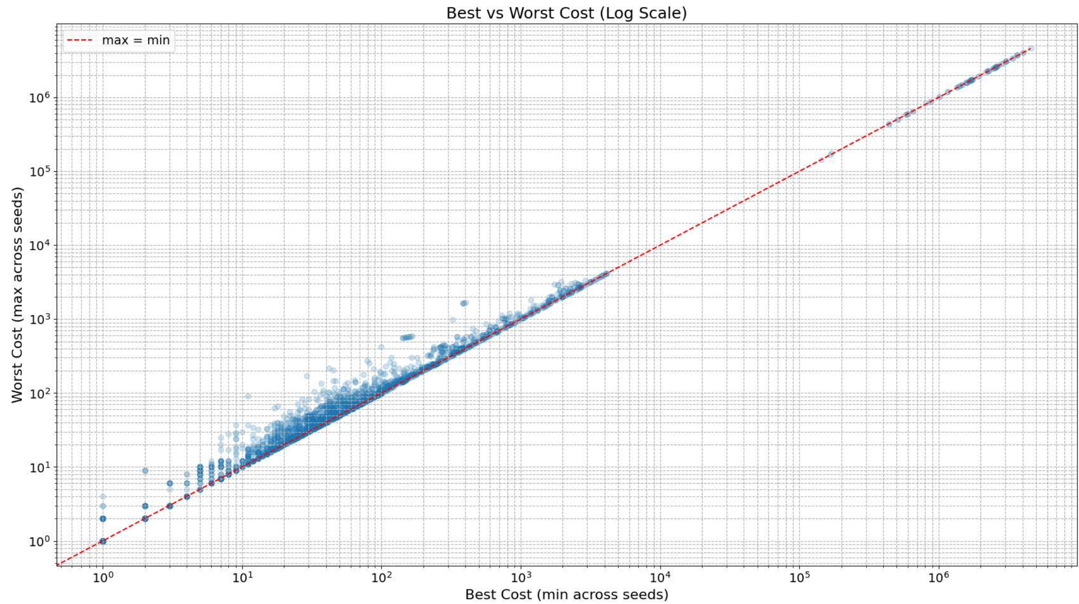

Figure 4.8: Effect of random seed on the outcome variability of the Shortest Path approach.

are minimal. To better understand whether this behavior is due to the ineffectiveness of UCS itself or to the choice of interval size, another experiment was conducted. This time, larger segments of the plan were considered: the first 30%, the middle 40%, and the last 30%. Figure 4.12 shows the results of this experiment.

Refining the last segment of the initial solution appears to be the most effective approach. This is likely due to the way the subproblem is derived from the original one. Taking, for instance, 30% from a central portion of the solution may not be a good idea: in order for the final solution to remain consistent, the final state of the subproblem must include all the preconditions required by the subsequent actions that are not being modified. As a result, the final state carries more constraints than the initial problem provided, leaving USC with little flexibility. By contrast, focusing on the last 30% of the solution avoids this issue, because once the final state is reached, there are no further actions that need to be satisfied. In any case, attempting to refine the initial solution obtained via the Shortest Path does not seem particularly useful. Even when focusing on refining the last 30% of the initial solution, the improvement over the Shortest Path is minimal. Using a larger window may be even less efficient, as the computation time for UCS increases exponentially with the number of nodes in the graph.

