Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option Artificial Intelligence and Semantic Web

PDDL4J: Test and Analysis of Algorithms and Heuristics

Samuel Aaron Boyd

23rd June 2016

Research project performed at Laboratoire d'Informatique de Grenoble

Under the supervision of:
Damien Pellier and Humbert Fiorino, LIG Labs: Team MAGMA

Defended before a jury composed of:
Prof E. Gaussier
Prof T. Ropars
Prof M. Reza Amini
Prof N. Brauner
Prof J. Euzenat
Prof A. Demeure

June 2016

Abstract

PDDL4J is an open source library that is used for finding solutions to classical planning problems. The main way that this is achieved is by using a search algorithm and a heuristic to guide the search. From research conducted we discovered that were better search algorithms that could be used instead of the current A* algorithm. What we found was that implementing a new search algorithm provided more problems solved overall as well as completing the problems in a faster time and using less memory, as well as discovering some interesting anomalies between admissible and not admissible heuristics. Within this thesis we implemented Greedy Best First Search with Enforced Hill Climbing search algorithm as well as the H^m heuristic.

Acknowledgements

I would like to thank my family, friends and the best girlfriend ever for helping me get to where I am today. Without them, all of this would not be possible, and I am eternally grateful.

Would also like to thank Damien and Humbert for giving me the opportunity to work within team MAGMA and with the PDDL4J library.

List of Figures

Contents

— 1 —
Introduction

1.1 In the Beginning

What is planning? It can be defined as follows: "Explicit deliberation process that chooses and organises actions by anticipating their outcomes" [31]; in other words, from a predefined initial state we predict the set of actions used to achieve a predefined goal state. There are many ways we can do this for different scenarios within the domain of planning, but this thesis will concentrate on classical planning, which is a subset of the planning domain.

So, what is classical planning? "It refers generically to planning for restricted state-transition systems" [31]. This is the oldest type of planning and is also known as STRIPS programming (Stanford Research Institute Problem Solver). STRIPS was a planner developed at Stanford University in 1971 [41] and its goal was to find a sequence of actions that could prove a goal formula true.

We can class planners in the domain of 'State Transition Systems'. In the example at Figure 1.1 we have multiple blocks and a robot. The robot can pick up and put down the blocks from one location to another. Each picture is a set of states and the arrows represent each new state. We say each state transition is deterministic, meaning that it leads to just one other state. In

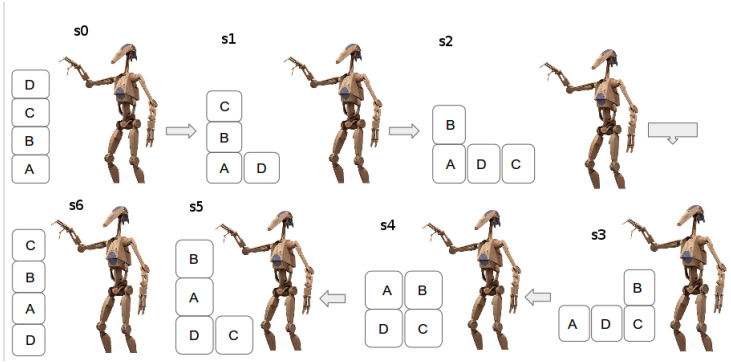

Figure 1.1: Example of state transition, from one to another

Figure 1.1 we have 7 states in total ranging from $s_0, s_1, s_2, s_3, s_4, s_5, s_6$. After an action has been performed it leads us to the next state. Now we can put the classical planning definition into some perspective, for example, we have 4 blocks, labelled A, B, C and D. A robot R1, an initial state onTable(A, B) onTable(C, D) meaning that all 4 blocks are placed on a table, and a simple goal state of on(A, B) on(C, D) meaning that we need to stack $(A \wedge B)$ and $(C \wedge D)$. This is a

simplistic way of explaining the methods involved in planning and how a planner would solve this problem. What a person would do is assess the quickest and most efficient way (to them) to solve. Taking block A and putting it on top of B, and the same for block C and block D. A planner (depending on the algorithm) will create a type of search tree and explore each state by generating successors of already-explored states.

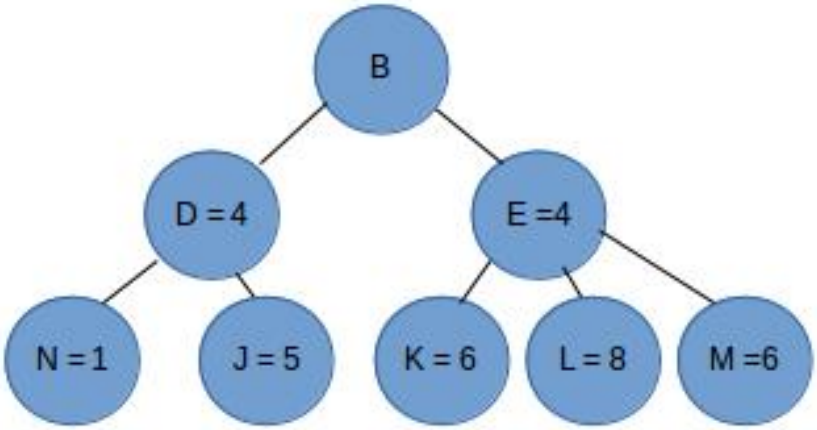

Figure 1.2: Basic example of a search tree with starting node B and goal node N[31]

As you can see in Figure 1.2, if the goal was to find the shortest path from an initial state B to a goal state N, the planning algorithm would expand D and E, and from this would assess that N is a child of D. The algorithm would quickly find the node N in the tree graph and the cost for getting from B to N would be 5.

1.2 The Goal

The goal for this thesis is to evaluate and test the techniques used in the PDDL4J library. In my opinion, the range of algorithms that can be used for planning is very wide and the current approaches are not optimal. In the following sections we will discuss the types of technologies that will be used throughout this thesis. We will start with the PDDL language in which domains and problems are encoded, then I will speak briefly about the PDDL4J library, introduce heuristics and finish with satisfiability problems.

1.3 PDDL

With the basics on planning covered, we can see how it is incorporated into the PDDL (Planning Domain Description Language), which is the main language for classical planning problems and domains. PDDL is used to express what predicates there are, what actions are possible, action structures and what the effects of those actions would be.[10] Each type of problem for example, 'Blocksworld' or 'Mystery', will have a number of problems ranging from easy to hard, as well as a domain. The domain will provide the actions of what can be done, for example picking up a block as in Figure 1.1. The problem file will begin with the objects that tell the planner how many items there are, such as 'objects: A B C D - blocks'; the more

objects, the harder the problem. It will also define an initial state which will let the planner know what state the blocks are in, for example '(onTable B)', and will provide a goal state which the planner must satisfy. The main components of the PDDL represented as a planning task are:

- Objects = The blocks in this case (A, B, C, D)
- Predicates = Properties of objects – these can be true or false. For example: Is A a block?
- Initial state and goal state = The starting state of the world and the goal state of the world. All blocks are on the table and the goal is for them to be stacked in a specific order.
- Operators/Actions = The actions that can be used to complete the task, broken down into preconditions and effects. Each precondition and effect can be positive or negative. ...

Each of the above are split between the domain file and the problem file. The domain handles the predicates and actions while the problem handles the initial state, goal state and the objects.

If we look at the domain we can see the schema for the Blocksworld problem (see Figure 1.3 below). Inside we have some requirements at the top which tells the planner if it's able to solve the problem. It does this by informing the planner of the types of actions it will find within the domain, conditional effects, equality etc. As PDDL is a huge language, there are multiple subsets that have been created because very few planners would be able to incorporate all versions of the PDDL language.

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

Figure 1.3: Example of domain schema from Blocksworld domain

Next the actions associated with that domain will be specified. The actions will reflect the requirements (Figure 1.4) and the planner will use these actions to help it solve the problem. The action in Figure 1.4 is a pick-up action; it tells the planner that it can only pick up one block at one time and, when it is picked up, it is at location x. An action can only be executed when the preconditions of that action are fulfilled; when the execution takes place, this generates effects, i.e. changes within the environment, for example:

1. Block A has been picked up = Action
2. Clear A must be true before the robot can pick it up, therefore, nothing can be on top of block A = Precondition of the action PickUp

3. The negative effect of this action is that the Block A is now not on the table and the robot's hand is not empty. The positive effect of this action is that the robot is now holding Block A = Negative/Positive effects

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

Figure 1.4: Example of an action pick-up from Blocksworld domain

The planner will use actions like these, and others that are coded in a similar fashion, to complete a problem. Below is a problem example which shows the objects (in this case 4), the initial problem stating the blocks that are clear, the blocks that are on the table and that the robot's hand is empty, and then finally the goal state.

```
(define (problem BLOCKS-4-0)
(:domain BLOCKS)
(:objects D B A C - block)
(:INIT (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D) (ONTABLE C) (ONTABLE A)
 (ONTABLE B) (ONTABLE D) (HANDEMPTY))
(:goal (AND (ON D C) (ON C B) (ON B A)))
```

Figure 1.5: Example of a problem from Blocksworld domain

To tie all of the above together, the domain will provide the planner with the actions along with preconditions and effects for that action, and the problem will provide an initial state and a goal state that the planner must reach in order to satisfy the problem.

Now we can define a PDDL task by a 3-tuple $\Pi = (S_0, G_n, O)$ where

- S_0 = Is a finite set of ground atoms called the initial state
- G_n = Is a closed formula called the goal formula
- O = Is a finite set of PDDL actions [18]

The initial state and goal state are the same as previously explained but, as the initial state begins at 0 and the goal state at n, we are not sure where it is within the sequence. The axioms help us define predicates based on basic predicates. As seen by the example "given an *ontop* predicate, we can define its transitive closure *above* with the two axioms $above(x,y) \leftarrow ontop(x,y)$ and $above(x,z) \leftarrow \exists y(ontop(x,y) \wedge above(y,z))$" [18] With axioms like this, we ensure the evaluation is well defined and contains specific rules such as "ontable(A) is true when ontable(A) is actually false".

1.4 PDDL4J

As discussed earlier, PDDL4J was created by Damien Pellier. The main components of the planner break down into three broad categories:

- Preprocessing
- Search algorithm
- Heuristics

Using these three categories in unison creates a well-formed planner. The preprocessing side will take in all the facts from the domain of a PDDL problem and change each action condition using the variables, such as Block A, Block B etc.

```
(:action pick-up
:parameters (A - block)
:precondition (and (clear B)(ontable B)(handempty))
:effect
(and(not(ontable A))
```

In each problem, the objects are known from the beginning, so the preprocessing side of the planner will instantiate each operator with all possibilities. As well as, this there are a number of steps that need to taken into account before the final representation is presented to the search algorithm and heuristic. When the domain and problem are encoded, logical simplification is performed to convert the problem to a bitset representation as the final outcome.

There are a number of different planning algorithms that can be used to find a solution to a problem; the PDDL4J library uses an algorithm called A*. The main idea behind A* is that it searches through all possible paths to the goal to find the shortest path. It is an iterative algorithm and, after each iteration, it decides which of the nodes to expand of the partial plan it has already made until it reaches the goal and computes the fastest route. It does this by using the simple function $f(n) = g(n) + h(n)$ [39] where

- n = the last node in the path
- $g(n)$ = the cost of the path from the start to n
- $h(n)$ = the heuristic that estimates the cost from n to the goal

The A* algorithm will use the function above and then try to find the best and less costly route. It does this by examining the vertex n that has the lowest $f(n)$ score. For example, if the cost from getting from a starting node A to a goal node D had two routes, one giving a total cost of 4 and another 6, A* will always choose the route with minimal cost. But A* cannot work on its own to find a solution to a problem, it needs to use a heuristic to assist it and guide it to the goal.

The role of a heuristic is to help look for an answer to a particular problem. It does not tell the algorithm what to find but only how to look for a potential solution. Within the PDDL4J library and with the A* algorithm, each heuristic plays a different role when used to help find a solution but will provide the same basic level of help, which is estimating the distance to the

goal.

In the context of mathematics, the Manhattan distance is the distance between two points measured along axes at right angles[6]. It can be written as $|x_1 - x_2| + |y_1 - y_2|$ when we have two points p_1 at (x_1, y_1) and p_2 at (x_2, y_2). This type of function can be used with the A* algorithm in the form of a heuristic to find the shortest path from p_1 to p_2. If we translate that into a heuristic function in the Java programming language, it will look more like this:

```
int distance = Math.abs(x1 - x2) + Math.abs(y1 - y2);
```

If we wanted to incorporate the function above with A*, we would create a small algorithm that would increase the value of g by *distance* and decrease h by *distance* as we move closer to the goal. So, regarding, the A* function *f(n)*, the A* function should have the same value as the *distance* function defined above. If the two match then we would say that this heuristic is "admissible".

Within the world of heuristics, two definitions exist: "admissible" and "non admissible". What is an admissible heuristic? We can say a heuristic h is admissible if $h(s) \leq h^*(s)$ for all states s. In other words, when estimating the distance from a starting node to a goal node, it does not overestimate. Then for a non admissible heuristic, there is a possibility that it will overestimate or underestimate the distance. Within the PDDL4J library there are two admissible heuristics, H^+ and H^1, and, H^m but these will be discussed later.

1.5 Satisfiability Problem

In the previous subsections, we looked at PDDL and how it works. This section and one in Chapter 2 dedicated to a set of problems termed satisfiability (SAT) problems. These are a set of expressions which are boolean, which means there are only two possible outcomes, i.e. either true or false. It is expressed in propositional logic and uses the operators AND(conjunction, $\wedge$), OR(disjunction, $\vee$) and NOT(negation, $\neg$). We say a problem is satisfiable if all the variables can be set to true or false. There is a particular structure that needs to be kept when working with satisfiability problems – we will look at the notation briefly.

- A literal can be positive or negative – p_1 is positive and $\neg p_2$ is negative.
- A clause is a disjunction of literals.
- A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses.[29]

As stated above, we have p_1 and $\neg p_2$. $p_1 \vee \neg p_2$ is a clause and $(p_1 \vee \neg p_2) \wedge (\neg p_1 \vee p_2 \wedge p_3) \wedge \neg p_1$ is a formula in conjunctive normal form. If we set p_1 = false, p_2 = false we get $(F \vee T) \wedge (T \vee F \vee p_3) \wedge T$
(p_3 is arbitrary which means it is not assigned a value)

Using CNF rules[34], we can deduce the above formula as $T \wedge T \wedge T$ = True. As this formula has been classed as true because of the rules applied in propositional logic, we begin to see how it can be useful in terms of planners and solving complex problems.

