A Multi-Language Heuristic Driven HTN Planner in Python

Cael Milne

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
of the
University of Aberdeen.

Department of Computing Science

2022

Declaration

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged

Signed: 

Date: 2022

Abstract

This dissertation explores the concepts of HTN planning and reports on the design, implementation, and evaluation of a HTN planner. We invoke a selection of heuristics and search strategies to test the performance capabilities of the planner. The developed planner encourages future work and extensions through the developed feature of interchangeable components which can be used to alter or completely re-write search procedure. We focus on the HDDL input language along with the standards of the International Planning Competition (IPC) but also experiment with JSHOP.

Acknowledgements

I would like to direct my gratitude and appreciation to my project supervisor Prof. Felipe Meneguzzi for his unwavering support and enthusiasm throughout this project. I would also like to give thanks to all my friends and family for all their support.

Contents

Chapter 1

Introduction

Planning is all around us in everyday life, plans exist in all magnitudes from building plans for skyscrapers to simple meal plans. As humans, we create plans continuously on varying subjects without hesitation.

Planning is not a human-only capability since computers are also able to process information about an environment and concoct a plan. Currently, AI planning exists in many cutting-edge applications pushing the boundaries of technology. Arguably, the most exciting current development in technology is self-driving vehicles. Self-driving cars use planning to convert sensor data into a plan for manoeuvring the vehicle safely [35].

Robotics and AI planning are closely intertwined with reams of work published. Robot motion planning has been the subject of research and development for a while [27]. Wide-scale adoption of autonomous self planning robots has gripped warehouses. In such warehouses, a central planning system plans which tasks each robot should carry out [7].

As the world shifts more and more towards digital solutions, AI systems will only become more prevalent [44]. Increasing uptake in AI systems not only by engineers, researchers, and enthusiasts but by small businesses and organisations is essential. AI planning has the potential to be a common tool used for optimising everyday tasks, but systems need to be developed with inexperienced users in mind. As well as inexperienced users, people studying AI planning should have a system they are able to easily interact with, learn from, and modify.

AI planners can tackle more than strictly real world motion problems. AI researchers have been studying and creating planning systems that can produce plans for any given environment and goal. These planning systems take many forms such as Classical Planners [29], Hierarchical Goal Network (HGN) Planners [42] and, Hierarchical Task Network (HTN) Planners [15].

This report focuses on HTN planning and covers the design, implementation, testing, and evaluation of a HTN planning system. The proposed system will be compatible with two input languages and be driven by search heuristics. A further consideration is that of usability, creating a system that is easy to use and modify as future users see fit. By allowing for modifications to the system new features can easily be tested - such as heuristics and search procedures [23].

1.1 Motivation

Although the planning community has generated a number of HTN planner implementations [24, 33, 36, 20, 28, 40], these planners are not very easy for future users to modify and extend. Thus, our project aims to develop a HTN planner that is easily modifiable. A platform encouraging

people to get hands-on experience in HTN planning is key in building interest in the field, which in turn accelerates progress made.

1.2 Objectives

The main objectives of this project are to:

1. **Develop a HTN planner capable of receiving HDDL and JSHOP input languages.**

   The central objective of this project is to create a HTN planner that can operate using both HDDL and JSHOP problems. The system should be able to parse, search, and give output for both input languages.

2. **Include support for total ordered and partial ordered problems.**

   Planning problems can be either total ordered or partial ordered. Differences in approach is required when handling each distinct type of problem. This system should be able to solve both problem types.

3. **Implement at least one search heuristic to improve the efficiency of planning.**

   Searching for results can be a time-consuming task, especially in big problems. The system should include at least one search heuristic to make search more efficient.

4. **Build a Modifiable and Future Ready System.**

   The system architecture should be considerate of future additions and modifications. Such as new heuristics, input languages, and search procedures. Design patterns such as hexagonal[1] would allow for additions to the system without impacting the core components. Components of the system should be modular to allow for readable and easy to modify code.

5. **Utilise unit testing to verify the correctness of all aspects of the system.**

   Autonomous and consistent unit testing is an essential segment of any system that encourages modifications and additions. The unit tests should use known solutions to problems to validate the correctness of the system.

6. **Report on the Development and Evaluation of a HTN Planning System.**

   Reporting on the developed system should be in-depth and include details of decisions taken during the design and development stages. Descriptions of the functionality and intentionally expandable components need to be presented and explained.

[1] https://alistair.cockburn.us/hexagonal-architecture/

Chapter 2

Background & Related Work - Planning Formalisms

2.1 Planning

The aim of AI planning is to choose and organise Actions by anticipating their outcomes. Planning can be defined as having multiple types such as navigation planning and manipulation planning [16]. In this report domain independent planning will be the center of focus. Domain independent planning allows for any problem environment to be defined using the specific representations and knowledge required. Models of the problem environment are computationally evaluated and considered until the best course of action for the unique problem is found. A key element of AI planning is considering the conditions, which must be met before a specific Action can be executed. Models must be able to determine when it is possible to carry out an Action in the defined environment [16].

The Stanford Research Institute Problem Solver (STRIPS) [14] is an example of a popular AI planner in which, a model is defined using well-formed formulas of the first-order predicate calculus. The motivation for the STRIPS planner was to develop a system capable of considering large collections of facts. STRIPS also includes functionality for rules to be defined which are used to set practical limitations on what Actions the solver can carry out on the model. For example an item can only be in one location at a time [14]. Besides STRIPS, there is a vast selection of similar planning systems which have been designed for specific requirements [19].

The input language defined and used by STRIPS has been adopted for use in other planning systems. ARMS, SLAF and LOCM are examples of planning systems that utilise STRIPS [2]. The aim of the Action-Relation Modelling System (ARMS) is to be able to autonomously infer information regarding an Action based on names and parameter lists. ARMS uses the initial PDDL version (STRIPS) as its operational language [45].

2.1.1 Formalisms & Definitions

Classical Planning systems follow differing problem representations and definitions. Representations control what can and cannot be defined within the planning system. Examples of representations are *set-theoretic representation*, *classical representation*, and *state-variable representation*. Definitions of which are given by Malik Ghallab, Dana Nau, and Paolo Traverso [16] and are detailed in this section.

2.1.1.1 Set-Theoretic Representation

Set-theoretic representations are split into domains and problems.

Definition 1. Set-Theoretic Domains

- L is a set of propositional symbols
- $\sum = (S, A, \gamma)$ is a state-transition system.
- $S \subseteq 2^L$; S is the power set of L . $s \in S$; s is a state within S.
- A is the set of actions. $a \in A$; a is an action within A.
- $a = (precond(a), effects^-(a), effects^+(a))$ where:
  - $precond(a)$ is the preconditions of a
  - $effects^-(a)$ is the deletion effects of a
  - $effects^+(a)$ is the addition effects of a
- $effects^-(a) \cap effects^+(a) = \emptyset$
- An action a can only be applied to a state s if $precond(a) \subseteq s$.
- $\gamma(s,a) = (s - effects^-(a)) \cup effects^+(a) = s'$ is valid if $precond(a) \subseteq s$.

Definition 1 describes the make up of Set-Theoretic Domains. The set of propositional symbols L holds all the possible symbols to describe states, $L = \{onground, holding, at_1, at_2\}$. S is the set of all possible states in the domain, $S = s_0, ..., s_n$. A state $s \in S$ represents a possible configuration of the environment such as, $s = \{onground, at1\}$. Actions ($a \in A$) are used to modify the environment changing which state is currently being observed.

The action $take \in A$ is defined with three distinct sets. We can define the action $take = (\{onground\}, \{onground\}, \{holding\})$. The first set defined within an action is the preconditions ($precond(take)$). For an action to be executed the items in the preconditions set must be found in the environment state. The second set represents the delete effects ($effects^-(take)$), the elements provided in this set need to be removed from the environment state during execution of the action. The third and final set details the addition effects ($effects^+(take)$), this set is the opposite of the aforementioned $effects^-(take)$ set. The elements found in the $effects^+(take)$ set need to be added to the environment state. If we assume the current environment state is $s = \{onground, at_1\}$, upon executing the *take* action the resulting state would be $s = \{holding, at_1\}$.

Definition 2. Set-Theoretic Problems

- $P = (\sum, s_0, g)$
- $s_0 \in S$; s_0 is the initial state.
- $g \in L$; g is the goal propositions.
- The set of goal states can be denoted as: $S_g = s \in S | g \subseteq s$

Definition 2 defines Set-Theoretic Problems which combine with Domains defined in Definition 1 to create a searchable environment. Set-Theoretic Problems consist of two components an

initial state (s_0) and goal propositions (g). These components define the beginning and the ending of search.

Definition 3. Set-Theoretic Plans

- Plans are a sequence of actions $\pi = \langle a_1 ... a_k \rangle$; $k \geq 1$ and $|\pi| = k$.

Definition 3 lays out the make up of plans which are the solutions for given domains and problems. A plan is a sequence of actions that transform the initial state into a state which satisfies the goal propositions.

2.1.1.2 Classical Representation

Classical Representation builds upon the definitions of Set-Theoretic Representation using first order predicates.

Definition 4. Classical Representation of operators
$o = (name(o), precond(o), effect(o))$ where:

- $name(o)$ is the unique name of the operator
- $precond(o)$ is the preconditions which must be satisfied to use the operator
- $effect(o)$ is the effects the operator has on the current state

Unlike Set-Theoretic Representation shown in Definitions 1 and 2, preconditions and effects of Classical Representation operators are literals opposed to propositions. Definition 4 details the structure of operators in Classical Representation.

Definition 5. Classical Representation of Literals, Preconditions, and Effects

- For a set of literals L. L^+ is the set of all atoms in L, and L^- is the set of all atoms whose negations are in L.
- Similarly, $precond^+(o)$ is the positive preconditions of o while $precond^-(o)$ are the negative preconditions of o.
- $effect^+(o)$ is the positive effects of o while $effect^-(o)$ is the negative effects of o.

Figure 2.1 gives an example of an operator in the Classical Representation standards defined in Definition 4. The $\neg$ symbol signifies a negative precondition or effect, both of which are specified in Definition 5. The operator in Figure 2.1 has three preconditions, two positive and one negative. Positive conditions must be present in the environment state whereas, negative conditions must not be present in the environment state. Similarly, positive effects are predicates to be added to the environment state and negative effects are predicates to be removed.

2.1.1.3 State-Variable Representation

State-Variable Representation is similar to both Set-Theoretic and Classical Representations. The difference is the reliance on functions instead of relations. Predicates are treated like functions with information regarding the state of the environment being returned from function calls. Figure

move(r, l, m)
precond: adjacent(l, m), at(r, l), ¬occupied(m)
effects : at(r, m), occupied(m), ¬occupied(l), ¬at(r, l)

Figure 2.1: Classical Representation Operator Example

2.2 shows an example of function definitions. States s_0 and s_1 define the return values for function calls with specific parameters. In the case of s_0 the function *pos* can be called with three distinct parameters. If the parameter used is c_1 the returned result will be l_1 whereas, if c_2 is the parameter l_2 will be yielded.

$$s_0 = \{\text{loc}(r_1) = l_1, \text{load}(r_1) = \emptyset, \text{pos}(c_1) = l_1, \text{pos}(c_2) = l_2, \text{pos}(c_3) = l_2\}$$
$$s_1 = \{\text{loc}(r_1) = l_1, \text{load}(r_1) = c_1, \text{pos}(c_1) = r_1, \text{pos}(c_2) = l_2, \text{pos}(c_3) = l_2\}$$

Figure 2.2: State Variable Predicate Function Definition

Definition 6. State-Variable Representation Operator
Operator $o = (name(o), precond(o), effects(o))$ where:

- $precond(o)$ is a set of expressions on state variables and relations
- $effects(o)$ is a set of state variable values in the form $x(t_1...t_k)$

Figure 2.3 puts forth an operator definition in the State-Variable format presented in Definition 6. The preconditions of the operator compare the values given as results to function calls. The effects of the operator set new results to be returned for future function calls. The first effect sets the result for the *load* function when being called with the object corresponding to parameter *r*.

load(c,r,l)
precond: loc(r) = l, pos(c) = l, load(r) = ∅
effects : load(r) ← c, pos(c) ← r

Figure 2.3: State Variable Operator

Definition 7. State-Variable Domain
Domain $\sum = (S, A, \gamma)$ where:

- $S \subseteq \prod_{x \in X} D_x$ where:
  - X is the set of ground state variables
  - D_x is the range of ground state variable x
  - $c \in D_x$
  - $s = \{(x = c) \mid x \in X\}$
- R is the set of rigid relations
- A is the set of ground operators which meet the relations in R
- $\gamma(s, a) = \{(x = c) \mid x \in X\}$ where c is specified by an assignment in effects(a), otherwise $(x = c) \in s$

Definition 7 exhibits how the components of State-Variable Domains are cemented together to develop a searchable planning domain. State-Variable Representation rounds up our overview of planning formalisms. Understanding the formalisms which construct planning problems is only the tip of AI planning. Effectively navigating these planning problems to obtain a valid plan is the core objective of AI planning.

2.1.2 Searching

Now we have defined the bedrock of planning, we proceed to discussing how a planner can produce plans to problems. The three types of planning representation we covered - Set-Theoretic, Classical, and State-Variable - do not have any limitations of what search strategies can be employed. Strategies can often be adapted to satisfy each representation in a similar fashion.

The most basic search method is simply repeatedly applying all possible operators to all states until a solution is found [14]. Consider a planning problem with three defined actions *Action A*, *Action B*, and *Action C*. Beginning from a defined initial state, Figure 2.4 illustrates the search procedure of such a rudimentary algorithm. This sequence of expansion continues in a breadth-first or depth-first manner until a solution is found. This search strategy is valid but very inefficient with larger problems. Since the amount of states per layer increases exponentially - where a layer is described as a the set of all states with the same depth e.g. from Figure 2.4 layer 0 = {Initial State} and layer 1 = {State 1, State 2, State 3}. The total number of states in this type of search can be expressed as:

$$1 + \sum_{n=1}^{x} y^n$$

Where x is the amount of layers (excluding the initial layer) and y is the number of actions. Using the example in Figure 2.4, $x = 3$ and $y = 3$ meaning in total 40 states would be required. However, if 10 layers were required to find a solution then 88573 states would be used. If 5 actions were defined and 10 layers used then 12207031 states would be considered.