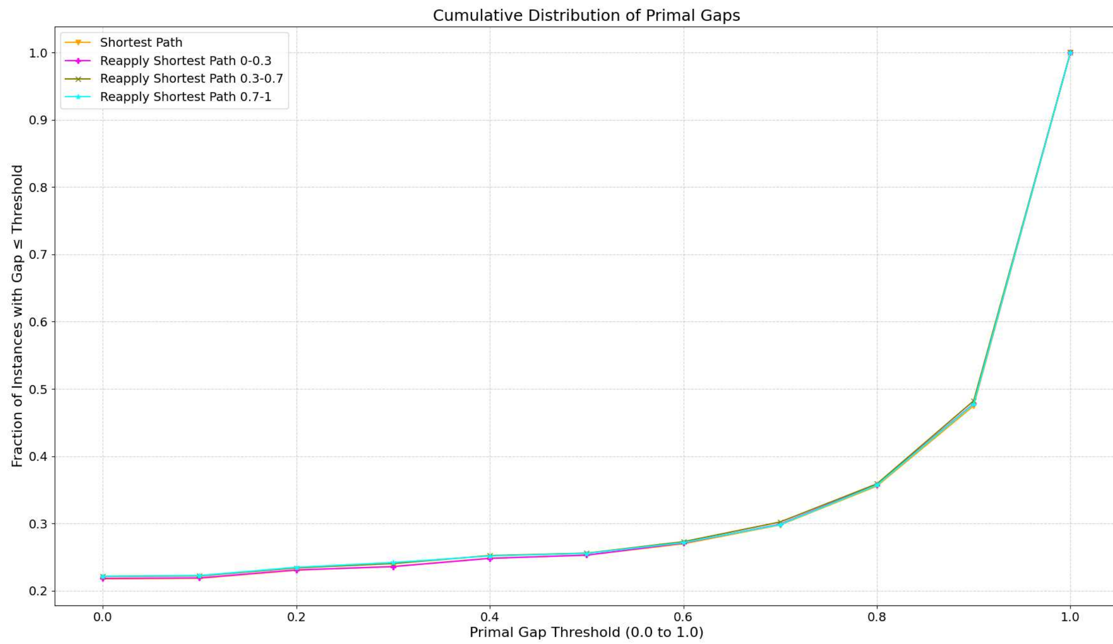

Figure 4.9: Comparison of cumulative distributions for Shortest Path reapplied to different plan segments.

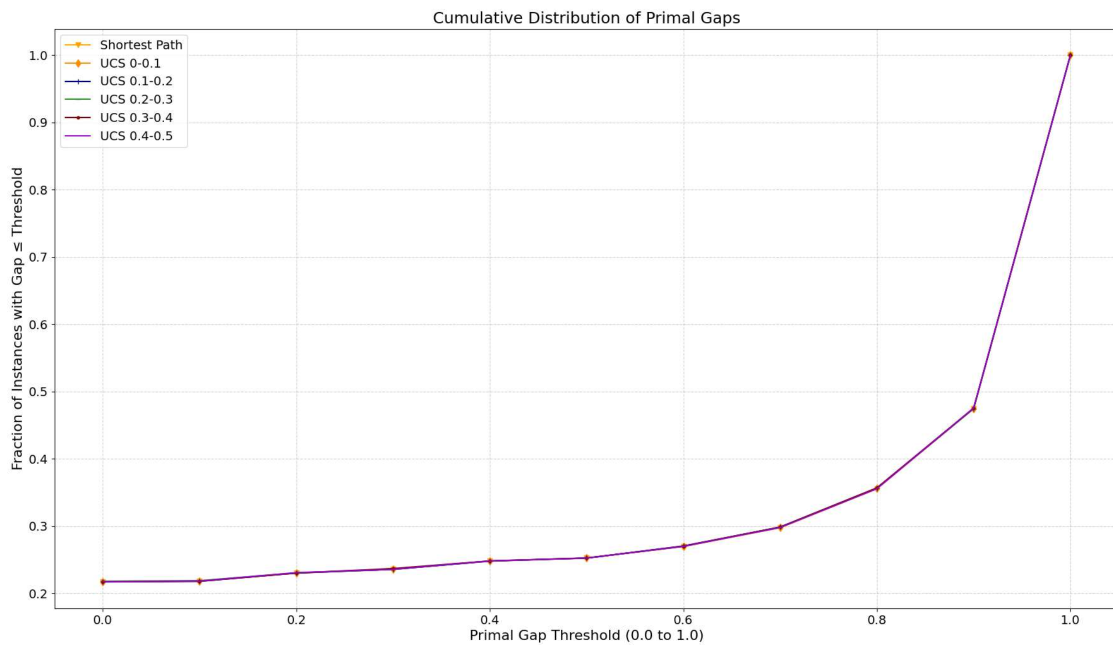

Figure 4.10: Cumulative distributions for UCS refinement applied to different 10% segments of the initial solution (first five intervals)

