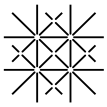


University of Basel

Optimistic Best-First Search with Goal-Preferred Actions in Fast Downward

Bachelor's thesis

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

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

Lisette Maureira
lisette.maureiradominguez@stud.unibas.ch
2020-709-937

10.10.2025

Abstract

Planning in Artificial Intelligence is the search for a sequence of actions applicable on the initial state to reach a given goal. The reliance on delete-relaxation graphs that ignore delete-effects of actions for the cost estimation for nodes in this search could potentially threaten the preservation of goal components achieved along the search. One such heuristic is the FF-heuristic. A novel approach to guide the search towards actions that don't delete goal components can be found in the concept of goal-preferred actions defined by Vidal [10]. Furthermore, the consideration of actions that are not deemed helpful by the FF-heuristic and yet are applicable, could avoid the labeling of a solveable problem as unsolveable, thus preserving completeness. We have implemented an Optimistic Best-first search algorithm with goal-preferred actions and rescue actions acting complementary to the helpful actions in an attempt to preserve completeness. We have found that a solution is reached in far fewer expansions with our implementation for some problems in comparison to Optimistic Best-First Searches utilizing the FF-heuristic without our added notions.

Table of Contents

1 Introduction

In Chapter 2, the background knowledge necessary for the understanding of the paper will be explained. In the following Chapter 3 and 4, this will be continued with the introduction of a running example for the concept of delete-relaxation and concepts introduced by Vidal. Then, in Chapter 5 the implementation in Fast Downward [4] will be discussed, of whose configurations were utilized for the experiments of which results are analyzed in the subsequent Chapter 6. We conclude the paper in Chapter 7, summarizing and formulating open questions.

2
Background

In Artificial Intelligence, planning is to determine a sequence of actions that will reach a predetermined goal. Planning formalisms are models that provide the language to formalize the structure in which planning takes place.

2.1 STRIPS

STRIPS is a planning formalism developed by Nils Nilsson and Richard Fikes [2].

Definition A STRIPS planning task is defined as a 4-tuple:

$$\Pi = \langle V, I, G, A \rangle \tag{2.1}$$

Where V is the finite set of state variables, $I \subseteq V$ the initial state, $G \subseteq V$ the goal set containing goal state variables and A the finite set of actions.

Definition In STRIPS, an action $a \in A$ is further defined with:

- $pre(a) \subseteq V$, the preconditions of a
- $add(a) \subseteq V$, the add effects of a
- $del(a) \subseteq V$, the delete effects of a
- $cost(a) \in \mathbb{N}_0$, the cost of a

In the original definition of STRIPS by Nilsson and Fikes [2], action costs are not involved, and thus neither by Vidal in his paper [10]. But for the algorithm implementation in Fast Downward [4] non-uniform action costs will be assumed and therefore, the definition above for actions is used.

An action a is applicable on a state s iff all of its preconditions are satisfied in s, in other words, if $pre(a) \subseteq s$. Applying the action a would then deliver the new state $s' = s \uparrow \langle a \rangle = (s \setminus del(a)) \cup add(a)$.

Definition A given STRIPS planning task induces the state space:

$$S(\Pi) = \langle S, A, cost, T, s_I, S_G \rangle \tag{2.2}$$

Where A and cost remain as in Π and:

- S is the set of states $\mathcal{P}(V)$[1]
- T is the set of transitions, where a given transition $t = s \xrightarrow{a} s'$ with $s, s' \in S$ and $a \in A$ is in T iff a is applicable on s and $s' = s \uparrow \langle a \rangle$
- $s_I = I$ is the initial state
- S_G is the set of goal states where $s \in S_G$ iff $G \subseteq s$

Paths are a sequence of actions. A path $P = \langle a_1, a_2, ..., a_n \rangle$ is valid for a state s if a_1 is applicable in state s and leads to a state s_1, and if actions $a_2, ..., a_n$ are applicable consecutively and lead to the states $s_{2,...,n}$ respectively after each application. If such a path exists for a given s and s_n, s_n is reachable from s. Furthermore, if $G \subseteq s_n$, P is a plan, and a solution for s.

Definition An optimal solution for the state s has minimal cost when compared to other solutions in the set of all solution plans.

Let $P_1 = \langle a_1, a_2, ..., a_n \rangle$ and $P_2 = \langle b_1, b_2, ..., b_n \rangle$ be paths. Their concatenation $P_1 \oplus P_2$ delivers the path $P' = \langle a_1, a_2, ..., a_n, b_1, b_2, ..., b_n \rangle$. Accordingly, for a given action a_{n+1}, the concatenation $P_1 \oplus \langle a_{n+1} \rangle$ delivers the path $P'_1 = \langle a_1, a_2, ..., a_n, a_{n+1} \rangle$.

2.2 SAS$^+$

A SAS$^+$ planning task is a 5-tuple defined as:

$$\Pi = \langle V, dom, I, G, A \rangle \tag{2.3}$$

Where V is the finite set of variables, dom is a function $dom(v)$ mapping each variable $v \in V$ to a non-empty finite domain, I the initial state, G the goal set and A the finite set of actions. A state s according to the SAS^+ planning formalism, is a total assignment of the variables $v \in V$, which means all variables are assigned a value. The initial state I is one such state. The goal is a partial assignment, defining the assignment of some, or all variables.

Definition In SAS$^+$, an action $a \in A$ is further defined with:

- $pre(a)$, the preconditions of a

[1] The powerset of V

- $\mathit{eff}(a)$, the effects of a
- $cost(a) \in \mathbb{N}_0$, the cost of a

Where the effects and preconditions constitute of partial assignments of the variables $\in V$. SAS$^+$ induces a state space as STRIPS, yet its definition will be omitted for this paper.

2.3 Heuristic

Definition Let ζ be a state space with states S. A heuristic function or heuristic for ζ is a function:

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\} \tag{2.4}$$

Which maps each state to a non-negative number, or ∞. Heuristic functions provide estimates of the cost to reach a goal from a given state.

2.4 Perfect heuristic

Definition The perfect heuristic h* for a given state space ζ, maps each state $s \in S$:

- to the cost of an optimal solution for s, or
- to ∞ if no solution exists

2.5 Properties of Heuristics

A heuristic h for the state space ζ is called:

- safe if $\mathrm{h}^*(s) = \infty$ for all $s \in S$ with $\mathrm{h}(s) = \infty$
- goal-aware if $\mathrm{h}(s) = 0$ for all goal states $s \in S_g$
- admissible if $\mathrm{h}(s) \leq \mathrm{h}^*(s)$ for all states $s \in S$
- consistent if $\mathrm{h}(s) \leq cost(a) + \mathrm{h}(s')$ for all transitions $s \xrightarrow{a} s'$

Admissible heuristics are often referred to as optimistic, while inadmissible heuristics are referred to as pessimistic.

2.6 Best-First Search

Search algorithms explore paths in a space to find a goal from an initial state, by expanding nodes, applying actions to acquire successor nodes. Graph searches keep track of visited nodes with a closed list, and potential candidates among its successors for expansion in an open list as opposed to tree searches, that only keep track of the latter. Best-First Search can be implemented as a graph or tree search, evaluating search nodes with an evaluation function f, and always expands a node N with minimal $f(N)$ value. For this paper, we will assume a graph search implementation.

2.6.1 Evaluation Functions

Heuristic search algorithms use heuristic functions to partially, or fully, determine node expansion order.

- In the case of Greedy Best-First Search, the evaluation function $f(N) = h(N)$.
- For A* [3], $f(N) = h(N) + g(N)$, where $g(N)$ refers to path costs.
- Weighted A* [7] possesses a parameter $w \in \mathbb{R}_0^+$ in the evaluation function $f(N) = w * h(N) + g(N)$ that affects the influence of the $h(N)$ on f, and thus calibrates the resemblance to greedy best-first search and A*. This means, the higher the w value, the more the search regards $h(N)$ much like in Greedy Best-First Search.

2.6.2 Properties

Optimality If a search algorithm always delivers the optimal plan if a plan exists, then the search algorithm is defined to be optimal. Greedy Best-First Search is not optimal. A* is optimal when utilized with an admissible heuristic.