The explosion of states required to solve problems using this generic search algorithm is why planners add optimisations to make searching more efficient. Optimisations can take many forms

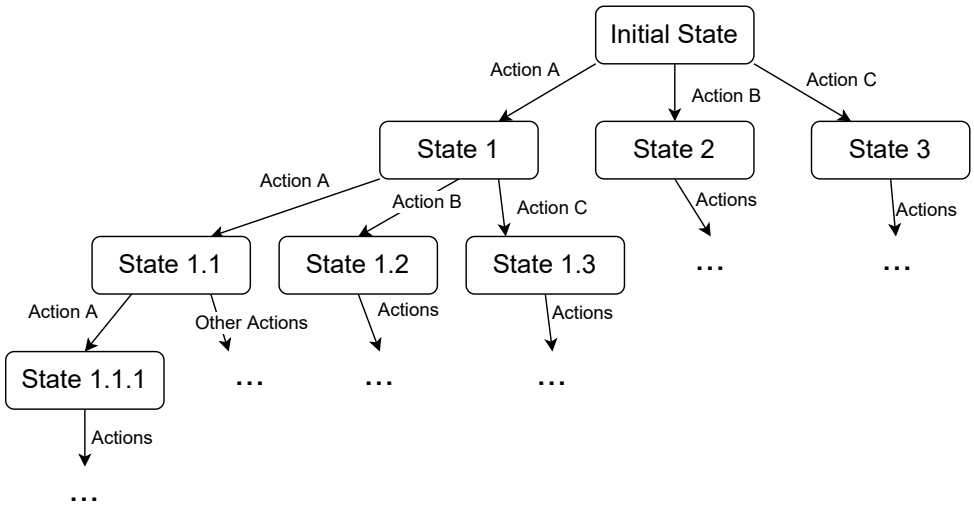


Figure 2.4: Generic Search

such as selecting the most promising states to expand first or checking for duplicate states to limit unnecessary searching.

2.1.3 PDDL

PDDL is the language used in the International Planning Competition's (IPC) classical track [9]. Since its inception, PDDL has undergone several stages of evolution. Each bringing different functionality for defining planning problems. A base version of PDDL is outlined by Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins for the AIPS-98 planning competition [1]. Although we do not use PDDL in this project, its relationship as the basis for HDDL makes it valuable to the understanding of this project.

PDDL provides processes for defining planning problem domains using the accepted information encoding. The PDDL syntax allows an environment to be contained in a model. The state of model is denoted using predicates.

Definition 8. PDDL Domain

A PDDL domain can be represented as a tuple $D = (L, T, A)$ where:

- L is the set of predicate logic
- T is the set of types
- A is the set of actions

PDDL uses domains to construct components capable of representing a searchable environment. Definition 8 outlines the definable components when creating a domain. PDDL uses types to differentiate objects into categories.

Definition 9. PDDL Type

A type $t \in T = (N, t_P)$ where:

- N is the type name
- t_P is the parent type

```
(:types
    location cube - object
    room - location
)
```

Figure 2.5: PDDL Type Definition

Definition 9 produces the definition of types. Figure 2.5 shows the definition of types in PDDL domains. In this example the types *location* and *cube* both have the parent type *object*, which is the base type and in present in all domains. The type *room* then uses the type *location* as its parent. Types can be assigned to objects and parameters.

Definition 10. PDDL Parameters

P is a set of parameters. $p \in P = (N, T)$ where:

- N is the parameter label
- T is type required

```
?l1 ?l2 - location
```

Figure 2.6: PDDL Parameter Definition

Parameters are defined in PDDL to detail the objects that are required to satisfy actions and predicates. Definition 10 shows the make up of parameters. Figure 2.6 is an example of parameter definition, the parameters *?l1* and *?l2* must satisfy the type *location*. If we again consider the type declaration in Figure 2.5, an object of type *room* can be passed as a parameter fulfilling the types of *room* and *location* as well as any other preceding parent types.

Definition 11. PDDL Predicate

A predicate $PR \in L = (N, O)$ where:

- N is the name of the predicate
- L is the optional list of parameters

```
(:predicates
    (at ?l - location)
    (unloaded)
)
```

Figure 2.7: PDDL Predicate Definition

Predicates are key to representing the state of the environment. Definition 11 puts forth the build up of predicates for PDDL. Predicates are defined as tuples with the first element denoting the predicate name and any subsequent elements denoting objects. Figure 2.7 gives an example of how predicates are defined in a PDDL domain. The first predicate *(at ?l - location)* is an example of a predicate with a parameter object. The second predicate *(unloaded)* is an example of a predicate with no parameters.

Definition 12. PDDL Actions

An action $A = (N, P, C, E)$ where:

- N is the name of the action
- P is a list of parameters
- C is a set of conditions
- E is the set of effects

```
(:action move
:parameters
       (?l1 - hallway
        ?l2 - hallway)
 :precondition
       (and (connected ?l1 ?l2) (at ?l1))
 :effect
       (and (not (at ?l1)) (at ?l2))
)
```

Figure 2.8: PDDL Action Definition

The final component of PDDL domains is Actions, which modify predicates and lead the model into new states. Definition 12 defines the sections of a PDDL Action. Actions have a set of parameters, preconditions and effects.

The Action shown in Figure 2.8 boasts a *:precondition* tag with the contents *(and (connected ?l1 ?l2) (at ?l1))*. In this case the predicates *(connected ?l1 ?l2)* and *(at ?l1)* must be in the environment state where, *?l1* and *?l2* are substituted for the objects given at run time. The action also contains two effects, the first is *(not (at ?l1))* which removes the predicate *(at ?l1)* from the state. The second effect is *(at ?l2)* which introduces a new predicate to the state.

Definition 13. PDDL Problem

A planning problem $P = (D, O, s_I, g)$ where:

- D is a planning domain
- O is a list of objects
- $s_I \in S$ is the initial state
- g is the goal description

All of the information relating to the domain is stored in the appropriately named *domain file*. Complementary to the domain file is the problem file, where the initial state, goal description, and objects are described. Definition 13 outlines the components present in a PDDL problem. The domain is a big part of a PDDL problem as it contains information regarding environment structure. During search the information contained in the domain such as action preconditions are essential to the search procedure.

The initial state is a set of predicates that describe the environment. The predicates in the

initial state are defined along with corresponding objects. An example of an initial state with only one predicate is as follows: *(:init (at room1))*.

The goal description details which conditions must hold in any solution. Plans for PDDL problems are produced when all goal conditions are met. Definition 3 from Section 2.1.1.1 details the composition of resulting plans.

Objects defined in Definition 14 are rather simple components of a PDDL problem, objects simply have a name and a type - which must be defined in the domain. Figure 2.9 gives the definition for two objects both with the type *room*. The objects defined are free to be passed as parameters to Actions and predicates.

Definition 14. PDDL Object
An object $O = (N, T)$ where:

- N is the object name
- T is the object type

```
(:objects
    room1 room2 - room
)
```

Figure 2.9: PDDL Object Definition

PDDL does have some assumptions about the environment it operates in. Firstly, the models of actions are deterministic, i.e. carrying out an action from a specific state will always result in the same state being reached. Secondly, the initial state is known and well-defined. Lastly, The environment is static i.e. the environment only changes when the planner performs an Action [18].

2.2 Input Languages

The input languages which we use to define domains and problems for this project are HDDL and JSHOP. These input languages have similarities in functionality but are very different in syntactical composition.

2.2.1 HDDL

The Hierarchical Domain Definition Language (HDDL) is an extension to PDDL designed specifically for hierarchical planning. The PANDA input language [20] was also used as a basis for HDDL. HDDL builds upon PDDL with the inclusion of constructs for task decompositions, specifically, methods and task orderings (total or partial). Further than syntactical definition changes HDDL adds compatibility for constraints to be added outwith the precondition scope provided by PDDL. Whereas HDDL has multiple versions and interpretations, the standard of HDDL we will be following is the requirements defined for the International Planning Competition.

HDDL operates under slightly different definitions as those traditionally used to represent HTN problems [21].

Definition 15. HDDL Formalism

HDDL uses a lifted formalism $L = (P, T, V, C)$. Where:

- P is the set of Predicates
- T is the set of Types
- V is the set of typed Variables
- C is the set of typed Constants

All of which are finite sets.

Definition 15 details some of the key components that underpin the architecture of HDDL and build upon the already established HTN features. Firstly, predicates (P) and types (T) are present in HDDL in the same manner as PDDL. Following on from predicates and types, objects also remain the same in HDDL as their counter parts in PDDL. The set of typed variables (V) refers to the set of parameters, which are again inherited from PDDL. Definitions 11, 9, 10, and 14 from Section 2.1.3 define predicates, types, parameters, and objects respectively.

```
(:constants a - A)
```

Figure 2.10: HDDL Constant Definition

The set of typed constants (C) is unique to HDDL. Constants are objects which are essential to solving and are defined in the domain of a problem to guarantee availability during search. Since constants are simply objects which are defined in the domain of a problem they follow the same definition as objects (Definition 14). Figure 2.10 gives an examples of a constant definition, object a is defined with type A.

Definition 16. HDDL Task Network

A Task Network can be denoted as a tuple $tn = (I, \prec, \alpha, VC)$ given the set X of task names.

- I is the set of task identifiers
  - $I = (N, P)$ where N is the the name of the task to be used and P is a collection of objects to be used as parameters
- $\prec$ is the ordering over I
- α: $I \rightarrow X$ maps task identifiers to task names
- VC is the set of variable constraints

A task network is a collection of Tasks, Methods, and Actions that are still to be decomposed. Definition 16 defines the composition of a task network. A task network can be considered like a recipe, the ordering of the steps are essential. In the event of items being removed or added to the task network maintaining order is a top priority. For example if a task network contains two tasks *m1* and *m2*. When *m1* is expanded to expose subtasks s1 and *s2*. Then *s1* and *s2* must be expanded before *m2*. $\prec$ maintains the order of expansion for all identifiers in the task network.

```
:constraints (and (not (= ?x ?y)))
```

Figure 2.11: HDDL Parameter Constraint

The set of variable constraints (VC) refers to parameter constraints, which are comparable to preconditions with regards to structure and evaluation. These constraints are used to check the validity of parameters given to methods. Parameters can be required to be equal or different using the operators available. Figure 2.11 gives an example of a parameter constraint which simply asserts that parameters *?x* and *?y* are not equal.

Definition 17. HDDL Domain
A planning domain $D = (L, T_P, T_C, M)$ where:

- L is the set of predicate logic
- T_P is the set of primitive tasks
- T_C is the set of compound tasks
- M is the set of decomposition methods

Definition 17 details the components of a HDDL domain. The first component mentioned is predicates which has already been discussed earlier in this section. The remaining components are primitive tasks, compound tasks, and decomposition methods. In HDDL these elements take the form of Actions, Tasks, and Methods respectively. Actions in HDDL follow the same composition as PDDL Actions as displayed in Definition 12.

Definition 18. HDDL Task
A Task $T = (N, P)$ where:

- N is the task name
- P is a collection of parameters

Definition 18 constructs HDDL Tasks which are rather simple components, consisting of only a name and list of parameters. During search a Task can be decomposed by multiple methods, each method yields differing subtasks to the task network. Figure 2.12 exhibits an example of a task.

```
(:task swap
    :parameters (?x ?y)
)
```

Figure 2.12: Swap task in Basic problem

Definition 19. HDDL Method

A Method $M = (N, P, T, C, R, S)$ where:

- N is the name of the method
- P is a collection of parameters
- T is a tuple $(T_N T_P)$ where:
  - T_N is the name of the parent task
  - T_P is a list of parameters received from the parent task during decomposition
- C is the set of conditions
- R is the set of parameter constraints
- S is a list of tuples $(S_N S_P)$ where:
  - Each tuples defines a subtask that needs to be expanded
  - S_N is the name of a task/action
  - S_P is a collections of parameters to be passed to the subtask

Definition 19 establishes the formal definition of a Method. For the purposes of explanation we will use Figure 2.13 as an example. The Method put forth decomposes the Task shown previously in Figure 2.12. The inclusion of a parent Task is unique to Methods, it is used to specify which Task the Method can decompose. In this case the *swap* Task can be decomposed, the parameters *?x* and *?y* are specified by the Task. In some scenarios the Task will only supply some of the parameters, in which case the others must be selected. The specification of subtasks is similar to the effects of Actions however, instead of modifying predicates subtasks add Tasks and Actions to the task network.

```
(:method have_first
    :parameters (?x ?y)
    :task (swap ?x ?y)
    :precondition (and
      (have ?x)
      (not (have ?y))
    )
    :ordered-subtasks (and
      (drop ?x)
      (pickup ?y)
    )
)
```

Figure 2.13: have_first method in Basic problem

Definition 20 outlines the components of a HDDL problem. Most of the entries are the same as PDDL problems defined in Definition 13. The only difference is the inclusion of the initial task network (tn_I), which sets out which Tasks must be fully decomposed during search along with any parameters. Figure 2.14 details an initial task network with one Task along with objects to use as

parameters.

Definition 20. HDDL Problem

A planning problem $P = (D, s_I, tn_I, g)$ where:

- D is a planning domain
- $s_I \in S$ is the initial state
- tn_I is the initial task network
- g is the goal description

```
(:htn :subtasks (swap banjo kiwi))
```

Figure 2.14: Initial Task Network Definition

As well as standard preconditions, HDDL also includes functionality for *for-all* conditions which evaluate a predicate condition for an entire collection of objects. Definition 21 specifies these conditions. The selector (S) is used to select all objects of a specific type. All of the chosen objects are evaluated against the predicate condition (P). A *True* result is only achieved if all selected objects satisfy P.

Definition 21. HDDL For-All Condition

A for-all condition takes the form $P = (S, P)$ where:

- S is a selector in the form of a tuple (S_N, S_T) where:
  - S_N is the parameter name to be used when evaluating each selected object
  - S_T is the type of object to select
- P is a predicate condition to be evaluated for each selected object

Figure 2.15 shows an example of a for-all condition in a HDDL problem. In this case the selector is the statement *(?a - A)* which selects all the objects of type A. These selected objects are each evaluated by the predicate condition *(foo ?a ?b)* where *?a* is the selected object and *?b* is a given object.

```
(forall (?a - A) (foo ?a ?b))
```

Figure 2.15: Forall condition

2.2.2 JSHOP

The Simple Hierarchical Ordered Planner (SHOP) was developed as an entire planning system with its own language and planning algorithm [31]. SHOP has multiple variations and extensions like HDDL, in this paper we are concerned with the JSHOP standard.

Many of the aspects we focus on within JSHOP are similar or have related counterparts in HDDL. A major distinction between JSHOP and HDDL is the syntax. JSHOP only defines Methods and Operators which are comparable to HDDL's Methods and Actions. Tasks however, are present in JSHOP but in a discrete capacity.

When considering a JSHOP Method it could be understood to include a named Task and multiple unnamed decompositions, where each decomposition is comparable to a HDDL Method. Definition 22 sets out the composition of Methods in JSHOP. When considering the tuple (H,S), H could be seen as a HDDL Task (Definition 18) and each $s \in S$ is similar to a HDDL Method (Definition 19). A JSHOP Method can have multiple decompositions but, only the first valid decomposition is used during search [21].