— 2 —
Literature Review

2.1 All About Research

Within this chapter we will discuss the research undergone that has helped to reach the conclusion of the overall hypothesis. This will be discussed later on in the chapter. The beginning will be a brief overview and foundation of research, followed by research papers grouped by ideas, a model of processes carried out and the hypothesis of this paper.

2.2 PDDL

The research undergone for PDDL was based around the language as there are multiple versions that have been released over the years since it was first introduced in 1998 for the first IPC (International Planning Competition) competition[35].

- 1998: STRIPS and ADL
- 2000: STRIPS and ADL subset
- 2002: PDDL2.1 Numeric and temporal planning
- 2004: PDDL2.2 Derived predicates and timed initial literals
- 2006: PDDL3 Soft goals and trajectory constraints

We have discussed STRIPS and ADL in Chapter 1; the next version of PDDL incorporated numeric and temporal planning which was broken down into two subsets, PDDL 2.1 level 2 was the introduction of numeric fluents and PDDL 2.1 level 3 which was level 2 plus action duration. Numeric fluents(actions) allows the addition of integers within the conditions of an action[33]. If we need to compare these numeric fluents, it is only possible between other numeric fluents. The effects of an action can be changed in such a way as to incorporate an increase or decrease in these numeric values, for example:

```
(:types bottle)
(:functions
(amount ?b - bottle
(capacity ?b - bottle)
```

```
(fluent number))
(:action empty
:parameters (?b1 ?b2 - bottle)
:precondition (>= (capacity ?b1) (amount ?b2)
:effect (and (change (amount ?b1) 0)
change (amount ?b2)
(+(amount ?b1)(amount ?b2))
```

With this problem we have two bottles and some liquid which we use as the numeric fluent. The goal is to take the liquid from the first bottle and pour it into the second bottle but only if the second bottle has the *capacity* to hold the amount of water from the first bottle. We use the word *change* to update the numeric fluent of each bottle. You can see that the effect of the change in contents from bottle 1 to bottle 2 will result in bottle 1 having the numeric fluent value of *0* as there is no liquid left after the action *empty* has completed.

A temporal action can be seen as a durative action as a way of modelling the temporal properties of a planning domain[33]. Durative actions can be broken down into discretised and continuous actions. One way it can be expressed is with *duration*. Duration can be applied at the start of an action, in the middle of an action or at the end of an action, so it needs to be placed in the correct area in the domain and should read accordingly.

```
:duration (= ?duration 10)
```

An example of the syntax is above and this shows that an action will cost a duration of 10 (could be seconds, minutes, hours etc).

The next version of PDDL, PDDL 2.2, was introduced by Stefan Edelkamp and Jorg Hoffman[43]. They created the term *derived predicates* meaning that the predicates do not get effected by the actions. So a set of derivation rules are used instead in the form of:

IF $\varphi(\bar{x})$ **THEN** $P(\bar{x})$[23]

P is the predicate that can be derived with the vector variable $\bar{x}$ so then instances of P can be true or false.

Timed initial literals are used to express a restricted form of externally derived events. In other words, at a certain time point in the plan, a fact will become true or false. These times are known to the planner from the beginning in the initial state. As PDDL is moving towards real life events and how to take into account events that do not start out being true but then change as time moves on, using timed initial literals gives that impression. Let's say we have a queue of customers waiting to pay for their shopping with one sales assistant working on a checkout. In the beginning, we have a group of people and one checkout operator. We can say that in the beginning, checkout operator >1 = false. But at some stage, another checkout operator arrives so there are two checkouts now open to serve the customers. Checkout operator >1 has now changed to true.

PDDL 3 was released in 2006 and introduces the concept of strong and soft constraints on plan trajectories[3]. The proposition of soft constraints and goals are not required to be completed by the planner to have a valid plan but are more of a desire. A strong constraint or goal requires the planner to satisfy these requirements for a plan to be valid. With regards to soft constraints, a planner should try to satisfy as many of these as possible as it gives better plan

quality, however, for strong constraints, these must be met by the planner. The soft constraint aspect can carry a penalty for not satisfying one of these goals/constraints. In other words, when a planner starts to solve a problem in PDDL 3, it must check first for the soft goals/constraints and if not all of them are achievable then the planner must select the costliest subset of soft goals to complete in order to minimise the penalty. This can be extremely difficult as there may be multiple soft goals/constraints that carry different penalty weights and even more so if the planner has a time limit in which to solve the problem. There are many ways to express soft and hard goals/constraints but see below as an example:

```
(forall (?p - players)
(preference ScoreGoal
(sometime (at ?x Score)))
```

This could represent a football (soccer) problem where every player on the team must score a goal. This would enforce a penalty for the total number of players that did not score a goal during the plan. We can set this penalty at any cost for each player that did not score. For instance, a striker (forward player) not scoring would have a penalty of 5 because that is his role but a defender would have a penalty of 1 because that is not his role within a game of football.

Now we can look at state trajectories that could be set as soft or strong constraints in the Blocksworld problem.
"Each block should be picked up at least once"

```
(forall (?b - block) (sometime(holding ?b)))
```

This can be set as a hard or soft constraint depending on the requirements of the problem. One way to complete this task for a planner could be to solve the problem with the required blocks by remembering which were used, then pick up the remaining blocks and put them back down after the hard goals/constraints have been satisfied.

As it is extremely difficult for a planner to incorporate all of the PDDL, in each research paper they use what is called the benchmark for classical planning. This is the recommended set of problems that have been used at the IPC competitions over the past number of years[26]. For this thesis, STRIPS and ADL have been the choice for testing algorithms and generating results. It will be IPC 1-8, each with 5 domains per ICP, as this will enable fair comparisons to previous work done with STRIPS and ADL to be carried out.

2.3 Classical Planning Algorithms

In this section, we will discuss the two types of algorithms (forward and backward search) in terms of planning. In state-space planning, we have forward and backward search. In plan space planning we search through a space of partial plans to find a solution. SAT planning turns the problem into a propositional satisfiability problem and given to a SAT solver. Finally we have HTN (Hierarchical Task Network) which is more of an approach rather than a technique of planning, but for this thesis we will focus on state-space planning for classical planning representation.

2.3.1 State-Space Planning

Within state-space algorithms for planning, the main focus is to create a subset of the state space termed the search space. We can break it down into 3 sub-categories, i.e. each node corresponds to a state of the world, each arc corresponds to a state transition and the current plan corresponds to the current path in the search space[31]. We then have three types of algorithm that we can implement for state-space planning: forward search, backward search and a combination of the two. We are able to search in a forward or backward way because of the preconditions and effects.

Forward Search

Best First Search (BFS) is an algorithm that came to light in 2003. The algorithm was developed from the *general search* algorithm described in the book 'Artificial Intelligence: A modern approach'. We can edit this general search algorithm to enforce a best first search algorithm by providing a knowledge area on the queueing function[44]. The knowledge gained from the queueing function allows us to determine the next node to be expanded. So the general search algorithm can be easily changed to incorporate a knowledge-type function which evaluates nodes, and the node with the best evaluation is expanded first. Once we have the BFS algorithm, we can then incorporate a heuristic function $h(n)$ which estimates the cost of the least expensive path from the state n to the goal state. We use this heuristic function to help select the next node to expand but it is not always the best node overall. When we use a heuristic with BFS, it changes to Greedy Best First Search (GBFS). We consider the algorithm to be susceptible to false starts[27]. As an example, say we have a starting node A and we must get to the goal node D. In a tree-like approach, the GBFS algorithm will look for a direct route to the goal and, instead of expanding the two children of A which are B and C, it will only expand one, which could lead the algorithm to a dead end. This means that the algorithm can get stuck and break or we need to introduce a method that will detect repeated states. So if the algorithm goes back to A, it does not go back to B, but instead expands C. Another example of how the algorithm works is shown below. In Figure 2.1 the goal is to find the highest cost in the tree, i.e. expand the nodes and generate the highest score.

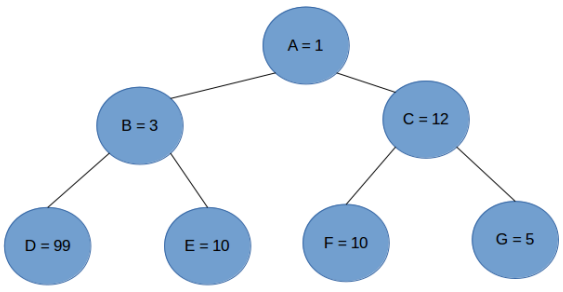

Figure 2.1: GBFS algorithm trying to find the highest score from the nodes

The GBFS algorithm will work by selecting the nodes with the highest cost due to the goal that has been set. The algorithm will select *node C* instead of *node B* because 12 is higher than 3. It will then select *node F* because 10 is higher than 5. The algorithm works in such a way that it will never reach the highest scoring node, *node D*. As you can see it works by selecting the

optimal immediate choice instead of expanding both nodes *B and C* and checking the successor.

Using a forward search algorithm means that we will start the search from an initial state and use actions within the domain in a certain sequence until a goal state is reached. One of the most popular and well used algorithms that works in a forward search manor is A*. This is the current algorithm implemented in the PDDL4J planner. It was created in 1968 and is an extension to the Dijkstra algorithm[39]. As well as being a forward searching algorithm, A* uses heuristics to guide its search towards the goal:

Algorithm 1 A*

$openList = setcontainingstart$
$closedList = emptySet$
$start(g) = 0$
while $(openList$ **is not** $empty)$ **do**
    $current = openList(withlowestfcost)$
    if $current = goal$ **then**
        $constructPath(goal)$
    $remove(current)$ **from** $openList$
    for $eachNeighbour \in currentNeighbour$ **do**
        if $neighbour$ **is not in** $closedList$ **then**
            $add(neighbour)$
        else $openNeighbour = neighbour \in openList$
        if $neighbour = neighbour \in openList$ **then**
            $openNeighbour(g) = neighbour(g)$
            $openNeighbour(parent) = neighbour(parent)$

We will look closely at this algorithm (see Algorithm 1 above) and the use of the weight functions $f(n) = g(n) + h(n)$. This essentially is the use of two types of algorithms, Best First Search and Dijkstra. If $h(n) = 0$ then $f(n)$ becomes $g(n)$ which is Dijkstra's shortest path algorithm. If $h(n) = large\ number$ then $g(n)$ is ignored and it becomes Best First Search. So the use of an admissible heuristic with A* helps to balance out the two functions that make $f(n)$ which in turn creates a direct shortest path algorithm.[45] From this we can see that A* is realistically more of a method than a point A to point B algorithm. Overall, it is a container for a greedy algorithm combined with a heuristic search function.

Enforced Hill Climbing (EHC) is an algorithm that was implemented in a famous planner called Fast Forward (FF) [24]. EHC is an updated version of a generic hill-climbing algorithm. Hill climbing is used in mathematical optimisation in the area of local search. The main function in hill climbing is that it starts with an arbitrary solution and iteratively changes a single element in the original solution; if the new solution is better than the old then the old is replaced and the algorithm will continue to do this until no changes occur in the newest solution. Regarding EHC, what we want to achieve is a sequence of actions that in turn leads to a better successor if, at that moment, none are present. One of the main issues with EHC is that, with a heuristic to guide the search, EHC can get stuck in a dead-end or be unable to escape a plateau, meaning that within the search space, the heuristic values of all successors are greater than or equal to the best nodes already explored[5]. What we would want to achieve when using EHC

is to provide a method to escape the algorithm being stuck and never finding a solution within a specified time limit.

Algorithm 2 Enforced Hill Climbing

```
openList = initialState
bestHeuristic = heuristicValue(initialState)
while (openList is not empty) do
    currentState = popstate from headofopenList
    successors = thelistofstatesvisiblefromcurrentState
    while successors is not empty do
        nextState = removeastate from successors
        h = heursticValue(nextState)
        if nextState = goalState then
            return(nextState)
        if hisbetterthanbestheuristic then
            clear(successors)
            clear(openList)
            bestHeuristic = h
        placenextstateatbackofopenlist
```

In algorithm 2 above, we can see the comparison of the heuristic function h and whether it is better than the best heuristic function available.

Backward Search

A backward search algorithm does exactly what you would expect in that it works in the opposite way to a forward search algorithm by starting from a goal state and using actions in a certain sequence until an initial state is reached. There are many algorithms like A* and Dijkstra that can be used for forward search or backward search.

Algorithm 3 DIJKSTRA

```
dist[s] ← 0
for all v ∈ V − (s) do
    dist[v] ← ∞
S ← 0
Q ← V
while Q ≠ 0 do
    u ← minDistance(Q, dist)
    S ← S ∪ (u)
    for all v ∈ neighbours[u] do
        if dist[v] > dist[u] + w(u, v) then
            d[v] ← d[u] + w(u, v)
return(dist)
```

The distance to source is marked as 0 and we set all others to infinity. We then select the elements of Q with the minimum distance and add u to the list of visited vertices. Following this, we perform a check to see if a new shortest path can be found. If so, set the new shortest path value and return the distance. Dijkstra performed in a backward search will use the same approach and vertices with each arc $w(u,v)$ becomes $w(v,u)$ (see algorithm 3).

2.4 SAT and SAT4J

SAT4J takes in files of CNF format as explained in Chapter 1. SAT4J is a Java library for solving boolean satisfiability and optimization problems[13]. It is able to solve a range of different types of problems and its written in Java. It is open source and uses a pseudo-boolean solver as a universal engine which translates the problems[13].