Completeness A search algorithm is complete if it is guaranteed to find a solution if one exists. A Best-First search is complete if h is safe.

2.7 Delete-relaxation

In delete-relaxation, delete effects of a given action or planning task are disregarded.

Definition The delete-relaxation a^+ of a STRIPS action a is the action:

- $pre(a^+) = pre(a)$
- $add(a^+) = add(a)$
- $del(a^+) = \emptyset$
- $cost(a^+) = cost(a)$

And thus, the relaxed planning task Π^+ of a STRIPS planning task $\Pi = \langle V, I, G, A \rangle$ is the task $\langle V, I, G, \{a^+ \mid a \in A\} \rangle$. Furthermore, plans of relaxed planning tasks are relaxed plans of Π.

2.8 Relaxed Planning graphs

Definition Relaxed planning graphs represent how and which state variables are reached in Π^+. It consists of variable layers V^i and action layers A^i. Variable vertex $v_0 \in V^0$ for all $v \in I$
goal vertex g if $v^n \in V^n$ for all $v \in G$, where n is the last layer.
Directed layers:

- Precondition edges: from v^i to a^{i+1} if $v \in pre(a)$
- Effect edges: from a^i to v^i if $v \in add(a)$
- No-op edges: from v^i to v^{i+1}
- Goal edges: from v^n to g if $v \in G$

3
Running example

In this chapter we introduce a small example of a planning task that will be referred to in the next chapters to illustrate central concepts of the paper.

3.1 Gripper

The original gripper [6] planning task's setting is that of a robot with two arms that is able to move a ball in each arm at most between two rooms that contain an arbitrary amount of balls. In our restricted running example, there is one ball, a robot with one arm, and two rooms A and B, the initial state illustrated in Figure 3.1. Our goal to achieve is for the ball's location to be room B, and additionally for the robot to also be in room B. These changes to the domain were done to best illustrate concepts introduced in the following chapter, Chapter 4.

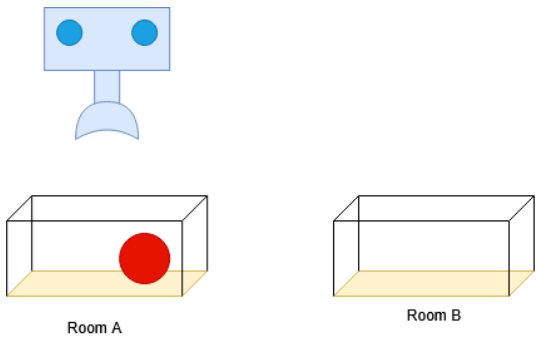


Figure 3.1: Gripper running example initial state.

3.2 Gripper in STRIPS

$V = \{at_{1A}, at_{1B}, at_{gA}, at_{gB}, in, free\}$
$I = \{at_{1A}, at_{gA}, free\}$
$G = \{at_{1B}, at_{gB}\}$
$A = \{move_{AB}, move_{BA}, pick_{A1}, pick_{B1}, drop_{A1}, drop_{B1}\}$

$pre(move_{AB}) = \{at_{gA}\}$
$add(move_{AB}) = \{at_{gB}\}$
$del(move_{AB}) = \{at_{gA}\}$
$cost(move_{AB}) = 2$

$pre(move_{BA}) = \{at_{gB}\}$
$add(move_{BA}) = \{at_{gA}\}$
$del(move_{BA}) = \{at_{gB}\}$
$cost(move_{BA}) = 2$

$pre(pick_{A1}) = \{at_{gA}, at_{1A}, free\}$
$add(pick_{A1}) = \{in\}$
$del(pick_{A1}) = \{at_{1A}, free\}$
$cost(pick_{A1}) = 3$

$pre(pick_{B1}) = \{at_{gB}, at_{1B}, free\}$
$add(pick_{B1}) = \{in\}$
$del(pick_{B1}) = \{at_{1B}, free\}$
$cost(pick_{B1}) = 3$

$pre(drop_{A1}) = \{in, at_{gA}\}$
$add(drop_{A1}) = \{at_{1A}, free\}$
$del(drop_{A1}) = \{in\}$
$cost(drop_{A1}) = 1$

$pre(drop_{B1}) = \{in, at_{gB}\}$
$add(drop_{B1}) = \{at_{1B}, free\}$
$del(drop_{B1}) = \{in\}$
$cost(drop_{B1}) = 1$

3.3 Gripper in SAS+

$V = \{robby, arm, ball\}$
$dom(robby) = \{at_A, at_B\}, dom(arm) = \{free, hold\}, dom(ball) = \{at_A, at_B, in\}$
$I = \{robby \mapsto at_A, ball \mapsto at_A, arm \mapsto free\}$
$G = \{ball \mapsto at_B, robby \mapsto at_B\}$
$A = \{move_{AB}, move_{BA}, pick_{A1}, pick_{B1}, drop_{A1}, drop_{B1}\}$

$pre(move_{AB}) = \{robby \mapsto at_A\}$
$eff(move_{AB}) = \{robby \mapsto at_B\}$
$cost(move_{AB}) = 2$

$pre(move_{BA}) = \{robby \mapsto at_B\}$
$\mathit{eff}(move_{BA}) = \{robby \mapsto at_A\}$
$cost(move_{BA}) = 2$

$pre(pick_{A1}) = \{robby \mapsto at_A, ball \mapsto at_{1A}, arm \mapsto free\}$
$\mathit{eff}(pick_{A1}) = \{ball \mapsto in, arm \mapsto hold\}$
$cost(pick_{A1}) = 3$

$pre(pick_{B1}) = \{robby \mapsto at_B, ball \mapsto at_{1B}, arm \mapsto free\}$
$\mathit{eff}(pick_{B1}) = \{ball \mapsto in, arm \mapsto hold\}$
$cost(pick_{B1}) = 3$

$pre(drop_{A1}) = \{ball \mapsto in, robby \mapsto at_A\}$
$\mathit{eff}(drop_{A1}) = \{ball \mapsto at_A, arm \mapsto free\}$
$cost(drop_{A1}) = 1$

$pre(drop_{B1}) = \{ball \mapsto in, robby \mapsto at_B\}$
$\mathit{eff}(drop_{B1}) = \{ball \mapsto at_B, arm \mapsto free\}$
$cost(drop_{B1}) = 1$

4

Delete-relaxation heuristics

Delete-relaxation heuristics utilize delete-relaxation graphs to acquire a sense on the reachability of states, and thus the goal. This because the omission of delete effects allows for more actions to be applied, rather than fewer. Once a relaxed plan is found, the estimation on possible path costs to reach the goal can be extracted in different ways, with a drive to avoid the downsides of ignoring the information that delete effects of actions provide and thus gain accuracy. Vidal [10] defined a new way in which the loss of that information through the use of delete-relaxation heuristics could be reincorporated, which will be introduced in this chapter.

4.1 h_{add} heuristic

h_{add} [1] is a pessimistic heuristic that assumes all precondition state variables are to be reached independently from each other. As a result, it fails to recognize positive synergies between subgoals and overestimates the actual costs. It is safe, goal-aware, but neither admissible nor consistent. Its lack of admissibility makes it unsuitable for optimal planning.

- Variable vertex costs:
  - Layer 0: 0
  - Otherwise: minimum of the costs of predecessor vertices
- Action/goal vertex costs:
  - Goal vertices: Sum of predecessor vertex costs
  - Action vertices a^i: $cost(a)$ added to sum of predecessor vertex costs
- Termination criterion:
  Terminate if $V^i = V^{i-1}$ and costs of all vertices equal V^i equal corresponding vertex costs in V^{i-1}
- Heuristic value:
  Value of goal vertex

We can look at the relaxed graph of our running example from the previous section to illustrate h_{add}.