Definition 22. JSHOP Method

A Method $M = (:method, H, S)$ where:

- $H = (N, P)$ where:
  - N is the name of the method
  - P is a list of parameters
- S is a set of tuples (C, E) where:
  - C is a set of conditions
  - E is a set of subtasks

Each tuple $s \in S$ defines a possible decomposition of the method.

JSHOP Operators are comparable to PDDL's Actions defined in Definition 12 with the only difference being the syntactical composition. Definition 23 details the formal make-up of operators.

Definition 23. JSHOP Operator

An Operator $O = (:operator, H, C, D, A)$ where:

- H is the primitive task
  - $H = (N, P)$ where:
  - N is the operator name - which must begin with !
  - P is a list of parameters
- C is the conditions which must be satisfied before the operator can be carried out.
- $D \in S$ is the predicates to be deleted from the state
- A is the predicates to be added to the state

Like HDDL, JSHOP offers compatibility for *for-all* conditions. However, where HDDL only allows for objects to be selected based on type, JSHOP selects objects based on a condition. Definition 24 describes for-all conditions in JSHOP.

Definition 24. JSHOP For-All Condition

A For-All Condition takes the form $P = (N, S, P)$ where:

- N is the parameter name to be used when evaluating each selected object
- S is a selector in the form of a list of predicate conditions
- P is a predicate condition to be evaluated for each selected object

JSHOP includes support for for-all effects in Operators. Definition 25 describes the formation of JSHOP's for-all Effects.

Definition 25. JSHOP For-All Effect

A For-All Effect takes the form $P = (N, S, E)$ where:

- N is the parameter name to be used when evaluating each selected object
- S is a selector in the form of a list of predicate conditions
- E is a set of effects to be executed for each selected object

Figure 2.16 displays an example for Effect. In this example each selected *?z* which satisfies the conditions *((package ?z) (in ?z ?t))* is used to add the predicate *(at ?z ?y)* to the state.

```
(forall (?z) ((package ?z) (in ?z ?t)) ((at ?z ?y)))
```

Figure 2.16: For-all Effect in JSHOP

Objects in JSHOP problems are defined differently to their counterparts in HDDL. Whereas, in HDDL objects are explicitly defined, JSHOP objects are not. Instead, objects in JSHOP are used in predicates without any prior definition.

2.3 International Planning Competition

HTN planning has recently gained some traction and was a part of the International Planning Competition (IPC) for the first time in 2020. HDDL was chosen as the common language for all planners in the contest, so a close connection was maintained between the HTN and Classical Planning contests [3]. The HTN planning contest was split into sections of the partial order track and the total order track [4].

Chapter 3

Background & Related Work - Planning Approaches

3.1 HTN Planning

Building upon the classical domain independent planning from Section 2.1, Hierarchical Task Network (HTN) planning introduces some differences which make it faster to compute and more scalable. HTN planning makes use of a defined initial state and a defined objective state. The planner makes use of the described primitive and compound tasks to find a network of tasks which lead from the initial state to the objective state. A primitive task is one which the planner can execute immediately, whereas a compound task is a larger task which needs to be split into multiple subtasks until only primitive tasks remain. Compared to Classical Planning, HTN planning depends upon more detailed domain knowledge being defined which in turn helps provide details on solving the problem given [15].

A paper written by Kutluhan Erol, James Hendler, and Dana S. Nau outlines the basic planning procedure required for HTN planning. As well as the basic functions of HTN planning, more technically complicated additions are also mentioned briefly such as heuristics [12]. This breakdown of the basic principles is a great starting point when considering the technical components of a HTN planner.

Problems in HTN planning can be defined as totally or partially ordered, which require total-order decomposition strategies or partial-order decomposition strategies respectively. Totally ordered problems mean that the subtasks corresponding to a Method are defined in the order they must be executed. Whereas partial ordered subtasks only contain some rules corresponding to the ordering of subtasks, for example subtask A must be carried out before subtask B. There is sophisticated planners developed for both total and partial orderings. During an experiment between total ordering and partial ordering it was found that total-order forward search is better suited to tackle complex problems [32].

Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomás Balyo [41] define HTN planning problems as follows.

Definition 26. HTN Planning Problem

The planning problem P is a tuple. Where $P = (s_0, g, T, O, M)$

- s_0 is the initial state
- g is the goal conditions
- T is the set of tasks
- O is the set of operators
- M is the set of methods

The definition of HTN planning problems detailed in Definition 26 outlines the core aspects of HTN planning. The initial state (s_0) and goal conditions (g) are the same as those defined for PDDL problems in Definition 13. An Operator ($o \in O$) is the same as PDDL Actions which are defined in Definition 12. The set of Tasks (T) and Methods (M) are unique of HTN planning. These extra components are necessary due to the search strategy of decomposition used by HTN planning. A Task can be decomposed by Methods which belong to it. During decomposition a Method replaces the Task being decomposed with a set of Tasks and Operators. Operators are only applicable when a Method introduces it during decomposition. This strategy limits what Tasks, Methods, and Operators can be applied during search.

Definition 27. HTN Solution

A solution to a planning problem P can be defined as $\pi = \langle a_1 ... a_k \rangle$ where:

- $a \in \pi$ is a task, method, or action
- $k = |\pi|$
- If $k = 0$ then s_0 satisfies g
- If $k \geq 1$ then when each $a \in \pi$ is applied to some state s in order, s will satisfy g assuming that initially $s = s_0$.

Resulting plans to HTN problems are similar to plans of State-Theoretic problems defined in Definition 3. Definition 27 details the architecture of HTN plans. Instead of a plan consisting strictly of Actions, HTN plans can contain Tasks and Methods. In some cases HTN plans are displayed in the same format as PDDL plans, by only including the Actions used.

HTN planners are not limited to one type of input language, each language provides differing functionality which can suit particular scenarios. Examples of input languages include HDDL [21], SHOP [31], HTN-PDDL [17], HATP [10], and ANML [43].

3.1.1 HTN Planning Components

HTN planning problems are constructed of components such as Tasks, Methods, Operators, and predicates to create a searchable environment.

3.1.1.1 Tasks, Methods, and Actions

Tasks are used as the starting point during search. Tasks are defined with parameters that must be fulfilled during search, Tasks also have a selection of Methods which can be used to decompose them using the parameters defined.

Methods are essentially the 'middle-men' of HTN planning. Methods contain subtasks which are to be carried out upon decomposition, these subtasks can be either Tasks or Actions. In HDDL once an Operator has been grounded it is known as an Action. Grounding refers to the process after parsing where Tasks, Methods, and Operators are prepped for search. Actions interact with the state of the environment by adding or removing predicates. Actions in HTN problems contain the same properties as Actions in PDDL defined in Definition 12.

3.1.1.2 Goal Conditions

Search ends when no non-primitive tasks remain in a task network. At this point, all tasks in the task network are fully executable from the initial state. Some formalisms also allow the inclusion of goal conditions to be checked in the state resulting from executing all primitive tasks. Goal conditions can be considered similarly to preconditions that are evaluated at the end of search to verify the final state of the search model is valid.

3.2 Search Algorithms

In this section we consider search algorithms and the different procedures available during search. The book *Artificial Intelligence: A Modern Approach* by Stuart J. Russell and Peter Norvig explores the algorithms we discuss in this section [38].

3.2.1 Breadth First Search

Breadth First Search is a basic search procedure which assumes that all expansions have the same cost. Nodes are expanded based on the number of predecessor nodes, with the least nodes being expanded first. In practice this means that the root node is expanded, then all the successors of the root node, than all their successors. This continues until all nodes have been searched, or a goal is found. This strategy ensures the shortest path is always found. Listing 3.1 shows the pseudocode algorithm for *Breadth First Search*; this algorithm uses a *FIFO* (First In, First Out) queue to order nodes. Since the algorithm adds all children of an expanded node to the queue, we can guarantee that the nodes are dispensed in the aforementioned order of number of predecessor nodes.

Figure 3.1 visualises the order of expansion using *Breadth First Search*, with the boxes representing nodes. The directed lines indicate which nodes are children of a node. The numbers on the nodes illustrate the order of node expansion.

3.2.2 Depth First Search

Depth First Search is essentially the opposite of *Breadth First Search* where instead of ordering nodes based on the least amount of successors, nodes are ordered based on the most successors. The *Breadth First Search* algorithm detailed in Listing 3.1 can be easily adapted to accommodate *Depth First Search*. If the *FIFO* queue is substituted for a *LIFO* (Last In, First Out) queue then the node with the most successors will be dispensed from the queue. Unlike *Breadth First Search* this search strategy is not guaranteed to find the shortest solution. An issue with *Depth First Search* is infinite loops which send the algorithm down an infinite path which never yields a solution, this makes the algorithm incomplete. Figure 3.2 visualises the execution order of the *Depth First Search* algorithm.

```
function Breadth−First−Search(problem)
    frontier <− a FIFO queue
    add Node(problem.Initial_State) to frontier
    reached <− {Initial_State}

    while frontier.LENGTH > 0 DO
        n <− frontier.pop()
        for each child in EXPAND(n)
            if Is_Goal(child)
                return child
            if child.State is not in reached
                add child.State to reached
                add child to frontier
    return ∅

function Expand(problem, node)
    s <− node.STATE
    for each action in problem.Actions(s) DO
        s' <− problem.Result(s, action)
        cost <− node.Path−Cost + problem.Action−Cost(s, action, s')
        yield Node(State=s', Parent=node, Action=action, Path−Cost=cost)
```

Listing 3.1: Breadth First Search Pseudocode

3.2.3 Uniform Cost Search

The previously seen *Breadth First Search* and *Depth First Search* algorithms assume that each expansion costs the same, in cases where expansions do not cost the same a new strategy is required. *Uniform Cost Search* or *Dijkstra's algorithm* adapts the *Breadth First Search* procedure shown in Listing 3.1 to operate using path cost. For *Uniform Cost Search* the *FIFO* queue is exchanged for a priority queue which orders nodes based on the cost of the path so far plus the expansion cost. Each node must only appear in the priority queue once with the lowest path cost possible. Figure 3.3 visualises this ordering of execution. The values beside the lines represent the cost of expansion, the number inside the squared box within the nodes is the total cost of expanding the node. As with previous diagrams the number inside the node represents the order of expansion.

3.2.4 Greedy Best First Search

Greedy Best First Search can be achieved by adapting the *Breadth First Search* algorithm proposed in Listing 3.1. In this case the *FIFO* queue is substituted for a priority queue that orders nodes based on the estimated distance to goal. Estimating the distance to goal requires a domain specific heuristic function. As nodes are being added to the priority queue the heuristic function estimates the distance from the node to the goal then it is ordered appropriately in the queue. Figure 3.4 visualises the search procedure of *Greedy Best First Search*, the numbers at the top of a node do not denote order of expansion but order of discovery. The values contained within the box inside a node show the estimated cost to goal by a heuristic.

3.2.5 A-Star Search

A-Star Search combines the properties of *Uniform Cost Search* and *Greedy Best First Search* from Sections 3.2.3 and 3.2.4 respectively. *A-Star Search* adapts the algorithm described in Section 3.2.4 by also taking path cost into consideration when ranking nodes in the priority queue. The

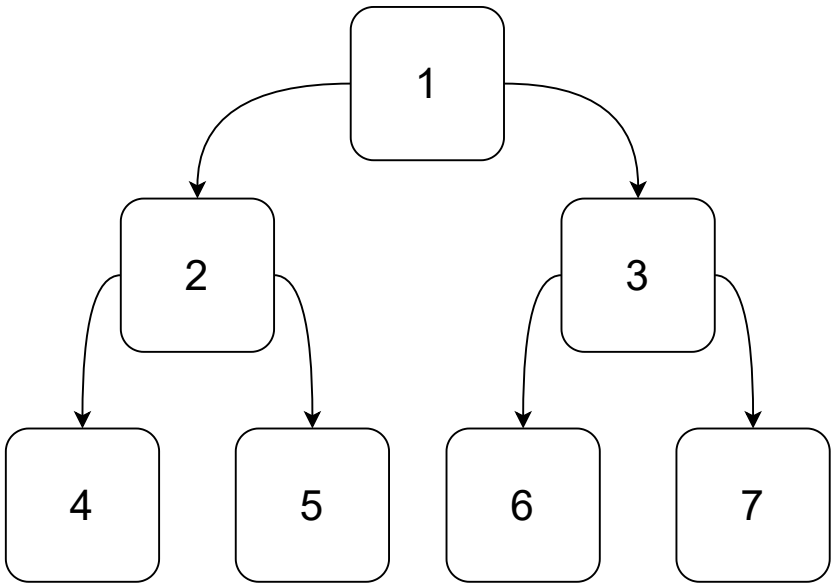


Figure 3.1: Breadth First Search Example

priority queue orders nodes based on the cost so far plus the estimated cost to goal provided by a heuristic. Figure 3.5 gives an example of *A-Star Search* in operation. In this diagram the values in the box within nodes are *heuristic estimation + cost so far = total cost*. The goal node is denoted with a *G*. We can see that the first time the goal becomes reachable is quite early on, after the second expansion the goal can be reached. *A-Star Search* does not automatically go for the goal when it is in reach, it continues to expand nodes in order in case a shorter path exists.

3.3 Task Ordering & Search

Forward decomposition means the search algorithm starts with the initial state and searches forwards until the goal is found. All search functionality for this project will be focused on forward search. Backward search -where the algorithm begins at the goal [34]- will not be in the scope of this project.

Total-orderings and partial-orderings were previously discussed in Section 3.1. Malik Ghallab, Dana Nau, and Paolo Traverso define algorithms for total-order forward decomposition and partial-order forward decomposition algorithms [16]. The algorithms defined are recursive and contain no structure to manage options during search, such as the queue used in the search strategies mentioned in Section 3.2. These algorithms are useful for considering how search in HTN space should progress. Applying primitive tasks to the state and decomposing non-primitive tasks which modify the task network are both handled in these algorithms.

3.4 Planning Heuristics

Heuristics are used to guide search algorithms efficiently towards a solution. In HTN planning such attempts to guide planners have taken many forms such as SHOP2's approach using state-based features to decide a course of action. Similar methods which determine the progress of search exist in other planners [23].

