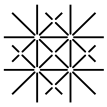


University of Basel

Exploring The Prioritized Incremental Heuristic

Bachelor thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Thomas Keller

Daniel Weissen
daniel.weissen@stud.unibas.ch
17-065-723

12.10.2020

Acknowledgments

I want to thank Prof. Dr. Malte Helmert for accepting my request to do this thesis and giving me the opportunity to work on such an interesting topic. Special thanks to my supervisor Dr. Thomas Keller for helping me throughout the completion of this thesis. Thomas was very generous with his time and his advice and feedback was always useful and very much appreciated.

Abstract

This thesis discusses the PINCH heuristic, a specific implementation of the additive heuristic. PINCH intends to combine the strengths of existing implementations of the additive heuristic. The goal of this thesis is to really dig into the PINCH heuristic. I want to provide the most accessible resource for understanding PINCH and I want to analyze the performance of PINCH by comparing it to the algorithm on which it is based, Generalized Dijkstra.

Table of Contents

1 Introduction

Heuristic search planners like HSP 2.0 [1], FF [2] or Fast Downward [3] transform planning problems described in a planning domain language (PDDL) into heuristic search problems. They derive heuristic values from the encoding of the planning problem and use these values to perform a heuristic forward or backward search in the space of world states. The goal is to find a path from a given initial state to a goal state. There are many different ways of extracting informed heuristic values from encoded planning problems. One of the most successful approaches is to consider a version of the planning problem where all delete effects of all actions are ignored (often referred to as a relaxed planning problem) and use the approximation of the relaxed planning cost as the heuristic value. One such way of approximating the relaxed planning cost is the additive heuristic.

When finding a plan for a problem, heuristic search planners spend the majority of the planning time calculating heuristic values, for HSP 2.0 about 80% of the planning time [4]. It therefore makes sense that, if we want to speed up the planning time, changing the way the heuristic values are computed should yield the most beneficial results.

For the additive heuristic, there are many different methods of computing the heuristic values. Perhaps the most intuitive one is to construct a relaxed planning graph and compute a plan for said graph. A different method uses a process called value iteration which iteratively updates variable values until none change in an iteration. An improvement on the value iteration approach is to order the value updates such that it leads to a quicker solution. An alternative way of speeding up the calculation uses information from previously determined heuristic values by performing incremental calculations.

The goal of this thesis is to explore a specific implementation of the additive heuristic. The PINCH (Prioritized, INCremental, Heuristic calculation) method, as described by Yaxin Liu, Sven Koenig & David Furcy [5], aims to combine both variable ordering and the reuse of information of previous heuristic values (incremental calculation). It is said to dramatically improve planning times compared to most alternative implementations of the additive heuristic[5]. My aim is to provide an in depth look at PINCH and to implement and test the PINCH method and describe both its benefits and failings.

2
Background

First it is important to explain the various concepts we will be using throughout the thesis. We will cover classical planning, relaxation heuristics and the additive heuristic.

2.1 Classical Planning

Planning refers to the process of finding a plan from a given initial state to a goal state. Classical planning refers to the process of finding a plan for a subset of problems, mainly those problems that are static, deterministic and fully observable. These problems are usually described with the use of a planning formalism. Planning formalisms allow us to encode problems in a concise and easily understandable way. Common planning formalisms are STRIPS, ADL, SAS+ and PDDL. For the purpose of this thesis we will focus on STRIPS-style planning tasks enhanced with action costs.

A problem, encoded in STRIPS, is given as input to a planner. The job of the planner is to find a solution (=plan) for the problem. To describe how a planner finds a solution it is useful to introduce a more formal definition of a STRIPS planning task.

A STRIPS planning task is a 4-tuple $\Pi = \langle V, I, G, A \rangle$ where:

- V: finite set of state variables
- $I \subseteq V$: the initial state
- $G \subseteq V$: the set of goals
- A: finite set of actions

For each action $a \in A$ we also define:

- $pre(a) \subseteq V$: 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

The solution for a STRIPS planning task is a plan from the initial state I to one of the goal states G and an optimal solution is a plan with the lowest cost. To find a solution, the planner induces a state space $\Omega(\Pi) := \langle S, A, cost, T, s_0, S_* \rangle$ where:

- S: set of states $= 2^V =$ (power set of V)
- A: actions as defined in Π
- $cost$: costs as defined in Π
- T: transitions $s \xrightarrow{a} s'$ for states s, s' and action a iff:
  - $pre(a) \in s$ (preconditions satisfied)
  - $s' = (s \setminus del(a)) \cup add(a)$ (delete and add effects are applied)
- s_0: initial state $s_0 = I$
- S_*: set of goal states $s \in S_*$ for state s iff $G \subseteq s$ (goals reached)

A planner can then find a solution for the state space Ω using different search algorithms. As they are generally faster heuristic based search algorithms such as A* are usually used. The question now is how heuristic values can be derived from STRIPS encodings.

2.2 Relaxation Heuristics

There are many different ways of deriving heuristic values from STRIPS encodings, one of the most successful approaches is to consider a relaxed version of the planning task.

A relaxed version of a planning task is identical to the original except for one important difference, for every action $a \subseteq A$ the delete effects $del(a)$ are ignored. This means that, in a relaxed planning task, applying an action to a state always leads to a state where more or an equal number of state variables are true. This trivializes the search significantly as applying any possible action at any possible step of the search is certain to at the very least not undo any progress we have made. To put it differently, every action brings us closer to the goal or it keeps us at a consistent distance to the goal, we never take a step backwards.

To define a relaxed planning task more formally let us first introduce a relaxed version a^+ of an action a with:

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

A relaxed version Π^+ of a STRIPS planning task $\Pi = \langle V, I, G, A \rangle$ is defined as:

$$\Pi^+ := \langle V, I, G, \{a^+ | a \in A\} \rangle$$

Let $s^0, ..., s^n \in S$ be states and $a^1, ..., a^n \in A$ be actions such that $s^0 \xrightarrow{a^1} s^1, ..., s^{n-1} \xrightarrow{a^n} s^n$. A plan is a sequence of actions $\pi = \langle a_1,..., a_n \rangle$ that leads from s^0 to s^n. The cost of a plan is defined as $cost(\pi) = \sum_{i=1}^{n} cost(a_i)$, the optimal plan π^* is the plan with minimum $cost(\pi)$ among all plans $\pi \in \Lambda$, with Λ being the set of all plans from a given s^0 and s^n. A plan for Π^+ is denoted by π^+ which is also called a relaxed plan for Π. $h^+(\Pi)$ denotes the cost of an optimal plan for Π^+ and $h^+(s)$ denotes the cost of an optimal plan for Π^+ starting in state $s \subseteq S$. h^+ is called the optimal relaxation heuristic.

The question now is how this relaxed version Π^+ of a STRIPS planning task Π helps us find informed heuristic values for Π. The general idea is to use the cost of an optimal relaxed plan $h^+(s)$ as heuristic values for each state $s \subseteq S$. Unfortunately, there is a catch with this idea. The computation of h^+ is NP-hard. It is not feasible to use h^+ as heuristic values since the computation would take too much time, therefore we have to approximate h^+ in a time efficient manner. The domain of relaxation heuristics cover different ways of approximating h^+. One such form of approximation is the additive heuristic.

2.3 Additive Heuristic

The additive heuristic (h^{add}) is a relaxation heuristic and therefore it has the central goal of approximating h^+ in a time efficient manner. In order to properly explain how h^{add} works, we first introduce the concept of a relaxed planning graph.

2.3.1 Relaxed Planning Graph

A relaxed planning graph is a graphical representation of the variables in Π^+ which can be reached and how they can be reached. To explain how a relaxed planning graph works and what information is displayed let us first look at the following example. Consider the relaxed planning task $\Pi^+ = \langle V, I, G, A \rangle$ with:

- $V = \{a, b, c, d, e, f, g, h\}$
- $I = \{a\}$
- $G = \{c, d, e, f, g\}$
- $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$
- $a_1 = a \xrightarrow{3} b, c$
- $a_2 = a, c \xrightarrow{1} d$
- $a_3 = b, c \xrightarrow{1} e$
- $a_4 = b \xrightarrow{1} f$
- $a_5 = d \xrightarrow{1} e, f$
- $a_6 = d \xrightarrow{1} g$

The relaxed planning graph for this task now looks as follows:

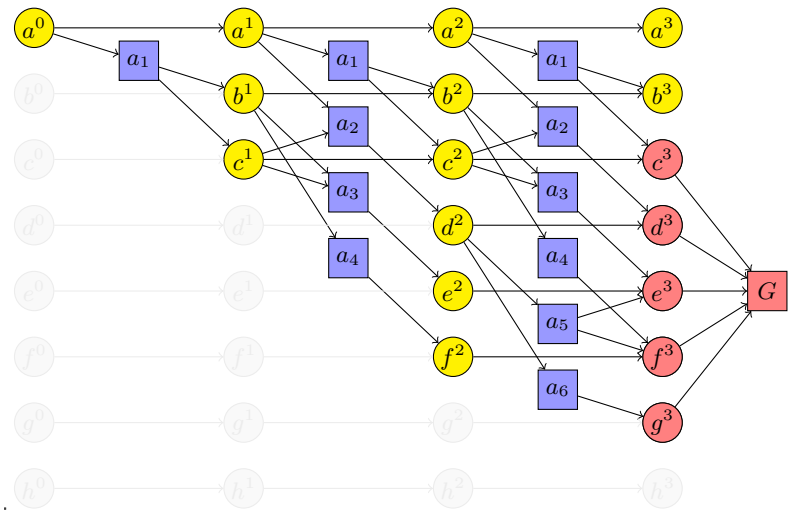


Figure 2.1: Relaxed planning graph

The graph is structured in a series of variable layers V^i and action layers A^i where:

- variable layer V^0 contains the variable vertex v^0 for all $v \in I$
- action layer A^{i+1} contains the action vertex a^{i+1} for action a if V^i contains the vertex v^i for all $v \in pre(a)$
- variable layer V^{i+1} contains the variable vertex v^{i+1} if the previous variable layer contains v^i or the previous action layer contains a^{i+1} with $v \in add(a)$
- goal vertices G^i are reached if $v^i \in V^i$ for all $v \in G$
- directed edges:
  - from v^i to a^{i+1} if $v \in pre(a)$ (precondition edges)
  - from a^i to v^i if $v \in add(a)$ (effect edges)
  - from v^i to G^i if $v \in G$ (goal edges)
  - from v^i to v^{i+1} (no-op edges)

We can now use this relaxed planning graph to help us understand how the additive heuristic works.