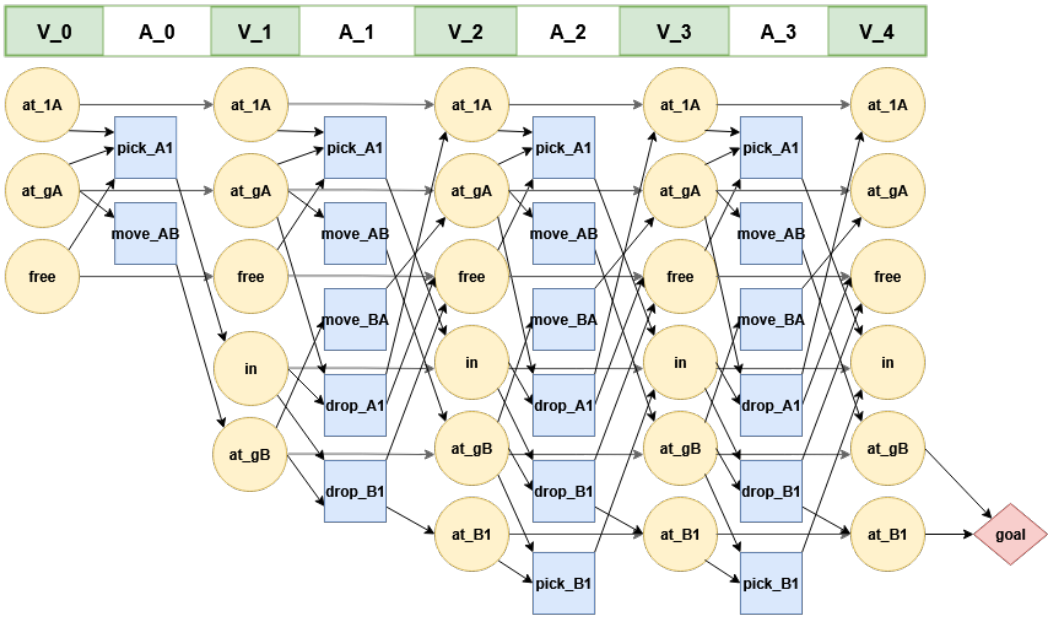


Figure 4.1: Relaxed graph for Gripper running example.

The heuristic value in h_{add} when looking at the costs is visible in Figure 4.2:

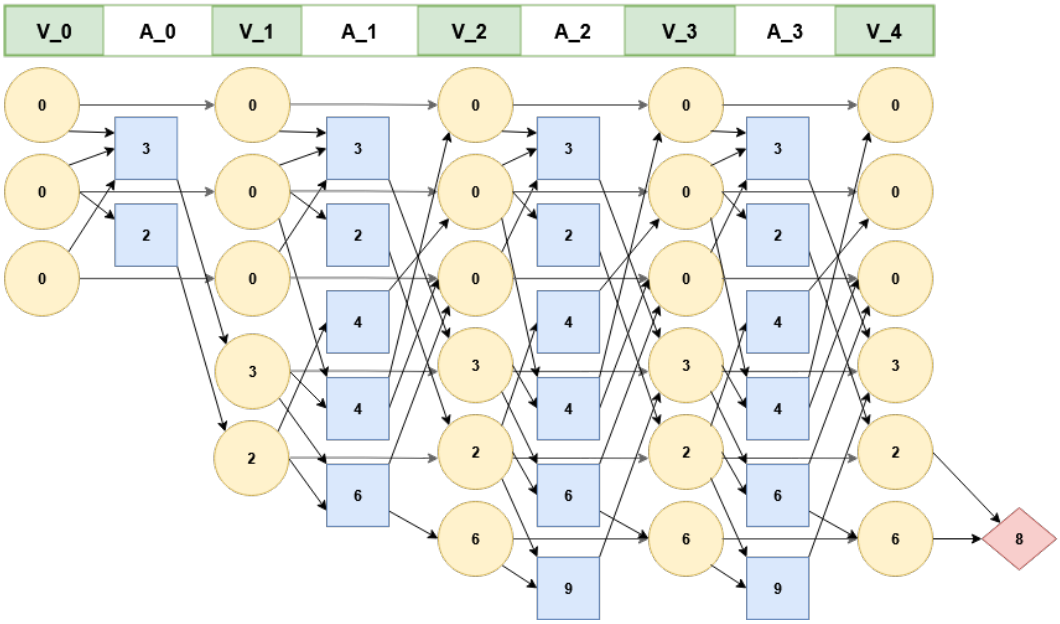


Figure 4.2: Calculation of h_{add} value for the Gripper running example.

4.2 FF-heuristic

The FF-heuristic [5] is identical to h_{add}, except with the following additional steps at the end:

- Marking rules:
  - Mark goal vertex
  - If marked action/goal vertex: mark all predecessors
  - If marked variable vertex v^i in layer $i \leq 1$: mark one predecessor with minimal h_{add} value. To tie-break, prefer variable vertices and otherwise choose arbitrarily
- Heuristic value:
  The actions that correspond to the marked action vertices build a relaxed plan, the cost of said plan is the heuristic value.

It is safe and goal-aware, but neither admissible nor consistent. It is guaranteed that it is at least as good as h_{add}.

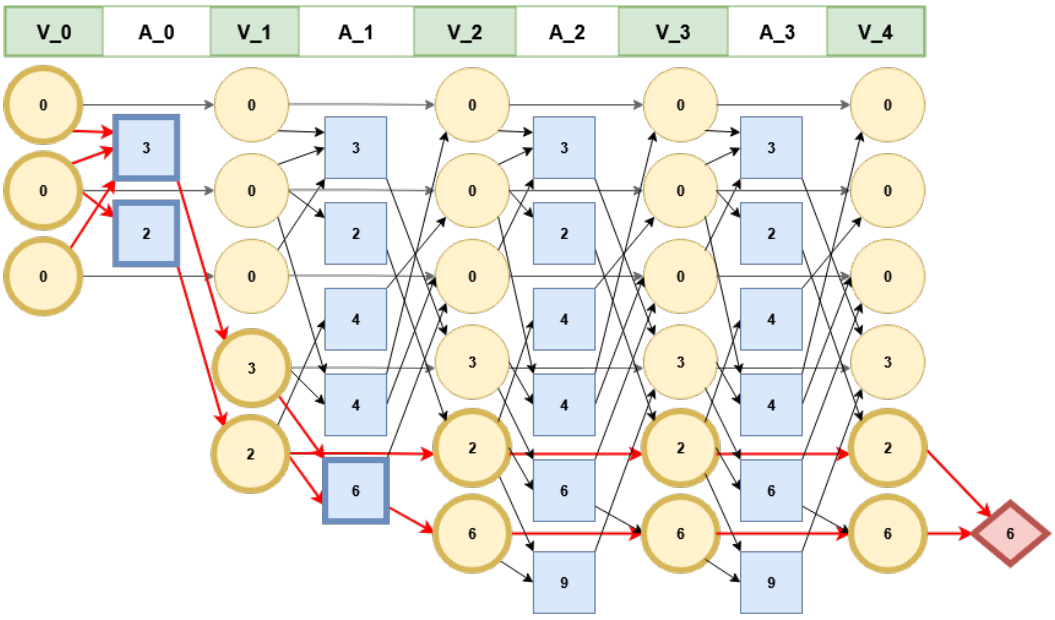


Figure 4.3: FF-heuristic marking rules for the Gripper running example.

Where h_{add} previously overestimated the heuristic value to be 8, the marking rules in the FF-heuristic allowed for the recognition that the cost of arriving to the state at_{gB} was not required to be counted twice, as that state variable can be achieved simultaneously as the robot is moved from room A to room B when carrying the ball. The latter goal state variable aids already in achieving the other goal state variable.
Let s be a state in the STRIPS planning task $\langle V, I, G, A \rangle$. Then, $h_{FF}(s) \leq h_{add}(s)$

4.3 FF Planner

Planners rely on search algorithms and heuristic functions to acquire a plan for a given problem comprised of an initial state, a goal state and actions. The FF planner [5] works in two phases. In the first phase, the FF-heuristic is utilized to prune applicable actions not deemed helpful in finding the goal state, namely by labeling actions helpful if they are present in the relaxed plan computed from a given state s as defined by the FF-heuristic and are applicable. The failure to consider all applicable actions caused by this pruning leads to the incompleteness of this first phase of the FF planner. The information gained by the helpful actions in that first phase is scrapped if a solution is not found, and the search is started anew in the second phase, that utilizes a complete best-first search algorithm. The FF planner is a satisficing planner, which means it is designed to find any solution that satisfies the goal condition, regardless of optimality. Satisficing planners forfeit optimatlity for the sake of finding any solution within time or resource constraints.