Although these solutions can be effective, they differ from the pure heuristic driven approaches found in Classical Planning. To further improve the efficiency of search, heuristics from

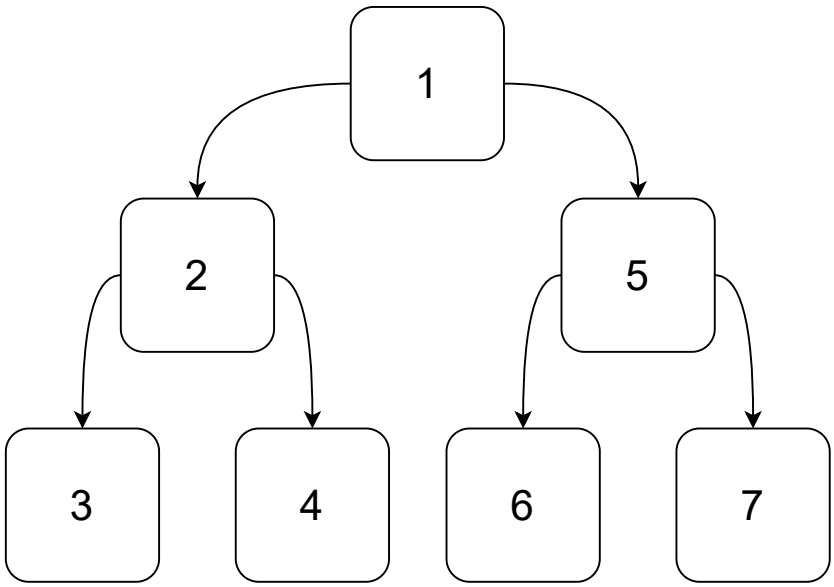


Figure 3.2: Depth First Search Example

Classical Planning can be applied to HTN planning if the HTN model is simplified to closely resemble a Classical Planning problem [23].

An example of a heuristic which uses a transformation to a Classical Planning problem is the *Delete and Ordering Relaxed Heuristic* [22]. This heuristic estimates the distance until a task network is fully decomposed by employing a bottom up reliability strategy. A key feature of this heuristic is that all delete effects of Actions and subtask orderings are completely ignored. This creates some interesting states, since predicates and their inverses can be in the state at the same time. For example *(have, A)* and *not(have, A)* can coexist in the state. When an Action is applied to the state its name is also added, since a Method is applicable when all its subtasks are present in the state. When a Method is decomposed the name of the Task it decomposes is added to the state. This strategy is iterative; each iteration collects all Actions and Methods which are applicable to the state. This continues until either all Tasks in the task network are present in the state or an iteration where no Actions or Methods are applicable. The number of iterations taken to add all Tasks in the task network to the state is returned as the estimated distance to goal.

The STRIPS planner follows a search procedure similar to that defined and used by the GPS system [11] but with a few additions. Similarly to the GPS system, the STRIPS planner considers the differences between the current state and the goal state before choosing which operator is the most likely to lead to a state closer to the goal. STRIPS stores each state in a search tree, a heuristic function is used to determine which state should be explored first. This function considers the number of goals remaining to be satisfied, amount of predicates remaining and their types, and the complexity of difference from the goal state [14].

3.5 Existing Planners

3.5.1 NOAH

The original HTN planner can be traced back to 1975 with the Nets of Action Hierarchies (NOAH) planner [15]. The paper *A Structure for Plans and Actions* details the inner workings of the NOAH

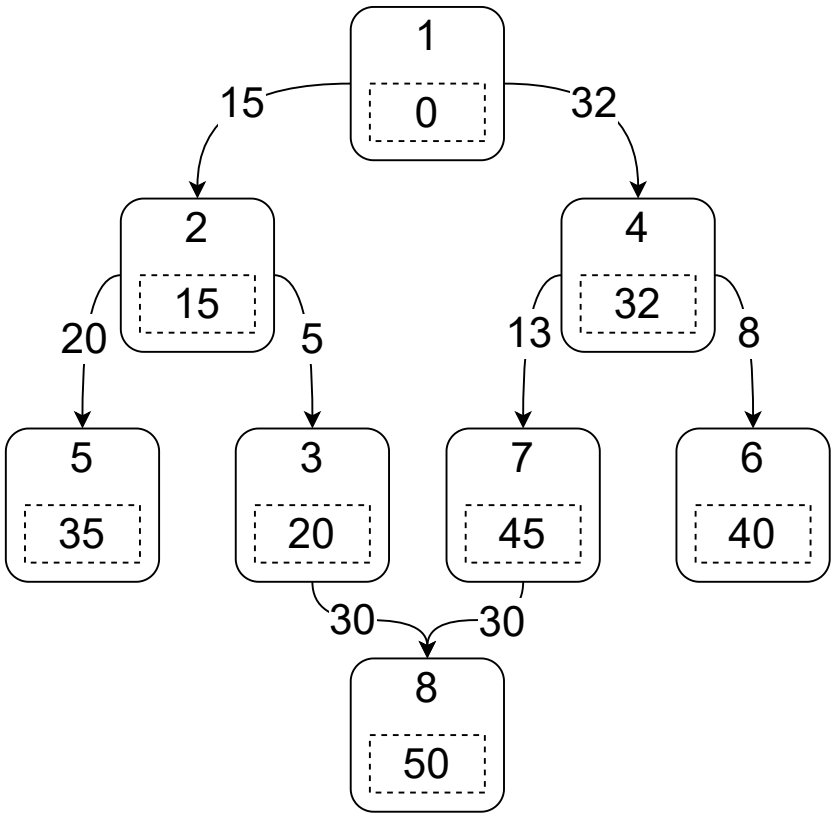


Figure 3.3: Uniform Cost Search Example

system. The NOAH system operated similarly to modern day HTN planners by expanding from and initial state via Actions until a goal is found. NOAH boasted an interesting feature of being able to save progress and continue execution at a later point in time. This was particularly useful when the system required more knowledge about its in environment to be defined before moving on [39].

3.5.2 GTOHP

The Ground Total Order Hierarchical Planner (GTOHP) is a planner which focuses on the concept of grounding a problem. Grounding a problem consists of computing all the possible instances of Operators and Methods. All typed parameters are replaced with constants, although the number of grounded instances increases rapidly when larger problems are considered. This strategy is able to find solutions to problems quickly. When compared to a SHOP planner GTOHP clocked in with better times in all cases tested [37].

3.5.3 HyperTensioN

HyperTensioN was the IPC 2020 HTN winner for the total-order track [4].

The HyperTensioN system is split into two parts. The first section containing the parsers, middle-end, and compilers in a module called *Hype*. The second section is the *HyperTensioN* module which solves the planning problem with the Ruby compiler. HyperTensioN provides functionality for multiple input languages and is built with support for new languages in mind. The current languages supported are PDDL, JSHOP, and HDDL [28]. The parsers from HyperTensioN could prove to be a useful point of reference for this project.

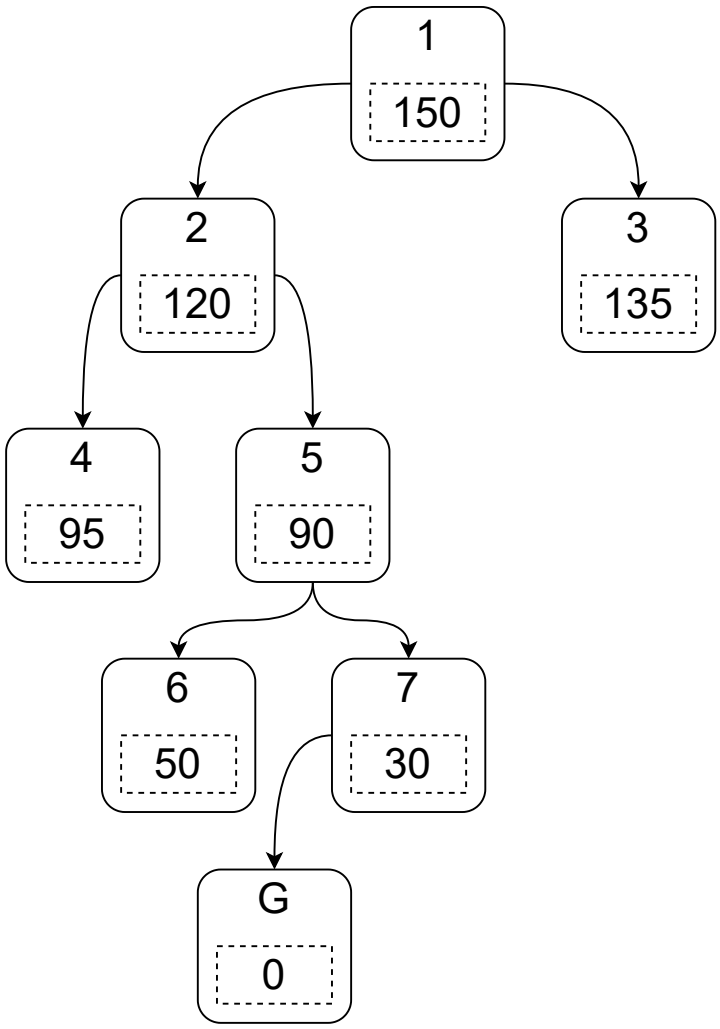


Figure 3.4: Greedy Best First Search Example

3.5.4 SIADEX

SIADEX was the IPC 2020 HTN winner for the partial-order track and was also a participant in the total-order track [4].

SIADEX does not use any heuristic guidance and instead relies on task unification and pre-condition unification when deciding on which actions to carry out [13]. SIADEX's partial-order ability might be a good examples for this project when considering functionality for partial-ordering.

3.5.5 LiloTane

LiloTane was the IPC 2020 HTN runner up for the total-order track [4].

Lifted Logic for Task Networks (LiloTane) is a Satisfiability based HTN planning system. LiloTane is only compatible with totally-ordered problems and operates using a lifted planning problem. Components from other planners are key in the design and operation of LiloTane, the parser used is PANDA's *pandaPIparser* [5], and the planning operations are similar to those found in *Tree-REX* [41] and *totSAT* [6]. LiloTane avoids grounding problems like a selection of other planners do, to thwart efficiency issues with large input problems. Instead, the lifted planning problem used by LiloTane is represented as propositional logic.

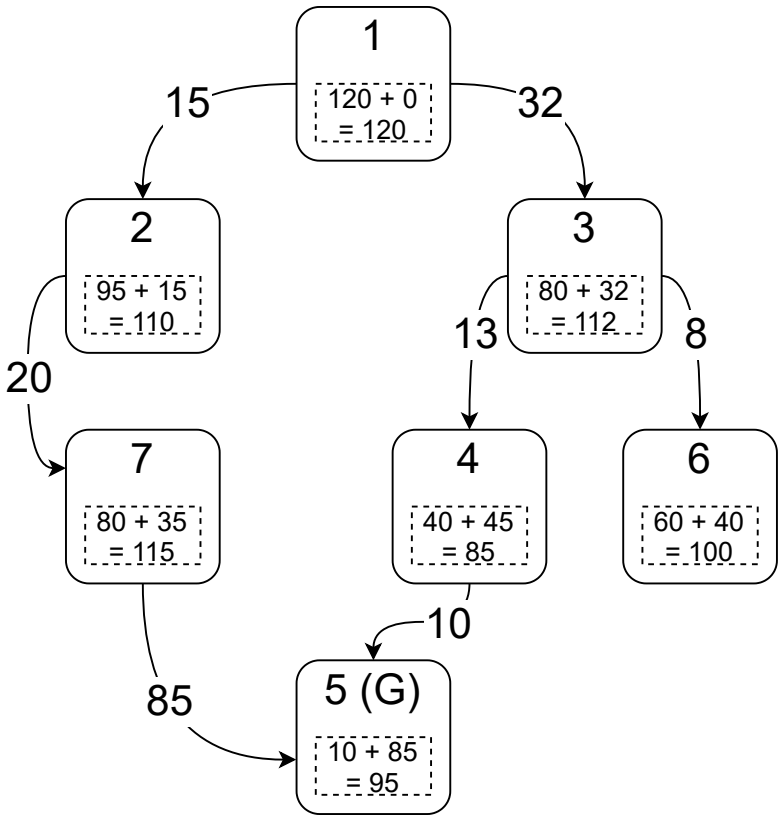


Figure 3.5: A-Star Search Example

During solving a layered approach is adopted with each layer containing a set of positions. Each position is acted on by a different operator to create the positions in the next layer of the algorithm. LiloTane attempts to be conservative when applying Methods to positions by only applying Methods that's subtasks are required, opposed to applying Methods with all possible parameters. To further improve the efficiency of solving, Methods and Actions with preconditions that are not satisfied by the current state are instantly rejected and not considered. Each of the remaining Methods are ranked based on its expected outcome. LiloTane prunes options during search by combining a Methods preconditions with that of its subtasks, if the resulting precondition holds then the Method can be applied. This prevents preconditions not being satisfied later in the search [40].

3.5.6 PyHiPOP

PyHiPOP competed in both the partial-order and total-order tracks; coming runner-up in the partial-order contest [4].

PyHiPOP operates by utilising prepossessing and grounding methods along with heuristics to solve planning problems. During the search, nodes which have already been visited are pruned to prevent duplicate searching [26]. Since PyHiPOP is written in Python it's code base could prove useful when considering design solutions for this project.

3.5.7 PDDL4J

PDDL4J took part in both the partial-order and total order-tracks in the IPC's 2020 HTN contest [4].

PDDL4J is a library of planning resources; within the library is a few different planners which utilise opposing search methods. Such planners include Totally Ordered Fast Downward and Partial Ordered Fast Downward planners [36].

3.5.8 The PANDA Planner

Planning and Acting in a Network Decomposition Architecture (PANDA) is a HTN planning system written in C++. PANDA puts emphasis on evolving with new search techniques to improve planning capabilities. Currently, PANDA uses HDDL and supports partially-ordered tasks as well as task insertion. PANDA also allows for plan search and progression search.

In plan search PANDA offers a selection of heuristics. Originally PANDA used landmarks which are Tasks that need to be in every plan, and are calculated using a Task Decomposition Graph. However, modern heuristics calculate the more common cost to reach goal. This cost is underpinned by assigning each Action, Method, and abstract Task a cost. Two heuristic methods use this cost information TDG_c and TGD_m. TDG_c calculates the estimated cost of Actions that need to be carried out to reach the goal whereas, TGD_m is used to estimate distance to goal. This makes TDG_c suitable for guiding a search algorithm and TGD_m useful for finding efficient solutions.

Progression search is handled in PANDA by only processing Tasks with no predecessor in the ordering relations. This means the planner commits to all Tasks being totally-ordered. Since heuristics for progression search are the same as heuristics used in Classical Planning some pre-possessing needs to be done. The problem being solved is modelled into a Classical Planning problem, which allows for Classical heuristics to be used [20].

The PANDA system operates using three separate tools. *pandaPIparser* [5] is the tool which focuses on parsing HDDL problems. The output from *pandaPIparser* can be configured to one of three options JSHOP2, HPDL, or *pandaPI's* encoding. *pandaPIgrounder* receives the parsed pandaPI encoding of the problem and executes a grounding process. After the grounding process the problem ready for solving by *pandaPIengine*.