2.3.2 Understanding the Additive Heuristic

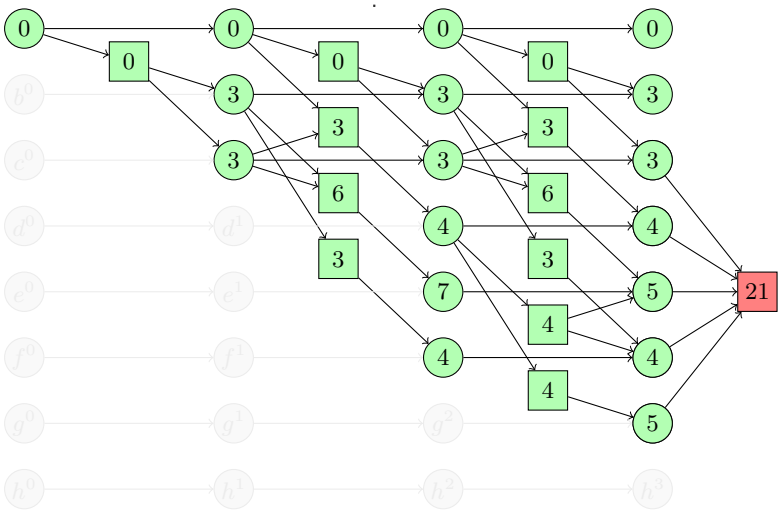
The central goal of all relaxation heuristics is to approximate h^+. We can think of h^+ as the optimal sequence of actions to reach the goal vertex G form variable layer V^0. In order to approximate h^+, h^{add} annotates all vertices and all actions with numerical values. These values estimate the cost to reach a vertex or to reach all preconditions of an action. We use $g_s(v)$ to denote the estimate cost of achieving variable $v \in V$ from state $s \subseteq S$, and $g_s(a)$ to denote the estimate cost of achieving all preconditions of action $a \in A$ from state $s \subseteq S$. h^{add} updates the variable and action values according to the following equations:

$$g_s(v) = \begin{cases} 0 & \text{if } v \in s \\ \min_{a \in A | v \in add(a)}[cost(a) + g_s(a)] & \text{otherwise} \end{cases} \tag{2.1}$$

$$g_s(a) = \sum_{v \in pre(a)} g_s(v) \tag{2.2}$$

Note that if $v \in pre(a) := \emptyset$ then $\sum_{v \in pre(a)} g_s(v) := 0$ and if $a \in A | v \in add(a) := \emptyset$ then $\min_{a \in A | v \in add(a)}[g_s(a)] := \infty$.

h^{add} makes the fundamental assumption that in order to reach action $a \in A$, all preconditions $pre(a)$ of a must be reached independently of another, hence the summation in (2.2). This makes h^{add} a pessimistic algorithm, as it assumes the worst case scenario of needing to reach all precondition variables independently of another. Now lets look at a version of the previous relaxed planning graph, but this time with the annotated h^{add} cost values:



$h^{\text{add}}(\{a\}) = 21$

Figure 2.2: Relaxed planning graph with h^{add} values

We will not cover how the relaxed planning graph updates its cost values layer by layer, the relaxed planning graph serves us as an intuitive visual aid, the underlying algorithm is not important for this thesis. Important to note is that the final variable and action layer give us the final h^{add} cost values, which all agree with equations (2.1) and (2.2).

The actual heuristic value is given by the cost value of the goal vertex. There are implementations of h^{add} that build this relaxed planning graph and determine the heuristic value by traversing the graph, but these implementations are very slow and currently rarely used [5]. For us this graph serves more so as an intuitive basis for understanding h^{add}, it is not to be confused with the actual algorithms that today are used to compute h^{add}. I would now like to explore some relevant implementations of h^{add}.

2.3.3 Implementations of the Additive Heuristic

While the construction of a relaxed planning graph is a very intuitive and easily comprehensible way of understanding h^{add}, it is not a good implementation of the heuristic in terms of performance. We will now look at relevant concepts for implementing h^{add}.

2.3.3.1 Value Iteration

The central idea of the value iteration (VI) approach is to iterate over all variables and operators and updated their cost values according to Equation (2.1) and (2.2). This is done until no cost values change in an iteration. This method is fast because it does not build a relaxed planning graph, but it is not ideal since it randomly iterates over all variables and operators. The majority of the variables/operators that are considered in each iteration will not change their cost value, the VI approach therefore wastes a lot of time looking at variables/operators which are irrelevant for the current iteration. Following is pseudo code for a version of VI, going forward we will use x_q (or x_v/x_a to be specific) to refer to the annotated cost values of a variable or operator:

Algorithm 1: Value Iteration

```
1 Function VI(state s):
2     for each q ∈ V ∪ A \ s do set x_q := ∞;
3     for each v ∈ s do set x_v := 0;
4     repeat
5         for each a ∈ A do set x_a := cost(a) + Σ_{v∈pre(a)} x_v;
6         for each v ∈ V \ s do set x_v := min_{a∈A | v∈add(a)} [x_a];
7     until the values of all x_q remain unchanged during an iteration;
8     return Σ_{v∈G} x_v;
```

2.3.3.2 Value Iteration with Value Ordering

Value ordering (VO) seeks to correct for the main deficiency of the VI approach. By ordering the value updates, a VO approach mitigates the amount of unnecessary work done and mainly considers only those variables or operators whose values are likely to change in an iteration.

2.3.3.3 Generalized Dijkstra

One of the most successful h^{add} algorithms is often referred to as Generalized Dijkstra (GD). GD takes VO to an extreme degree, it does not iterate over all variables/operators, it systematically updates variables in a fashion that guarantees that GD only has to update each variable and operator once. GD uses a priority queue to keep track of the variables whose cost values are most likely to change. It orders the value updates by first considering variables with a low cost and gradually works up to variables with higher cost. Following is pseudo code for GD:

Algorithm 2: Generalized Dijkstra

```
1  Function GD(state s):
2      queue = priority queue over (value, variable) pairs ordered by value;
3      forever do
4          for each v ∈ V do set x_v := ∞;
5          for each a ∈ A do set x_a := cost(a);
6          for each v ∈ s enqueue with (0, v);
7          while the priority queue is not empty do
8              (val, var) = pop element with highest priority;
9              if x_var < val then
10                 continue;
11             if if var is last unsatisfied goal variable then
12                 return Σ_{v∈G} x_v;
13             foreach a ∈ A|var ∈ pre(a) do
14                 x_a := x_a + val;
15                 mark precondition var of a as satisfied;
16                 if all preconditions of a satisfied then
17                     if x_a < x_add(a) then
18                         x_add(a) := x_a;
19                         enqueue with (x_a, add(a));
20             return deadend;
```

2.3.3.4 Incremental Value Iteration

Just like VI, incremental value iteration (IVI) approaches generally sweep over all variables/operators. The difference here is that previous information from previously calculated heuristic values is used to speed up the calculation. The central idea is to not recalculate annotated h^{add} cost values for those variables or operators, for which we know for certain that their cost values have not changed for the current calculation. The following pseudo code is different from VI by only setting $x_q := \infty$ once for all $q \in V \cup A \setminus s$. It generally assumes that most x_q remain unchanged, if this assumption holds true it can speedup the heuristic calculation significantly. Unfortunately this assumption can also lead to IVI not finding a solution, in which case normal VI is called. IVI has produced promising results in some domains when compared to normal VI [5].

Algorithm 3: Incremental Value Iteration

Function `IVI` (*state* s)**:**
  if *s is initial state* **then**
    for each $q \in V \cup A$ do set $x_q := \infty$;
    set threshold for number of iterations;
  for each $v \in s$ do set $x_v := 0$;
  set iterations counter := 0;
  repeat
    for each $a \in A$ do set $x_a := cost(a) + \sum_{v \in pre(a)} x_v$;
    for each $v \in V \setminus s$ do set $x_v := \min_{a \in A \mid v \in add(a)} [x_a]$;
    increase counter by 1;
    if *counter is above threshold* **then**
      return VI(s);
  until *the values of all x_q remain unchanged in an iteration*;
  return $\sum_{v \in G} x_v$;

3
The PINCH Method

Now that we have laid a foundation, let us talk about the main topic of this thesis, the Prioritized INCremental Heuristic (PINCH). PINCH was introduced by Yaxin Liu, Sven Koenig and David Furcy in their 2002 paper "Speeding Up the Calculation of Heuristics for Heuristic Search-Based Planning". It is a method for calculating h^{add} that combines the previously introduced Generalized Dijkstra Algorithm with the Incremental Value Iteration approach.

3.1 PINCH, The Best of Both Worlds

Previously we have discussed two methods for computing h^{add}, Generalized Dijkstra (GD) and Incremental Value Iteration (IVI). Both methods come with their strengths and weaknesses. GD is very efficient at computing h^{add} for a single state, but in the larger context of an entire state space GD doesn't make use of previously calculated h^{add} values. In other words, GD treats each state as its own entity, it doesn't take into consideration the similarities certain states might share. IVI methods account for similarities between different states, they remember the previously annotated h^{add} cost values but they are considerably worse at evaluating a single state when compared to GD. Another way of thinking about it is that IVI methods take a holistic approach, they consider, as best they can, information from the entire state space. GD takes a reductionist approach, its focus is solely on one state an makes its computation as efficient as possible.

PINCH aims to combine the strengths of both GD and IVI. This is a non trivial task as the differences between GD and IVI methods appear in many ways irreconcilable. GD makes the fundamental assumption that cost values cannot increase during the computation of heuristic values, IVI methods violate this property. To show how PINCH works we will look at the pseudo code for PINCH and work through the algorithm step by step. Following is the pseudo code for PINCH.

Algorithm 4: PINCH

```
1  Function AdjustVariable(q):
2      if q ∈ V then
3          if q ∈ s then
4              set rhs_q := 0;
5          else
6              set rhs_q := min_{a∈A|v∈add(a)}[1 + x_a];
7      else
8          /*q ∈ A*/
9          set rhs_q := 1 + Σ_{v∈pre(a)} x_v;
10     if q is in the priority queue then
11         delete it;
12     if x_q ≠ rhs_q then
13         insert q into the priority queue with priority min(x_q, rhs_q);

14 Function SolveEquations():
15     while the priority queue is not empty do
16         delete the element with the smallest priority from the queue and assign it to q;
17         if rhs_q < x_q then
18             set x_q := rhs_q;
19             if q ∈ V then
20                 foreach a ∈ A such that q ∈ pre(a) do
21                     AdjustVariable(a);
22             else
23                 foreach v ∈ add(q) with v ∉ s do
24                     AdjustVariable(v);
25         else
26             set x_q := ∞;
27             AdjustVariable(q);
28             if q ∈ V then
29                 foreach a ∈ A such that q ∈ pre(a) do
30                     AdjustVariable(a);
31             else
32                 foreach v ∈ add(q) with v ∉ s do
33                     AdjustVariable(v);

34 Function PINCH(state s):
35     if s is initial state then
36         empty the priority queue;
37         foreach q ∈ V ∪ A do
38             set x_q := ∞;
39         foreach q ∈ V ∪ A do
40             AdjustVariable(q);
41     else
42         foreach v ∈ (s \ s') ∪ (s' \ s) do
43             AdjustVariable(v);
44     SolveEquations();
45     set s' := s;
46     return 1/2 Σ_{v∈G} x_v;
```