4.4 Goal-preferred actions

Goal-preferred actions are those that do not delete a state variable needed for the goal that was not in the initial state. In SAS^+, such delete effects can be defined as setting a goal variable a different value than the ones that are required for the goal. Goal-preferred actions in Fast Downward [4], that works under the $SAS+$ finite domain representation formalism, were implemented under this assumption.

For the explanation of the two algorithms for detection of goal-preferred actions, SAS^+ variable assignments will be referred to as variable value pairs. Such variable value pairs are found in the sets defining effects, preconditions, and in states. Our first algorithm, we'll define as the naive goal-preferred algorithm, the pseudocode shown in Algorithm1.
Our naive approach to implement goal-preferred actions in Fast Downward with an algorithm that would discard actions not eligible to be considered goal-preferred, the following checks could be made:
First, in line 3, we check that the variable and value pairs present in the effects of the action are not to be found in the initial state, as any that do affect the initial state ... . Next, in line 7 we check whether we are affecting a goal variable or not. If we are, we check in line 10 whether we are setting the variable to a value unlike that in the goal. In this case, we assume a delete-action has ocurred, and we flag the action with this effect as not goal-preferred.

Algorithm 1 Naive goal-preferred action algorithm

```
1: let goal_preferred = True
2: for all eff ∈ operator.effects do
3:     if eff ∈ init then
4:         continue
5:     endif
6:     for all goal ∈ goals do
7:         if eff.variable ≠ goal.variable then
8:             continue
9:         endif
10:        if eff.value ≠ goal.value then
11:            goal_preferred = False
12:            break
13:        endif
14:    endfor
15:    if goal_preferred = False then
16:        break
17:    endif
18: endfor
19: if goal_preferred = True then
20:    gp_ids.insert(operator.id)
21: endif
```

Algorithm 1 fails to recognize that delete-effects of a goal variable value pair couldn't have ocurred if before the application of the action the goal value for the goal variable was not set. A possible option to acquire information on the previous state is by taking preconditions into consideration, as we know from the definition of applicable actions that they are only applicable if the preconditions are satisfied. In the case of taking preconditions into consideration, the following checks are made:
In what was formerly our last check, before we discard an action as not being goal-preferred based on the fact that it sets a goal variable to a value different than that present in the goal, we check in line 11 whether the goal variable we are looking at is anywhere to be found in the preconditions. If it isn't, then we are conservative and assume that the goal value could've been possibly formerly set. Otherwise, if the goal variable and goal value pair was in the preconditions, then we can be certain that a delete-effect has indeed ocurred, and label the action as not goal-preferred after this check in line 15.

Algorithm 2 Goal-preferred action algorithm

```
1: let goal_preferred = True
2: for all eff ∈ operator.effects do
3:     if eff ∈ init then
4:         continue
5:     endif
6:     for all goal ∈ goals do
7:         if eff.variable ≠ goal.variable then
8:             continue
9:         endif
10:        if eff.value ≠ goal.value then
11:            if goal.variable ∉ preconditions then
12:                goal_preferred = False
13:                break
14:            endif
15:            if ⟨goal.variable, goal.value⟩ ∈ preconditions then
16:                goal_preferred = False
17:                break
18:            endif
19:        endif
20:    endfor
21:    if goal_preferred = False then
22:        break
23:    endif
24: endfor
25: if goal_preferred = True then
26:    gp_ids.insert(operator.id)
27: endif
```

In our running example, the goal-preferred actions would be $GA = \{move_{AB}, move_{BA}, drop_{A1}, drop_{B1}\}$ without consideration of preconditions, and $GA = \{move_{AB}, move_{BA}, drop_{A1}, drop_{B1}, pick_{A1}\}$ with consideration of preconditions. In the case of the action $pick_{A1}$, the detrimental strictness of not taking preconditions into consideration is illustrated; an action that would've aided in setting the goal variable value pair $ball \mapsto at_B$ by concatenating said action with the path $\langle move_{AB}, drop_{A1} \rangle$ was considered not goal-preferred. This is because, as delete-effects in $SAS+$ were defined previously, it set a variable present in the goal, namely $ball$, to a different value than that of the goal with its effect $\{ball \mapsto in\}$. However, in this case the precondition is able to inform that the action is only applicable in cases where the precondition is unlike our goal state, so to define it as not goal-preferred would be erroneous, as no deletion of a goal variable value pair is executed by the action. The following solution of using preconditions as aid in acquiring information on the previous state to infer delete-effects does not cover all cases. In our solution, effect conditions for singular effects for example are ignored.

4.5 Helpful Actions & Nodes

The actions of the relaxed plan delivered by the FF-heuristic that are applicable in a given state s are considered helpful. In the Figure 4.3 of the Gripper running example, in the

initial state $I = \{at_{1a}, at_{ga}, free\}$ of the actions in the relaxed graph actions $pick_{A1}$ and $move_{AB}$ in action layer A^0 are applicable, and thus considered helpful for that state. Nodes labeled helpful by Vidal's algorithm due to their successful achievement of a relaxed plan through the sole use of goal-preferred actions are labeled helpful, and inserted into the open list alongside their helpful actions.

4.6 Rescue Actions & Nodes

Rescue actions are actions that are not helpful; They are applicable on the state s but not in the relaxed plan delivered by the FF-heuristic. Their consideration allows for completeness lost in the comparable first phase of the FF planner to be regained, as the union of helpful actions and rescue actions equals to the set of all actions that can be applied to the current state s. In the event that a relaxed plan can be found through the sole use of goal-preferred actions, a helpful node is created, and simultaneously, a search node labeled rescue alongside the state's rescue actions is inserted into the open list. If such a relaxed plan is failed to be found, all actions output by the relaxed plan computed utilizing all actions is stored in a node labeled rescue. Helpful Nodes are always preferred over Rescue Nodes, without regard for their heuristic value.

4.7 Lookahead states, Nodes & Paths

A state s' reachable from state s by a valid lookahead path P, whose node is treated like a direct descendant from the S node in the search graph, are called lookahead states. This would lead to two different types of arcs in the search graph, created by either a sole action, or a path. In the case that a goal state is found, the solution would be comprised of paths and single actions respectively for the two different kinds of arcs.
Lookahead paths are acquired in polynomial time with the FF-heuristic, as it creates a relaxed planning graph for each encountered state.

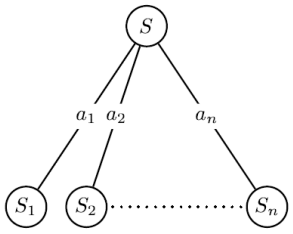


Figure 4.4: Node development.

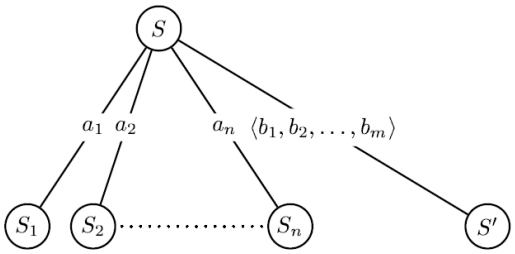


Figure 4.5: Lookahead node.

5
Implementation

The concepts 4.4 through 4.6 introduced in the previous Chapter as defined by Vidal [10] were implemented in the Fast Downward planning system [4]. Namely Goal-preferred Actions, Helpful Actions/Nodes and Rescue Actions/Nodes. For Goal-preferred actions two different configurations were implemented based on the Algorithms 1 and 2 defined in Section 4.4. As the Fast Downward planning system works with the $SAS+$ planning formalism, a translation of certain concepts from the STRIPS planning formalism assumed in Vidal's paper had to be undertaken. The structure of Fast Downward too required reworking of the algorithms that utilized these concepts.

5.1 Algorithm Explanation

Our implementation utilizes the FF-heuristic to acquire a relaxed plan and thus actions labeled helpful as opposed to rescue.

5.1.1 Goal-preferred Actions

The first relaxed plan built for the state is comprised of solely Goal-preferred Actions as defined in our Algorithms 1 and 2 for the respective two configurations. In Vidal's paper, at this point a Lookahead Node is computed as well, but our implementation omits this step. In the case that the creation of the relaxed plan using Goal-preferred Actions succeeds, that means that it deems the goal reachable through the disregard of delete-actions, we create a helpful node for the helpful actions for that state and create a Rescue Node for all other applicable actions, and store both nodes in the open list. If this relaxed plan fails, all applicable actions for the construction of a second relaxed plan, and both helpful actions and the rescue actions are stored in a rescue node.