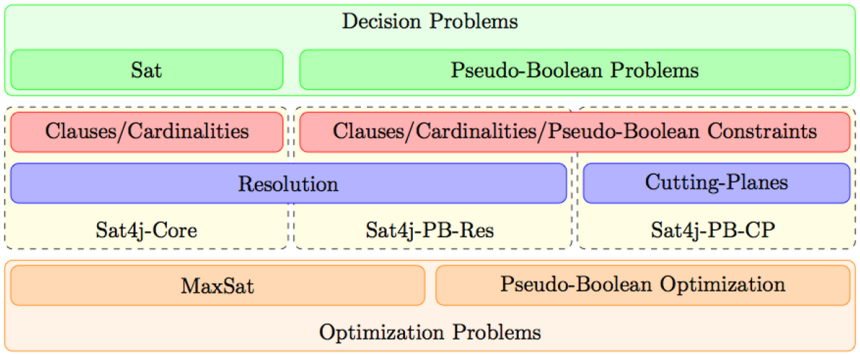

Figure 2.2: Overview of the SAT4J features

In Figure 2.2 you can see an overview of the SAT4J system[13]. SAT was the first problem within the planning domain to be NP-complete based on the Stephen Cook theorem[11]. NP represents for Non-deterministic Polynomial time and NP-complete means that any problem can be solved in polynomial time using a non-deterministic Turing machine. This is a theoretical concept in the field of computer science. Below we show the basic semantic for a CNF file. If we have a formula:

```
(p(1) OR (NOT p(3)))
AND
(p(2) OR (p(3) OR (NOT p(1)))
```

we can convert this into CNF format like so:

```
p cnf 3 2
1 -3 0
2 3 -1 0
```

The first line *p cnf 3 2* means that this is the first line of the problem; it is in conjunctive normal form; and there are 3 variables and 2 clauses. The next lines below are the clauses and they finish with a *0* to let the solver know that the clause has finished. We represent *OR* by a space in the clause and an *AND* with a new line as seen above. The SAT solver will then decide if this problem is satisfiable (or not) depending on the input by the user.

```
c 2 constraints processed
s SATISFIABLE
v -1 -2 -3 0
c Total wall clock time (in seconds) : 0.023
```

(The 0 at the end has been outputted to show the end of the clause) We can see from the above output that the problem was satisfiable in a total time of 0.023 seconds. Translating it back to logic it looks like this:

```
(NOT p(1) OR p(2) OR p(3)
```

2.5 Heuristics

As discussed in Chapter 1, a heuristic function will involve a relaxed version of the problem. In some heuristics, the plan is to relax the problem so that a small subset of it can be solved so it makes it less computationally expensive to solve a problem. When the planning problem is relaxed, the deleted list of actions is ignored. These are the negative effects of an action.[25]

```
(:action put-down
   :precondition (holding ?x)
   :effect
   (and (not (holding ?x))
```

Taking a look at this example above shows how, within PDDL, the negative effect of an action is coded within the domain but, with heuristic functions, we ignore these notations of $and(not(xxxx)$ for negative effects as they provide no help to the state of the environment. The area of heuristics can be broken down into 2 categories: admissible and not admissible. An admissible heuristic h for all $s \in S : h(s) \leq h^*(s)$. This will mean that it does not overestimate the distance to the goal node. Not admissible heuristics can over-estimate or underestimate the distance to the goal node. In the PDDL4J library there are a number of heuristics that have already been implemented; the table below shows a list of heuristics and if there are admissible or not admissible.

Heuristics	Method	Admissible/Not Admissible
Adjusted Sum	$sum(cost(pi)) + delta(S)$	Not Admissible
Adjusted Sum 2	$cost(S) + delta(S)$	Not Admissible
Adjusted Sum 2M	$cost(S) + delta(S)$ with mutex	Not Admissible
Combo	$sumheuristic(S) + setlevel(S)$	Not Admissible
Fast Forward	Uses a relaxed graph to ignore negative effects	Not Admissible
Set Level	Return the level of the planning graph where all propositions of the goal are	Admissible
Sum	Sum all the individual costs of each action	Not Admissible
Sum Mutex	Sum heuristic where mutual exclusions are computed	Not Admissible
Max	Calculates the most expensive atom	Admissible

As there are multiple heuristics implemented within the PDDL4J planer, I will explain briefly a not admissible heuristic Sum and how it can be changed with one line to make it the Max heuristic which is admissible.

2.5.1 Sum H^+

This heuristic function is called $h_0(s)$, otherwise known as the sum algorithm.

$$
\begin{aligned}
&Delta(s)\\
&\quad \textbf{for each } p \textbf{ do} : if\, p \in s \textbf{ then } \triangle_0(s,p) \leftarrow 0\\
&\quad else \triangle_0(s,p) \leftarrow \infty\\
&\quad U \leftarrow s\\
&\quad Iterate\\
&\quad \textbf{for each } a \textbf{ such that } \exists u \in U, precond(a) \subseteq u \textbf{do}\\
&\quad U \leftarrow u \cup effects^+(a)\\
&\quad \textbf{for each } p \in effects^+(a) \textbf{ do}\\
&\quad \triangle_0(s,p) \leftarrow min \triangle_0(s,p), cost(a) + \sum q \leq precond(a) \triangle_0(s,q)
\end{aligned}
\tag{2.1}
$$

We set the costs of the propositions appearing in the initial state as 0 and the cost of the rest ∞. We then update the costs until a fixed point is reached. So if we have a planning problem and each action has a $cost(a)$, we can try to estimate the distance to a proposition p or g. We do this by summing all the individual costs of each action.[12] If there is a solution to the problem, the sum algorithm will achieve it in polynomial time. Although it is a very informative heuristic, it is not admissible.

2.5.2 Max H^1

Max is a popular heuristic but is not widely used in planning competitions because, for any given state, max calculates the most expensive atom in the state. It works by considering a relaxed version of the problem. The max algorithm is the same as the sum algorithm except for one line.[31] The *for* loop for the positive effects changes to:

$$\triangle_1(s,p) \leftarrow min\triangle_1(s,p), 1 + max\triangle_1(s,q) | q \in precond(a) \tag{2.2}$$

The max heuristic is not as informative as other heuristics as it computes the most costly subgoals.

2.6 State of the Art

The best way to view state of the art in state-space planning is the International Conference on Automated Planning and Scheduling, otherwise known as the International Planning Competition (IPC) [26]. This competition is for planners to be tested against others on different domains and problems. The competition has been running since 1998 and uses different domains each time that are submitted by teams incorporating different levels of PDDL. For each of the conferences there have been new PDDL additions as discussed earlier. The competition breaks the planners down into sub-categories.

- Deterministic part
- Learning part
- Probabilistic part

Within each of these sub-sections are domains and problems that correlate to a specific task as it is difficult for all planners to be able to participate in all events. In this thesis, the deterministic section will be the main focus.

2.6.1 Search Algorithms and Heuristics

Looking at a number of planners that participated in the IPC competition over the past years, A* has been used various times as the main search algorithm but has been incorporated with other search algorithms to give a better overall search. A main algorithm used is one designed by Malte Helmert for the Fast Downward planner[20]. It is based upon the FF (Fast Forward) planner that was designed by Jorg Hoffmann and Bernhard Nebel.[24] The FF planner won the IPC competition in 2000 and is based upon two functions: a search algorithm called Enforced Hill Climbing with a Best First Search as its "back-up" search function, and a heuristic function

called Graphplan. Figure 2.3 shows the system architecture for the FF planner; the heuristic Graphplan is called at every state by the EHC algorithm and it is a forward search planner.

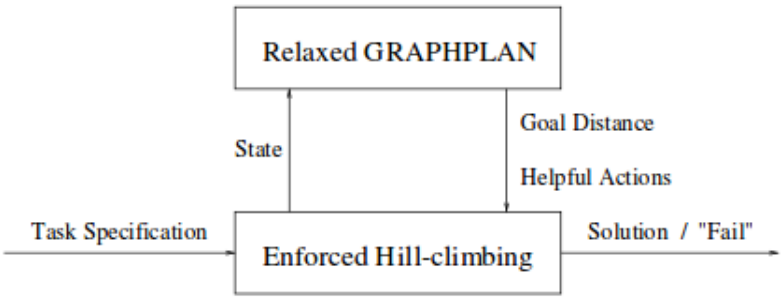

Figure 2.3: System architecture for the FF planner

Fundamentally, it uses the EHC algorithm to find a solution to a problem but, as the algorithm is susceptible to getting stuck in dead ends as pointed out in [24], there is a complete heuristic search engine built into FF that will start from the beginning if EHC fails, which helps to guarantee a solution. As well as the heuristic function, it has a backup algorithm Best First Search – and all of these functions together help build a very sound planner.

Fast Downward is based on this approach and it was the winner of the IPC competition in 2004 [2]. It is constructed in the same way as FF as it is a heuristic search planner but it uses a different heuristic function called Casual Graph Heuristic. This heuristic approximates goal distances by solving a hierarchy of local planning problems [32]; these problems only include a single state variable and the dependant variable. Within a casual graph we think of nodes as the variables in the problem and the links within the graph show the dependencies among the variables. The estimate that it provides is the number of operators needed to reach the goal from a state s in terms of estimated costs of changing the value of each variable v that appears in the goal from its value $s(v)$ in s to its value $s_*(v)$ in the goal.[32] Malte Helmert and Hector Geffner describe the casual graph estimate function as:

$$h(s) = \sum_{v \in dom(s_*)} cost_v(s(v), s_*(v)) \tag{2.3}$$

The Fast Downward planner incorporates two actual search algorithms to create a plan. It uses a Greedy Best First Search (GBFS) with the casual graph heuristic, and the second is multi-heuristic BFS which combines different heuristic evaluators. The workings of the GBFS have been discussed earlier. The multi-heuristic BFS works in a very simple way in that it will use multiple heuristics to direct the search towards a common goal. The idea behind it is that the heuristics will hopefully complement each other by having different weaknesses. So if one heuristic gets stuck in one section of the plan, the other heuristic may be able to find a solution for that particular problem efficiently and quickly and, therefore, negating the possibility of plateaus. But instead of combining the heuristic estimates into a single value, each heuristic alternates expanding a state from each open list.[20]

Fast Downward uses a combination of a Casual Graph Heuristic and a Fast Forward heuristic (FF). The documentation for the Fast Downward planner [19] explains that both the Casual Graph heuristic and the FF are not admissible. From work previously carried out, a heuristic is not admissible when it may overestimate optimal costs.[38][36] [17] This shows that the Fast

Downward planner in this sense is not optimal because it uses heuristics that are not admissible. Overall it means that when using these heuristics combined with the multi-heuristic BFS algorithm, the plans found are not optimal in terms of steps taken to achieve a goal. In the most recent IPC competition, the first-place planner in the deterministic track, the basis was Fast Downward and FF so we can see that using these algorithms provides non-optimal plans but this is sacrificed for speed and memory consumption, as well as solving problems.

One of the main heuristics used by multiple planners is the Fast Forward heuristic, developed by Hoffmann and Nebel[28]. Its main function is that from a current state, it finds a relaxed version of the problem and also ignores the negative effects from actions. The basis of the Fast Forward heuristic is that it uses a relaxed version of GraphPlan to compute the heuristic value.

The algorithms over the years have not changed much but the heuristic functions have changed drastically. The PDDL4J planner has optimal heuristic functions that can be used with the A* search algorithm but A* is only an optimal search algorithm if the heuristic is admissible.

From the two main planners we looked at in this section, the FF planner attempted 284 problems in the IPC 2002 competition and successfully completed 237 of them, giving it an 83% success rate. The Arvanherd planner, the winner of the IPC 2014 competition, was built upon the Fast Downward planner and uses the FF heuristic. It was able to solve a total of 161 problems[42]. The total success rate is unknown for Fast Downward.

2.7 Hypotheses

In this section we will define the hypothesis for this thesis. Based within the PDDL4J library, there is reason to believe that the A* algorithm is not as optimal as other search algorithms. Yes, provided with the correct heuristic, it provides an optimal plan but is it optimal in terms of speed and memory consumption? Also, are the heuristics within the planner optimal in terms of plan size and are they truly admissible or not?

As stated in the previous section, there are other search algorithms and heuristics that provide fast and efficient results for classical planning problems. The goal is to implement a new search algorithm for the PDDL4J planner based on the research conducted and also a new heuristic. The PDDL4J library will be tested against the new algorithm and heuristic as well as tested against other planners from the IPC competition.

The last part of the thesis will show how a different area of planning can complete classical planning tasks. We will use the SAT4J library to run classical planning tasks and compare the results from the SAT4J library to the results from the PDDL4J library to see if satisfiability problems are easier to solve than classical planning approaches using PDDL.

— 3 —
Implementation and Methods

3.1 What to do with this research

In this section we will discuss the methods that have been compiled for the literature review discussed in Chapter 2 and how algorithms have been constructed. We will also detail the testing procedure for the new algorithms and how to assess the validity and reliability of these methods. In the final section will discuss the initial assumptions.

3.2 Algorithms and Design

We will explore in detail the algorithms that will be implemented for the PDDL4J library and how the research brought us to the conclusion that these algorithms are more efficient than the existing ones. The search algorithm and heuristic have been taken from previous work in the classical planning field and applied to the PDDL4J planner.

As the PDDL4J library is written in Java, the best way to implement the new algorithms is to use the existing parser and lexer for the library and then remove all existing code for the A* algorithm and implement the new GBFS and EHC at the top level. As creating the parser and lexer would be a lot of work, we felt this was the better option as the existing code was efficient at parsing the domain and problem files.

The two search algorithms were available in pseudo-code which made the implementation a lot easier. The enforced hill climbing has been used multiple times so information regarding it was widely available[22]. The two new algorithms will be broken down into the following sub-sections and evaluated.

3.2.1 Greedy Best First Search with Enforced Hill Climbing

From the research conducted on search algorithms, we have made the decision to implement Enforced Hill Climbing (EHC) which was used in the Fast Forward planner [24] but instead of using A* or a modified version of it, we have decided to use a generic Greedy Best First Search algorithm[44]. The main reason for implementing EHC is due to its success at previous IPC competitions as well as being able to be implemented alongside another search algorithm like GBFS. We felt that it would improve the PDDL4J library in terms of speed, memory and solving problems. As GBFS expands the node that appears closest to the goal and uses the function $f(n) = h(n)$. This means that it only uses the heuristic to guide the search. So if we

incorporate GBFS as a back-up algorithm for EHC with the Fast Forward heuristic, this will make a faster/more memory efficient planner than A* with the Fast Forward heuristic which is what is used within the library at present. From the work carried out by Blai Bonet and Hector Geffner, they proposed changing an old planner HSP [8] from a hill climbing algorithm to a Best First Search algorithm, and the results showed that there was a dramatic increase in the number of solved problems compared to just using EHC [7]. One reason we want to combine these two algorithms is that the EHC algorithm is not complete and it has a problem with potentially getting stuck in infinite loops so we want to include GBFS as well as a relaxed heuristic like Fast Forward. Another main driving point why we think that A* can be beaten in terms of speed/memory consumption is that, in previous research, it states that A* is slower than other searching algorithms due to the computation of $h(s)$ for every new state, which is computationally expensive[8], therefore, we want to avoid this computation if possible. The decision was then made to incorporate the EHC and GBFS together to create potentially a fast, memory conscious planner that would rival A*.