3.2 The Algorithm in Depth

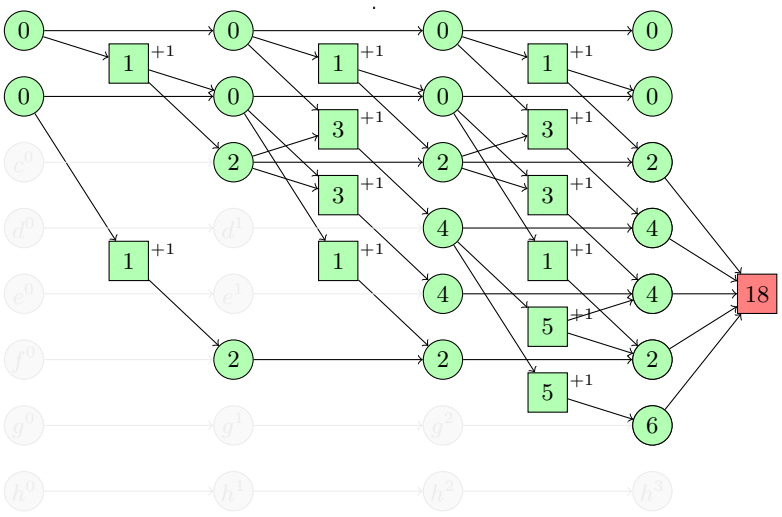
I want to explain PINCH by taking the algorithm through an example. Lets consider the following planning task $\Pi^+ = \langle V, I, G, A \rangle$ with:

- $V = \{a, b, c, d, e, f, g\}$
- $I = \{a, b\}$
- $G = \{c, d, e, f, g\}$
- $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$
- $a_1 = a \xrightarrow{1} b, c$
- $a_2 = a, c \xrightarrow{1} d$
- $a_3 = b, c \xrightarrow{1} e$
- $a_4 = b \xrightarrow{1} f$
- $a_5 = d \xrightarrow{1} e, f$
- $a_6 = d \xrightarrow{1} g$

We will compute PINCH(s) for the current state s. For the computation PINCH will use information from the previous state s'. s and s' are defined as follows:

- $s' = I = \{a, b\}$
- $s = \{a, c\}$

Since PINCH is an incremental algorithm the cost values of s' are relevant for the computation of s, therefore lets take a look at s':



$h^{\text{add}}(s') = 34/2 = 9$

old state s'		
q	x_q	rhs_q
a	0	0
b	0	0
c	2	2
d	4	4
e	4	4
f	2	2
g	6	6
a_1	1	1
a_2	3	3
a_3	3	3
a_4	1	1
a_5	5	5
a_6	5	5

In the table on the right you will find the annotated cost values x_q for all $q \in V \cup A$, you will also find a column for a property named rhs_q, we will explain the utility of this property as we go along. On the left you see the relaxed planning graph (RPG) for s'. PINCH does not build a RPG but this visualization will help us understand the algorithm.

You may have noticed that the annotated cost values aren't calculated the way that was introduced in equations (2.1) and (2.2), this is because PINCH changes these equations as follows:

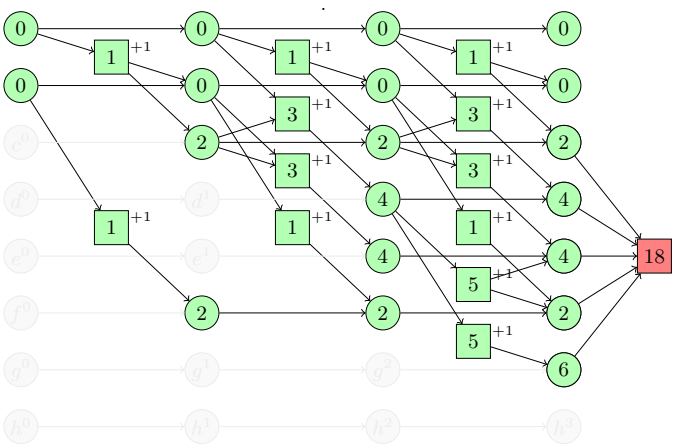
$$g'_s(v) = \begin{cases} 0 & \text{if } v \in s \\ \min_{a \in A | v \in add(a)}[1 + g'_s(a)] & \text{otherwise} \end{cases} \tag{3.1}$$

$$g'_s(a) = 1 + \sum_{v \in pre(a)} g'_s(v) \tag{3.2}$$

It is trivial to show that $g_s(v) = 1/2\, g'_s(v)$ and therefore $h^{add}(s) = \sum_{v \in G} g_s(v) = 1/2 \sum_{v \in G} g'_s(v)$.[5] This version of PINCH works for planning tasks with unit cost, meaning that every action $a \in A | cost(a) = 1$. We will explore the reason for PINCH changing these equations in chapter 3.3. For now I want to provide an intuitive example that gives a basic impression of how PINCH works, a more formal exploration of the algorithm follows in chapter 3.3.

Starting with the algorithm, I want to focus on lines 42 and 43. Since s is not the initial state this is where the algorithm begins. Here we call AdjustVariable(q) for each $v \in (s \backslash s') \cup (s' \backslash s)$. In our example $v \in (s \setminus s') = c$ and $v \in (s' \setminus s) = b$. Before we call AdjustVariable(q) on c and b, lets first think about the state of our state s. State s is currently an exact copy of s' (which you can see on page 12). In PINCH we never reinitialize any x_q to an arbitrary value, the x_q and rhs_q values at the beginning of the computation of s are therefore identical to its predecessor state s'.

Lines 1 - 13 in the pseudo code cover the AdjustVariable(q) procedure. In this procedure the rhs_q component is set according to equations (3.1) and (3.2) and the q are inserted into the priority queue (PQ) with value $\min(x_q, rhs_q)$ if $x_q \neq rhs_q$. Here the utility of the rhs_q component starts to show, rhs_q is used to compare the current cost value of q, represented by rhs_q, with its previous cost value, represented by x_q. We have just updated rhs_q and we only insert it into the PQ if $rhs_q \neq x_q$. Lets see what happens to s after we call AdjustVariable(q) on b and c.



new state s		
q	x_q	rhs_q
a	0	0
b	0	2
c	2	0
d	4	4
e	4	4
f	2	2
g	6	6
a_1	1	1
a_2	3	3
a_3	3	3
a_4	1	1
a_5	5	5
a_6	5	5

PQ
b:0
c:0

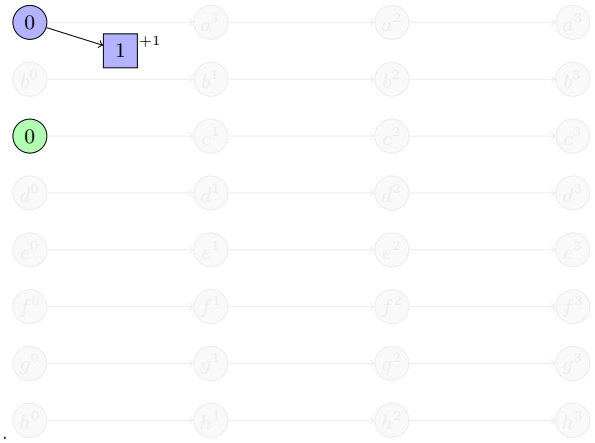
The rhs_q component of b was set to 2 and the one of c to 0. The RPG hasn't changed as it represents the x_q and not the rhs_q. The numbers in red represent the q whose values have changed compared to the previous picture. Additionally we now track the state of the priority queue, that currently holds b and c with value $\min(x_q, rhs_q)$.

Now we move on to Line 44 and therefore to the SolveEquations() procedure. This procedure spans from line 14 - 33 and makes up the majority of the algorithm. The main purpose of this procedure is to set the x_q values and to call AdjustVariable(q) on all the q that are affected by the newly set x_q values. The q that are popped out of the PQ are treated based on the relationship between rhs_q and x_q.

If $rhs_q > x_q$ then x_q is set to ∞ and we call AdjustVariable(q) for the current q and all q that follow from the current q. This is done because in this scenario we do not know the correct values for x_q and therefore we dont know the values for all q that depend on the current q.

if $rhs_q < x_q$ then x_q is set to rhs_q. If this scenario occurs we know for certain that rhs_q holds the correct value. This comparison of x_q and rhs_q in combination with the PQ is what allows PINCH to order value updates in a fashion similar to GD while also including incremental value calculations.

Let us take a look at our example after b and c were popped and processed in SolveEquations().



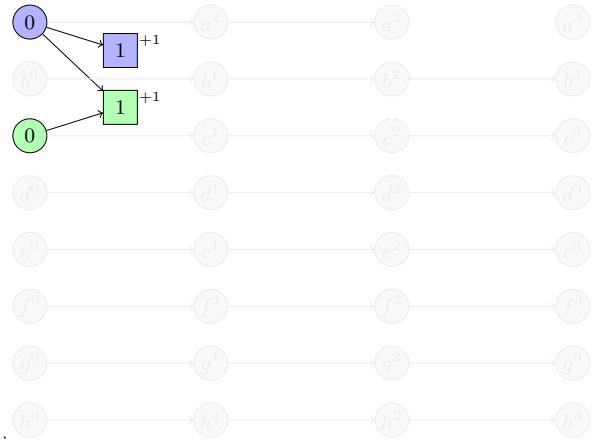
new state s		
q	x_q	rhs_q
a	0	0
b	∞	2
c	0	0
d	4	4
e	4	4
f	2	2
g	6	6
a_1	1	1
a_2	3	1
a_3	3	∞
a_4	1	∞
a_5	5	5
a_6	5	5

PQ
a_2:1
a_4:1
b:2
a_3:3

Since the value of b was set to ∞, we can now infer that everything that relies on b might have to be recalculated. This is why the rhs_q of a_3 and a_4 were set to ∞. The RPG represents the current state by having removed every node that depends on b directly or indirectly.

The blue nodes in the RPG show us the incremental aspect of PINCH. Since a was never inserted into the PQ, a itself and all q that solely depend on a (In this case a_1) do not have to be recalculated. The green nodes show us those q which PINCH had to recalculate and for which PINCH has set the final x_q value.