5.1.2 Expansion Criteria

To decide on what successor node from the open list to expand next of the search, Helpful Nodes are preferred over Rescue Nodes. To tie-break, next the heuristic value is regarded. In Vidal's paper, the heuristic value is calculated as in WA* with W=3, but we used solely the

heuristic value provided by the FF-heuristic. Vidal too utilizes a third tie-breaker, namely the length of the relaxed plan computed by the FF-heuristic, that we did not implement.

5.2 Algorithm Adaptation

The following pseudocode, Algorithm 3, is as defined by Vidal [10]:

Algorithm 3 LOBFS() and compute_node() as per Vidal's paper [10]

```
1: let Π = ⟨A, I, G⟩
2: let GA = {a ∈ A|∀f ∈ Del(a), f ∉ (G \ I)}
3: let open = ∅
4: let close = ∅
5:
6: function LOBFS()
7: compute_node(I, ⟨⟩)
8: while open ≠ ∅ do
9:     let ⟨S, P, actions, h, flag⟩ = pop_best_node()
10:    for all a ∈ actions do
11:        compute_node(S ↑ ⟨a⟩, P ⊕ ⟨a⟩)
12:    endfor
13: endwhile
14: end
15:
16: function compute_node(S, P)
17: if S ∉ close then
18:    if G ⊆ S then
19:        output_and_exit(P)
20:    endif
21:    close ← close ∪ {S}
22:    let ⟨RP, H, R⟩ = compute_heuristic(S, GA)
23:    if RP ≠ fail then
24:        open ← open ∪ {⟨S, P, H, length(RP), helpful⟩, ⟨S, P, R, length(RP), rescue⟩}
25:        let ⟨S′, P′⟩ = lookahead(S, RP)
26:        if length(P′) ≥ 2 then
27:            compute_node(S′, P ⊕ P′)
28:        endif
29:    else
30:        let ⟨RP, H, R⟩ = compute_heuristic(S, A)
31:        if RP ≠ fail then
32:            open ← open ∪ {⟨S, P, length(RP), H ∪ R, rescue⟩}
33:        endif
34:    endif
35: endif
36: end
```

The reworking of the algorithm was prompted by two reasons. The first reason are the lines of code 17-20 seen in the pseudocode of Algorithm 3 that conflicted with the structure of Fast Downward, where those steps are usually found in the search algorithm code. The second reason is that since we are assuming non-uniform action costs, we choose to utilize the heuristic value calculated by the FF-heuristic as opposed to the length of the relaxed plan

in our reworked algorithm. These changes are reflected then in the following pseudocode, Algorithm 4:

Algorithm 4 OBFS with goal-preferred actions() and compute_node()

```
1:  let Π = ⟨A, I, G⟩
2:  let GA = {a ∈ A|∀f ∈ Del(a), f ∉ (G \ I)}
3:  let open = ∅
4:  let close = ∅
5:
6:  function OBFS()
7:  compute_node(I, ⟨⟩)
8:  while open ≠ ∅ do
9:      let ⟨S, P, actions, h, flag⟩ = pop_best_node()
10:     if S ∉ close then
11:         if G ⊆ S then
12:             output_and_exit(P)
13:         endif
14:         close ← close ∪ {S}
15:         for all a ∈ actions do
16:             compute_node(S ↑ ⟨a⟩, P ⊕ ⟨a⟩)
17:         endfor
18:     endif
19: endwhile
20: end
21:
22: function compute_node(S, P)
23: let ⟨RP, H, R⟩ = compute_heuristic(S, GA)
24: if RP ≠ fail then
25:     open ← open ∪ {⟨S, P, H, hff_val, helpful⟩, ⟨S, P, R, hff_val, rescue⟩}
26: else
27:     let ⟨RP, H, R⟩ = compute_heuristic(S, A)
28:     if RP ≠ fail then
29:         open ← open ∪ {⟨S, P, H ∪ R, hff_val, rescue⟩}
30:     endif
31: endif
32: end
```

6
Experimental evaluation

We devised experiments for our implementation that allows conclusions to be drawn on how well it fares when compared with implementations that utilize related concepts, heuristics and search algorithms.

6.1 Setup

As the algorithm was implemented in Fast Downward [4], the python package Downward Lab by Seipp et al. [9] was utilized to run experiments on the SciCORE scientific computing centre at the University of Basel. The experiments were run on 64 Core AMD EPYC 7742 2.25GHz processors. The time limit for each run was set to 30 minutes, and the memory limit to 3.5 GiB. The implementation was tested on a collection of IPC benchmark instances[2]. For analysis and comparison, the Eager Greedy Best-First Search with the FF-heuristic (with and without preferred actions), and the first component of the Lama planning system introduced by Richter and Westphal [8] were chosen. The planning tasks were ran too for our 2 configurations, namely the naive goal-preferred action definition, and the goal-preferred action definition considering preconditions mentioned in Section 4.5, totaling to 5 configurations. In tables and graphs, they will be referred to as Eager Greedy FF, Greedy FF pref, LAMA first, Naive Goalpref and Goalpref.

6.2 Results

As our implementation is in the realm of suboptimal planning, what we are looking for is to arrive to a solution fast, without running out of memory, and less so to achieve low costs for plans. However, if said costs were to be comparable to other similar algorithms, this too could inform us on the behavior of the algorithm, so this metric will be considered as well. We also want to verify the completeness defined in the theoretical grounds of our implementation in Section 4.6 on Rescue Actions, by confirming that solveable problems are not marked unsolveable.

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

6.2.1 Metrics

The metrics looked at in this section are defined as follows:

Coverage The number of problems for which a solution was found within given time and memory constraints.

Out of memory The number of times an out of memory error ocurred for the given configuration, comprised of translator out of memory errors and search out of memory errors.

Out of time The number of times the search passed the time limit and thus raised an out of time error.

Expansions The number of expanded states for solved problems.

Search Time The time taken by the search to solve each problem.

Cost The cost of the plan found by the search algorithm.

Operator count The number of operators of the given problem.

Goal-pref count The number of operators marked goal-preferred by the implementation out of the operators.

Goal-pref ratio The percentage of goal-preferred operators compared to all operators.

6.2.2 Coverage

In the first run of the experiments, it became apparent that both variations of the implementation were incomplete, as they erroneously flagged solveable problems as unsolveable for certain domains. Said domains have been omitted from the following results to avoid skewing the expansions and search time metric, as finding that a problem is unsolveable is considered as having come to a solution, and therefore said early aborted searches as having arrived fast to a solution with few expansions. The omission of said domains allow for those two metrics to remain comparable among all configurations. The reduction in domains left the problem instances to a total of 1715 across 65 domains. In Table 6.1, the coverage for the 1715 instances can be seen for the configurations, as well as the causes for non-coverage.

Table 6.1: Coverage vs. errors

Sum	Eager Greedy FF	Greedy FF pref	LAMA first	Goalpref	Naive Goalpref
Out of memory	87	70	109	760	829
Out of time	506	393	89	0	0
Other errors	5	6	3	9	4
Coverage	1117	1246	1514	946	882
Coverage %	65.13%	72.65%	88.28%	55.16%	51.4%

The implementations for goal-preferred operators differ in coverage, the reworked version allows for more solutions to problems to be found. This implies that our hypothesis on the possible detrimental aspects of the naive goal-preferred algorithm on our running example in Section 4.6 was likely indicative of a larger trend.
The frequency of the search out of memory error can be attributed to the fact that we store helpful actions, but also rescue actions explicitly for each state. The latter could potentially be acquired dynamically when regarding the current applicable actions without storing them. The frequency of this error also explains the lack of out of time errors for our implementations, as the search ran out of memory before it ran out of time.

6.2.3 Expansions

The following Table 6.2 shows that the reduced number of expansions for both our implementations compared to Eager Greedy FF, Greedy FF with preferred operators and Lama first is consistent across all problems, as the use of the geometric mean avoids bias towards extreme values present in the data set. Here too, the reworked algorithm shows improvement in comparison to the initial naive approach.