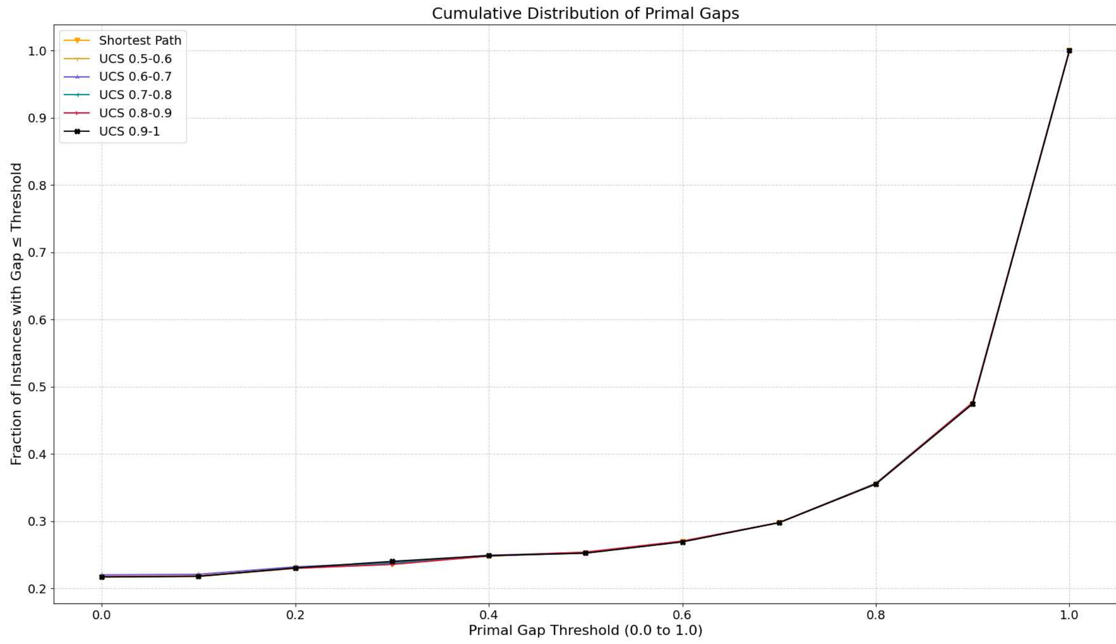

Figure 4.11: Cumulative distributions for UCS refinement applied to different 10% segments of the initial solution (last five intervals)