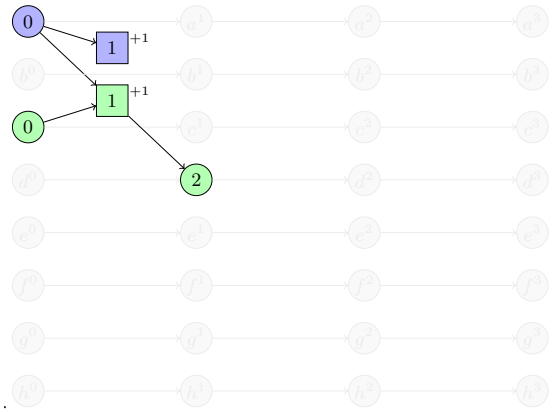
If we now pop a_2 and process it the example looks as follows:



new state s		
q	x_q	rhs_q
a	0	0
b	∞	2
c	0	0
d	4	2
e	4	4
f	2	2
g	6	6
a_1	1	1
a_2	1	1
a_3	3	∞
a_4	1	∞
a_5	5	5
a_6	5	5

PQ
a_4:1
d:2
b:2
a_3:3

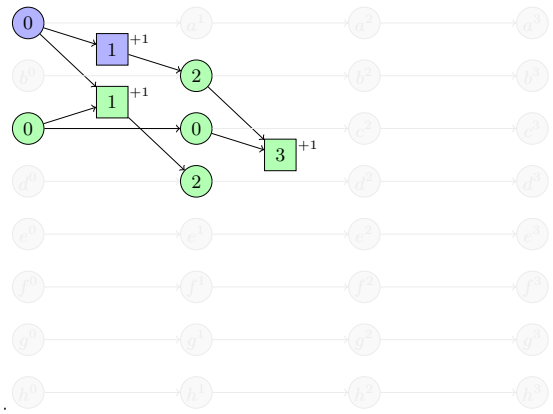
We will now see how the algorithm progresses as we continue to process the q. This is the example after we pop $a4, d$:



new state s		
q	x_q	rhs_q
a	0	0
b	∞	2
c	0	0
d	2	2
e	4	4
f	2	6
g	6	6
a_1	1	1
a_2	1	1
a_3	3	∞
a_4	∞	∞
a_5	5	3
a_6	5	3

PQ
f:2
b:2
a_5:3
a_6:3
a_3:7

The example after we pop f, b:

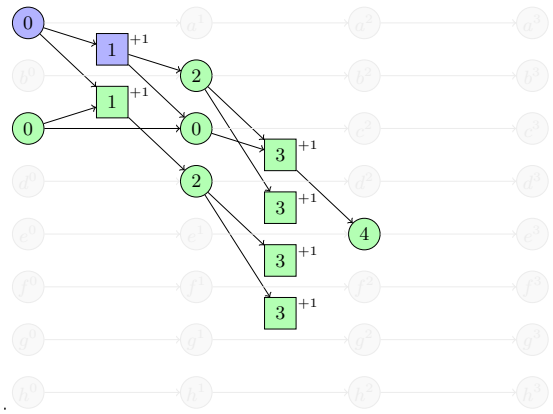


new state s		
q	x_q	rhs_q
a	0	0
b	2	2
c	0	0
d	2	2
e	4	4
f	∞	6
g	6	6
a_1	1	1
a_2	1	1
a_3	3	3
a_4	∞	3
a_5	5	3
a_6	5	3

PQ
a_5:3
a_6:3
a_4:3
f:6

Interesting to note here is that the rhs_q of a_3 was set to 3, which is the same as the cost value from the previous state. If a_3 had additional q that solely rely on a_3, then PINCH would not recalculate the cost values for those q, meaning that the algorithm would once more benefit from the incremental calculations.

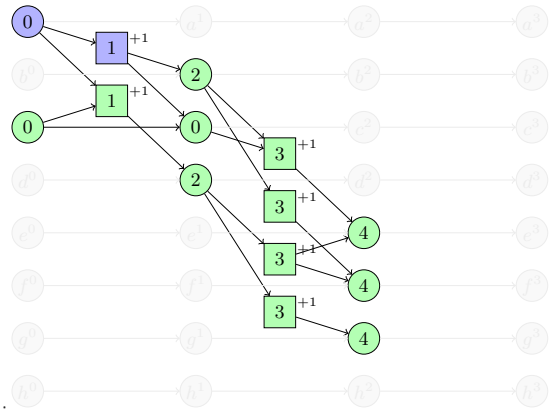
The example after we pop a_5, a_6, a_4:



new state s		
q	x_q	rhs_q
a	0	0
b	2	2
c	0	0
d	2	2
e	4	4
f	∞	4
g	6	4
a_1	1	1
a_2	1	1
a_3	3	3
a_4	3	3
a_5	3	3
a_6	3	3

PQ
f:4
g:4

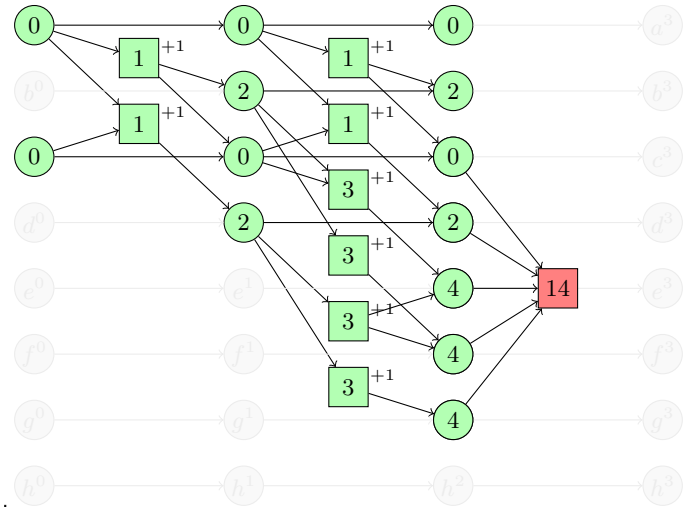
The example after we pop f,g:



new state s		
q	x_q	rhs_q
a	0	0
b	2	2
c	0	0
d	2	2
e	4	4
f	4	4
g	4	4
a_1	1	1
a_2	1	1
a_3	3	3
a_4	3	3
a_5	3	3
a_6	3	3

PQ

The priority queue is now empty and therefore the SolveEquations() procedure is complete. PINCH now sets $h^{add}(s) = 1/2 \sum_{v \in G} x_v$. If we fill in the remaining no-op edges the final RPG looks as follows:



$h^{\text{add}}(s) = 14/2 = 7$

new state s		
q	x_q	rhs_q
a	0	0
b	2	2
c	0	0
d	2	2
e	4	4
f	4	4
g	4	4
a_1	1	1
a_2	1	1
a_3	3	3
a_4	3	3
a_5	3	3
a_6	3	3

PINCH will now set $s' = s$ which can be seen in Line 45 of the Algorithm. Now the next state is computed in the same fashion that we have just seen. Lines 35 - 40 set up PINCH for its first computation of the initial state I. Since there is no s' in this instance, PINCH will treat the initial state I in a fashion similar to GD.

3.3 PINCH more Formally

We have just seen an Example of how PINCH processes the computation of the h^{add} value for state s. Now I want to further explain some of the details of the algorithm and give a more formal explanation of how and why PINCH works.

For us to understand PINCH more formally we first have to introduce the concept of a strict weakly superior function (SWSF). A SWSF is a function $g(x_1, ..., x_j, ..., x_k) : \mathbb{R}_+^k \rightarrow \mathbb{R}_+$ where for every $j \in 1...k$ it is monotone non-decreasing in variable x_j and satisfies: $g(x_1, ..., x_j, ..., x_k) \leq x_j \rightarrow g(x_1, ..., x_j, ..., x_k) = g(x_1, ..., \infty, ..., x_k)$.

The SWSF fixed point (in short: SWSF-FP) problem is to compute the unique fixed point of k equations, namely the equations $x_i = g_i(x_1, ..., x_k)$, in the k variables $x_1, ..., x_k$, where the g_i are SWSF for $i = 1...k$. The dynamic SWSF-FP problem is to maintain the unique fixed point of the SWSF equations after some or all of the functions g_i have been replaced by other SWSF's. DynamicSWSF-FP solves the dynamic SWSF-FP problem efficiently by re-calculating only the values of variables that change, rather than the values of all variables. [5]

3.3.1 Why PINCH Changes Equations (2.1) and (2.2)

PINCH uses DynamicSWSF-FP, it therefore requires for functions g_i to be SWSF. This is why equations (2.1) and (2.2) have to be replaced with equations (3.1) and (3.2), as equation (2.2) specifically is not a SWSF. Following is an example to help us understand how all these concepts relate to PINCH.

Consider the following planning task $\Pi^+ = \langle V, I, G, A \rangle$ with:

- $V = \{a, b, c, d\}$
- $I = \{a\}$
- $G = \{b, c, d\}$
- $A = \{a_1, a_2\}$
- $a_1 = a \xrightarrow{1} b, c$
- $a_2 = c \xrightarrow{1} d, a$
- $s = I = \{a\}$

Now consider the following functions: $g_i(x_a, x_b, x_c, x_d, x_{a_1}, x_{a_2})$ and $g'_i(x_a, x_b, x_c, x_d, x_{a_1}, x_{a_2})$, where $x_q \in (x_a, ..., x_{a_2})$ denote the current cost values for $q \in (a, ..., a_2)$. The functions are defined as follows:

$g_i(x_a, x_b, x_c, x_d, x_{a_1}, x_{a_2}) = g_s(i)$, where i refers to the q (not x_q) of the ith entry of the input arguments of $g_i(x_a, ..., x_{a_2})$ and $g_s(i)$ is equation (2.1) if $i \in V$ or equation (2.2) if $i \in A$. $g'_i(x_a, x_b, x_c, x_d, x_{a_1}, x_{a_2})$ alternatively uses equation (3.1) and (3.2).

Now let us apply these functions to our example, imagine that the x_q are currently set as follows: $(x_a = 0, x_b = \infty, x_c = \infty, x_d = \infty, x_{a_1} = 0, x_{a_2} = \infty)$. Following we test if equations g_1 to g_6 are SWSF. The ✓ indicate that the equation is a SWSF, the × indicate that the equation is not a SWSF.

$g_1(0,\infty,\infty,\infty,0,\infty) = g_s(a) = 0 \rightarrow g_1(0,\infty,\infty,\infty,0,\infty) = g_1(\infty,\infty,\infty,\infty,\infty,\infty) = 0$✓
$g_2(0,\infty,\infty,\infty,0,\infty) = g_s(b) = 1 \rightarrow g_2(0,\infty,\infty,\infty,0,\infty) = g_2(0,\infty,\infty,\infty,0,\infty) = 1$✓
$g_3(0,\infty,\infty,\infty,0,\infty) = g_s(c) = 1 \rightarrow g_3(0,\infty,\infty,\infty,0,\infty) = g_3(0,\infty,\infty,\infty,0,\infty) = 1$✓
$g_4(0,\infty,\infty,\infty,0,\infty) = g_s(d) = \infty \rightarrow g_4(0,\infty,\infty,\infty,0,\infty) = g_4(0,\infty,\infty,\infty,0,\infty) = \infty$✓
$g_5(0,\infty,\infty,\infty,0,\infty) = g_s(a_1) = 0 \rightarrow g_5(0,\infty,\infty,\infty,0,\infty) \neq g_5(\infty,\infty,\infty,\infty,\infty,\infty) = \infty$×
$g_6(0,\infty,\infty,\infty,0,\infty) = g_s(a_2) = \infty \rightarrow g_6(0,\infty,\infty,\infty,0,\infty) = g_6(0,\infty,\infty,\infty,0,\infty) = \infty$✓

Equation g_5 is not a SWSF since it does not fulfill the properties of a SWSF. We can therefore conclude that we cannot use equations (2.1) and (2.2) for the DynamicSWSF-FP algorithm on which PINCH is based. Now let us test if equations g'_1 to g'_6 are SWSF, note that x_{a1} is now set to 1 to make the two examples equivalent:

$g'_1(0,\infty,\infty,\infty,1,\infty) = g'_s(a) = 0 \rightarrow g'_1(0,\infty,\infty,\infty,1,\infty) = g'_1(\infty,\infty,\infty,\infty,\infty,\infty) = 0$✓
$g'_2(0,\infty,\infty,\infty,1,\infty) = g'_s(b) = 2 \rightarrow g'_2(0,\infty,\infty,\infty,1,\infty) = g'_2(0,\infty,\infty,\infty,1,\infty) = 2$✓
$g'_3(0,\infty,\infty,\infty,1,\infty) = g'_s(c) = 2 \rightarrow g'_3(0,\infty,\infty,\infty,1,\infty) = g'_3(0,\infty,\infty,\infty,1,\infty) = 2$✓
$g'_4(0,\infty,\infty,\infty,1,\infty) = g'_s(d) = \infty \rightarrow g'_4(0,\infty,\infty,\infty,1,\infty) = g'_4(0,\infty,\infty,\infty,1,\infty) = \infty$✓
$g'_5(0,\infty,\infty,\infty,1,\infty) = g'_s(a_1) = 1 \rightarrow g'_5(0,\infty,\infty,\infty,1,\infty) = g'_5(0,\infty,\infty,\infty,\infty,\infty) = 1$✓
$g'_6(0,\infty,\infty,\infty,1,\infty) = g'_s(a_2) = \infty \rightarrow g'_6(0,\infty,\infty,\infty,1,\infty) = g'_6(0,\infty,\infty,\infty,1,\infty) = \infty$✓

All functions g'_i are SWSF. PINCH can use equations (3.1) and (3.2) since $g_s(v) = 1/2\ g'_s(v)$ and therefore $h^{add}(s) = \sum_{v \in G} g_s(v) = 1/2 \sum_{v \in G} g'_s(v)$.

To give a more intuitive basis for the utility of SWSF, lets reexamine the example from chapter 3.2, this time we use equations (2.1) and (2.2) to update the cost values (The RPG is irrelevant for what we are about to discuss, hence it is absent):

new state s		
q	x_q	rhs_q
a	0	2
b	0	1
c	1	0
d	2	2
e	2	2
f	1	1
g	3	3
a_1	0	0
a_2	1	1
a_3	1	1
a_4	0	0
a_5	2	2
a_6	2	2

PQ
b:0
c:0

The state of the algorithm is identical to the one on page 14. Now lets pop c:

new state s		
q	x_q	rhs_q
a	0	0
b	0	1
c	0	0
d	2	2
e	2	2
f	1	1
g	3	3
a_1	0	0
a_2	1	0
a_3	1	0
a_4	0	0
a_5	2	2
a_6	2	2

PQ
b:0
a_2:0
a_3:0

After we pop c the priority queue contains 3 entries with the value 0. Here we run into a tie breaking issue as we have no guarantee over which q will be popped next. Now lets pop first a_3 and then b:

new state s		
q	x_q	rhs_q
a	0	0
b	∞	1
c	0	0
d	2	2
e	2	1
f	1	1
g	3	3
a_1	0	0
a_2	1	∞
a_3	0	∞
a_4	0	0
a_5	2	2
a_6	2	2

PQ
a_3:0
a_2:1
b:1
e:1

After we pop a_3 PINCH mistakenly believed to have assigned the correct value to a_3, only for it to be reinserted into the priority queue after we pop b. This means that PINCH at a minimum will update the cost value of a_3 3 times. The example demonstrates how the value ordering of PINCH relies on equations (3.1) and (3.2), if we use equations (2.1) and (2.2) PINCH loses its vital property of only having to update each cost value at most twice. [5]

3.3.2 DynamicSWSF-FP

If we view PINCH through the lens of DynamicSWSF-FP, then we are merely searching for the unique fixed point of the k equations $x_i = g_i(x_1, ..., x_k)$, in the k variables $x_1, ..., x_k$ where g_i are SWSF for i = 1...k.

If $x_i = g_i(x_1, ..., x_k)$, then we call x_i consistent, otherwise it is called inconsistent. If $x_i < g_i(x_1, ..., x_k)$ we refer to it as underconsistent and if $x_i > g_i(x_1, ..., x_k)$ we refer to it as overconsistent. PINCH has set the correct cost values for all $x_q \in V \cup A$ once they are all consistent. PINCH uses the variables rhs_i to keep track of the current values of $g_i(x_1, ..., x_k)$. It always holds that $rhs_i = g_i(x_1, ..., x_k)$ (Invariant 1). PINCH compares x_i and rhs_i to check whether x_i is overconsistent or underconsistent. PINCH maintains a priority queue that always contains exactly the inconsistent x_i with priorities $\min(x_i, rhs_i)$ (Invariant 2). [5]

PINCH calls AdjustVariable() for each x_q to ensure that Invariants 1 and 2 hold before it calls SolveEquations() for the first time. It needs to call AdjustVariable() only for those x_q whose function g_q has changed before it calls SolveEquations() again. The invariants will automatically continue to hold for all other x_q.[5]

SolveEquations() then adjusts the values of the inconsistent x_q. It always removes the x_q with the smallest priority from the priority queue. If x_q is overconsistent then SolveEquations() sets it to the value of rhs_q. This makes x_q consistent. Otherwise x_q is underconsistent and SolveEquations() sets it to infinity. This makes x_q either consistent or overconsistent. In the latter case, it remains in the priority queue. SolveEquations() then calls AdjustVariable() to maintain the Invariants 1 and 2. Once the priority queue is empty, SolveEquations() terminates since all x_q are consistent. One can prove that PINCH changes the value of each x_q at most twice, namely at most once when it is underconsistent and at most once when it is overconsistent, and thus terminates in finite time. [5] [6]

The authors of DynamicSWSF-FP have proved its correctness, completeness, and other properties and applied it to grammar problems and shortest path problems. [6]

3.4 Expanding PINCH Beyond Unit Cost

PINCH, as it was introduced by Yaxin Liu, Sven Koenig and David Furcy in 2002 only works for planning tasks $\Pi^+ = \langle V, I, G, A \rangle$ where for every action $a \in A | cost(a) = 1$. [5]

As a part of this thesis, we have expanded PINCH to be applicable on planning tasks Π^+ where for every action $a \in A | cost(a) > 0$. In order to do this we have to adjust equations (3.1) and (3.2) as follows:

$$g_s^*(v) = \begin{cases} 0 & \text{if } v \in s \\ \min_{a \in A | v \in add(a)} [cost(a) + g_s^*(a)] & \text{otherwise} \end{cases} \tag{3.3}$$

$$g_s^*(a) = cost(a) + \sum_{v \in pre(a)} g_s^*(v) \tag{3.4}$$

Since equations (3.3) and (3.4) are SWSF like equations (3.1) and (3.2) we are allowed to use them for the computation of PINCH. By adding $cost(a)$ in both equations, the useful property of $g_s(v) = 1/2\ g_s^*(v)$ remains intact. Therefore $h^{add}(s) = \sum_{v \in G} g_s(v) = 1/2 \sum_{v \in G} g_s^*(v)$. This means that we have successfully expanded PINCH beyond unit costs without raising its complexity.

3.5 Improved PINCH

The PINCH algorithm can be improved with a number of small optimizations. Consider, for example, the case where $q \in A$ has the smallest priority during an iteration of the while-loop in SolveEquations() and $rhs_q < x_q$. At some point in time, SolveEquations() then executes $for\ each\ v \in add(q)\ with\ v \notin s\ do\ AdjustVariable(v)$. The for-loop iterates over all variables that satisfy its condition. For each of them, the call AdjustVariable(v) executes set $rhs_q := \min_{a \in A | v \in add(a)} [1 + x_a]$. The calculation of rhs_q therefore iterates over all actions that contain v in their add list. However, this iteration can be avoided. Since $rhs_q < x_q$ according to our assumption, SolveEquations() sets the value of x_q to rhs_q and thus decreases it. All other values remain the same. Thus, rhs_v cannot increase and one can recalculate it faster as follows: set $rhs_v := \min(rhs_v, 1 + x_q)$. This and more optimizations are implemented in the following Improved PINCH algorithm.[5] This is the algorithm we use for our Evaluation, we have also added Equations 3.3 and 3.4 to make it applicable to planning tasks with $cost(a) > 0$ for all $a \in A$.

Algorithm 5: Improved PINCH

```
1  Function AdjustVariable(q):
2      if x_q ≠ rhs_q then
3          if q is not in the priority queue then
4              insert it with priority min(x_q, rhs_q);
5          else
6              change the priority of q in the priority queue to min(x_q, rhs_q);
7      else
8          if q is in the priority queue then
9              delete q from the priority queue;
10 Function SolveEquations():
11     while the priority queue is not empty do
12         assign the element with the smallest priority in the priority queue to q;
13         if q ∈ V then
14             if rhs_q < x_q then
15                 delete q from the priority queue;
16                 set x_old := x_q;
17                 set x_q := rhs_q;
18                 foreach a ∈ A such that q ∈ pre(a) do
19                     if rhs_a = ∞ then
20                         set rhs_a := cost(a) + Σ_{v∈pre(a)} x_v;
21                     else
22                         set rhs_a := rhs_a − x_old + x_q;
23                     AdjustVariable(a);
24             else
25                 set x_q := ∞;
26                 if q ∉ s then
27                     set rhs_q := min_{a∈A|v∈add(a)}[cost(a) + x_a];
28                     AdjustVariable(q);
29                 foreach a ∈ A such that q ∈ pre(a) do
30                     set rhs_a := ∞;
31                     AdjustVariable(a);
32         else
33             if rhs_q < x_q then
34                 delete q from the priority queue;
35                 set x_q := rhs_q;
36                 foreach v ∈ add(q) with v ∉ s do
37                     rhs_v = min(rhs_v, cost(a) + x_q);
38                     AdjustVariable(v);
39             else
40                 set x_old := x_q;
41                 set x_q := ∞;
42                 set rhs_q := cost(a) + Σ_{v∈pre(a)} x_v;
43                 AdjustVariable(q);
44                 foreach v ∈ add(q) with v ∉ s do
45                     if rhs_v = cost(a) + x_old then
46                         set rhs_v := min_{a∈A|v∈add(a)}[cost(a) + x_a];
47                         AdjustVariable(v);
48 Function PINCH(state s):
49     if s is initial state then
50         empty the priority queue;
51         foreach q ∈ V ∪ A do
52             set rhs_q := x_q := ∞;
53         foreach a ∈ A with pre(a) = ∅ do
54             set rhs_a := x_a := cost(a);
55         foreach p ∈ S do
56             rhs_v := 0;
57             AdjustVariable(v);
58     else
59         foreach v ∈ (s \ s') do
60             rhs_v := 0;
61             AdjustVariable(v);
62         foreach v ∈ (s' \ s) do
63             rhs_v := min_{a∈A|v∈add(a)}[cost(a) + x_a];
64             AdjustVariable(v);
65     SolveEquations();
66     set s' := s;
67     return 1/2 Σ_{v∈G} x_v;
```

4
Evaluation

Now it is time to see how PINCH stacks up against its non incremental counterpart Generalized Dijkstra by comparing the two algorithms in a variety of domains.