Table 6.2: Geometric mean of expansions

Geometric Mean	Eager Greedy FF	Greedy FF pref	LAMA first	Goalpref	Naive Goalpref
Expansions	439.97	250.34	227.57	152.15	190.84

We can verify the tendency to a low number of expansions of our implementation in more detail through the use of a scatter plot. The reworked version of the Optimistic Best-First Search with goal-preferred actions' number of expansions for a given problem instance is present in both following plots 6.1 6.1 on the y-axis. The comparison is the eager greedy FF configuration, with its number of expansions for a given problem is reflected on the x-axis. The diagonal indicates where the number of expansions is identical. In the case that the dot representing the problem instance is underneath the diagonal, the number of expansions is lower for our implementation compared to eager greedy FF. Vice versa, if the dot is above the diagonal, then eager greedy FF required fewer expansions than our implementation. This notion is reflected too in the "lower for x tasks" label in both configurations. From those values, and the cluster of problem instances visible around the middle underneath the diagonal, we can say that our implementation had fewer expansions more often than eager greedy FF. This could be due to the guidance of goal-preferred actions avoiding the deletion of goal variable value pairs and thus avoiding the need to reinstate a goal variable value pair after deletion.

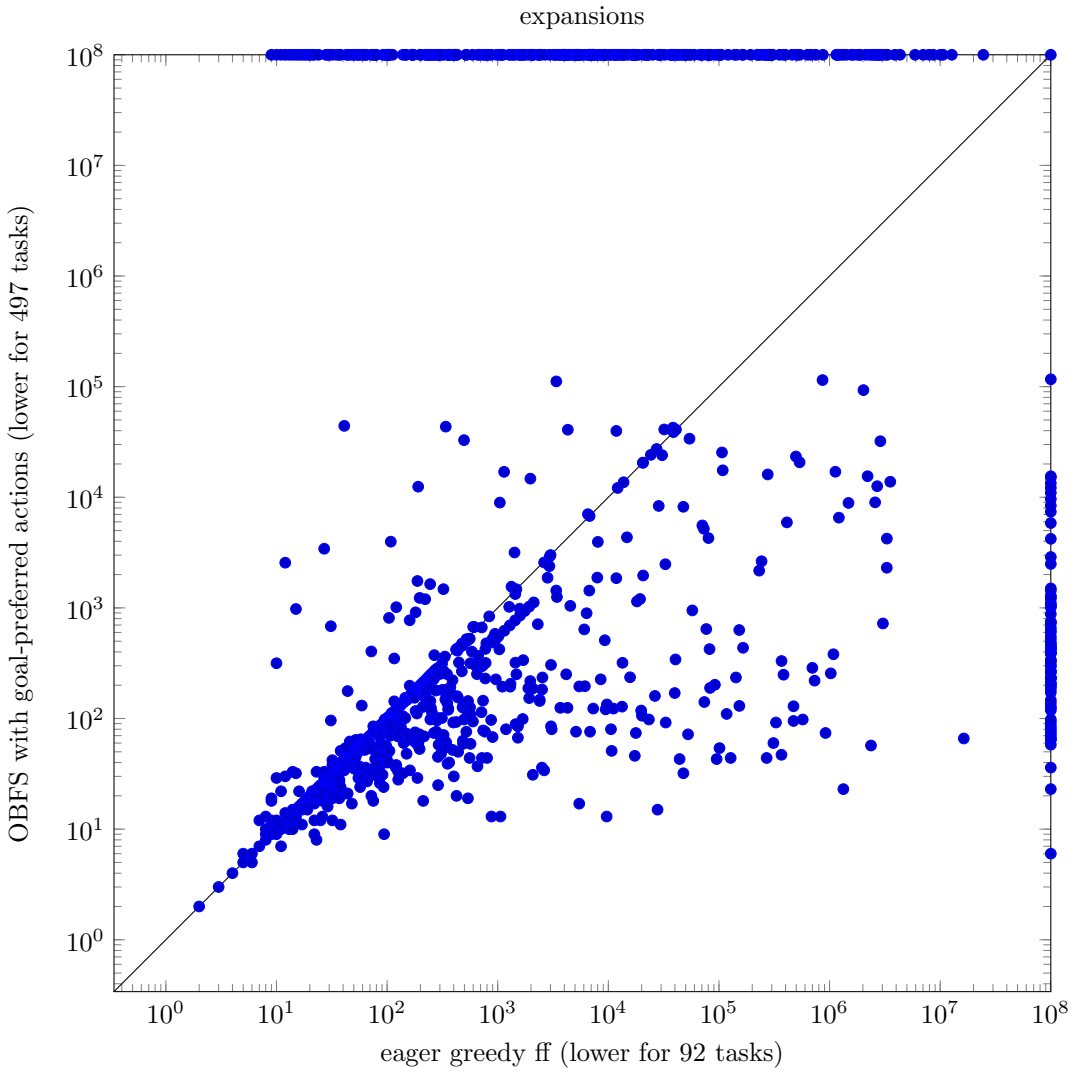
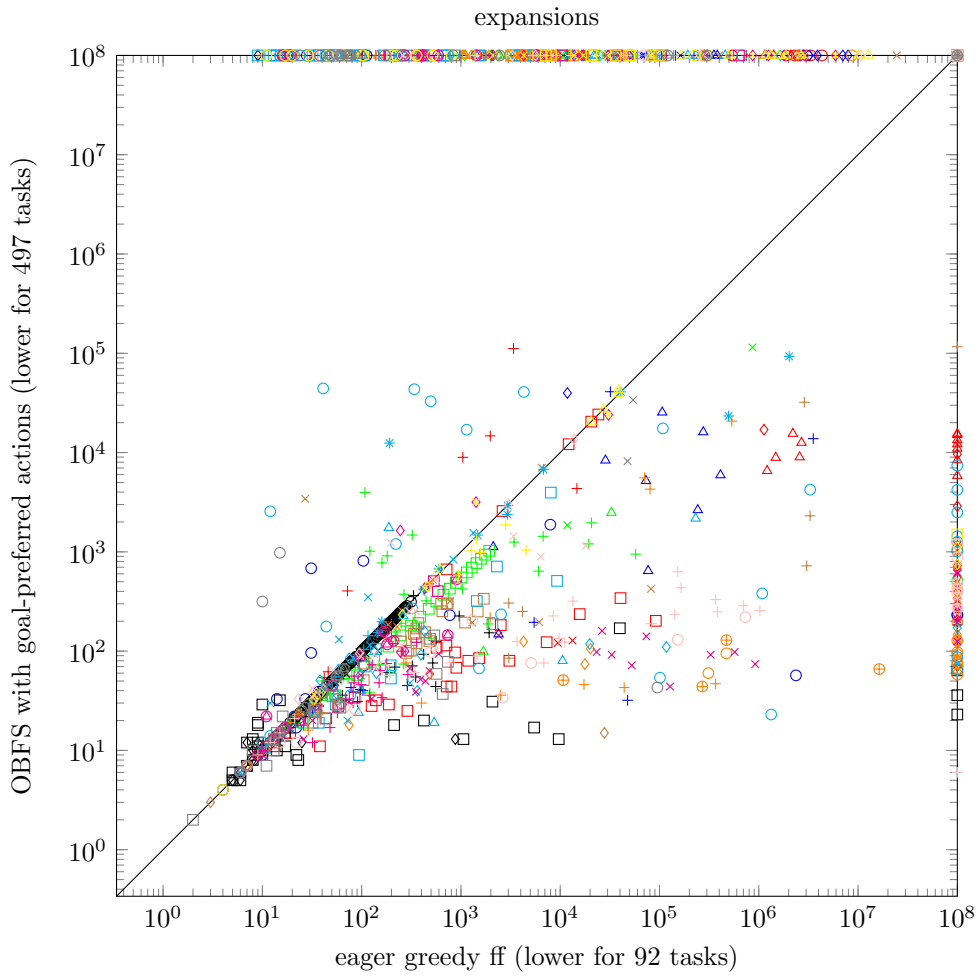


Figure 6.1: Expansions unlabeled

To find out for which domains the expansions of our implementation are lower for, we can take a look at a labeled scatter plot. On Figure 6.2 we can see some of the specific domains for which fewer expansions were needed for our implementation to find a solution. Namely for elevators-sat08-strips, tpp, mprime, rovers, barman-sat14-strips, logistics98, quantum-layout-sat23-strips. Mixed results were achieved for the pipesworld-notankage, driverlog and the freecell domain. Curiously, for gripper the number of expansions was reduced by seemingly a fixed amount, as a line parallel to the diagonal is visible underneath. This domain also has among the lowest ratio of goal-preferred actions found of the domains. Where the majority range from 90% - 100%, the gripper problem possessed an arithmetic mean of 75% on the goal-preferred ratio. Though no correlation could be found on the difference in goal-pref ratio for different domains and their coverage, implying the difference lies in the nature of the domains themselves.