PANDA is a very powerful system with capabilities to handle a lot of functionality. When considering the aims of this project it is clear that the studying the usage of heuristics in PANDA could provide transferable knowledge during implementation.

Chapter 4

Design & Requirements

4.1 Requirements

4.1.1 Functional Requirements

1. **Support HDDL and JSHOP input languages**

   Operating with both HDDL and JSHOP is the most essential requirement of the project.

2. **Handle Total Ordered problems**

   Support for total-ordered problems is essential for any HTN planner. The solution should efficiently process totally-ordered subtasks.

3. **Handle Partial Ordered problems**

   Accepting partial-ordered problems is not an essential feature of HTN planners but it gives another layer of usability to the system. The developed solution should be careful not to become inefficient when dealing with partial-orderings.

4. **Plan output**

   The system should detail the developed plan's steps and also the final state of the environment.

5. **Testing Framework**

   The system should come with its own unit tests which check each component thoroughly. The autonomous tests should use some example problems and compare output against expectations.

6. **Domain and problem input**

   The system should validate given user input and only accept recognised files. The provided problem file should match the domain file already parsed.

7. **Interchangeable Search Components**

   One of the most essential aims of this project is to allow future users to extend the functionality of the system and test their own ideas. A key part in this is to allow for search related components to be defined by future users and ensure the components can be used by the system seamlessly.

4.1.2 Non-Functional Requirements

1. **The system should be well documented and consist of readable code**

   The code base will be developed with future users in mind. People should be free to experiment with their own ideas and projects using the developed code base. Extensive and clear documentation is vital to this, so users can locate the sections of code they want to visit or adapt.

2. **Future updates to HDDL and JSHOP should be easily implemented**

   The parsers should leave consideration for future updates to the languages and leave opportunities to implement any further features where possible.

3. **Clear output**

   Clear output enables easier comparison of plans with ones generated by other planners. Debugging and solution verification will both be aided by a good clear output structure.

4. **Effective Error Handling and Reporting**

   Dealing with user input syntax is always going to require some consideration being given to errors. During parsing there is a sea of inputs that could be invalid. Clearly reporting these errors is important to ensure that users can find and resolve errors in their problems or components.

4.2 Design

In this section we dive into the design of the developed system and the justifications for its composition. We cover the responsibilities and objectives of components while also discussing strategies adopted to implement the functionality of HTN problems.

4.2.1 System Overview

The requirement of a modular system means that the hexagonal software design architecture is perfect for this project. Figure 4.1 shows a high level design of the system architecture. This architecture puts emphasis on having core components which remain the same, and outer components that can be modified. This makes it easier to add more parsers for different languages, Heuristics, Solvers, Parameter Selectors, and output methods. Each of the components within the system design diagram have their own internal architecture.

The hexagonal architecture is also effective at isolating the three main stages of execution; parsing, solving, and output. These stages are especially evident when viewing Figure 4.1 sequentially from left to right. In each stage of execution, methods and data from previous stages should not be meddled with. Instead only data which has been explicitly passed between stages should be interacted with. This design choice helps eliminate bugs since methods will only be initiated when it is safe to do so.

4.2.2 Runner

As apparent in Figure 4.1 the *Runner* class is the instigator of the system, it receives the input domain and problem files and initiates a parser. The *Runner* chooses which parser to use based on the suffix of the files given as input. Any future support for other input languages would

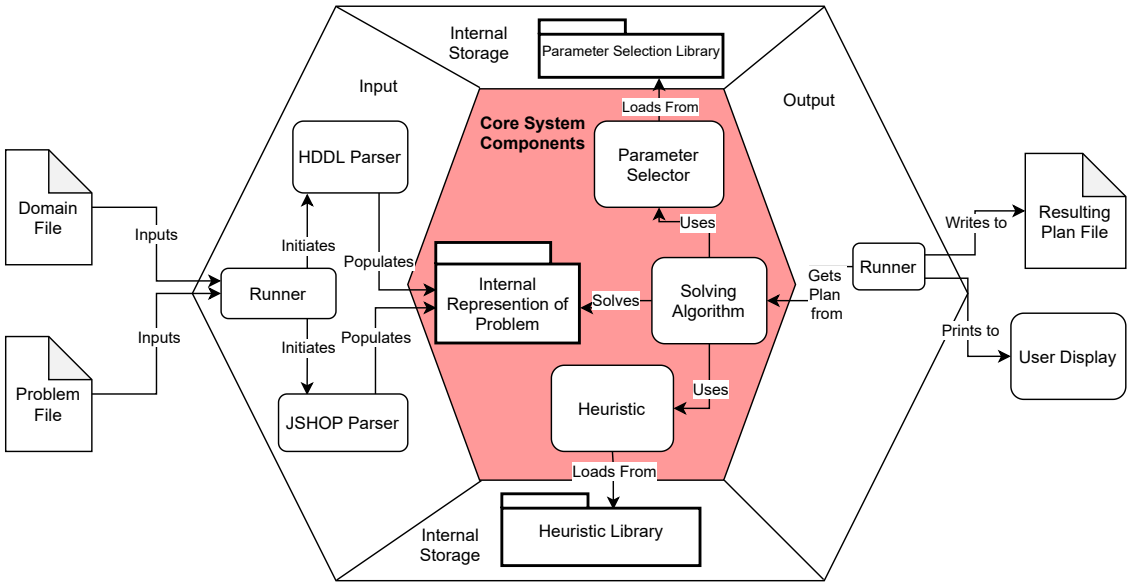


Figure 4.1: System Architecture Design

need to update this class to map the suffix of the new files to the new parser. The role of the *Runner* class goes further beyond only choosing a parser, Heuristics, Solvers, Search Queues, and Parameter selection classes can be selected too. There are two ways in which the *Runner* class can be interacted with; the command line interface or as an object in a third party script.

4.2.2.1 Module Selection

Heuristics (Section 4.2.7), Solvers (Section 4.2.6), Search Queues (Section 4.2.5.3), and Parameter Selectors (Section 4.2.8) can be selected using the *Runner* class. There are two methods of selecting classes, the first is the command line and the second is via a method[1].

The default solving algorithm is the *PartialOrderSolver* mentioned in Section 4.2.6.2. The *Total Cost* search queue, *Pruning* heuristic method, and *RequirementSelection* parameter selection classes defined in Sections 4.2.5.3, 4.2.7.1, and 4.2.8 are also used by default.

4.2.2.2 Plan Output

As standard the plan found by the planner is printed to the screen if the system is being run from the command line. As well as outputting to the screen the plan can be written to a file. Both of these output options are also available when using the Runner class as part of a third party script.

4.2.3 Parsers

Parsers are the components which make sense of the input files given to the system. Information extracted from the files is loaded into dedicated objects. These objects make up the *Domain* and *Problem* classes -covered in Sections 4.2.4.1 and 4.2.4.2 respectively- objects from the *Internal Representation* package shown in figure 4.1.

From figure 4.2, it is clear that the parser class is comprised of solely private variables and methods - with the exception of the two main parse methods. The parser class is designed to be very private, since after the parsing stage it should not be interacted with again. No other objects should be utilising its methods or reading its data.

[1] https://github.com/C-Milne/4th-Year-Dissertation/blob/main/README.md

During parsing, the objective of a parser is to develop objects to populate the *Domain* and *Problem* objects ready for solving. Figure 4.2 displays the relationship between parser and the domain and problem objects.

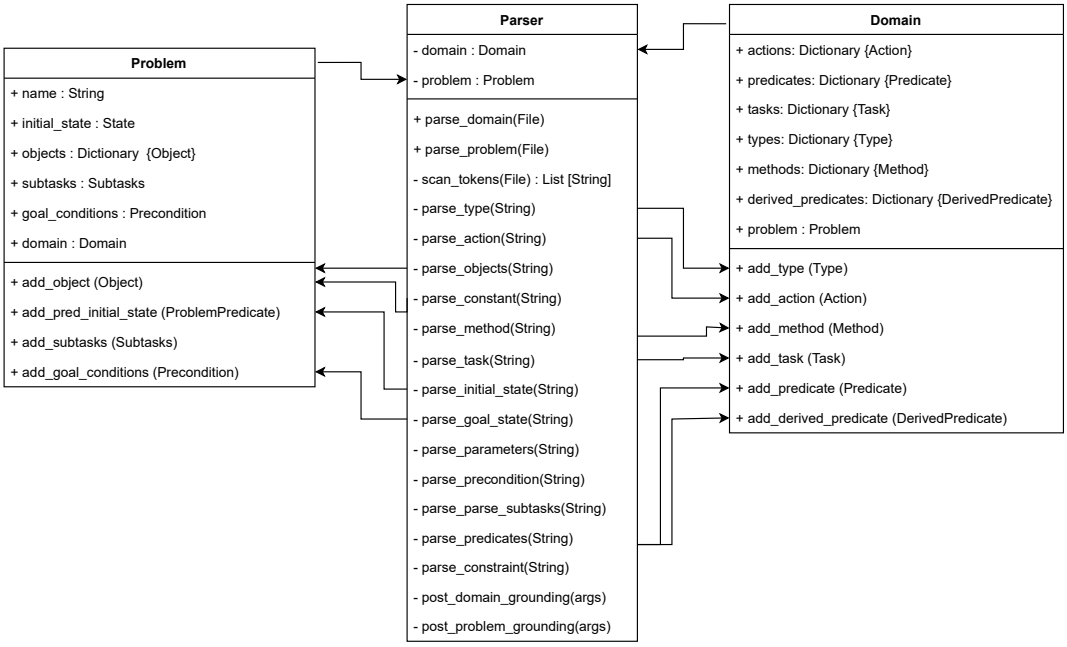


Figure 4.2: Parsing Design

Although the main relationships of a parser are between the *Domain* and *Problem* classes, objects of other classes are required to populate the domain and problem. So in practice a parser has access to the majority of the classes in the *Internal Representation* package, the classes in this package are explored in Section 4.2.4. Figure 4.3 illustrates the modules populated and returned by methods in the parser class.

Since this project aims to allow for multiple languages to be used, multiple parsers will be required. A central challenge of this requirement is ensuring that the parsers produce the same internal representations. To do this the implementation of parsers take advantage of class inheritance. The *Parser* class defines all the main methods required by a parser by using the *@abstractmethod* decorator. This decorator forces any sub-classes to implement the method decorated. For the *Parser* class only two methods are absolutely essential, methods for parsing a domain and parsing a problem. Listing 4.1 shows an example of the error raising design. Since the *parse_domain* method is public and instigated from the *Runner* module it must be implemented in every *Parser* sub-class with the *file_path* parameter. This design strategy ensures all parsers cover the essential features of HTN planning and are compatible with the *Runner* component of the system.

Other methods such as those for creating Actions, Methods, Tasks etc are also defined but raise a *Not Implemented Error* unless a subclass defines them. This design choice creates a template for future sub-class implementations to follow. Although this does not force the use of any methods it acts as a gentle guide. Currently there are two sub-classes of the *Parser* class developed, the *HDDLParser* and *JSHOPParser* classes.

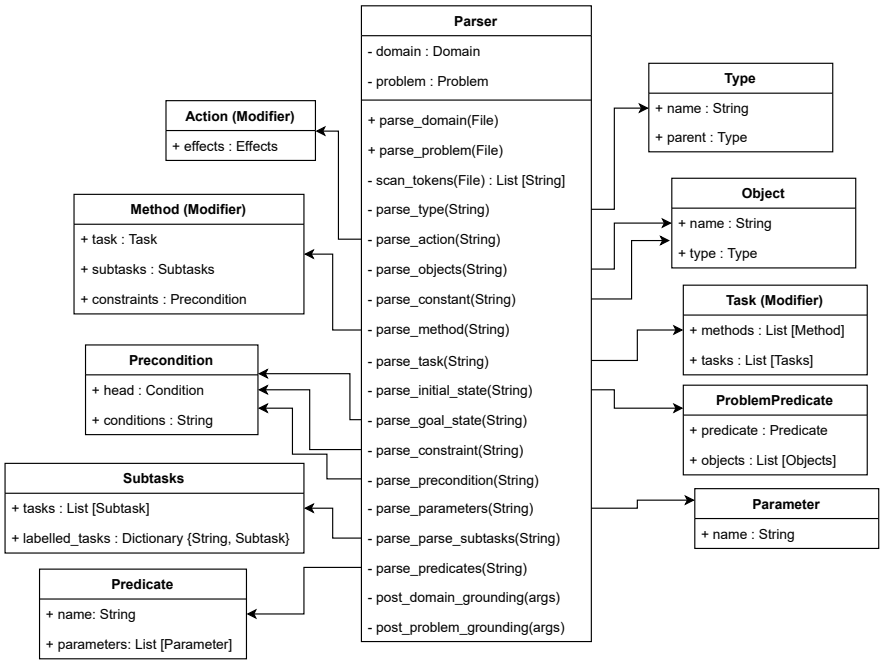


Figure 4.3: Modules Populated by Parser

```
@abstractmethod
def parse_domain(self, file_path: str):
    raise NotImplementedError
```

Listing 4.1: Example of Abstract Method Inheritance Design

4.2.4 Internal Representation

The *Internal Representation* package is comprised of classes which represent HTN problems. The *Domain* and *Problem* classes are the two main components for representing the given problem. These components are built-up of other objects within the package. Classes within the package are well connected to ensure the required information can be found quickly at run time. A high level of inter-class connectivity allows for run time operations such as finding which Methods belonging to a specific Task to be concluded quickly.

4.2.4.1 Domain

The *Domain* class shown in Figure 4.2 is used to store and organise all components parsed from the domain file of a problem. After parsing the domain object is used by solving algorithms detailed in Section 4.2.6 during search.

4.2.4.2 Problem

The *Problem* class shown in Figure 4.2 is used to store all objects parsed from the problem file of a problem. Like the previously mentioned domain, the problem object is also used by solvers during search to find Objects and evaluate the any goal conditions.

4.2.4.3 Modifiers

Throughout the software solution we use the concept of Modifiers, we define these as all the operations available during search. For HDDL the modifiers we have are Tasks, Methods, and Actions. We use the concept of Modifiers since these operations have some features in common, namely

parameters. To ensure the common features are preserved and standardised within the planner inheritance is adopted. Figure 4.4 presents a structure for the definition of modifier classes. Only *Method* and *Action* objects both have preconditions, hence the explicit definition of the *evaluate_preconditions* method in the *Task* class which always returns *True*.

JSHOP modifiers can be represented in the same classes as HDDL modifiers. The adaptions to convert JSHOP problems into HDDL problems discussed in Section 2.2.2 allow for the same modifier classes to be used in the representation of JSHOP problems.