We have already introduced the EHC in Chapter 2 as well as talked a little about GBFS but now we will describe how these two algorithms will work in depth and how they can complement each other. The use of a GBFS means that we will greedily expand the first successor that is found with a better heuristic value than its parent. But the algorithm will keep the parent stored so that the remaining children can be evaluated later. This, in turn, means that when we generate a successor state, two things can happen:

1. if the heuristic value of a successor is better than the parent, the parent is placed behind the successor in the search queue; or

2. if the heuristic value is not better than the parent, we use the value of the successor to place it in a priority queue and we continue the search.

In algorithm 4, we can see the pseudo-code for the GBFS algorithm. The aim is to perform heuristic evaluations that will be low as we envisage fewer successors.[4]

An advantage in GBFS is that it can be a forward or backward search algorithm. In this case we will use it as a forward search and it falls into the category *state space algorithm*, i.e. it looks through the space of possible states instead of looking for partial plans. With the implementation of these two new algorithms I purposed that the use of the Fast Forward heuristic that has been built within the PDDL4J library will be able to solve more problems than A* with Fast Forward and in faster times, as well as use less memory.

Algorithm 4 Greedy Best First Search

```
state = initialState
h = initialHeuristic
while searchQueue is not empty do
    currentQuery = pop item from front of searchQueue
    currentState = state from currentQuery
    currentHeuristic = heuristic from currentQueue
    applicableActions = array of actions applicable in currentState
    for all index ∈ applicableActions ≥ counter do
        currentAction = applicableActions
        successorState = currentState.apply(currentAction)
        if successorState = goal then
            return(plan)
        successorHeuristic = heuristicValue(successorState)
        if successorHeuristic < currentHeuristic then
            inset(currentState, currentHeuristic)
            insert(successorState, successorHeuristic)
        else insert(successorState, successorHeuristic
```

3.2.2 H^m Heuristic

Here we will define the H^m heuristic that was developed by Patrik Haslum and Hector Geffner[17]. The H^m heuristic is a domain-independent admissible heuristic which can be used to create optimal plans. This heuristic is part of the family of heuristics discussed in 2 where we change a small section of the Max heuristic to create the H^m heuristic. The new heuristic is based on computing admissible estimates of the costs of achieving sets of atoms from the initial state[17]. This means that when we have a set of these atoms of size 1, it is basically equivalent to the Max heuristic.

$$\begin{aligned} h^m(g,s) &= 0 \textbf{ if } g \subseteq s \\ h^m(g,s) &= \infty \textbf{ if } g \not\subseteq s \end{aligned} \tag{3.1}$$

The above equation is the beginning of the algorithm. It is stating that if the goal exists within the state s then it is 0, and if it is not within the state then the node is unexplored so its set to infinity.

$$\begin{aligned} &\textbf{if } |g| \leq k \\ &min_a(1 + \triangle^*(s, \lambda^{-1}(g,a)) | a \; relevant \; for \; g) \\ &\textbf{otherwise} \\ &max_{g'}(\triangle_k(s,g') | g' \subseteq g \; and \; |g'| = k \end{aligned} \tag{3.2}$$

This was taken from [31] and it means that when g has at most k propositions, we take the exact value $\triangle^*$ but if that is not possible then we approximate the distance $\triangle_k$ to g. We then can break down this algorithm to give it more meaning – looking through [31] we see that $\lambda^{-1}(g,a)$ can be defined as:

$$\lambda^{-1}(g,a) = (g - effects^+(a) \cup precond(a) \tag{3.3}$$

Then further looking redefines a to be relevant for g as:

$$g \cap effects^+(a) \neq 0 \wedge g \cap effects^- = 0 \tag{3.4}$$

In[31][17] they both have recommended that the value of k is kept at 2 because, as problems get harder with more objects, it is computationally expensive to generate subsets. The complexity of the heuristic makes the computation a polynomial N^k where N is the number of atoms.

3.3 How to Evaluate

Based on a research paper by Carlos Linares Lopez, the new algorithms will be evaluated using the techniques discussed in [9]. We will use all of the evaluation techniques as this will give a better idea of how effective the new algorithms are for the PDDL4J planner. The evaluation criteria is broken down into the following 4 categories:

- Objectivity – We want to have the same setting for all tests
  - Representation Language – Keep the same order for each domain and problem as this can affect the outcome. Also the language used i.e. PDDL
  - Evaluation Metric – Use one metric like time, plan size etc or if using multiple, keep it consistent throughout

  - Validation – Use a reliable system to validate the plans. It will ensure that the solution plans are logically sound

- Exhaustiveness – Use problems of a different nature, i.e. STRIPS and ADL, as well as multiple problem files within a domain which keeps the same problem structure but changes initial states and goal states

- Comparability – Using well studied algorithms as a baseline for the new algorithms

- Reproducibility – The test environment must be detailed so it can be replicated

3.3.1 How We Will Achieve This

Within the domain of classical planning, the IPC competition has a number of domains and problems that are used each year to test the planners. So for the testing procedure, we have taken a number of domains and problems from each year of the IPC competition to make an extensive set of testing domains which will give a fair analysis of the new algorithms and methods. The tests will be run on a Linux server that has 24 CPUs, each Intel(R) Xeon(R) 2.30GHZ, as well as having 264.13GB of memory, which is more than capable of running the new algorithms. Each problem will be allowed 10 minutes to find a solution but if no solution is found within the time limit then the planner will skip and go on to the next problem.

We will be using a piece of software called GNU parallel[14] so the problems can be run sequentially to save time, however, the issue is that the threads that have been created to execute problems will be sharing resources so we need to try to limit the resources so that each thread has a specific amount. Below shows an example of one way we can use GNU parallel to test the program. The *seq* allows us to start with the first problem and finish at the final problem (in this case 20). The $-j6$ tells the server that we want 6 processes to be created and to run sequentially. The next part is allocating memory to each of the processes. We are opting for 8048 megabytes as we think this is a fair amount of memory for the planner.

```
seq -w 20 | parallel -k -j6 java
-javaagent:build/libs/pddl4j-3.1.0.
jar -server -Xms8048m -Xmx8048m
fr.uga.pddl4j.planners.hsp.HSP -o
pddl/benchmarks_STRIPS/ipc1/gripper/domain.pddl -f
pddl/benchmarks_STRIPS/ipc1/gripper/p{}.pddl
-i 8 '>>' AstarGripper.txt
```

There will be a variety of STRIPS and ADL problems within the tested domains. Each of the domains will have a range of problems from small to medium and hard. All of the actions within each domain will be listed in the same order, as well as the problem files. The domains that we will be testing are in the table below: the left column shows the domain name, the middle column shows the number of problems in each domain and the right column shows which IPC competition they are from.

Domain	Number of Problems	IPC
Gripper	20	IPC1
Logistics	30	IPC1
Movie	30	IPC1
Mprime	30	IPC1
Mystery	30	IPC1
Blocksworld	35	IPC2
Elevator	100	IPC2
Freecell	60	IPC2
Schedule	100	IPC2
Depots	22	IPC3
Driverlog	20	IPC3
Rover	20	IPC3
Satellite	20	IPC3
Zenotravel	20	IPC3
Airport	50	IPC4
Optical Telegraph	14	IPC4
Philosophers	29	IPC4
Pipesworld	50	IPC4
Psr	50	IPC4
Openstacks	30	IPC5
Pathways	30	IPC5
Storage	30	IPC5
Tpp	30	IPC5
Truck	30	IPC5
Pegsol	30	IPC6
Sokoban	30	IPC6
Transport	30	IPC6
Barman	20	IPC7
Nomystery	20	IPC7
Parking	20	IPC7
Childsnack	20	IPC8
Hiking	20	IPC8
Thoughtful	20	IPC8

The main test is the search algorithms, and they will be tested with the same procedure and the results will be compared in three areas:

- Time – How fast the planner can find a valid plan
- Memory – What is the consumption of memory used to compute a valid plan
- Plan size – What are the number of actions been completed to solve a problem

Both search algorithms will use the heuristic Fast Forward and will be tested in the same environment. The new heuristic will be tested against three other heuristics: Max, Sum and Fast Forward. They will be tested using the same search algorithm (A*) and then compared in the

same format as above but with one extra measurement, i.e. plan quality. This is where we look at each plan individually and assess when actions were executed to see if the admissibility of certain heuristics is true or false. Also, we will see how not admissible heuristics complete a plan and the steps they take compared to an admissible heuristic.

3.3.2 Validity

Validity is defined as *"the quality of being logically or factually sound; soundness or cogency"*. [15] In the context of this thesis, we will be using a plan-checking software that was used for the IPC 2002 competition[1] which incorporates STRIPS and ADL (the use of which was decided in Chapter 2 of this thesis). The plan-checking software was built for the competition by Stephen Cresswell and it works by generating parse trees for PDDL 2.1 syntax. It uses a range of different coding languages and functions to check if a plan is logically sound. *"A state-space planner provides a plan as a sequence of actions"*[31]. This is the goal for the new algorithms, i.e. to provide a sequence of actions that are logically sound, for example:

```
0: pick-up b [1.00]
1: stack b a [1.00]
2: pick-up c [1.00]
3: stack c b [1.00]
4: pick-up d [1.00]
5: stack d c [1.00]
```

Figure 3.1: Example of a sound plan from Blocksworld

We can see from Figure 3.1 above that there are no instances where the plan is incorrect. So at each step in the plan, the actions can be executed. An example of a non logically sound plan would be if, at step 2, the planner executed the action *pick-up b*; this would not be possible unless the planner unstacked *b* and *a* first.
For validity we used a system called VAL which is a plan validator [40]. It takes in a domain file, a problem file, a plan file, and tests to see if the plan is feasible given the set of actions in the domain and the problem.

3.4 Initial Assumptions

The initial assumptions for the new search algorithm (GBFS with EHC using the Fast Forward heuristic) are that it will be able to provide a plan in a faster time and with less memory consumption than A* as well as being able to solve more problems from within the total amount. The new heuristic (H^m) will provide a critical path from starting node s_0 to a goal node s_n. In other words, the heuristic will sacrifice time and memory consumption but provide the optimal path. In comparison to other admissible heuristics where the plan length will be the same, we assume that the H^m heuristic will complete different actions than other admissible heuristics in order to achieve a goal.

— 4 —
Evaluation

4.1 Before Testing

In this Chapter we start the evaluation and testing procedure for the new search algorithm and heuristic. The first step is a preprocessing test, and when the results are analysed, we will start the preliminary tests of the new algorithms.

4.1.1 Preprocessing Test

The first thing to do before any tests could be completed was to check all of the IPC domains that would be used in the testing process for any errors. It would also give us a general idea of how long the parsing and encoding would take before the actual search could start. The goal was to remove any domains that had issues or that caused complications for the PDDL4J planner to encode. It would not make the testing environment fair if a domain was unable to be encoded properly and the search algorithm or heuristic ran for a long period of time when no solution would ever be potentially available. What we saw was issues with some of the domain files like Barman for example from IPC7. There were other domains like this and we addressed the issues and attempted to generate a new domain file or see if there was a small error inside the file that we could change.

4.1.2 Preliminary Test

In this section, preliminary tests were carried out to see if both the heuristic and the search algorithms were going down the right path to complete the hypothesis. We tested the search algorithm first by running it on the Blocksworld domain from the IPC2 competition. It was tested against the current A* algorithm and the preliminary results showed that the new search algorithms had out-performed A* by a considerable amount. The time for completing all 35 problems was much faster than A*, and also, A* was unable to complete the final 5 Blocksworld problems within the 600 second time limit.

This gave a really good insight to the capabilities of the new algorithm, as well as being on track to prove the hypotheses set out in Chapter 2. These results, however, could not be used as they were run on my own computer which does not fit in with the testing environment set out in Chapter 3.

The same environment was used to test the H^m heuristic and it was tested on the Mystery domain from IPC1. We tested the new heuristic with Fast Forward and Max to compare the

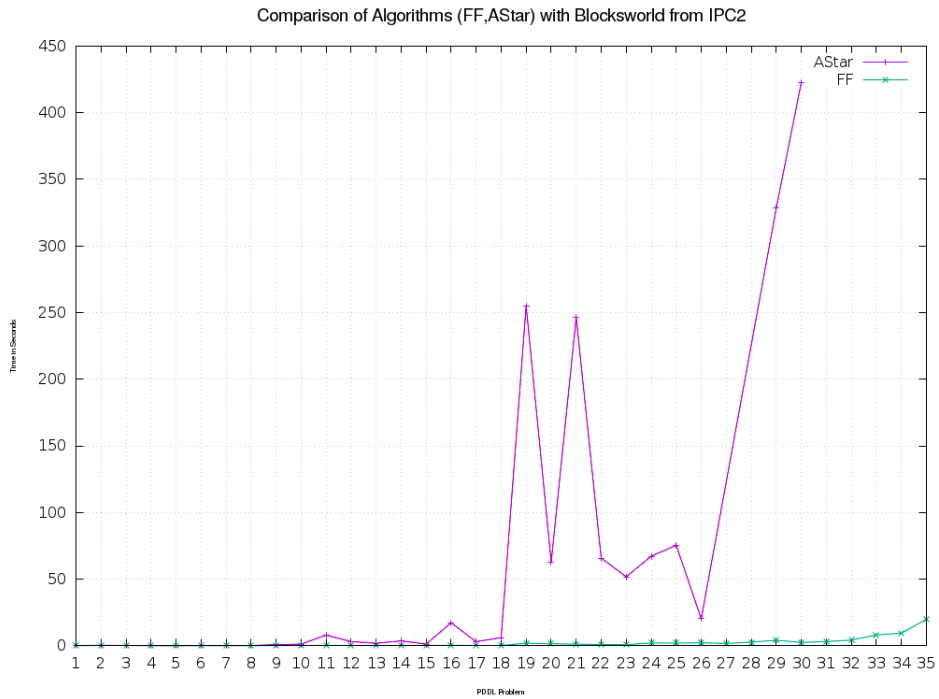

Figure 4.1: Preliminary results on the Blocksworld domain for the new search algorithm vs A*

time needed to complete the problems. We achieved good results in that the new heuristic was similar in speed versus Fast Forward for completing the tasks although, as expected, the Max heuristic took longer.