× agricola-sat18-strips	+ airport	○ barman-sat11-strips
△ barman-sat14-strips	□ blocks	◇ childsnack-sat14-strips
× data-network-sat18-strips	+ depot	○ driverlog
△ elevators-sat08-strips	□ elevators-sat11-strips	◇ floortile-sat11-strips
× floortile-sat14-strips	+ freecell	○ ged-sat14-strips
△ grid	□ gripper	◇ hiking-sat14-strips
× logistics00	+ logistics98	○ miconic
△ movie	□ mprime	◇ mystery
× nomystery-sat11-strips	+ openstacks-sat08-strips	○ openstacks-sat11-strips
△ openstacks-sat14-strips	□ openstacks-strips	◇ organic-synthesis-sat18-strips
× organic-synthesis-split-sat18-strips	+ parcprinter-08-strips	○ parcprinter-sat11-strips
△ parking-sat11-strips	□ parking-sat14-strips	◇ pathways
× pegsol-08-strips	+ pegsol-sat11-strips	○ pipesworld-notankage
△ pipesworld-tankage	□ psr-small	◇ quantum-layout-sat23-strips
× rovers	+ satellite	○ scanalyzer-08-strips
△ scanalyzer-sat11-strips	□ snake-sat18-strips	◇ sokoban-sat08-strips
× sokoban-sat11-strips	+ spider-sat18-strips	○ storage
△ termes-sat18-strips	□ tetris-sat14-strips	◇ thoughtful-sat14-strips
× tidybot-sat11-strips	+ tpp	○ transport-sat08-strips
△ transport-sat11-strips	□ transport-sat14-strips	◇ trucks-strips
× visitall-sat11-strips	+ visitall-sat14-strips	○ woodworking-sat08-strips
△ woodworking-sat11-strips	□ zenotravel	

Figure 6.2: Expansions labeled

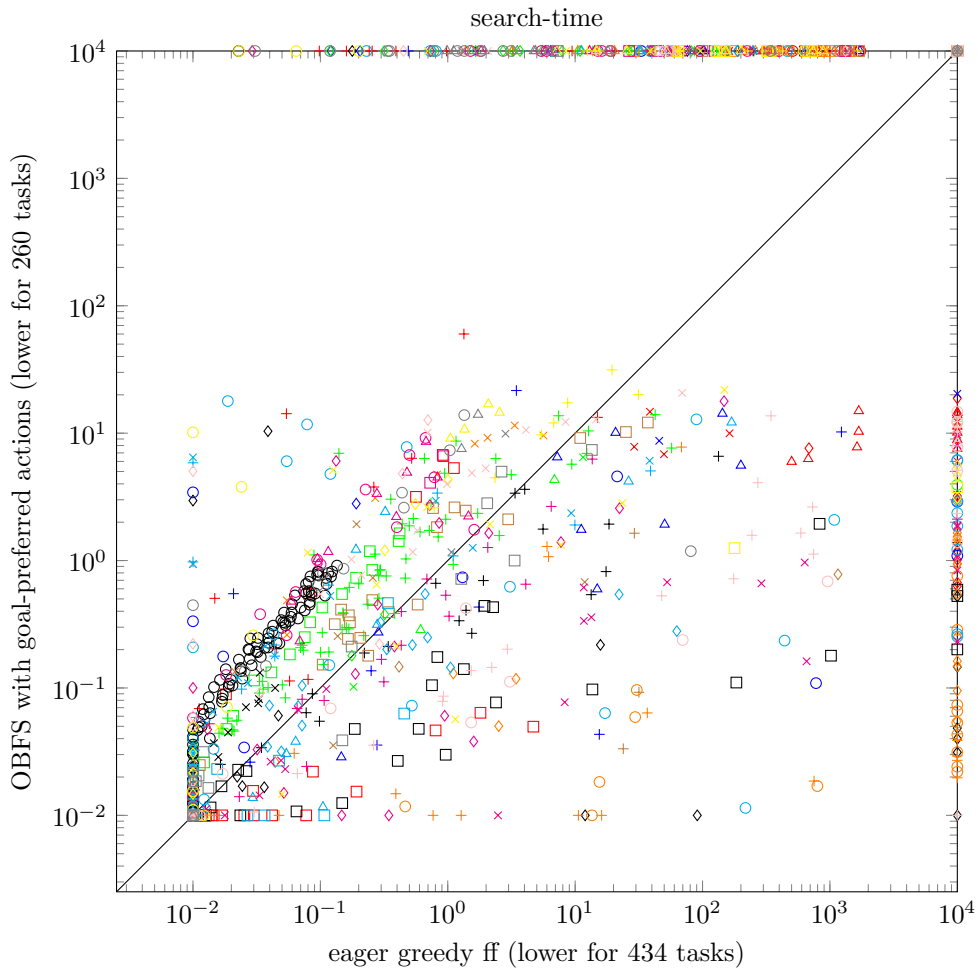
6.2.4 Search Time

For the search time, no significant difference was expected, as our implementation didn't make use of the caching structure contained in Fast Downward that other algorithms were able to uzilize. However, at first glance on this table the geometric mean on the search time indicates comparable search times for the goal-preferred algorithm to the eager greedy FF configuration and the greedy FF with preferred operators configuration. It is also visible that the reworking of the goal-preferred algorithm from its naive version made a positive difference on the search time.

Table 6.3: Geometric mean of search time

Geometric Mean	Eager Greedy FF	Greedy FF pref	LAMA first	Goalpref	Naive Goalpref
Search time	0.18	0.14	0.05	0.16	0.26

As was the case for the expansions, on the labeled scatter plot 6.3 we want to see a minimization of the values for our implementation, and thus problem instances being reflected below the diagonal is desirable. On the labeled plot, the fixed difference in the search time because of the use of caching of search algorithms other than our implementation is visible by the parallel clusters built above the diagonal. Regarding search time, we can see that our implementation was lower for less planning tasks than for the eager greedy ff implementation. Here we see elevators-sat08-strips, mprime, tpp, quantum-layout-sat23-strips and barman-sat14-strips represented again as domains for which our implementation does well for, but also newly parcprinter-08-strips.