4.1 Summary of Methods

For our implementation we have used the Fast Downward Planner. The search component of the planner is written in c++. For our evaluation we compare (improved) PINCH and GD. Both algorithms use a bucket based priority queue.

For the comparison we will use weighted A* with a weight of 2. The algorithms are compared on a big collection of domains (Satisficing Track of IPC 1998-2018) with in total 2542 instances. The instances induce state spaces with action costs > 0. We track a number of factors such as coverage, search time and different types of errors.

The experiments were done at the center for scientific computing (sciCORE) in Basel.

4.2 Evaluation

4.2.1 General Overview

I want to start the evaluation by giving a basic picture of what the results are like.

Results		
Property	GD	PINCH
Number of Runs	2542	2542
Coverage	1663	1630
Search out of Memory	138	229
Search out of Time	696	638
Search Time	1.08	1.84

Table 4.1: Summary of the Results, Search Time is the mean Search Time in seconds over all Runs

Let us discuss the Results that we can see in Table 4.1. The coverage, meaning the number of instances for which a plan was found, is comparable between the 2 algorithms. GD covers a slightly larger amount, which will make sense as we continue to explore the evaluation. Search out of Memory and Search out of Time tells us how many runs were interrupted either due to memory or time constraints, the planning time was restricted to 30 minutes and the memory to 3584Mib. PINCH runs out of memory more often than GD, this is likely due to the priority queue storing both variables and actions in PINCH, whereas GD only stores variables.

Perhaps the most interesting property is the Search Time. GD, on average, finds plans about 1.7 times more quickly than PINCH.

4.2.2 Search Time

We have just seen that GD outperforms PINCH on average by a significant margin. Figure 4.1 shows us how GD and PINCH compare in terms of search time. The further away an entry is from the diagonal line, the bigger the disparity for that entry between the two algorithms. Here we can see that, while GD outperforms PINCH on average, PINCH clearly has plenty of instances in its favour. To be exact, PINCH outperforms GD on 449 of the 2542 instances. Assuming that PINCH and GD roughly cover the same instances, that would be 449 of the $\sim$1600 covered instances, meaning PINCH outperforms GD on circa one fourth of the covered instances.

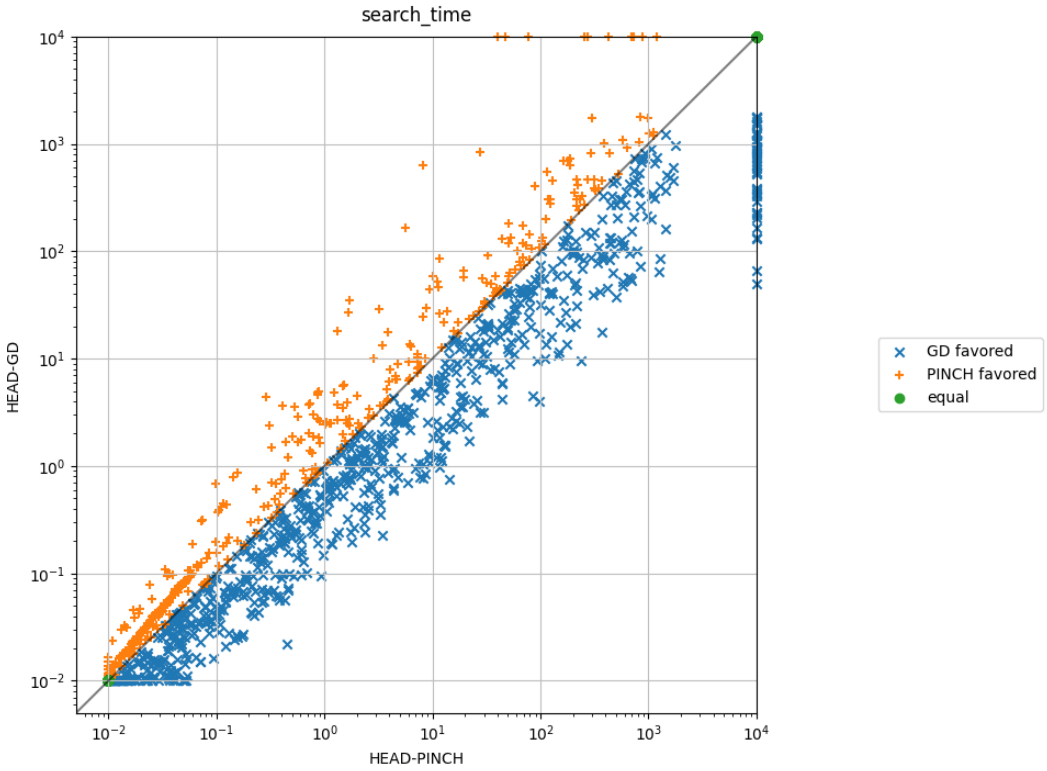


Figure 4.1: Search Time comparison Plot

I now want to explore the reasons for the inconsistent performance of PINCH. In order to do this I have selected several domains from the set of domains in which PINCH either performs exceptionally well, about the same as GD or significantly worse than GD (Table 4.2). The PINCH and GD favored domains were chosen based on the number of instances in the domains and the degree to which either algorithm outperforms the other one. The PINCH and GD favored domains are the 4 *best* domains for either algorithm when comparing mean search time. The domains for the Equal grouping were chosen based on the number of instances in the domains and the similarity in performance between PINCH and GD. The domains in the Equal grouping are the 3 most similar domains between PINCH and GD when comparing mean search time.

Domain Groups		
PINCH favored	Equal	GD favored
satellite (SAT)	miconic-fulladl (MIC)	freecell (FRE)
logistics98 (LOG)	caldera-sat18-adl (CAL)	pipesworld-notankage (PIP)
woodworking-sat11-strips (WOO)	schedule (SCH)	spider-sat18-strips (SPI)
maintenance-sat14-adl (MAI)		agricola-sat18-strips (AGR)

Table 4.2: grouping of domains according to search time performance

Going forward we will often use the abbreviations that you can see in parenthesis in Table 4.2 to refer to the domains. We will also use the colors from Table 4.2 (green/blue/red) to show which domain is part of which group. If we plot only the instances from the domains from Table 4.2 the plot looks as follows:

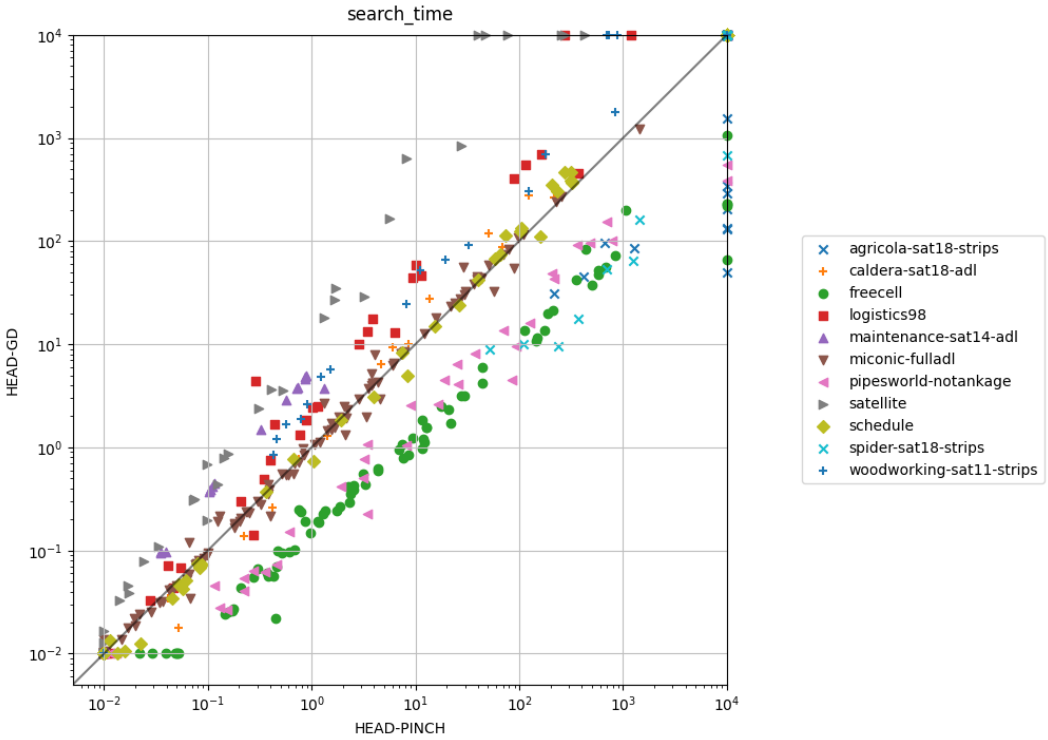


Figure 4.2: Search Time comparison Plot for the instances from the domains from Table 4.2

Interesting to note is that the instances in the PINCH favored and GD favored groupings consistently outperform and have a tendency to outperform more strongly with increasing size. The instances from the Equal group tend to favor GD on small instance sizes and slowly start to favor PINCH with increasing size.

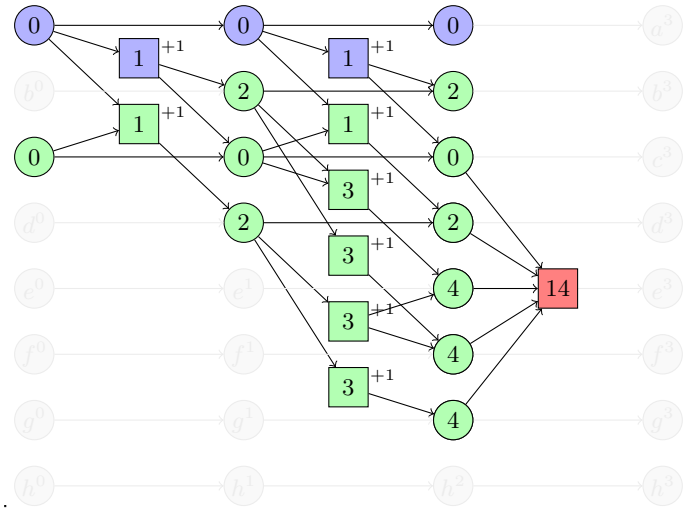
Now that we have selected our domains, we will analyze these domains for information that might explain the performance of PINCH.

4.2.3 The Incremental Benefit

The main difference between GD and PINCH are the incremental calculations that PINCH uses. In order for us to quantify how big the incremental benefit is that PINCH can derive from a given instance, we have to come up with factors that are related to the incremental benefit.