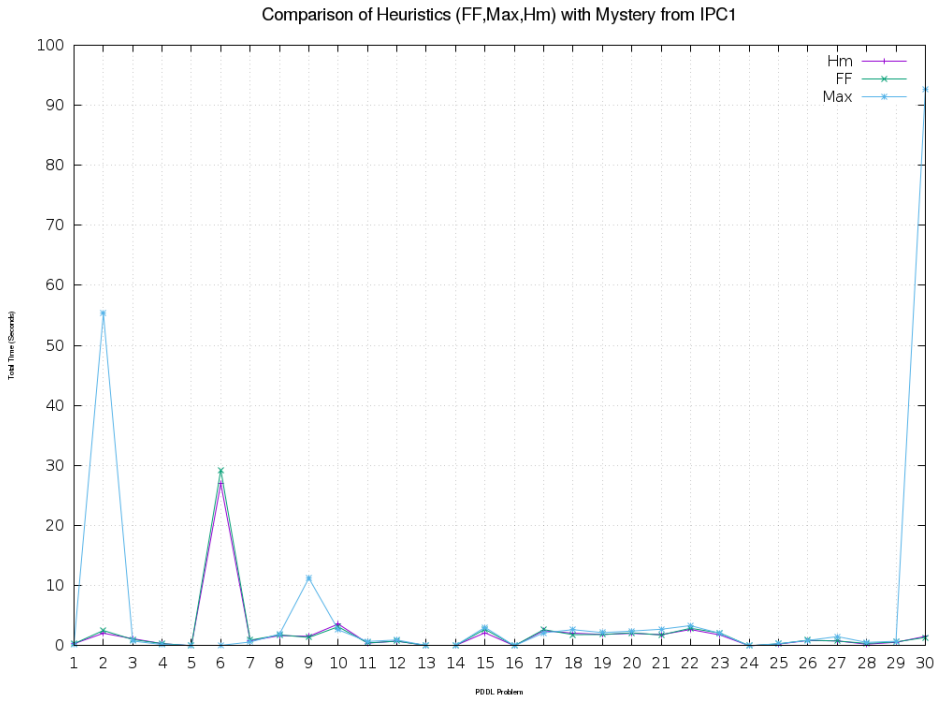

Figure 4.2: Preliminary results on Mystery domain, FF vs Max vs H^m

There were no differences in plan size and we think this is because the H^m heuristic is admissible [16], as is Max, so there should not be any differences in the plan length. Also, compared with Fast Forward, all plan lengths were the same. This is possibly due to the Mystery domain being a set of small problems with a small number of objects as well as all the heuristics running with the A* algorithm which should provide optimal plans (with the use of an admissible heuristic). A vigorous testing procedure in the next section should be able to provide the results that we are hoping for.

4.2 A* vs Greedy Best First Search and Enforced Hill Climbing

As the preprocessing tests and preliminary results showed impressive data for the new search algorithm and heuristic, the testing stage can start against all domains from IPC1-IPC8 as discussed in Chapter 3 The table below shows all the domains in which we tested the new algorithm against A*. It also shows the total number of problems within each domain and how many of those problems the planners managed to solve.

Domain	**GBFS + EHC**	**A***
Gripper	20/20	7/20
Logistics	0/30	7/30
Movie	30/30	30/30
Mprime	14/30	24/30
Mystery	13/30	16/30
Blocksworld	35/35	30/35
Elevator	99/99	99/99
Freecell	60/60	59/60
Schedule	0/100	0/100
Depots	15/22	16/22
Driverlog	16/20	14/20
Rover	0/20	2/20
Satellite	14/20	12/20
Zenotravel	20/20	13/20
Airport	0/50	28/50
Optical Telegraph	1/14	2/14
Philosophers	10/29	5/29
Pipesworld	16/50	15/50
Psr	41/50	47/50
Openstacks	30/30	7/30
Pathways	7/30	5/30
Storage	17/30	15/30
Tpp	15/30	7/30
Truck	13/30	8/30
Pegsol	29/30	27/30
Sokoban	25/30	25/30

Transport	0/30	0/30
Barman	0/20	0/20
Nomystery	7/20	13/20
Parking	3/20	0/20
Childsnack	2/20	0/20
Hiking	0/20	0/20
Thoughtful	7/20	5/20
Solved Problems Total	559/1089	538/1089
Percentage	51.33%	49.40%

Table 4.1: Table for displaying results of GBFS with EHC vs A*

These are the results for the new algorithm for the PDDL4J planner versus A*. We used the same technique as described in Chapter 3 to run the tests. A bash script was created so all the domains could be run one after the other and save the results in a text file which was then used to create diagrams with GNUPLOT. We can see some good results from the new search algorithm as well as some strange results, namely from the Rover domain where A* was able to solve only 2 problems out of a possible 20. Then with Airport from IPC4, A* was able to solve 28 out of a possible 50 but GBFS with EHC was unable to solve any of the problems. With Openstacks, GBFS with EHC was able to solve all 30 but A* could only manage 7. The total number of problems in all of the domains is at the bottom of the table along with the total number of solved problems and the number of problems attempted against problems solved is shown as a percentage. From this we can deduce that GBFS with EHC was able to solve 1.93% more problems than A*. Even though this is a small percentage, it is still an improvement in regards to the PDDL4J planner and it proves the hypothesis correct. We will now look at some of the problems in detail and explain the findings. All results for the tests are placed in Appendix A at the end of the report.

4.2.1 Graphs and Analysis

There are many sets of domains that we could choose from but have narrowed it down to 4 in order to demonstrate the difference in time plus memory consumption between the two search algorithms. The domains are Movie from IPC1, Sokoban from IPC6, Blocksworld from IPC2 and Psr from IPC4. The full set will be displayed on GitHub and all information regarding access can be found in Appendix A.

The Movie domain has always the same goal, i.e. to get lots of snacks before watching the movie. The planners finish when all the snacks set out in the problem have been accumulated as well as the success of all other parameters set out in the problem. Each problem requires ADL, as discussed in Chapter 2. The problems for both planners are relatively easy but in terms of speed at achieving the goal, the new algorithm fairs better in every problem than A*. You can see from the graph (GBFS with EHC red, A* green) that the paths of the lines never cross or are even close to each other. Along the X axis is the time in seconds, the Y axis has the number of problems. We can see that the new algorithm as well as A* had a steady increase

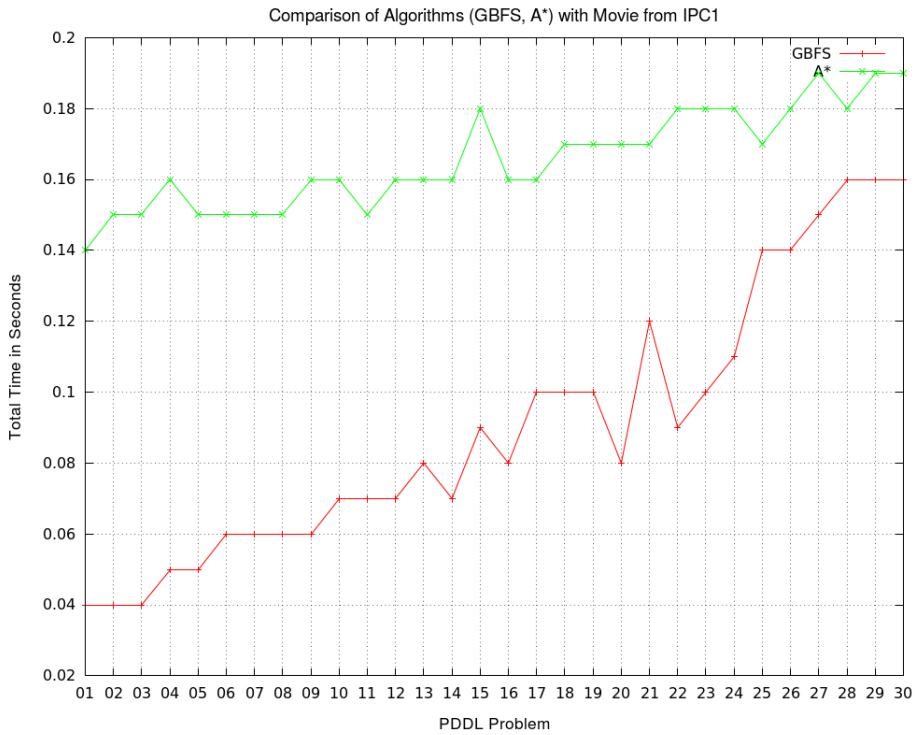

Figure 4.3: Movie domain from IPC1, GBFS + EHC vs A*

in line with the increase in difficulty of the problem and the time taken to solve them but even then, the times are not similar. From the raw data that can be accessed from the information in Appendix A, EHC failed for every problem but GBFS with EHC was still able to complete each problem faster than A*.

Let's look at the Sokoban domain which was used for the IPC6 competition. Sokoban is a game where there are crates in certain locations and the goal is to take one crate at time from the starting point to an end point but being careful not to block yourself in or get stuck. The domain has a total of 30 problems but both algorithms could only solve 25 each, however, the results regarding time are mostly better for GBFS with EHC. From the graph it is clear that GBFS with EHC was able to solve the same number of problems but in a faster time; with some of the problems, the time taken is relatively close but in others there is a huge difference.

Looking at the results on memory consumption for both A* and GBFS with EHC in the Sokoban domain, Figure 4.5 shows that an increase in time yields an increase in memory consumption with the largest disparity showing up in problem 15. Here, GBFS with EHC was able to solve the problem in 52.99 seconds and use 17.82Mbytes of memory, whereas A* took 496.38 seconds and used 284.31 Mbytes, which is a difference of 443.39 seconds and 266.49Mbytes. To put this in perspective, GBFS with EHC would need to solve problem 15 in Sokoban 15 times in order to match the same memory consumption used by A*. Earlier in Chapter 4, we presented results of a preliminary test in the Blocksworld domain, showing the speed of GBFS with EHC. Now, within the same domain, we will look at the memory usage for problems 01-35 (see Figure 4.6).

From the graph we can see a very slow, steady increase in memory usage as the plans become more complicated for the GBFS with EHC algorithm, however, for the A* algorithm, things

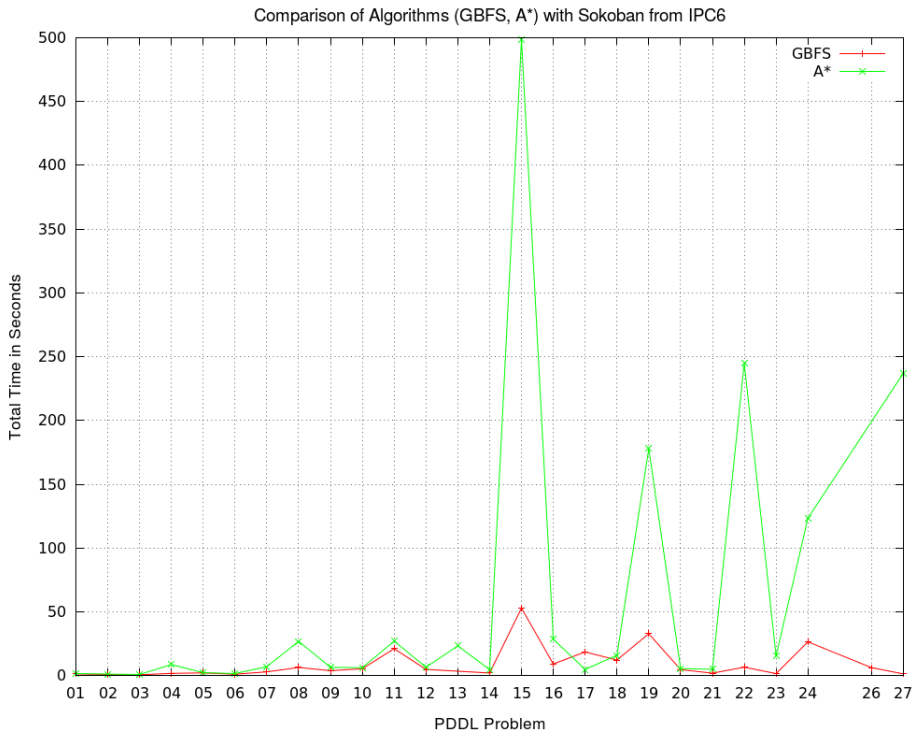

Figure 4.4: Sokoban domain from IPC6, GBFS + EHC vs A*

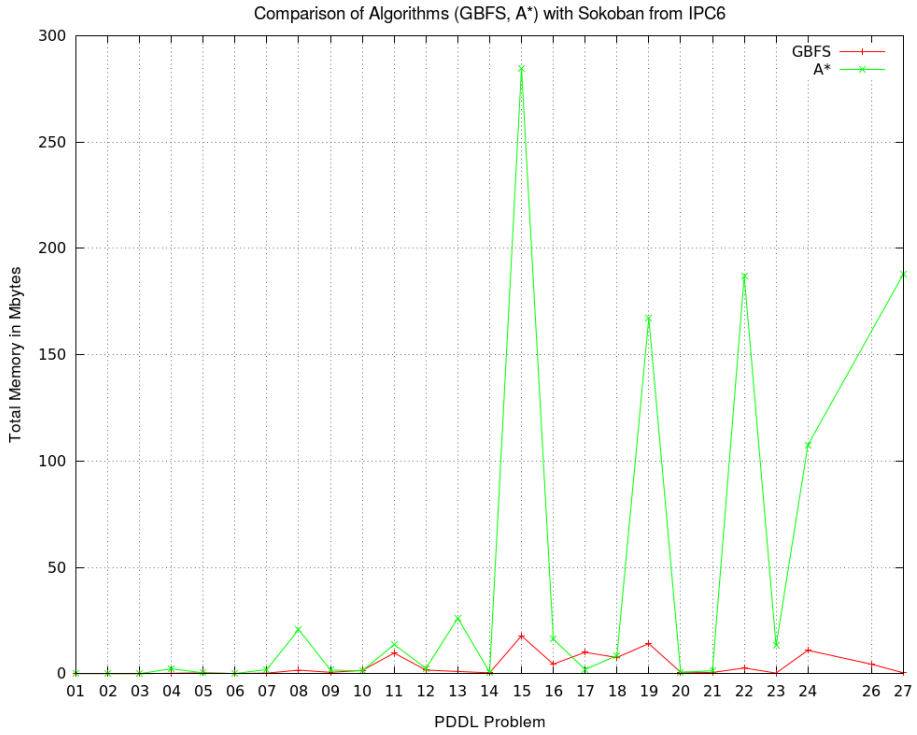

Figure 4.5: Sokoban domain from IPC6, GBFS + EHC vs A*

were more complicated. The maximum memory used by GBFS with EHC was done so when solving the hardest problem in the domain, i.e. 12.05Mbytes was used by GBFS with EHC for