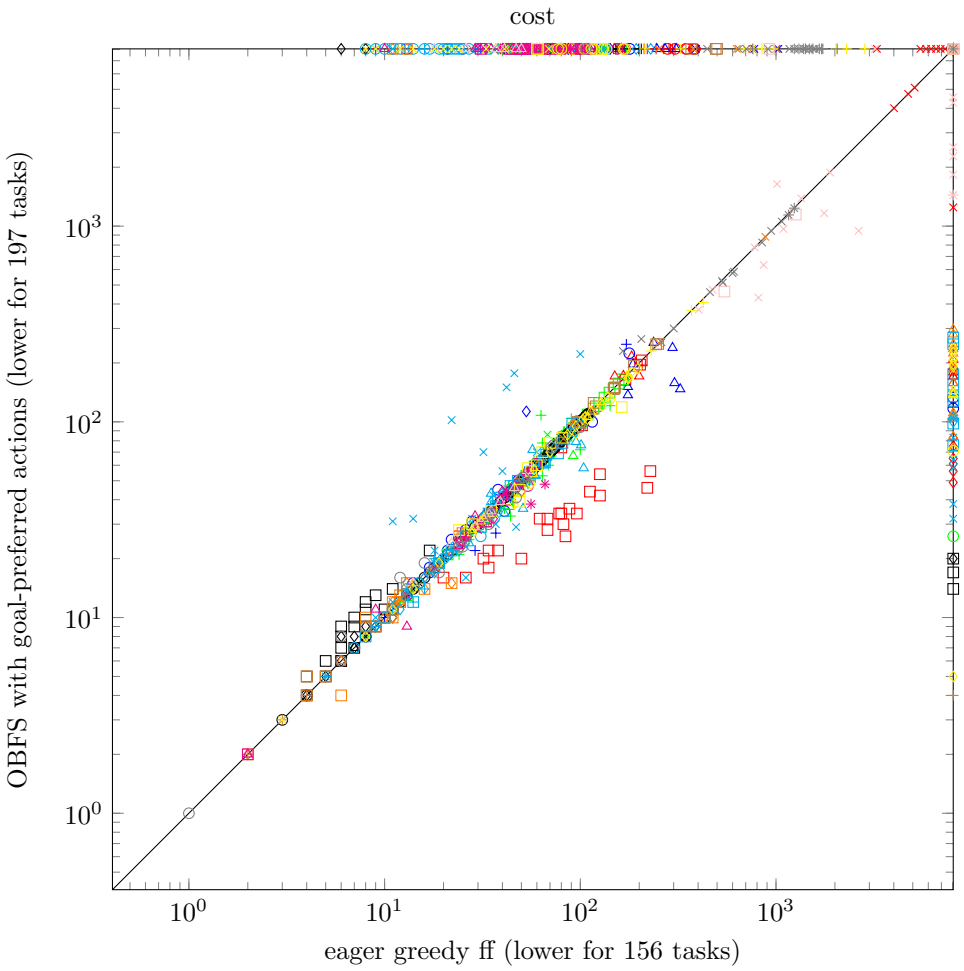


agricola-sat18-strips	airport	barman-sat11-strips
barman-sat14-strips	blocks	childsnack-sat14-strips
data-network-sat18-strips	depot	driverlog
elevators-sat08-strips	elevators-sat11-strips	floortile-sat11-strips
floortile-sat14-strips	freecell	ged-sat14-strips
grid	gripper	hiking-sat14-strips
logistics00	logistics98	miconic
movie	mprime	mystery
nomystery-sat11-strips	openstacks-sat08-strips	openstacks-sat11-strips
openstacks-sat14-strips	openstacks-strips	organic-synthesis-sat18-strips
organic-synthesis-split-sat18-strips	parcprinter-08-strips	parcprinter-sat11-strips
parking-sat11-strips	parking-sat14-strips	pathways
pegsol-08-strips	pegsol-sat11-strips	pipesworld-notankage
pipesworld-tankage	psr-small	quantum-layout-sat23-strips
rovers	satellite	scanalyzer-08-strips
scanalyzer-sat11-strips	snake-sat18-strips	sokoban-sat08-strips
sokoban-sat11-strips	spider-sat18-strips	storage
termes-sat18-strips	tetris-sat14-strips	thoughtful-sat14-strips
tidybot-sat11-strips	tpp	transport-sat08-strips
transport-sat11-strips	transport-sat14-strips	trucks-strips
visitall-sat11-strips	visitall-sat14-strips	woodworking-sat08-strips
woodworking-sat11-strips	zenotravel	

Figure 6.3: Search time labeled

6.2.5 Cost

For the scatter plot 6.4 we too want to minimize costs if possible, or at least remain comparable to other suboptimal search algorithms, and thus have our implementation show problem instances appearing underneath the diagonal. Here, no significant difference is shown between both configurations for the domains, except for the blocks planning task, for which our implementation found plans of lower cost. With the majority of the problem instances remaining close on the diagonal, and the number for which the costs were lower for both configurations not differing greatly compared to plots of other metrics, we can say that the achievement to acquire plans with comparable costs was reached.



× agricola-sat18-strips	+ airport	○ barman-sat11-strips
△ barman-sat14-strips	□ blocks	◇ childsnack-sat14-strips
× data-network-sat18-strips	+ depot	○ driverlog
△ elevators-sat08-strips	□ elevators-sat11-strips	◇ floortile-sat11-strips
× floortile-sat14-strips	+ freecell	○ ged-sat14-strips
△ grid	□ gripper	◇ hiking-sat14-strips
× logistics00	+ logistics98	○ miconic
△ movie	□ mprime	◇ mystery
× nomystery-sat11-strips	+ openstacks-sat08-strips	○ openstacks-sat11-strips
△ openstacks-sat14-strips	□ openstacks-strips	◇ organic-synthesis-sat18-strips
× organic-synthesis-split-sat18-strips	+ parking-sat11-strips	○ parking-sat14-strips
△ pathways	□ pegsol-08-strips	◇ pegsol-sat11-strips
× pipesworld-notankage	+ pipesworld-tankage	○ psr-small
△ quantum-layout-sat23-strips	□ rovers	◇ satellite
× scanalyzer-08-strips	+ scanalyzer-sat11-strips	○ snake-sat18-strips
△ sokoban-sat08-strips	□ sokoban-sat11-strips	◇ spider-sat18-strips
× storage	+ termes-sat18-strips	○ tetris-sat14-strips
△ thoughtful-sat14-strips	□ tidybot-sat11-strips	◇ tpp
× transport-sat08-strips	+ transport-sat11-strips	○ transport-sat14-strips
△ trucks-strips	□ visitall-sat11-strips	◇ visitall-sat14-strips
× woodworking-sat08-strips	+ woodworking-sat11-strips	○ zenotravel

Figure 6.4: Costs labeled

6.2.6 Number of Goal-preferred Actions

The total amount of actions amounted to 36'917'800, out of which the following Table 6.4 specifies how many of them were recognized to be goal-preferred by the two different configurations of our implementation.

Table 6.4: Goal-preferred actions

Sum Metric	Goalpref	Naive Goalpref
Goal-preferred actions	35'433'194	22'143'251
Goal-preferred ratio arithmetic average %	92%	64%

Like we hypothesized in Section 4.6 on our running example, the reduced coverage for the configuration utilizing the naive goal-preferred algorithm could be due to the strictness in discarding an action as a goal-preferred candidate without regarding the information preconditions are able to provide on the previous state reflecting itself negatively on the ability to find a solution.

6.3 Discussion

The storing of Helpful and Rescue Nodes explicitly for each state in our implementation caused the search to run out of memory in many instances, which goes against the point of using suboptimal algorithms. The incompleteness of the implementation diverges with the theoretical aspects as seen in previous chapters, indicating the presence of a logic error. Despite this, it was discovered that for some problems far fewer expansions were needed by both goal-pref configurations to arrive to plans in comparable time with comparable costs.

7
Conclusion

We implemented a partial version of the LOBFS algorithm described by Vidal [10], implementing the notion of goal-preferred actions, helpful actions/nodes and rescue actions/nodes, taking into consideration action costs, disregarding some tie-breaking aspects for the open list and excluding the lookahead algorithm.

As a future project, more aspects mentioned in Vidal's paper could be implemented. For example, in the pop_best_node() algorithm the heuristic value was calculated as in WA* (f(s)= $W * h + length(P)$), with W= 3, so the effect of the weight on the algorithm could be investigated, and the choice of W= 3 justified. Within the confines of this project, the lookahead algorithm was omitted, but can be implemented and incorporated into the existing solution. Our implementation is incomplete, and is memory-inefficient, yet showed comparable search times and plan costs to other planners. Regarding Vidal's work, there too are further variations of the lookahead algorithm YAHSP2 [11] and YAHSP3 [12], that could possibly contain significant optimizations if implemented in Fast Downward [4].

Bibliography

[1] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

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

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

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

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

[6] Drew M. McDermott. The 1998 AI planning systems competition. *AI magazine*, 21: 35–35, 2000.

[7] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1: 193–204, 1970.

[8] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[9] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.399255, 2017.

[10] Vincent Vidal. A Lookahead Strategy for Heuristic Search Planning. 2004.

[11] Vincent Vidal. YAHSP2: Keep it simple, stupid. *Proceedings of the 7th International Planning Competition (IPC-2011)*, pages 83–90, 2011.

[12] Vincent Vidal. YAHSP3 and YAHSP3-MT in the 8th international planning competition. *Proceedings of the 8th International Planning Competition (IPC-2014)*, pages 64–65, 2014.