4.2.3.1 Introducing the Factors

I want to explain the factors that I use to quantify the incremental benefit by demonstrating them on our example from chapter 3.2. Remember that the final result of the Example from chapter 3.2 looks as follows:



$h^{\text{add}}(s) = 14/2 = 7$

new state s		
q	x_q	rhs_q
a	0	0
b	2	2
c	0	0
d	2	2
e	4	4
f	4	4
g	4	4
a_1	1	1
a_2	1	1
a_3	3	3
a_4	3	3
a_5	3	3
a_6	3	3

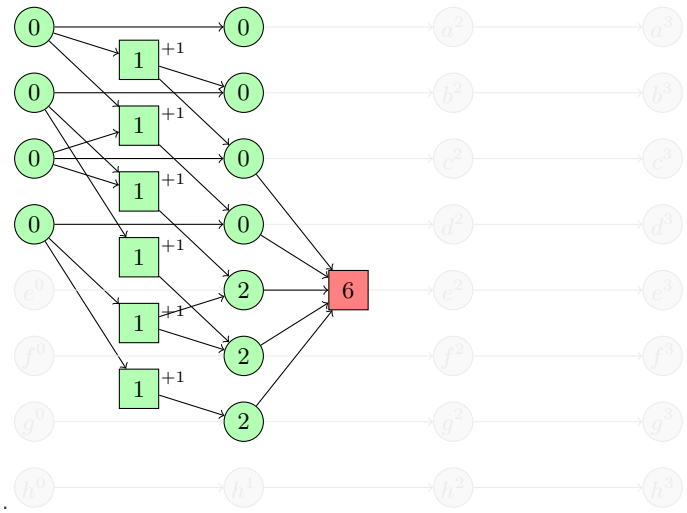
The blue nodes tell us which q did not have to be recalculated and were taken over from s'. In this example s and s' were set as follows:

- $s' = I = \{a, b\}$
- $s = \{a, c\}$

Now lets consider a new example with the same state space from chapter 3.2 but with:

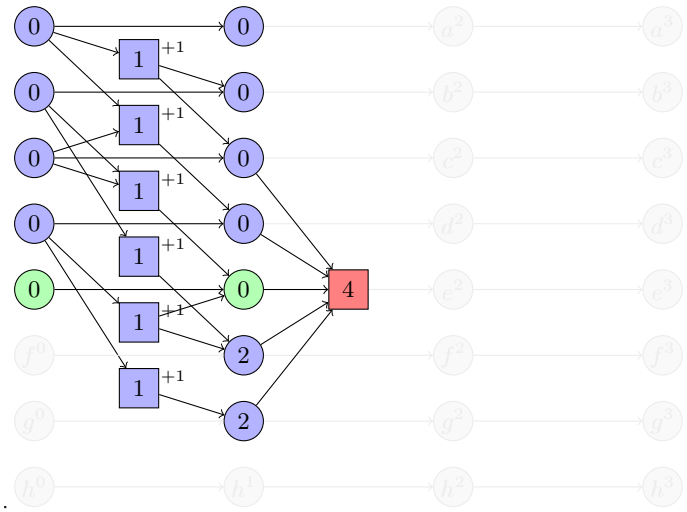
- $s' = \{a, b, c, d\}$
- $s = \{a, b, c, d, e\}$

The final result for s' in this instance looks as follows:



old state s'		
q	x_q	rhs_q
a	0	0
b	0	0
c	0	0
d	0	0
e	2	2
f	2	2
g	2	2
a_1	1	1
a_2	1	1
a_3	1	1
a_4	1	1
a_5	1	1
a_6	1	1

The final result for s in this instance looks as follows:



new state s		
q	x_q	rhs_q
a	0	0
b	0	0
c	0	0
d	0	0
e	0	0
f	2	2
g	2	2
a_1	1	1
a_2	1	1
a_3	1	1
a_4	1	1
a_5	1	1
a_6	1	1

As we can see the incremental benefit of PINCH appears to be much larger than in the previous example, I derive the following performance factors from this discovery:

Factor 1: *PINCH benefits from state s and s' being as similar as possible.*
Factor 2: *PINCH benefits from state s and s' including a large number of variables in relation to the total number of variables*

A further hypothesis we can make is that PINCH is more likely to benefit from its incremental aspects if the number of preconditions for most actions is low. You can see this in this example by looking at a_1 and a_2. For a_1 it is sufficient to reach variable a, so if a is part of s and s' then we will not have to recalculate a and a_1 for s. For a_2 we need to reach a and c, meaning that it is more likely that we will have to recalculate a_2 rather than a_1 for s.

Factor 3: *PINCH benefits from actions having a low number of preconditions.*

Perhaps for similar reasons as Factor 3, an additional hypothesis we can make is that PINCH is more likely to benefit if the ratio of variables and actions tends to favor the former. A lot of actions, which in turn all have varying amounts of preconditions, tend to make it difficult for PINCH to derive its incremental benefit.

Factor 4: *PINCH benefits from there being a high number of variables in relation to the number of actions.*

4.2.3.2 Results

To test for Factors 1 - 4 we have gathered information alongside our test runs. For Factor 1 we compare the similarity of states s and s' by tracking how many $v \in s' \cap s$ are present. We track this information for every s / s' combination we encounter during search. We then calculate the mean number of $v \in s' \cap s$ and finally divide this number by the mean number of $v \in s$. The output is a percentage number that tells us how many variables state s and s' share on average in relation to the total number of variables state s contains. The closer the number is to 1, the better it is for PINCH.

For Factor 2 we track the same information that we track in Factor 1, mainly the mean number of $v \in s' \cap s$, but this time we divide by the total number of v. The output is a percentage number that tells us how many variables state s and s' share on average in relation to the total number of variables. The closer the number is to 1, the better it is for PINCH.

For Factor 3 we simply calculate the average number of preconditions an action has, this can be done once at the beginning of the computation. The output is a decimal number that tells us the average number of preconditions per action for the given instance. The lower the number, the better it is for PINCH. For Factor 4 we divide the number of variables by the number of actions. The output is a decimal number that tells us the ratio of variables and actions for the given instance. The higher the number, the better it is for PINCH

For all factors the results are averaged over the total number of instances each domain contains. We can see the results in Table 4.3.

Factor 1-4 Results											
	SAT	LOG	WOO	MAI	MIC	CAL	SCH	FRE	PIP	SPI	AGR
Factor 1	0.96	0.95	0.97	0.99	0.92	0.95	0.95	0.90	0.94	0.98	0.96
Factor 2	0.23	0.06	0.35	0.49	0.41	0.47	0.47	0.21	0.47	0.33	0.40
Factor 3	0.54	0.87	1.60	0.5	1.85	8.24	1.64	1.72	1.32	4.41	2.95
Factor 4	0.017	0.183	0.074	0.365	0.092	0.002	0.243	0.006	0.06	0.009	0.001

Table 4.3: Green domains: PINCH favored. Blue domains: Equal. Red domains: GD favored

At first glance none of the factors appear to explain the search time performance by themselves. Especially for Factor 1 and 2 there are domains from all groupings with strong results, Factor 3 and 4 appear to be more informative. In Table 4.4 I have colored all entries according to their relation to the mean of Factors 1-4 over all domains. A green entry is *better* than the mean, a red entry is *worse*, where *better* and *worse* refer to the impact of the factor on PINCH (Factor 1 and 2: better closer to 1. Factor 3: better closer to 0. Factor 4: better closer to ∞). The mean for Factors 1-4 is in that order: (0.89,0.29,1.65,0.015).

Factor 1-4 Results											
	SAT	LOG	WOO	MAI	MIC	CAL	SCH	FRE	PIP	SPI	AGR
Factor 1	0.96	0.95	0.97	0.99	0.92	0.95	0.95	0.90	0.94	0.98	0.96
Factor 2	0.23	0.06	0.35	0.49	0.41	0.47	0.47	0.21	0.47	0.33	0.40
Factor 3	0.54	0.87	1.60	0.5	1.85	8.24	1.64	1.72	1.32	4.41	2.95
Factor 4	0.017	0.183	0.074	0.365	0.092	0.002	0.243	0.006	0.06	0.009	0.001

Table 4.4: The green entries perform better than the mean, the red entries perform worse than the mean, where better and worse refer to the impact on search time performance of PINCH

If we consider Table 4.4, we see a correlation between Factors 3 and 4 and the search time performance of PINCH. Figure 4.3 and 4.4 contain plots with domains whose Factor 3/4 is better than average or significantly better than average.

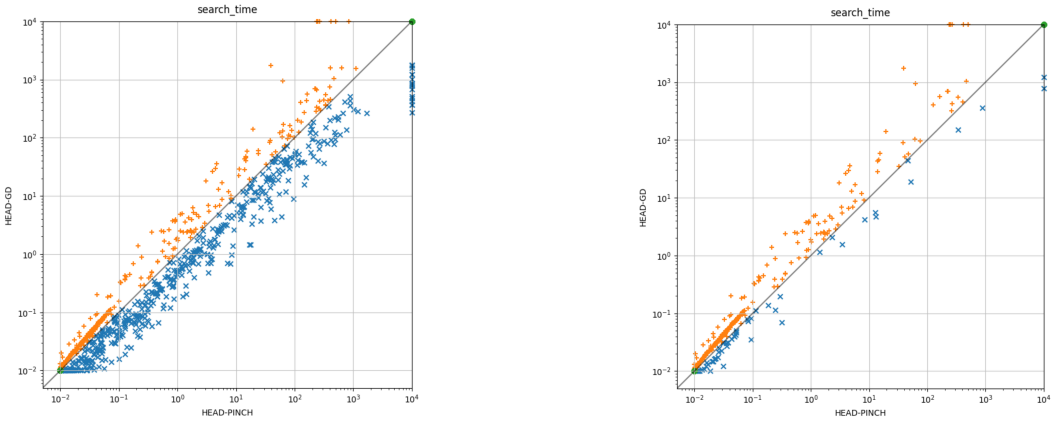


Figure 4.3: Search time plot with domains whose Factor 3 is less than 1.65 (left), less than 1.1(right). **Legend** Orange: PINCH favored, Blue: GD favored

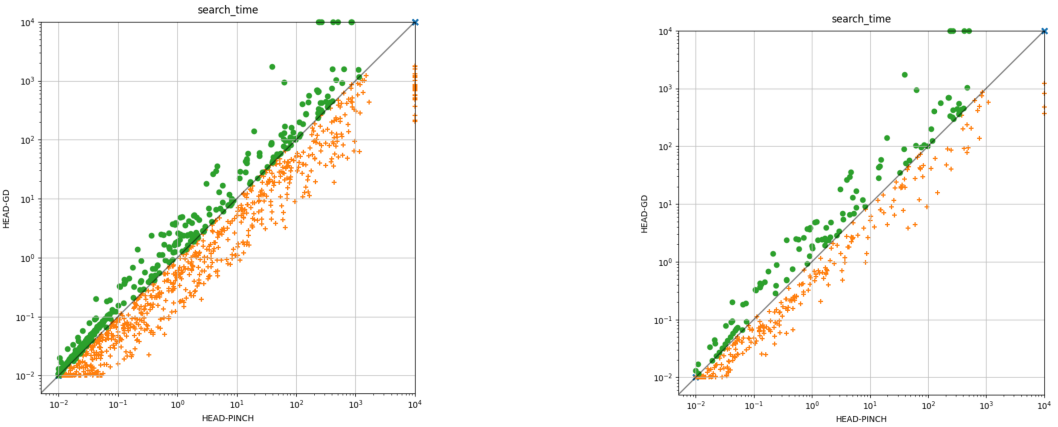


Figure 4.4: Search time plot with domains whose Factor 4 is more than 0.015 (left), more than 0.2 (right). **Legend** Green: PINCH favored, Orange: GD favored

As we continue to raise the standard for Factors 3 and/or 4 the performance of PINCH improves when compared to GD. There appears to be a strong correlation, in Figure 4.3 on the right picture PINCH beats GD by a factor of ∼3.3, the issue is that only a small number of domains contain the desired properties under which PINCH performs best. In Figure 4.5 we plot domains whose Factor 1/2 are *better* and significantly *better* than the mean.