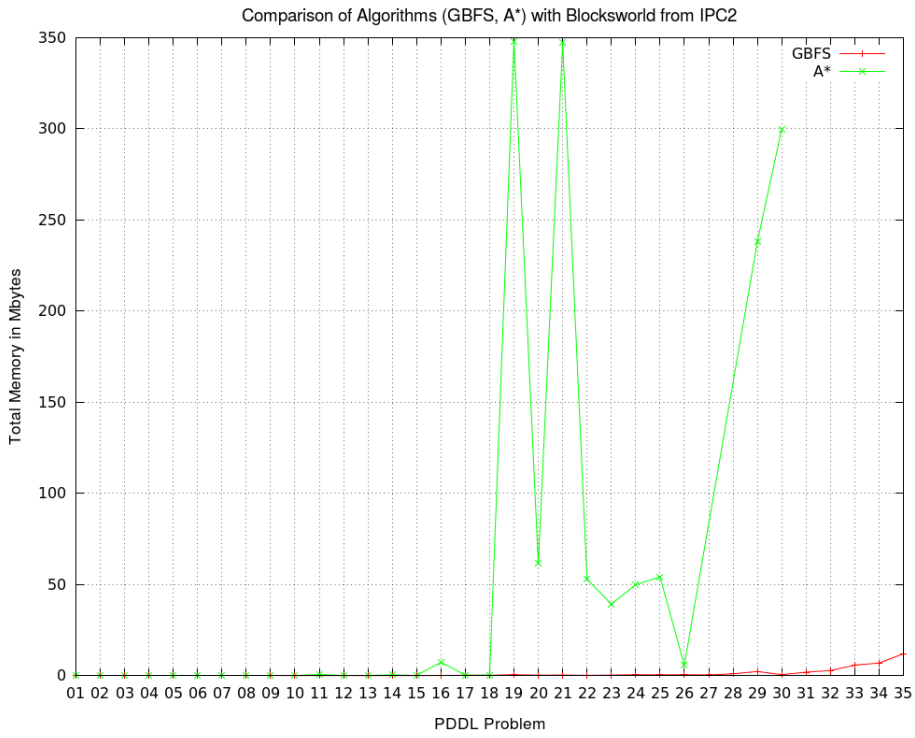

Figure 4.6: Blocksworld domain from IPC2, GBFS + EHC vs A*

problem 35. If we look at the maximum memory used by A*, it was 348.12Mbytes and 347.07 Mbytes for problems 19 and 21 respectively, neither of which are the hardest problem. As well as this, we can also see that A* was unable to complete Blocksworld problems 31-35.
If we look at Zenotravel from IPC3 Figure4.7, GBFS with EHC was able to solve all 20 of the problems within the domain, but A* was only able to solve 13. The time gradually increases as the problems get harder in the domain for both algorithms but for GBFS with EHC there is a huge increase of time from problem 18 to 19. This could be because of the increase of facts, as problem 18 has 585 facts whilst problem 19 has 760. Also in problem 17 there is a total number of 16880 possible action which increases to 20970 in problem 18, then 19 has an even bigger increase to 26500. This increase could be the reason why the time has dramatically risen. Then in contrast in Figure4.8, as the problems get harder, the memory consumption usually goes up along with it, which is what happened to A* but GBFS with EHC kept a steady increase of memory consumption with a small spike at problem 19. We can see by all the graphs shown that in terms of speed and memory consumption, GBFS with EHC is dominantly better. Now if we look at Figure 4.9, we took data from the IPC 2002 competition web-page [1] regarding the FF planner that won the competition in 2002. We can see by the results that the new search algorithm GBFS with EHC + Fast Forward heuristic was able to better the previous winner in some of the harder problems in the Zenotravel domain. We allowed some of the result even though they are above our time limit for comparison but in problems 15, 17, 18 and 20 the new algorithm was better than the FF planner.

All of the IPC 2002 results can be found in Appendix A, we also saw in the Freecell track that the hardest problem took the FF planner 1711.97 seconds to complete but the same problem with the new algorithm was able to complete the problem in 76.56 seconds

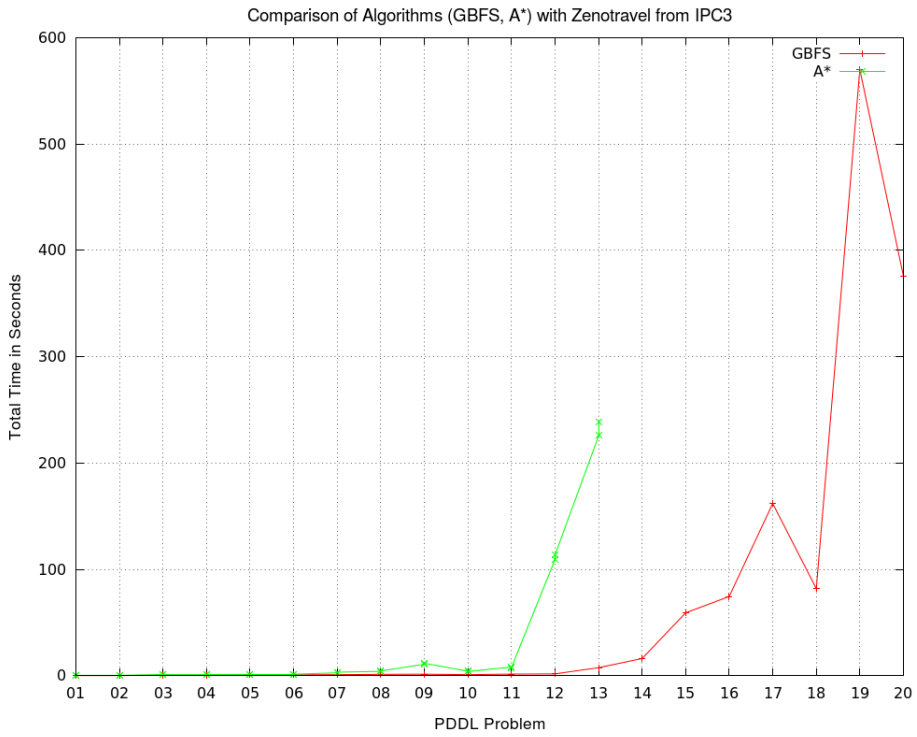

Figure 4.7: Zenotravel domain from IPC3, GBFS + EHC vs A*

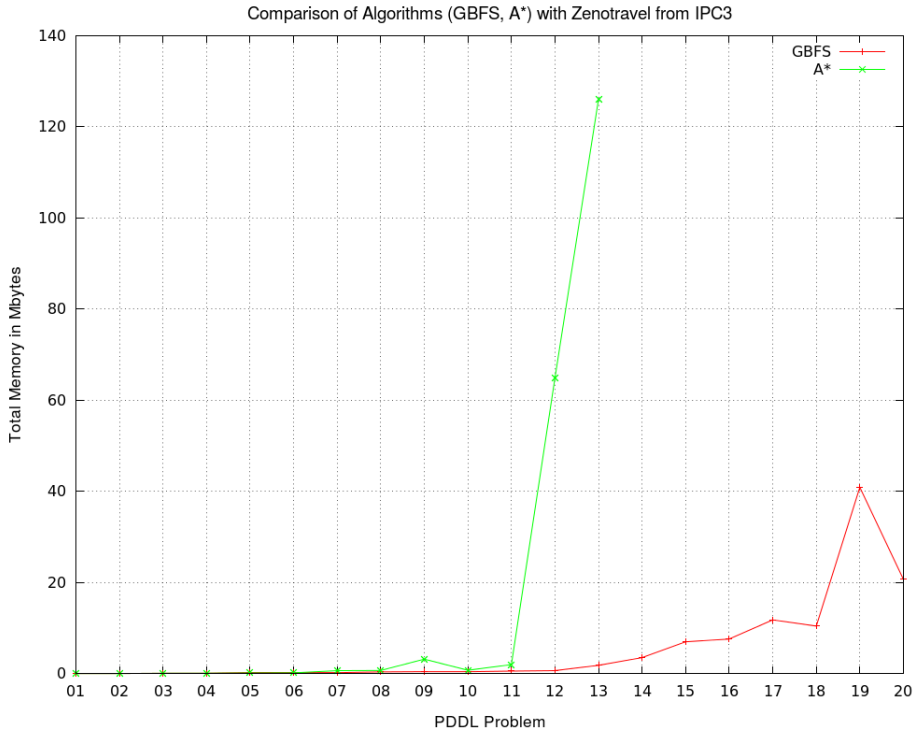

Figure 4.8: Zenotravel domain from IPC3, GBFS + EHC vs A*

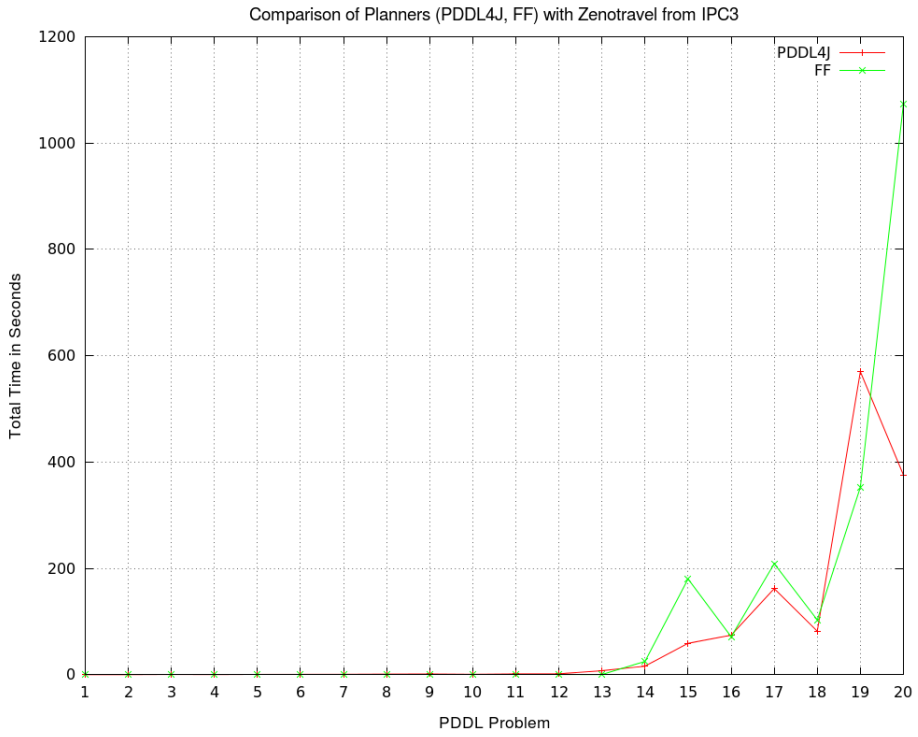

Figure 4.9: Zenotravel domain from IPC3, GBFS + EHC vs FF Planner

4.2.2 Null Plans

Even though a preprocessing test was carried out for all domains, there were some that did not parse correctly and further analysing of these tests showed that some of the domains did not work correctly or there was issues with them which made it impossible for the planners to even attempt the domains. We can see automatically which domains or problems had potential problems as both of the planners were unable to solve any of the problems and the text files pertaining to the problem output were empty. To name one, the Barman domain was unable to be parsed correctly and that is why each of the algorithms have 0 solved problems.

4.2.3 Validity

As discussed in Chapter 3, we used VAL[40] to check the new search algorithm to ensure that it was providing logically sound plans. As there were over 1000 tests, we selected a few problems to test using VAL. We felt that if these 5 tests came back positive then the algorithm was able to produce logically sound plans for all domains as we have a mix of STRIPS and ADL as well as easy, medium and hard problems. The 5 problems selected were:

- Problem 1 from PSR
- Problem 10 from Zenotravel
- Problem 20 from Blocksworld
- Problem 01 from Sokoban
- Problem 10 from Blocksworld

All 5 of the tests came back positive and all 5 were valid. As there we no errors in any of the plans, we can assume that all plans will be valid from the new search algorithm. All raw data files can be found in Appendix B.

4.3 H^m vs Fast Forward vs Max vs Sum

For comparison with regards to heuristics, we used the same search algorithm (A*) with all 4 of the heuristics. We tested them with the same script and same set of problems. As before, each problem was capped at 600 seconds to solve a problem. The results gained were not as good as the search algorithm and in terms of solved problems it was worse than expected. The H^m heuristic was unable to solve more problems than any of the other 3 heuristics but we assumed this could happen as it is costly to generate all subsets of a problem. Because of this, the H^m heuristic was only able to solve 12 of the Blocksworld problems within the 600-second time limit. The two admissible heuristics (H^m and Max) performed worse in the Blocksworld domain. Max was only able to solve 6 more problems than H^m but FF and Sum solved nearly all problems (see Figure 4.10).

If we then take a look at the Movie domain from IPC1 (see Figure 4.11), H^m was able to solve all 30 problems, as could the other 3 heuristics. H^m was a little slower than the other 3 in solving the problems but was able to do it within the 600-second time limit. Again, the not admissible heuristics proved to be better in terms of time within this domain. The memory

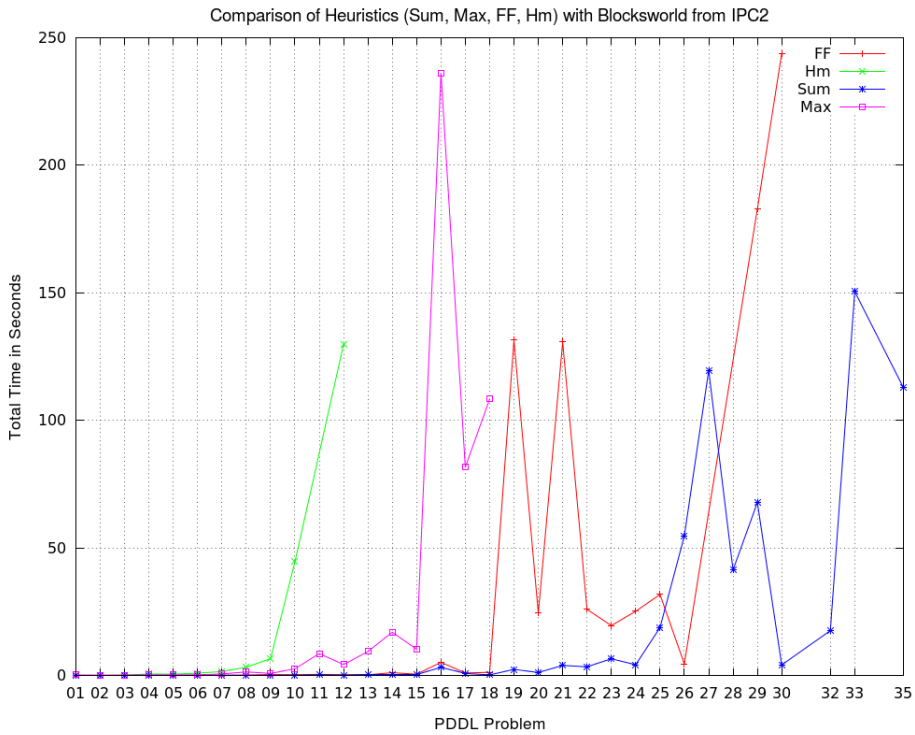

Figure 4.10: Blocksworld domain from IPC2, Sum vs FF vs Max vs H^m

consumption of each of the heuristics was exactly the same except for Sum being more efficient towards the end. The last strange result is that the plan size for all 4 heuristics was the same for each problem. Even though the Movie domain has many small problems, every problem having the same plan size for not admissible and admissible heuristics is not what we expected (see Figure 4.12). As stated in [37] the H^m heuristic exhausts memory and this was the case multiple times during testing. Within a lot of the domains, the heuristic failed to solve any problems and memory was exhausted, that being said, within some of the domains compared to an admissible heuristic like Max, the results were similar in terms of solved problems but it was still the worse of the three. In some of the domains like Freecell, the H^m heuristic was unable to solve any of the problems. What we saw was within the Blocksworld domain, there were anomalies between admissible and non admissible heuristic in terms of actions used to achieve a goal.