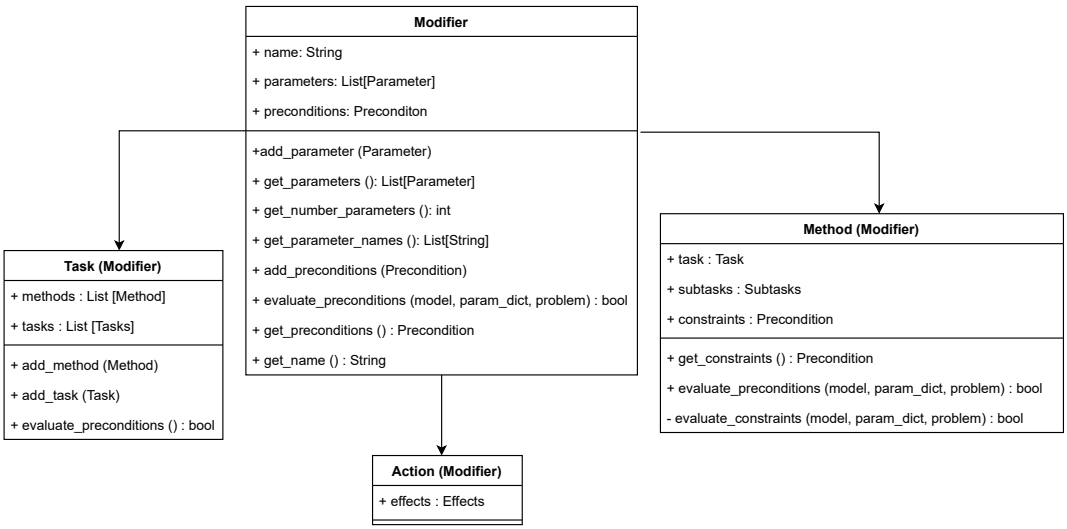


Figure 4.4: Design of Modifiers

4.2.4.4 Predicate

The *Predicate* class represents predicates as defined in Definition 11. The class is a straight forward containing only the defined name and parameters. The *ProblemPredicate* class represents predicates at run time, switching parameter definitions with objects that have been passed to fulfill parameters. Figure 4.5 illustrates the relationship between the *Predicate* and *ProblemPredicate* classes.

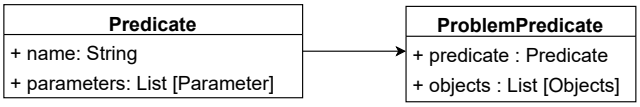


Figure 4.5: Design of Predicates

4.2.4.5 Parameter

HDDL problems only have one simple type of parameters, which can be represented using the *RegParameter* class shown in Figure 4.6. The intention of objects created using this class is to store the requirements for a parameter. As well as the *RegParameter* class, JSHOP also has functionality for lists as parameters which can be contained using the *ListParameter* class.

4.2.4.6 Type

Figure 4.7 shows the design of the *Type* class. The *parent* property references the parent type defined in Definition 9.

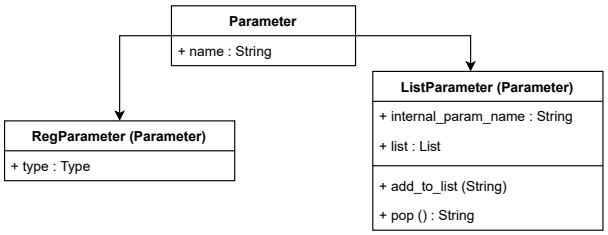


Figure 4.6: Design of Parameters

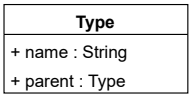


Figure 4.7: Design of Type Class

4.2.4.7 Preconditions

The core piece of intelligence in the *Precondition* class is to decide if a *Model* is in a valid state for an Action or Method to be carried out. Preconditions are comprised of several conditions, storing these conditions in a structure similar to that of a linked list provides an effective method of evaluation. Section 4.2.4.8 details which conditions are available to use as part of preconditions.

Figure 4.8 features the class for constructing preconditions. The *head* property stores the first condition in the precondition. For preconditions with more than one condition the *head* needs to be an *OperatorCondition* with the operator *'and'*.

Evaluating preconditions is done using the public *evaluate* method which initiates the *evaluate* method of the *head* condition. This begins a chain of *evaluate* method calls over all conditions. Figure 4.9 visualises how the method operates using the example from Listings 5.1 and 5.2. Each rectangle with curved edges represents a condition being evaluated. The lines show which child conditions are evaluated during the evaluation of a condition. The text next to the lines state what was returned from the block below. The boxes with square corners are the Boolean results of non-operator conditions.

In the example shown in Figure 4.9, evaluation begins with evaluation of the condition *['and', ['have', '?y'], ['not', ['have', '?x']]]* which instigates the evaluation of its two children *['have', '?y']* and *['not', ['have', '?x']]*. This chain of evaluation continues until all conditions have been evaluated.

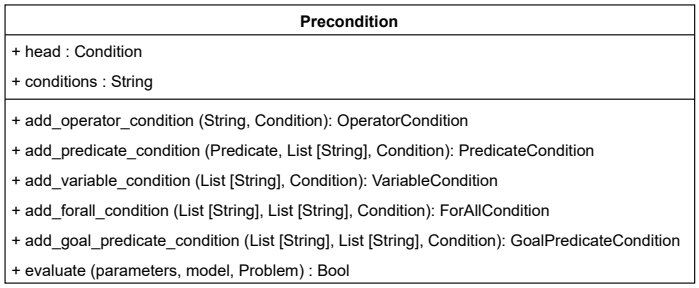


Figure 4.8: Design of Preconditions Class

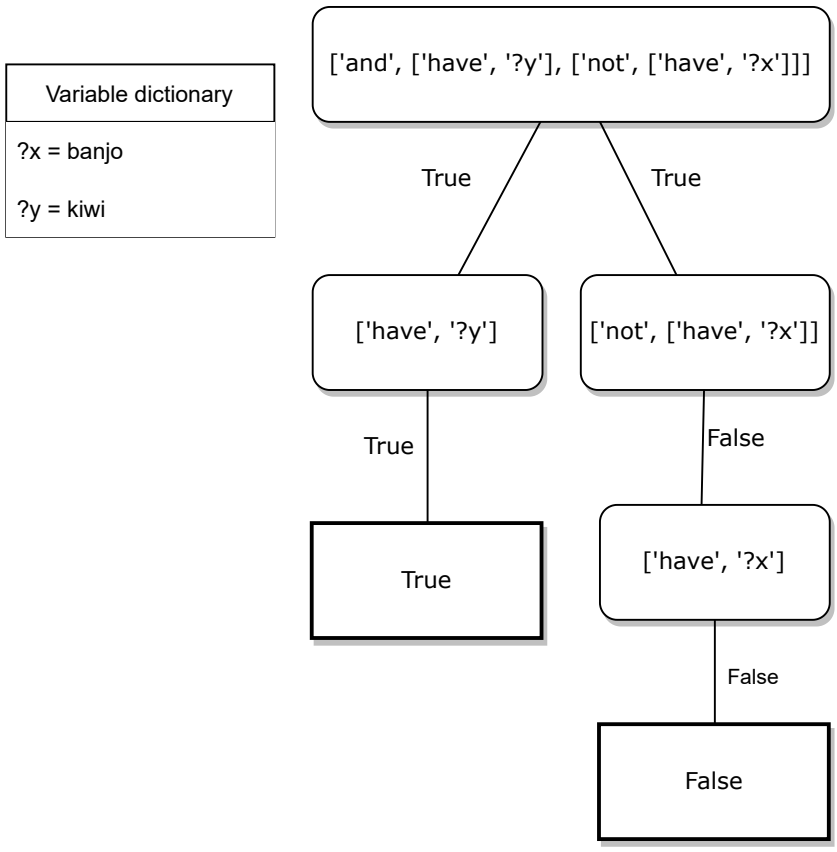


Figure 4.9: Precondition Evaluate Method - Example from Listings 5.1 and 5.2

4.2.4.8 Conditions

Multiple types of conditions exist, each with differing features and evaluation criteria. Figure 4.10 puts forth the condition types used to construct wider preconditions. Similar to the *Parser* class introduced in Section 4.2.3, the *Condition* class has utilises the @*abstractmethod* decorator. In this scenario the *evaluate* method is decorated to ensure its presence in any developed sub-classes. The *Condition* class is inherited by all condition types and contains the *parent* property which refers to the *OperatorCondition* which evaluates the condition at run time.

PredicateCondition's are used to simply check the presence of a particular predicate with given objects in the current state of the search model. The *ForAllCondition* class is used to represent *for-all* conditions, defined in Definitions 21 and 24.

OperatorCondition objects can have an *operator* value of *'and'*, *'or'*, *'not'*, or *'='*. During evaluation these operators act differently. The *'and'* operator ensures that all child conditions evaluate *True* whereas, the *'or'* operator requires at least one child condition to evaluate *True*. The *'not'* operator negates the result of its evaluated child finally, the '=' operator determines if the child objects are equal. If the *operator* attribute is equal to *'='* then the *children* list contains *VariableConditions*.

The purpose of *VariableCondition*'s is to store and retrieve a particular variable. This is used in conjunction with an *OperatorCondition* with an *operator* of *'='* to check if parameters are equal. This is used as part of HDDL constraints.

Lastly, *GoalPredicateCondition*'s is used as part of JSHOP's functionality to check the contents of the problems goal state.

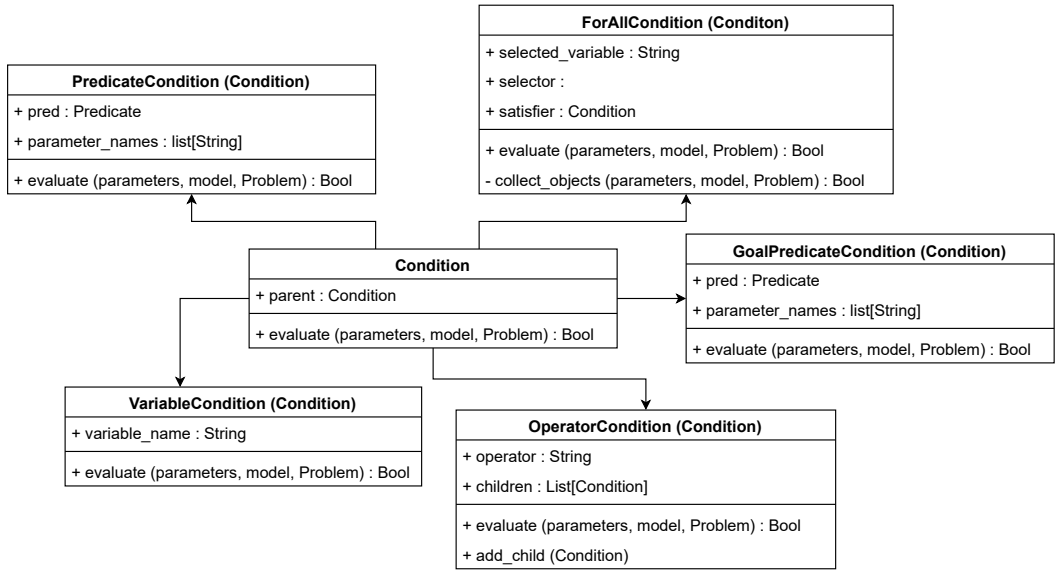


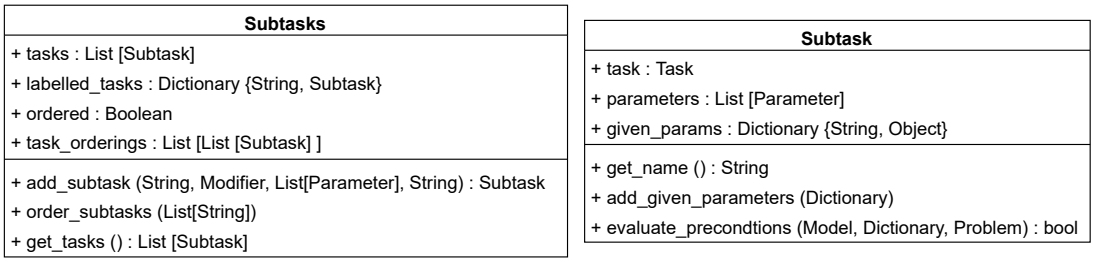
Figure 4.10: Design of Conditions

4.2.4.9 Subtasks

The *Subtasks* class defined in Figure 4.11a is used to represent the subtasks belonging to a Method. A *Subtasks* object is contains multiple *Subtask* objects shown in Figure 4.11b. *Subtask* objects construe each subtask to be added to the task network. The *given_params* property of the *Subtask* class stores what objects have been passed as parameters to the subtask during search. Since subtasks can be either total-ordered or partial-ordered, the attribute *ordered* is used to track if the subtasks represented are total-ordered or not. The *task_orderings* attribute is used to store all possible orderings of partial-ordered subtasks.

4.2.4.10 Object

Objects are a key component in HTN planning, since objects are passed between modifiers during search. The *Object* class shown in Figure 4.12, is used to represent objects defined in Definition 14.



(a) Subtasks Class Diagram

(b) Subtask Class Diagram

Figure 4.11: Subtasks and Subtask Class Diagrams

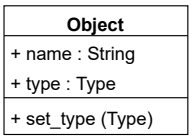


Figure 4.12: Object Class Diagram

4.2.4.11 State

The state of the search environment is an essential component of planning. The *State* class detailed in Figure 4.13 is responsible for representing states during search. This class is comprised of two attributes, the dictionary *index* and the list *elements*. The purpose of the *elements* attribute is to simply store all the predicates represented in the state. The second attribute, *index* is used to map predicate names to the indexes which they appear in the *elements* list. This mapping is used when finding predicates at run time to prevent iterating over the entire list of elements. On top of the attributes the *State* class contains a spread of both public and private methods. The methods provide functionality for adding, removing, and searching for predicates.

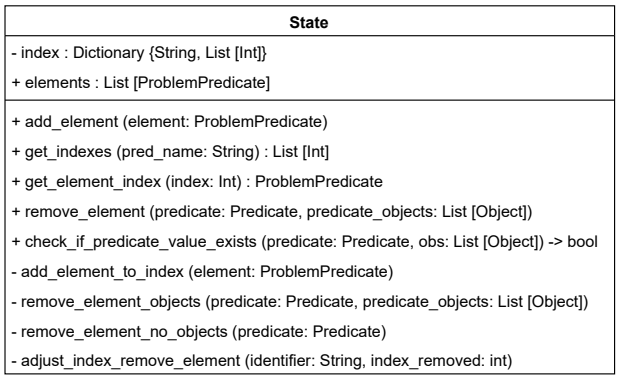


Figure 4.13: State Class Diagram

4.2.5 Solver

The *Solver* package is comprised of classes concerned with the solving of problems. In this section we introduce the idea of *Models* and continue on to discuss search strategies and recording steps taken during search.