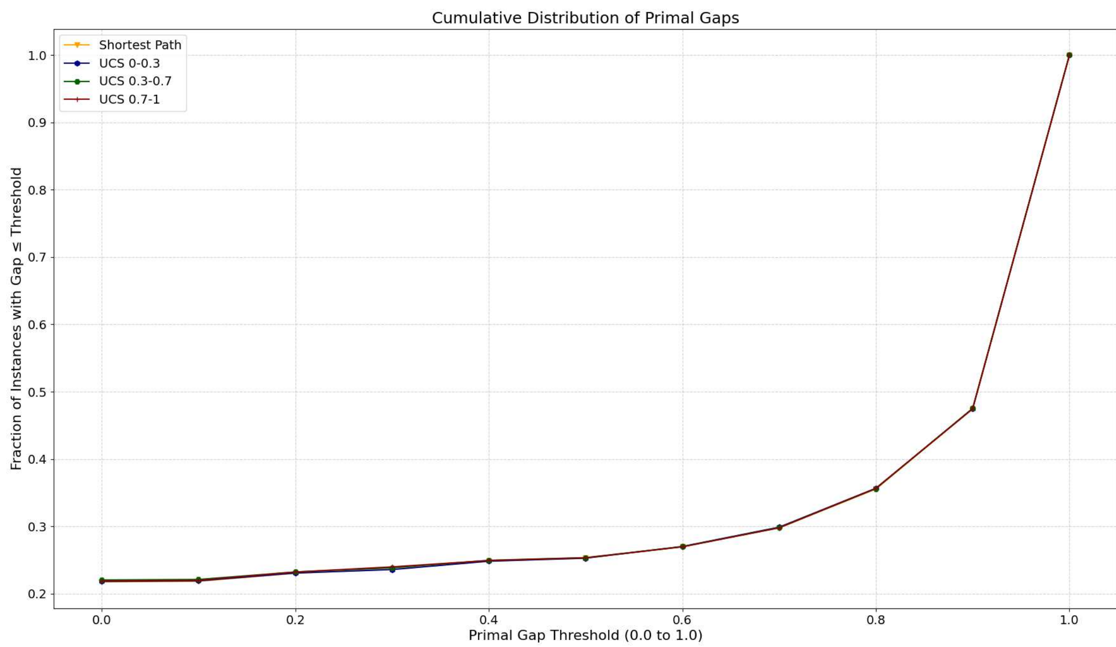

Figure 4.12: Cumulative distributions for UCS refinement applied to larger plan segments: first 30%, middle 40%, and last 30% of the initial solution

Chapter 5

Conclusions

This thesis investigated heuristic approaches for solving delete-free planning problems, a simplified yet challenging fragment of classical planning. By removing negative effects, delete-free planning yields a monotonic state space, which simplifies reasoning about reachability while preserving much of the inherent difficulty of plan generation.

The study first revisited several well-known heuristics and implemented them within a common experimental framework. Building on these foundations, a novel heuristic was proposed, designed to exploit the structural properties of delete-free tasks more effectively. The performance of all algorithms was systematically evaluated across a suite of benchmark instances, with a focus on both computational efficiency and the quality of the resulting plans.

The results highlight that the proposed heuristic achieves superior performance compared to established approaches, confirming the potential of structural exploitation in delete-free domains. The analysis of randomization further revealed that the stability of solution quality strongly depends on the informativeness of the heuristic: more informative heuristics consistently led to more stable outcomes across different random seeds, while less informative ones exhibited greater variability. Finally, refinement strategies, whether based on solving subproblems with Uniform Cost Search or on reapplying the heuristic itself to subproblems, did not yield substantial improvements. In both cases, the refined plans largely overlapped in cost with the originals, indicating that such post-processing techniques are not particularly effective in this setting.

At the same time, this work opens several avenues for further investigation. A natural next step would be to explore local branching using an external solver as a "black box", which could allow richer refinements of partial solutions than the UCS-based approach adopted here. Similarly, the pruning-enhanced variant of the $h^{\max}$ heuristic developed in this thesis could be integrated into A* search, enabling an optimal procedure to benefit

from early elimination of unhelpful actions. These directions suggest practical ways to extend the contributions of this work, while reinforcing the broader insight that delete-free planning offers a valuable framework for designing and evaluating new heuristic strategies.

Bibliography

[1] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001.

[2] T. Berthold, "Measuring the impact of primal heuristics," *Operations Research Letters*, vol. 41, no. 6, pp. 611–614, 2013.

[3] M. Fischetti and A. Lodi, "Local branching," *Mathematical programming*, vol. 98, no. 1, pp. 23–47, 2003.

[4] A. Felner, "Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 2, 2011, pp. 47–51.

[5] M. Helmert and G. Röger, *A beginner's introduction to heuristic search planning*, `https://ai.dmi.unibas.ch/misc/tutorial_aaai2015/`, 2015.