Below (Figures 4.13 for the two admissible heuristics and Figure 4.12 for not admissible) are the actions used to complete the Blocksworld problem 4. With regards to the two admissible heuristics H^m and Max, we can see some differences in the actions used to complete the problem. At step *02* the H^m heuristic guided the algorithm to pick up block d whilst Max unstacked e b. Fast Forward and Sum did the same action at this stage. Even though all plan sizes are the same in length, these steps can describe a more critical path to the goal from the starting node.

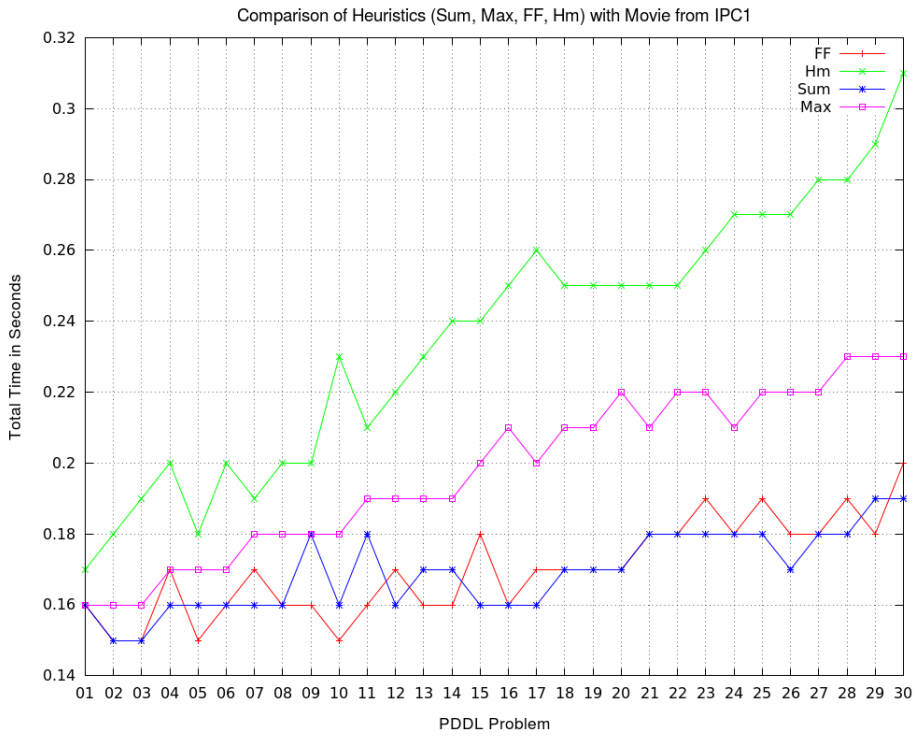

Figure 4.11: Movie domain from IPC1, Sum vs FF vs Max vs H^m

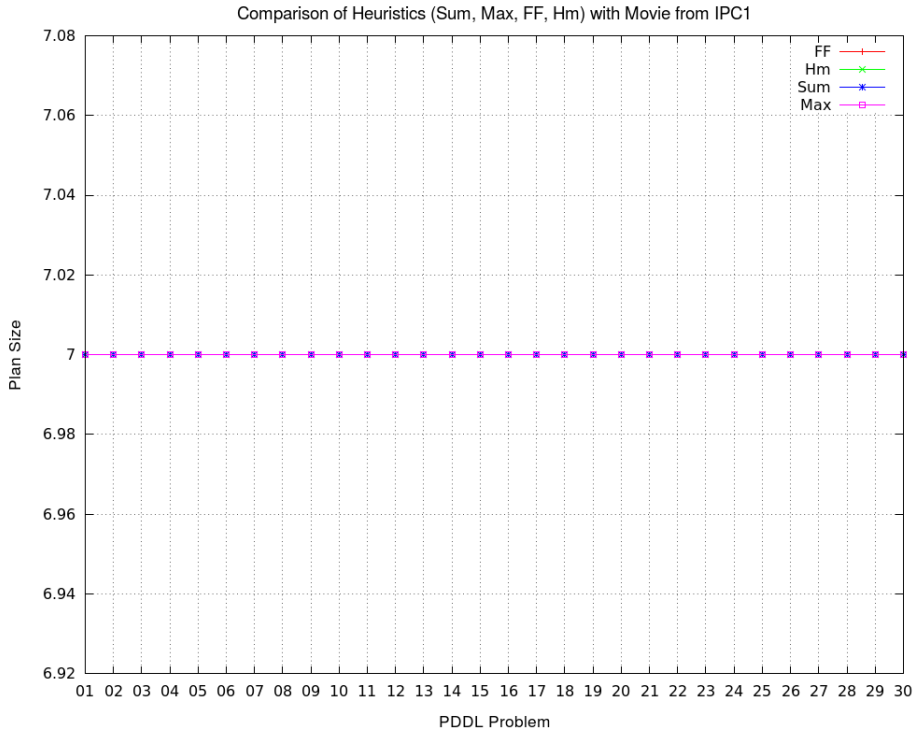

Figure 4.12: Movie domain from IPC1, Sum vs FF vs Max vs H^m

```
00: unstack c e [1.00]
01: put-down c [1.00]
02: pick-up d [1.00]
03: stack d c [1.00]
04: unstack e b [1.00]
05: put-down e [1.00]
06: unstack b a [1.00]
07: stack b d [1.00]
08: pick-up e [1.00]
09: stack e b [1.00]
10: pick-up a [1.00]
11: stack a e [1.00]
```

(a) FF: Actions used to achieve goal in Blocksworld problem 4

```
00: unstack c e [1.00]
01: put-down c [1.00]
02: pick-up d [1.00]
03: stack d c [1.00]
04: unstack e b [1.00]
05: put-down e [1.00]
06: unstack b a [1.00]
07: stack b d [1.00]
08: pick-up e [1.00]
09: stack e b [1.00]
10: pick-up a [1.00]
11: stack a e [1.00]
```

(b) Sum: Actions used to achieve goal in Blocksworld problem 4

Figure 4.13: Not admissible heuristic actions

```
00: unstack c e [1.00]
01: put-down c [1.00]
02: unstack e b [1.00]
03: put-down e [1.00]
04: pick-up d [1.00]
05: stack d c [1.00]
06: unstack b a [1.00]
07: stack b d [1.00]
08: pick-up e [1.00]
09: stack e b [1.00]
10: pick-up a [1.00]
11: stack a e [1.00]
```

(a) Max: Actions used to achieve goal in Blocksworld problem 4

```
00: unstack c e [1.00]
01: put-down c [1.00]
02: pick-up d [1.00]
03: stack d c [1.00]
04: unstack e b [1.00]
05: put-down e [1.00]
06: unstack b a [1.00]
07: stack b d [1.00]
08: pick-up e [1.00]
09: stack e b [1.00]
10: pick-up a [1.00]
11: stack a e [1.00]
```

(b) H^m: Actions used to achieve goal in Blocksworld problem 4

Figure 4.14: Admissible heuristic actions

4.4 SAT4J

We wanted to test to see how a satisfiability planner solved PDDL problems. Even though there are a lot of SAT planners[30][21] we decided to take a Blocksworld problems and encode them to SAT problems within the PDDL4J library, then give the output to a SAT solver (in this case SAT4J) and see how much of a difference there was in terms of speed at solving satisfiability problems. We encoded 9 PDDL problems from the Blocksworld domain (p01-p09) and provided the files to SAT4J. We discovered that SAT4J was able to complete Blocksworld problem 9 within 0.037 seconds. Compared to A* which was 0.82 seconds and GBFS with EHC was 0.10 seconds. It shows that using a SAT solver is nearly 10x faster at solving problems.

```
c org.sat4j.minisat.constraints.cnf.OriginalBinaryClause => 4
c org.sat4j.minisat.constraints.cnf.OriginalWLClause => 13
c ignored satisfied constraints => 17
c 34 constraints processed.
s SATISFIABLE
v -1 2 -3 -4 -5 -6 -7 -8 0
c Total wall clock time (in seconds) : 0.037
```

Due to time constraints, only the Blocksworld domain was tested. With more time, we could test all of the IPC domains for a comparable analysis.

4.5 GBFS with H^m

We combined the new search algorithm with the new heuristic to see if the plan size would be smaller when using a heuristic that should provide the optimal path. When the problems became harder, the GBFS with EHC + FF plan size grew but when applied with H^m, the plan size was considerably smaller as you can see from Figure 4.15. As the H^m heuristic was only able to complete 19 out of the 35 problems within the time limit, we selected only the first 20 problems to create the graph 4.15. Except for one instance (PDDL problem 5), the H^m heuristic plan size stayed below GBFS with EHC using Fast Forward. As H^m is an admissible heuristic it provided a better quality plan as it did not overestimate or under-estimate the distance from the starting node to the goal node. This was a good result for the Blocksworld problem but applying the same approach to Elevator from IPC2 did not provide the same set of results. We can see that with some of problems, the plan size was the same but with the majority of others the plan size for H^m was larger which is not what we expected given the results from the Blocksworld domain.

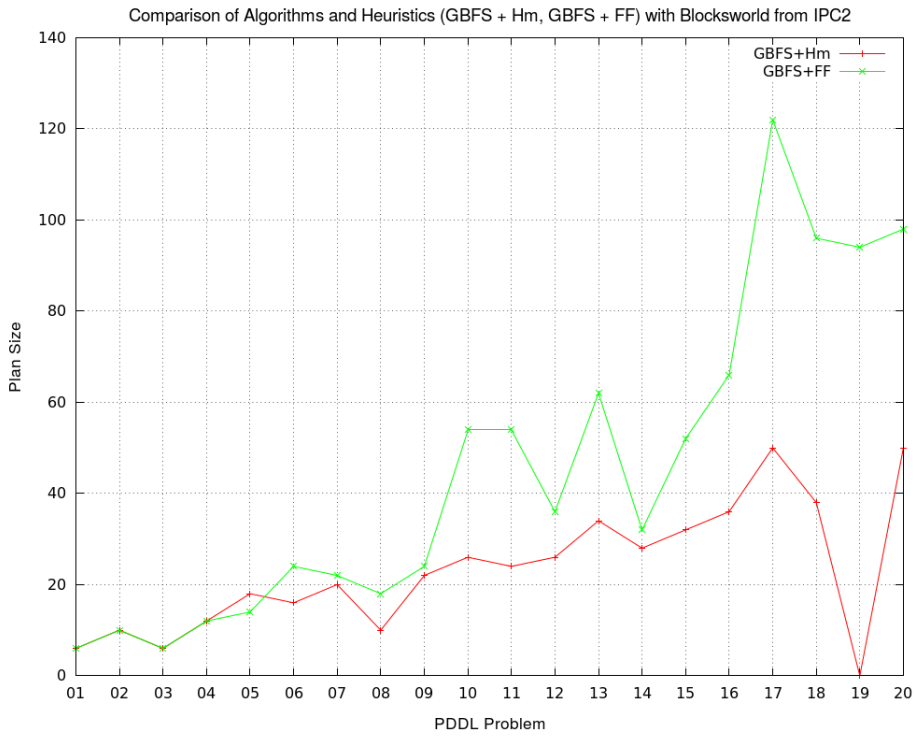

Figure 4.15: Blocksworld domain from IPC2, GBFS with EHC + FF vs GBFS + H^m

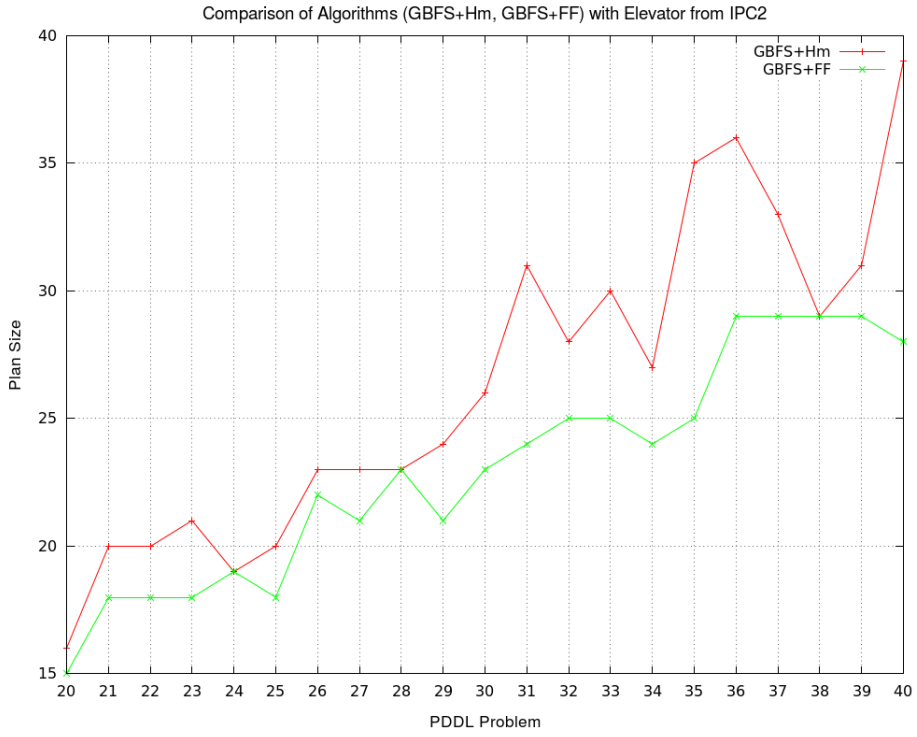

Figure 4.16: Elevator domain from IPC2, GBFS + FF vs GBFS + H^m

4.6 Test of Hypotheses

In Chapter 2, we made hypotheses about the new search algorithm (GBFS with EHC) and the new heuristic H^m. With regards to the search algorithm, we devised the hypothesis that it would able to solve more problems than the current algorithm (A*). From the final results, we can say that this hypothesis is now true, as GBFS with EHC was able to solve 21 more problems than A*. It is a small number in the grand scheme of how many problems were tested but it proves that the current algorithm is not better in terms of solved problems. We also stated that the new algorithm would be faster and use less memory than A*. From the results presented within this chapter and the raw data, we can safely say that this is true. The results in certain domains were close but in others GBFS with EHC proved to be the more superior algorithm for classical planning. With the new heuristic we saw that it was unable to complete more problems than the other heuristics within the library and also the speed and memory consumption was not better overall. We were able to spot some anomalies within the actions used to achieve a goal from the Blocksworld domain which suggest that the other heuristics are not using the most optimal approach when solving a problem. This concerns Max more as it is an admissible heuristic, even though it does not overestimate or under-estimate the distance to the goal, it uses actions that are possibly more costly in achieving a goal.