4.2.5.1 Model

One of the most fundamental aspects of HTN planning is the idea of search state, which store all the information regarding the environment during search. In this project the class *Model* undertakes the role of representing and managing a search state.

Definition 28 represents search models as a tuple. Since during search there are multiple options regarding which Methods to execute and with which parameters, unique search states are used to keep track of each option. The three features of a model are the state (S), the task network (T_N), and the list of decompositions used (D).

Definition 28. HTN Search Model

A search state $ST = (S, T_N D)$ where:

- S is the state of the environment
- T_N is the task network of the model
- D is a list of decompositions used thus far during search

At the beginning of search in a total-ordered problem usually only one model exists, during search as multiple options for decompositions appear more models are created to consider all decomposition options. Different decompositions add differing modifiers to the task network, this brings in the requirement for models to store and manage task networks.

Figure 4.14 displays the *Model* class along with its relationships to other classes. When considering the components of Definition 28, a list is sufficient for representing a task network since a list maintains order. Each model has a state which is stored in the form of a *State* object defined in Section 4.2.4.11. The final feature of a model is tracking the decompositions used, to do this the *ActionTracker* class defined in Section 4.2.5.2 is utilised. After each decomposition an *ActionTracker* object is created in stored. The *Model* class has two list attributes for storing *ActionTracker* objects. The first is *actions_taken* which only stores *ActionTracker* objects relating to *Actions*. The other is the *operations_taken* which stores all *ActionTracker* objects. We use these two lists since the IPC requires both the Actions executed and the entire list of decompositions used to be output [2].

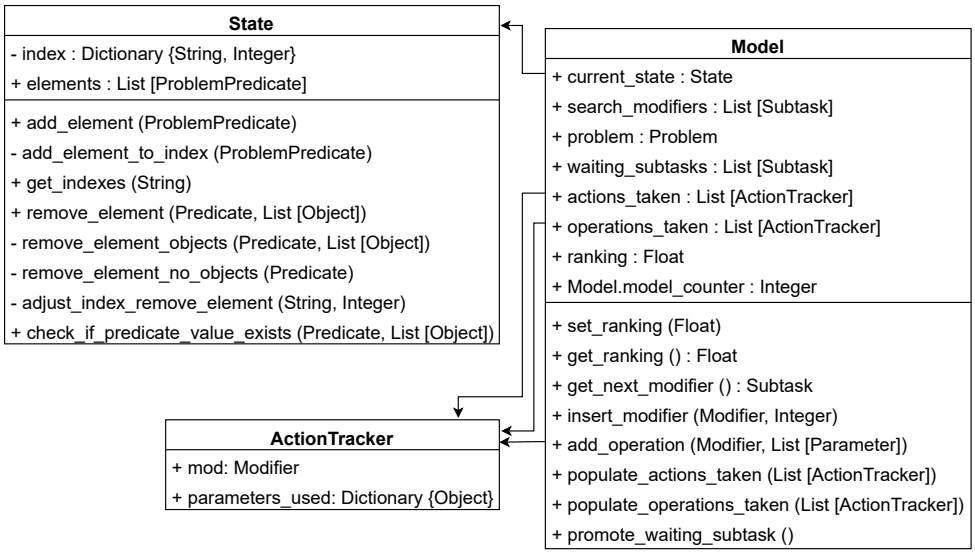


Figure 4.14: Search Components

4.2.5.2 Action Tracker

As mentioned in Section 4.2.5.1 the purpose of the *ActionTracker* class is to represent a decomposition taken during search. Figure 4.14 displays the simple nature of the class as it only contains two attributes since the main purpose of the class is strictly for storage. Apart from storage the

[2] https://gki.informatik.uni-freiburg.de/ipc2020/format.pdf

only other functional contribution of the class is for output which is developed using Python's *__str__* method.

4.2.5.3 Search Queues

In Section 4.2.5.1 we discussed that during search multiple models are created. To contain and order these models we use a Search Queue. The *SearchQueue* class is the default Search Queue and orders models based on the total estimated cost to goal, in this project we refer to this strategy as the *Total Cost* search queue. The total cost is calculated as the cost thus far plus the estimated cost to goal which is provided by a heuristic. Two other Search Queues inherit the *SearchQueue* class and order models using different techniques. The *Greedy Best First Search (GBFS)* queue orders models solely on the heuristic estimate whereas, the *Greedy Cost Search Queue* orders models by dividing the cost thus far by five and adding the estimated cost to goal. Figure 4.15 outlines the full composition of the *SearchQueue* class and its relationship with heuristics.

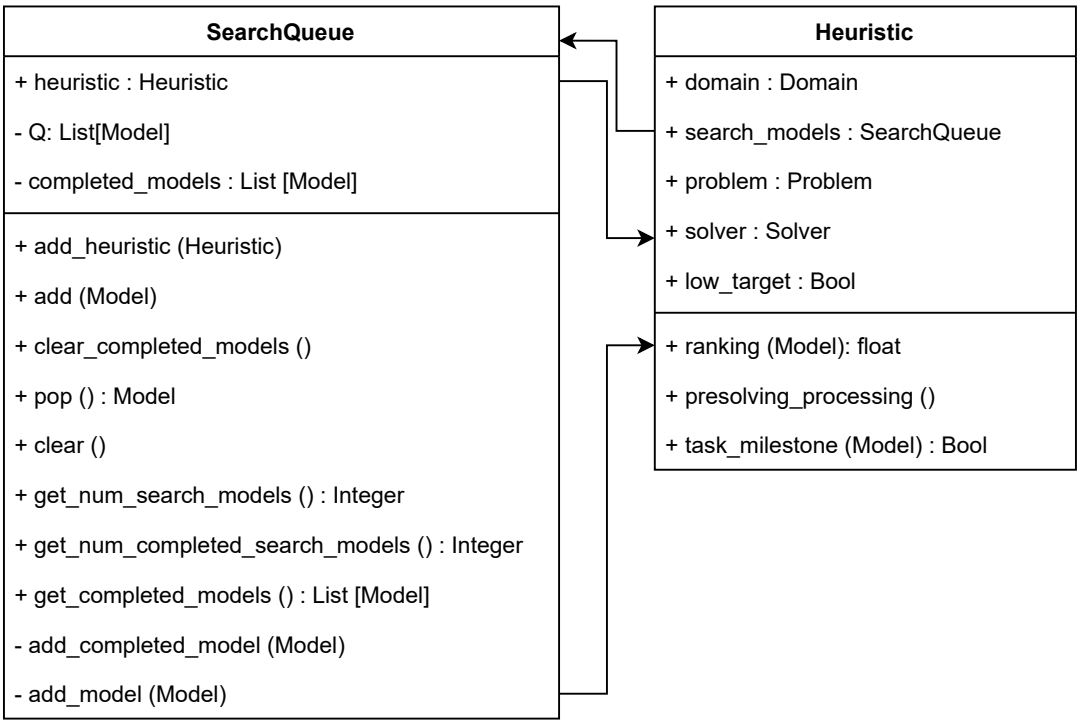


Figure 4.15: Search Queue and Heuristic Relationship

4.2.6 Solving Algorithms

The solving algorithms work closely with the aforementioned *Internal Representation* package, particularly the *Domain* and *Problem* objects outlined in Sections 4.2.4.1 and 4.2.4.2. The role of a solving algorithm is to apply Methods and Actions to models in order to find a valid solution.

Similar to the inheritance design of Parsers described in Section 4.2.3 solving algorithms must inherit the *Solver* class. Figure 4.16 exhibits the make up of the *Solver* class, the *@abstractmethod* decorator introduced in Section 4.2.3 is used to command the implementation of the *_expand_task*,*_expand_method*, and *_expand_action* methods in sub-classes. Two sub-classes of the *Solver* class are implemented in the system, the *TotalOrderSolver* and *PartialOrderSolver* classes are introduced in Sections 4.2.6.1 and 4.2.6.1 respectively.

The *task_expansion_given_param_check* attribute refers to a optimisation where the parameters explicitly passed to a Method are validated before any objects are chosen to satisfy supplementary parameters. This prevents validating the same set of conditions with different objects if they are all bound to fail. This optimiser can be deactivated if required by solvers or parameter selectors.

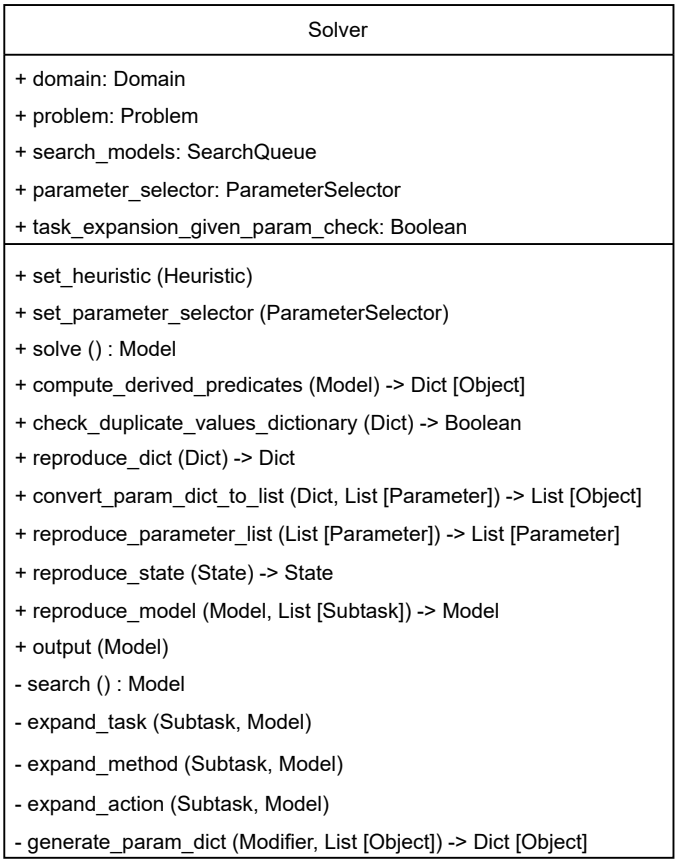


Figure 4.16: Solver Class Diagram

The total-order and partial-order forward decomposition algorithms mentioned in Section 3.3 are recursive algorithms. We use an iterative algorithm instead since, the recursive algorithm limits search to a depth first approach where as our iterative algorithm provides much more opportunity for different search strategies. For example a both a breadth first search and depth first search can be produced based on the *SearchQueue* used. In Section 3.2 we explored the search strategies possible from the expansion of a *Breadth First Search* algorithm. The Iterative strategy of search allows for all these other strategies to be used.

The *Solver* class defines the iterative search strategy used by both the *TotalOrderSolver* and *PartialOrderSolver*. Listing 4.2 details the iterative strategy, the functionality for *expand_task*, *expand_method*, and *expand_action* is implemented by sub-classes.

4.2.6.1 Total-Order Solver

The implementation of the aforementioned *expand_task*, *expand_method*, and *expand_action* methods is rather simple for total-order problems. Since the orderings are final, sequences of subtasks can simply be iterated over and added to the task network during the expansion of Methods. Listings 4.3, 4.4, and 4.5 step through the procedure for the *expand_task*, *expand_method*,

```
solve():
    initial_model = initial search model
    Initialise SearchQueue with initial_model

    While True:
        model = search_models.pop
        if model is ∅:
            RETURN No Plan Found

        modifier = model.search_modifiers.pop
        if modifier is a Task:
            expand_task(modifier, model)
        else if modifier is a Method:
            expand_method(modifier, model)
        else if modifier is an Action:
            expand_action(modifier, model)

        if model.search_modifiers == ∅
        and model satisfies goal conditions:
            RETURN model
```

Listing 4.2: Solving Algorithm

```
expand_task(modifier, model):
    For each method belonging to modifier:
        P = All parameters from parameter selector
        For p in P:
            M = new model with p as parameters for method
            search_models <− M
```

Listing 4.3: expand_task method

and *expand_action* methods respectively.

4.2.6.2 Partial-Order Solver

The *Partial-Order Solver* builds upon the functionality covered in Section 4.2.6.1 by adding consideration for different subtask orderings when decomposing Methods. When dealing with partially-ordered subtasks we adapt the algorithm seen in Listing 4.4 to include a loop over all the possible orderings. For this to be applicable all of the possible orderings of subtasks are calculated once during the grounding stage to save on computing power during search.

To calculate all of the possible task orderings we use *Kahns algorithm* [25] with some adaptions. Listing 4.6 exhibits the procedure for computing all the possible orderings. This algorithm is recursive to deal with the situation where more than once subtask can be added to an ordering

```
expand_method(modifier, model):
    i = 0
    for mod in modifier.tasks:
        model.task_network.insert(mod, i)
        i ++
    search_models <− model
```

Listing 4.4: expand_method method

```
expand_action(modifier, model):
    for e in modifier.effects:
        if e not ForAllEffect:
            model.current_state.change_elements(e)
        else:
            obs = objects that satisfy for−all selector
            for o in obs:
                model.current_state.change_elements(e, o)
    search_models <− model
```

Listing 4.5: expand_action method

```
kahns_algo(S, orderings)
    if S.LENGTH == 1
        add the only element in S to orderings
        S <− empty list
        add all subtasks with no outstanding predecessors to S
        return kahns_algo(S, orderings)
    else if S.LENGTH == 0
        return orderings
    else
        for each subtask in S
            add khans_algo(subtask, COPY(orderings)) to orderings
        return orderings
```

Listing 4.6: Calculating All Possible Orderings

at the same time.

4.2.7 Heuristics

Guiding the search of the algorithm is a key component when considering the speed and efficiency of the planner. Heuristics can be applied as a 'plug-in' components that are interchangeable. Figure 4.15 demonstrates how heuristic functions are applied to the ordering of Search Queues. The *Heuristic* class detailed has three functions, the *ranking* function returns a score for a given model. *presolving_processing* is a space for any required processing to be done prior to solving. The *task_milestone* method is for any processing after a subtask set in the problem file is fully decomposed. This method can be used to pruning models, returning a result of *False* means the model should be pruned.

4.2.7.1 Pruning

The basic ability of pruning which is available to heuristics in the planner is pruning via the aforementioned *task_milestone* method. The *Pruning* class inherits the *Heuristic* class and is available to be inherited by heuristics in place of the *Heuristic* class. The *Pruning* class prunes models after a Task from the initial task network is fully decomposed, by considering the model state. If the state has previously been seen after the decomposition of the same task, then the model is pruned[3].