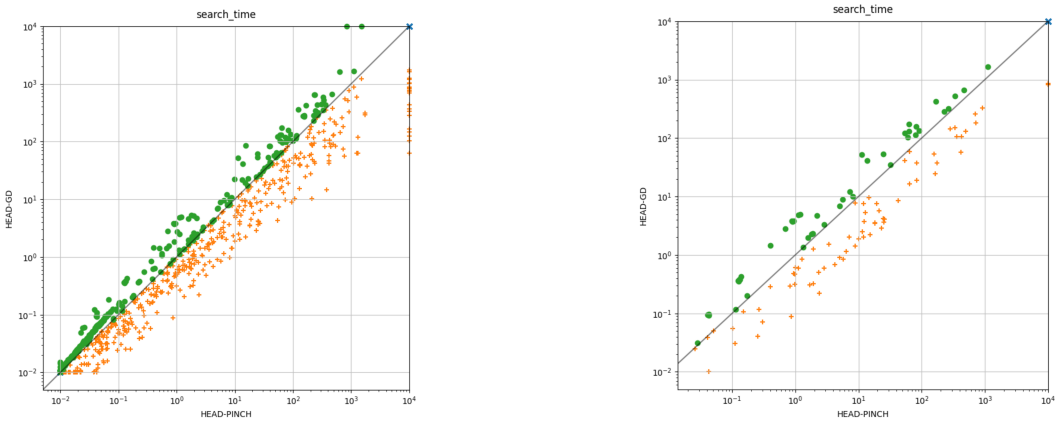


Figure 4.5: Search time plot with domains whose Factor 1 and 2 is above 0.9/0.29 (left), above 0.98/0.4 (right). **Legend** Green: PINCH favored, Orange: GD favored

As Table 4.4 suggested, insisting that all domains must be better than the mean in Factors 1 and 2 has not significantly improved the Performance of PINCH. If we raise the requirements to extreme amounts (Figure 4.5, right picture) we do get a slight improvement, with GD beating PINCH in roughly 2/3 of the covered domains, but we loose so many domains that it is hardly a good indicator.

4.2.3.3 Conclusion

Of the 4 factors that we have hypothesised would benefit the incremental aspect of PINCH and therefore PINCH as a whole, only 2 proved to have strong evidence in their favor. Perhaps the example, that was used in Chapter 4.2.3.1 from which Factor 1 and 2 were derived was too much unlike the types of state spaces that most instances induce. Factor 3 and 4 are clearly correlated with the performance of PINCH. We can conclude this since choosing domains that perform well in those factors drastically improves the performance of PINCH when compared to GD. What is especially interesting is that Factor 3 and 4 can be calculated prior to choosing a h^{add} implementation, meaning that, if Factor 3 and 4 both appear to be *good*, one could choose PINCH for the computation, and GD or a different h^{add} implementation otherwise. Further research is needed to say when exactly PINCH reliably outperforms GD given Factor 3 and 4. In my runs PINCH started to *beat* GD in more then 50% of the covered domains roughly at around 1.2 for Factor 3 and 0.3 for Factor 4 but these numbers are based on a small sample-size. I have also not considered the positive effects of having both factors above a certain threshold, I only looked at each factor in isolation.

4.2.4 Comparing PINCH and GD Directly

We can derive further factors if we compare PINCH and GD directly.

4.2.4.1 Introducing the Factors

One thing we can compare is how many times PINCH has to adjusts the cost values of all $q \in V \cup A$. As mentioned before, PINCH updates the cost value of a given q at most twice, while GD updates the cost value of all $q \in V \cup A$ exactly once. In the example from chapter 3.2, PINCH updates cost values 12 times. GD respectively would update cost values 13 times when applied to the same example.

Factor 5: *The algorithm that adjusts fewer cost values has an advantage.*

Factor 5 does not take into consideration that PINCH and GD update cost values differently. PINCH updates exactly one cost value per $q \in V \cup A$ that is popped from the priority queue. GD only inserts $v \in V$ into the priority queue, it updates action cost values in the same iteration as it updates variable cost values. For the example from chapter 3.2, PINCH pops exactly 12 q from the priority queue, GD only pops 7 v from its priority queue when processing the same example.

Factor 6: *The algorithm that pops fewer $q \in V \cup A$ from its priority queue has an advantage.*

4.2.4.2 Results

To test for Factor 5 and 6 we simply count how many times on average for a state $s \in S$ PINCH and GD adjust their cost values or pop q from their priority queues.

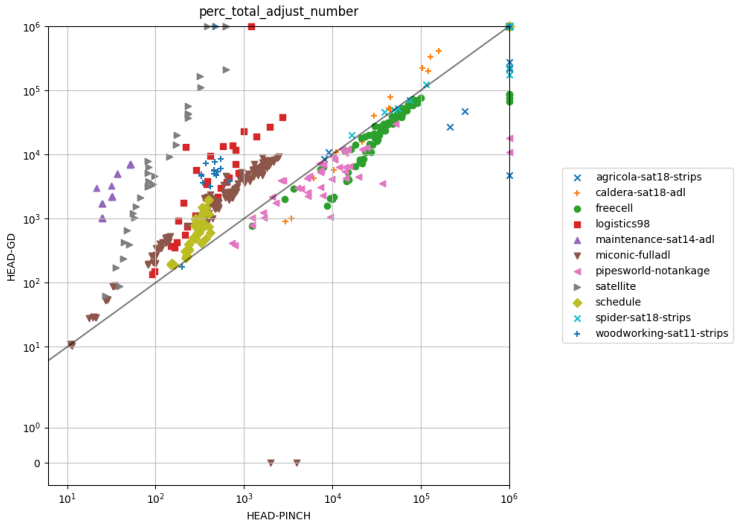


Figure 4.6: total number of cost adjustments comparison plot between PINCH and GD

Figure 4.6 shows us how many times PINCH adjusts its cost values on average compared to GD. PINCH updates its cost values at most twice, GD exactly once. In Figure 4.6 we can see that, thanks to the incremental benefit that PINCH has, PINCH tends to updated its cost values less than GD. Even for the GD favored domains the number of times cost values are adjusted are within a similar range.

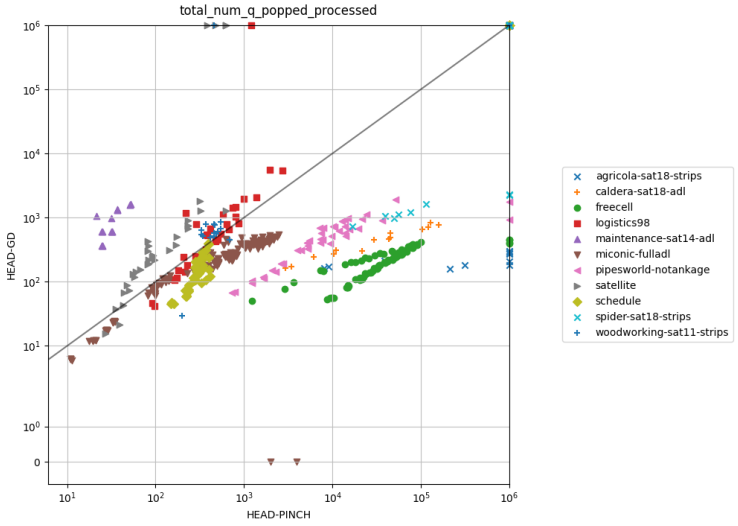


Figure 4.7: total number of variables/actions popped out of the priority queue comparison plot between PINCH and GD

Figure 4.7 shows us how many times PINCH pops a variable or action out of its priority queue compared to GD. Since GD only stores variables in its priority queue it is expected that it pops fewer variables out of its priority queue. Even for domains that are PINCH favored GD tends to compare quite well.

4.2.4.3 Conclusion

It is interesting, that Factor 5 appears to indicate that PINCH is the superior algorithm while Factor 6 does the opposite. While Factor 5 might be a good argument in favor of the benefits of PINCH, Factor 6 paints a different picture. Even though PINCH updates cost values less, it only achieves this by maintaining a fairly expensive priority queue. PINCH is required to store both variables and actions, in the worst case scenario it is required to insert and pop each of them twice. According to my testing maintenance of the priority queue is the single most time intensive part of PINCH, GD has an immense advantage here. The average number of q popped form the priority queue for PINCH is 15519, for GD it is 432. Comparatively the average number of cost adjustments for PINCH is 15502, for GD it is 15701. Even tough PINCH does update cost values less, the way it achieves this comes at a huge price.

5
Conclusion

The goal of this thesis was to give an in depth and intuitive explanation of PINCH. Additionally I have expanded PINCH beyond unit cost and tested PINCH against its non incremental counterpart GD. In my testing I have identified factors that indicate the performance of PINCH.

While I am slightly disappointed that PINCH did not perform as well as I had expected, I believe it is exactly the lackluster performance that motivated me to really dig into PINCH and figure out its strengths and weaknesses. Even though PINCH performs worse than GD on average, I have identified one group of domains for which PINCH is more likely to outperform GD. That group being domains with a low average number of preconditions per action and/or a high ratio of variables in relation to actions.

The biggest hindrance that PINCH unfortunately fails to overcome is the overhead associated with its priority queue. If I were to try to improve the algorithm my first instinct would be to adjust the priority queue such that it only stores variables like GD. However, I do not know if it is possible to make that adjustment and keep all or most of the good properties of PINCH.

It is promising that, despite its priority queue, PINCH still manages to beat GD in a good number of the covered instances. Even in its current form PINCH should prove useful if used for domains with the desired properties. While GD may have come out victorious at the end of my evaluation, I hope to have provided good enough reasons to argue for the benefits of PINCH. If the issue with its priority queue could be resolved, I believe that PINCH could be a fantastic option for the computation of h^{add}.

Bibliography

[1] Blai Bonet and Héctor Geffner. Heuristic search planner 2.0. *AI Magazine*, 22(3):77–77, 2001.

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

[3] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

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

[5] Yaxin Liu, Sven Koenig, and David Furcy. Speeding up the calculation of heuristics for heuristic search-based planning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI 2002)*, pages 484–491, 2002.

[6] Ganesan Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Daniel Weissen

Matriculation number — Matrikelnummer
17-065-723

Title of work — Titel der Arbeit
Exploring The Prioritized Incremental Heuristic

Type of work — Typ der Arbeit
Bachelor thesis

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

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

Basel, Oktober 12, 2020

Signature — Unterschrift