— 5 —
Conclusion

5.1 Conclusion of Work

In Chapter 1 we detailed the outline that would be the general theme for the thesis, Classical Planning approaches. We provided the main aspects of classical planning and what were the topics we would be covering in the next sections. Explaining the aspects of the PDDL4J library, PDDL, Satisfiability problems and the goal for this thesis.

We would then go on in Chapter 2 to discuss what areas we will be focusing on and how we would try to achieve the goals of this thesis. We also discussed many approaches in terms of state-space planning as we felt using one of these algorithms would prove our initial hypotheses set out at the end of Chapter 2. The overall goal of Chapter 2 was to grasp an understand into the world of planning, and through research, be able to come to a conclusion with what algorithms would prove our hypotheses.

Chapter 3 described how we would test the various assumptions, and based on [9] how we would test the new algorithms. We also described the implementation phase of the new search algorithm and heuristic function. These were based on [24] and [31] plus [37].

The last Chapter, 4, we compiled all the results from the multiple tests and discussed the results, as well as tested our hypotheses set out in Chapter 2.

In this thesis we were able to prove that A* was not the more optimal search algorithm for the PDDL4J planner, as the new search algorithm, Greedy Best First Search with Enforced Hill Climbing based on [24] was better overall. Even though there were some domains were A* was better, the more solved problems in a faster, less memory consumed manor went to GBFS with EHC. We can see by the results the difference between these two algorithms and in the next section below we discuss future additions to the algorithm and the PDDL4J planner.

The H^m heuristic on the other hand did not goes as well as expected, from the beginning we knew that it would be computationally expensive to compute but did not assume that it would not be able to solve a majority of the problems. Even though we were able to see some anomalies between actions, in the majority of harder problems in Blocksworld for example, it was unable to solve past problem 12. When paired together with the GBFS with EHC algorithm it provided the same base of results in terms of solved problems but in some of the domains the speed was worse for GBFS with EHC.

In the following sections, we converted PDDL problems into SAT problems and gave the problems to the SAT4J library. This provided fast results in solving some of the Blocksworld problems compared to PDDL4J (A* and GBFS with EHC). Further testing would have been

a plus in this area so more problems could be analysed but this has went into the future work section below as an extension to the PDDL4J planner.

5.2 Limitations

One of the biggest limitations for me was the fact that I did not create the PDDL4J planner, which meant that I had to spend a lot of time trying to figure out such things as how it all worked, how classes linked together, what methods were available to me, etc. This took up a large part of my time when it came to creating the two algorithms. Also, even though Java is a very popular coding language and I have worked with it throughout my University career, some of the classes use "Bitset", which is something I had never used before, so trying to familiarise myself with that as well as conducting research on a topic about which I know only a little was certainly challenging.

5.3 Future Work

For future work, I feel there is a great deal that could be incorporated in the PDDL4J library that has not as yet been implemented, one of those things being the incorporation of a SAT solver, which would enable the library to solve satisfiability problems as PDDL.

Another addition could be a potential learning algorithm, one that start to build knowledge of the4 actions with a domain, meaning that the initial problems would be run as normal but the more the planner runs problems within a certain domain like Blocksworld, the more it would understand how to use actions in an optimal manner.

An algorithm that could take into account two heuristics would be good for classical planning approaches as combining a not admissible heuristic and an admissible heuristic could complement each other in certain areas. So for example, if the not admissible heuristic was stuck in a section, the admissible heuristic could take over to push past or go back from a dead end and letting the not admissible heuristic take over again. Potentially, this could provide a fast classical planner that could provide semi-optimal plans.

Bibliography

[1] IPC 2004. *International Planning Competition 2002*. URL: http://ipc02.icaps-conference.org/.

[2] IPC 2004. *International Planning Competition Winners*. URL: http://idm-lab.org/wiki/icaps/ipc2004/deterministic/.

[3] Derek Long Alfonso Gerevini. "Preferences and Soft Constraints in PDDL3". In: (2005). DOI: http://www.cs.yale.edu/homes/dvm/papers/pddl-ipc5.pdf.

[4] Amanda Smith Andrew Coles. *Greedy Best-First Search when EHC Fails*. URL: http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/node11.html#modifiedbestfs.

[5] Amanda Smith Andrew Coles. *Marvin: A Heuristic Search Planner with Online Macro-Action Learning*. URL: http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html.

[6] Paul E. Black. *Manhattan Distance*. URL: http://www.nist.gov/dads/HTML/manhattanDistance.html.

[7] Hector Geffner Blai Bonet. "Planning as Heuristic Search". In: *Artificial Intelligence* 129 (2001), pp. 5–33. DOI: http://www.cs.toronto.edu/~sheila/2542/s14/A1/bonetgeffner-heusearch-aij01.pdf.

[8] Hector Geffner Blai Bonet. "Planning as Heuristic Search: New Results". In: *Recent Advances in AI Planning* 1809 (1999), pp. 360–372. DOI: www.dtic.upf.edu/~hgeffner/html/reports/hsp-r.ps.

[9] Sergio Jimenez Carlos Lopez Malte Helmert. "Automating the Evaluation of Planning Systems". In: *AI Communications* 26 (2013), pp. 331–354. DOI: http://www.dtic.upf.edu/~sjimenez/publications/carlos-aicom13/carlos-aicom13.pdf.

[10] AIPS-98 Planning Competition Committee. "PDDL - The Planning Domain Definition Language version 1.2". In: (1998). DOI: http://homepages.inf.ed.ac.uk/mfourman/tools/propplan/pddl.pdf.

[11] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings Third Annual ACM Theory of Computing* (1971). DOI: http://www.cs.utoronto.ca/~sacook/homepage/1971.pdf.

[12] Subbarao Kambhampati Daniel Bryce. “A Tutorial on Planning Graph-Based Reachability Heuristics”. In: (2007). DOI: http://www.cs.toronto.edu/~sheila/2542/s14/material/brykam-relheur-aim07.pdf.

[13] Anne Parrain Daniel Le Berre. “The Sat4j Library, Release 2.2”. In: *Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64. DOI: http://www.cs.toronto.edu/~sheila/2542/s14/A1/hoffmannebel-FF-jair01.pdf.

[14] GNU. *GNU Parallel*. URL: http://www.gnu.org/software/parallel/.

[15] Google. *Validity Definition*. URL: https://www.google.fr/search?client=ubuntu&channel=fs&q=when+actions+are&ie=utf-8&oe=utf-8&gfe_rd=cr&ei=5epOV46QMcGAaPjxqqgC#channel=fs&q=validity+definition.

[16] Patrik Haslum. “Admissible Heuristics for Automated Planning”. In: (2006). DOI: https://www.ida.liu.se/divisions/aiics/publications/Thesis-2006-Admissible-Heuristics-Automated.pdf.

[17] Patrik Haslum. “Admissible Heuristics for Optimal Planning”. In: *Proceedings From AIPS* (2000). DOI: https://www.aaai.org/Papers/AIPS/2000/AIPS00-015.pdf.

[18] Malte Helmert. “Concise Finite-Domain Representation for PDDL Planning Tasks”. In: *Artificial Intelligence* 173 (2009), pp. 503–535. DOI: http://ac.els-cdn.com/S0004370208001926/1-s2.0-S0004370208001926-main.pdf?_tid=87eb7262-3162-11e6-afeb-00000aacb35f&acdnat=1465821173_d97ddbe9a2703f51dda458b7b95fe002.

[19] Malte Helmert. *Fast Downward Heuristic Documentation*. URL: www.fast-downward.org/Doc/Heuristic.

[20] Malte Helmert. “The Fast Downward Planning System”. In: *Artificial Intelligence Research* 26 (2006), pp. 191–246. DOI: http://www.jair.org/media/1705/live-1705-2731-jair.pdf.

[21] Jorg Hoffmann Henery Kautz Bart Selman. “SatPlan: Planning as Satisfiability”. In: *Abstracts of the 5th International Planning Competition* (2006). DOI: http://www.cs.rochester.edu/u/kautz/papers/kautz-satplan06.pdf.

[22] Jorg Hoffmann. “A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-Climbing Algorithm”. In: (2000). DOI: https://www.ics.uci.edu/~dechter/courses/ics-271/fall-06/project/j.hoffmann00.pdf.

[23] Jorg Hoffmann. *Automatic Planning PDDL*. URL: http://fai.cs.uni-saarland.de/teaching/winter13-14/planning-material/planning03-pddl-post-handout.pdf.

[24] Jorg Hoffmann. “FF: The Fast-Forward Planning System”. In: *Artificial Intelligence Research* 14 (2001), pp. 253–302. DOI: http://www.cs.toronto.edu/~sheila/2542/s14/A1/hoffmannebel-FF-jair01.pdf.

[25] Jorg Hoffmann. “Where ’Ignoring Delete Lists’ Works: Local Search Topology in Planning Benchmarks”. In: *Artificial Intelligence Research* 24 (2005), pp. 685–758. DOI: http://www.jair.org/media/1747/live-1747-2543-jair.pdf.

[26] ICAPS. *International Planning Competition*. URL: http://www.icaps-conference.org/index.php/Main/Competitions.

[27] Robert Givan Jia-Hong Wu Rajesh Kalyanam. “Stochastic Enforced Hill-Climbing”. In: *ICAPS* (2005). DOI: https://www.aaai.org/Papers/ICAPS/2008/ICAPS08-049.pdf.

[28] Bernhard Nebel Jorg Hoffmann. “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *Artificial Intelligence Research* 14 (2001), pp. 253–302. DOI: http://www.cs.toronto.edu/~sheila/2542/s14/A1/hoffmannebel-FF-jair01.pdf.

[29] John Franco Jun Gu Paul W. Purdon. “Algorithms for the Satisfiability (SAT) Problem: A Survey”. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (1996). DOI: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=67BBCE044D04F18F16567A9F3255B00C?doi=10.1.1.41.2697&rep=rep1&type=pdf.

[30] Henry Kautz and Bart Selmen. “Blackbox: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Workshop on Planning as Combinatorial Search* (1998). DOI: http://www.cs.rochester.edu/u/kautz/papers/aips98-kautz.ps.

[31] Paola Traverso Malik Ghallab Dana Nau. *Automated Planning, Theory and Practice*. San Francisco: Elsevier, 2004.

[32] Hector Geffner Malte Helmert. “Unifying the Casual Graph and Additive Heuristics”. In: *Advancement of Artificial Intelligence* (2008). DOI: http://www.dtic.upf.edu/~hgeffner/malte-icaps08.pdf.

[33] Derek Long Maria Fox. “pddl2.1: An Extension to pddl for Expressing Temporal Planning Domain”. In: *Artificial Intelligence Research* 20 (2003), pp. 61–124. DOI: https://www.jair.org/media/1129/live-1129-2132-jair.pdf.

[34] Encyclopedia of Mathematics. *Conjunctive Normal Form*. URL: http://www.encyclopediaofmath.org/index.php.

[35] Drew McDermott. “The 1998 AI Planning Systems Competition”. In: *AI Magazine* 21.2 (1998). DOI: http://www.aaai.org/ojs/index.php/aimagazine/article/view/1506/1405.

[36] Nils J Nilsson. *Principles of Artificial Intelligence*. Berlin: Springer, 1982.

[37] Hector Geffner Patrik Haslum Blai Bonet. “New Admissible Heuristics for Domain-Independent Planning”. In: (2005). DOI: https://www.aaai.org/Papers/AAAI/2005/AAAI05-184.pdf.

[38] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. USA: Addison Wesely, 1984.

[39] Bertram Raphael Peter E. Hart Nils J. Nilsson. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions of Systems Science and Cybernetics* 4.2 (1968). DOI: http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf.

[40] M. Fox R. Howey D. Long. “VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL”. In: *Tools with Artificial Intelligence* 16 (2004), pp. 294–301. DOI: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.7467&rep=rep1&type=pdf.

[41] Nils J. Nilsson Richard E. Fikes. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence* 2 (1971), pp. 189–208. DOI: http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/strips.pdf.

[42] Martin Muller Richard Valenzano Hootan Nakhost. “ArvandHerd: Parallel Planning with a Portfolio”. In: (2011). DOI: https://webdocs.cs.ualberta.ca/~mmueller/ps/arvandherd.pdf.

[43] Jorg Hoffmann Stefan Edelkamp. “PDDL2.2: The Language for the Classical Part of the International Planning Competition”. In: (2004). DOI: https://www.researchgate.net/publication/228567744_PDDL2_2_The_language_for_the_classical_part_of_the_4th_international_planning_competition.

[44] Peter Norvig Stuart J. Russell. *Artificial Intelligence: A Modern Approach*. New Jersey: Alan Apt, 2004.

[45] Richard Korf Stuart Russell. *Informed Search Algorithms*. URL: https://courses.cs.washington.edu/courses/csep573/11wi/lectures/03-hsearch.pdf.

— A —
Appendix Title Here

- The link at the bottom is where all the raw data files, code for the search algorithm and heuristic algorithms are.
- I was advised to use an online repository because of the large amount of test documents that were generated from testing purposes for this thesis.
- Please read the file titled 'ReadMe' which will explain the contents of the repository.

GitHub was used to provide all the results in a easy to access fashion.
http://github.com/aaronboyd92/MasterThesis