To prune models in partial-ordered problems, a different approach is required. The *PartialOrderPruning* class works in a similar manner to the aforementioned *Pruning* class. The

[3]https://github.com/C-Milne/4th-Year-Dissertation/blob/main/Solver/Heuristics/pruning.py

difference between the two is the logic of deciding when to prune a model. Since partial-order problems can have the same tasks in differing orders, the *PartialOrderPruning* class concatenates the names of the remaining tasks. The resulting string from this concatenation is used to check a dictionary for seen states[4].

4.2.7.2 Breadth First

Breadth first is a simple search strategy as seen in Section 3.2.1. Since breadth first search works by expanding the option with the least amount of steps, where we define a step as any decomposition. This ensures that all models from a common parent are decomposed before moving on to any further decomposed models. In this system breadth first search is achieved by using the *Total Cost* search queue discussed in Section 4.2.5.3 and a heuristic function which always returns zero. We can combine breadth first search with both of the pruning methods outlined in Section 4.2.7.1.

4.2.7.3 Tree Distance

Since a HTN planning problem is considered complete when the task network is fully decomposed, we propose *Tree Distance* as a heuristic which estimates the cost to fully decompose a task network. Figure 4.17 shows an example of the estimated costs assigned to a Tasks, Methods, and Actions in a problem. The cost of an Action is always 1. The cost of a Method is the combined cost of all its subtasks plus 1. The cost of a Task is the cheapest cost of all its Methods plus 1.

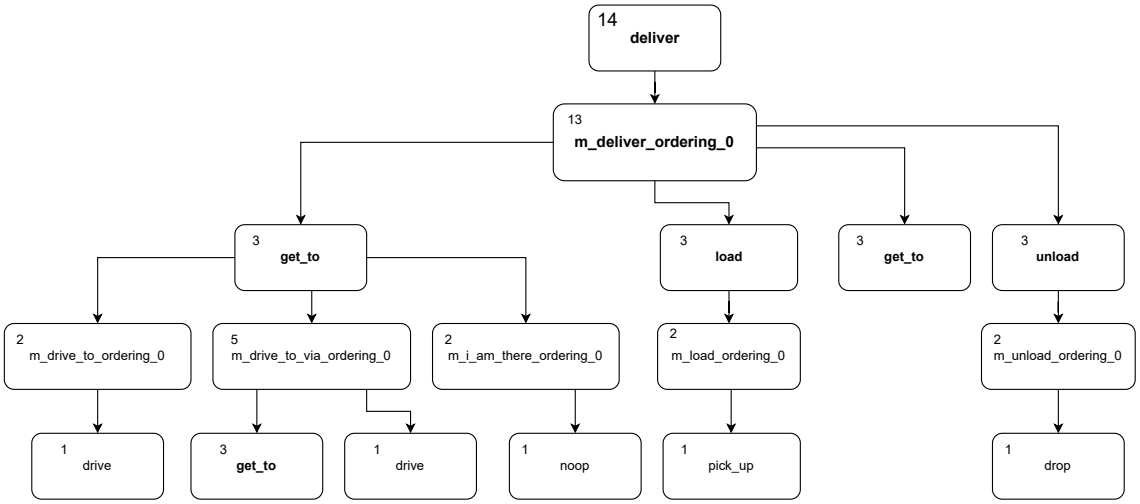


Figure 4.17: Tree Distance Example

4.2.7.4 Hamming Distance

Hamming Distance proposed by Richard Hamming [30] refers to the amount of times two strings differ. We adapt this principle to calculate the number of times a state differs from the goal conditions.

4.2.7.5 Delete Relaxed

The *Delete and Ordering Relaxed* heuristic explained in Section 3.4 rounds off the collection of heuristics. The process of transforming a HTN planning problem to a Classical Planning problem required the inclusion of some modified classes to operate the unique functionality utilised by the heuristic.

[4]https://github.com/C-Milne/4th-Year-Dissertation/blob/main/Solver/Heuristics/partial_order_pruning.py

4.2.8 Parameter Selection

As well as heuristics to guide search, selecting parameters is another optimiser. Only selecting parameters which are likely to be useful can save searching with parameters which will never yield a valid plan. With less *model*'s to search, a goal can be found quicker. Figure 4.18 details the *ParameterSelector* class, this class can is designed to be inherited to create new parameter selection techniques. The *presolving_processing* is similar to that found in the *Heuristic* class, it provides an opportunity for some processing to be done before solving commences. The main method is *get_potential_parameters* which returns a list of dictionaries where, each dictionary is a combination of objects to be used to satisfy the given modifier.

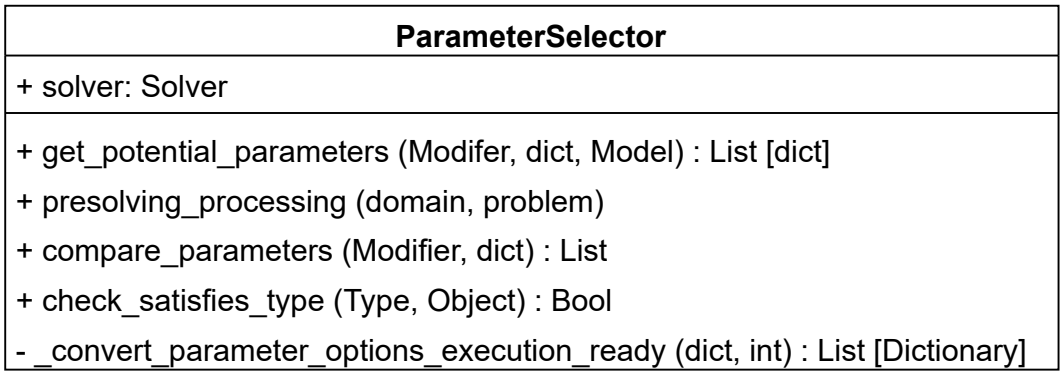


Figure 4.18: Parameter Selection Class

The *AllParameters*[5] parameter selection class is the base parameter selection technique. All combinations of objects which satisfy the parameter type are returned. The *RequirementSelection*[6] parameter selection class only selects parameters which satisfy the precondition constraints of a parameter. This strategy aims to reduce the amount of parameter options returned to the solving algorithm for consideration.

[5]https://github.com/C-Milne/4th-Year-Dissertation/blob/main/Solver/Parameter_Selection/All_Parameters.py
[6]https://github.com/C-Milne/4th-Year-Dissertation/blob/main/Solver/Parameter_Selection/Requirement_Selection.py

Chapter 5

Implementation

5.1 Methodology

The aim of this project is to develop a HTN planning system that encourages users to experiment with their own optimisation ideas. The capability for users to easily define their own functionality for parameter selection, heuristics, search queues, and search strategy are essential. As mentioned in Section 2.3 HTN planning is part of the International Planning Competition, which demands the use of experimental optimisations and strategies. To understand the requirements of a HTN planning system and develop a planner the following steps were employed:

1. Reading of background literature and development of knowledge relevant to HTN planning.
2. Investigation into current planners.
3. Identifying areas for this project to tackle.
4. Creation of a project plan.
5. Consideration of software tools and concepts necessary to develop the system.
6. Compiling both functional and non-functional requirements of the system.
7. Devising a future ready software architecture design.
8. Exploring concepts and different code structures.
9. Implementing the system with a test-driven approach.
10. Testing and evaluating the developed system.

5.2 Technologies Chosen

During the selection of technology there were some instances where multiple options were considered. This section outlines the technologies chosen and gives the reasoning behind the selection.

5.2.1 Python

Python[1] is the programming language of choice for the implementation of the system. Python is a good fit for this project since accessibility is a key concern; Python was ranked as the most popular programming language in the 2021 IEEE Spectrum Programming Language ranking [8]. Python works well across all platforms -Windows, MacOS, Linux- without complications. All this Makes Python a great choice when considering how many people the project will be accessible to. Python

[1] https://www.python.org/

is under represented in the HTN planner space given its immense popularity, as popular languages for HTN planners are C++[2] [20, 40, 13] or Java[3] [36].

Python comes with a plethora of libraries as standard such as unittest[4] and timeit[5]. Additional libraries are simple to install and manage with the pip package manager[6]. The capability to painlessly work with libraries is a great feature of Python.

5.2.2 Development Environment

For the development of this project the main IDE used will be JetBrains PyCharm[7] since it is an extremely powerful tool. PyCharm provides a good debugger as well as detailed code inspections. In the case that a secondary IDE is required, case Visual Studio Code[8] will be used.

5.2.3 Version Control

Git[9] will be used for version control, with the project being stored on GitHub. Git allows for branches to be used and switched between seamlessly. This is a key feature when considering the components of this project. The development of separate components can be kept apart from the master branch until all tests are completed.

5.2.4 Test Framework

Automated tests will be written using Python's built-in unittest[4] library. Unittest allows for *setUp* and *tearDown* methods to execute automatically when testing begins and ends. Also available is a range of powerful tools for asserting actions and values. The *assertRaises* method is perfect for checking exception types and messages.

5.2.5 Example Input Problems

Testing the system requires example domains and problems. The HDDL domains and problems used for the 2020 IPC makes for a good set of non-trivial problems that are confirmed to have no errors[10]. Complementary to the collection of domains and problems is a selection of test instances for individual features [11].

5.3 Implementation

When implementing this project a stage by stage approach was adopted. With a stage being completed and tested before considering future functionality. A key decision was to implement the most basic HTN planner as possible to begin with, then work on adding new features. Prior to commencing with the serious implementation some experimenting with concepts and ideas was done before a code structure was decided. This section details the stages of development after experimentation and the functionality brought forward in them. Backups of the code were taken after the completion of each section to minimise the risk of losing progress.

[2] https://isocpp.org
[3] https://openjdk.java.net
[4] https://docs.python.org/3/library/unittest.html
[5] https://docs.python.org/3/library/timeit.html
[6] https://pip.pypa.io/en/stable/
[7] https://www.jetbrains.com/pycharm/
[8] https://code.visualstudio.com/
[9] https://git-scm.com/
[10] https://github.com/panda-planner-dev/ipc2020-domains
[11] https://ipc.hierarchical-task.net/benchmarks/input-language

5.3.1 Representing Basic Functionality

The first stage of development was to create a very simple HTN planner. The *Basic HDDL* problem outlined in Listings 5.1 and 5.2 was set as the first problem to solve. Solving this problem required the code framework of the system to be established, most importantly the HDDL parser. When creating the HDDL parser attention was given the PDDL parser[12] provided by Felipe Meneguzzi and the Ruby implementation within the HyperTensioN planner [28]. The initial version of the HDDL parser included support for Tasks, Actions, Methods, predicates, and preconditions. Tasks, Methods, and Actions only included the essential components at this time, future work would build upon the base developed here.

The developed parser included a few checks and error raising conditions. For example if the parser comes across a token it doesn't know what to do with, an error is raised. It is key that the problem file given belongs to the domain that was defined by the domain file, a check for that condition is also in place.

In the current state there is a very limited selection of problems that can be represented by the parser. Only totally-ordered subtasks can be accepted moreover, the example problem used only uses subtasks and not any goal condition. As such support for goal conditions is not yet implemented in this version.

To ensure the work completed in this stage was not undone a *unittest* suite was devised to assert the correctness and stability of functionality brought forward.

5.3.2 Basic Solving Functionality

The core solving functionality was created in line with the basic parsing functionality. Creating a solver capable of solving the *Basic* problem shown in Listings 5.1 and 5.2 required the core solving functionality to be developed. Both the *Model* and *SearchQueue* classes defined in Section 4.2.5 are introduced to tackle solving. The simple solving procedure tackled the problem in a breadth-first manner and handled removing and inserting items into the task-network.

5.3.3 Adding Additional Functionality

Now the fundamental aspects of the system are implemented, the next stage of functionality can be developed. Some of the incoming additions are for the benefit of progressing system capabilities meanwhile, others are to make the system more efficient and safe.

The IPC Tests set out in Section 5.2.5 were employed to guide the implementation of functionality. Of the 11 tests for total-order planners 7 were selected as the focus of this stage. The 7 tests selected were *test01_empty_method*, *test02_forall*, *test03_forall1*, *test04_no_abstracts*, *test05_constants_in_domain*,*test06_synonymes*, and *test07_arguments*. These tests checked for compatibility with edge case examples such as a Method with no subtasks as well as core HDDL functionality like for-all conditions.

On top of these features required to handle the first 7 IPC tests, some other functionality was added in anticipation for the future IPC tests. Improvements to the grounding of components was improved to limit processing at run time. For example Methods that decompose a Task were stored within the *Task* object. Type checking was one of the future improved in this stage, with objects able to satisfy any preceding parent types that they may have. Parameter selection was also tackled

[12] https://github.com/pucrs-automated-planning/heuristic-planning

```
(define (domain basic)
 (:requirements :hierarchy
  :negative-preconditions :method-preconditions)
 (:predicates (have ?a))
 (:task swap :parameters (?x ?y))

 (:action pickup
  :parameters (?a)
  :precondition (not (have ?a))
  :effect (have ?a)
 )

 (:action drop
  :parameters (?a)
  :precondition (have ?a)
  :effect (not (have ?a))
 )

 (:method have_first
  :parameters (?x ?y)
  :task (swap ?x ?y)
  :precondition (and
   (have ?x)
   (not (have ?y))
  )
  :ordered-subtasks (and
   (drop ?x)
   (pickup ?y)
  )
 )

 (:method have_second
  :parameters (?x ?y)
  :task (swap ?x ?y)
  :precondition (and
   (have ?y)
   (not (have ?x))
  )
 :ordered-subtasks (and
   (drop ?y)
   (pickup ?x)
  )
 )
)
```

Listing (5.1) basic.hddl (basic domain)

```
(define (problem pb1)
 (:domain basic)
 (:objects kiwi banjo)
 (:init (have kiwi))
 (:htn :subtasks (swap banjo kiwi))
)
```

Listing (5.2) pb1.hddl (problem 1 of basic domain)

at this stage, the initial parameter selection technique developed was the *RequirementSelection* selector.

5.3.4 Additions to Solving Procedure

The Remaining 4 IPC tests focus on solving entire problems, not just small unique scenarios. These tests demand capability with goal conditions. Parsing the goal state and evaluating it against fully decomposed search models determined if a model could be returned as the result. The final component required to satisfy all IPC tests was Method constraints. Adding functionality for the parsing and evaluation of constraints, marked the completion of HDDL functionality. The method of model pruning discussed in Section 4.2.7.1 was implemented and tested as the final optimiser of this stage of development.

5.3.5 JSHOP Support

To be compliant with JSHOP problems, some classes were restructured to be more accommodating. The *Parser* was developed as a parent class to all potential parsers. Conditions got a similar structure to allow for JSHOP's different for-all conditions. Effects were the final component to be modified with a parent class to allow for JSHOP's for-all effects.

It was possible to handle most of JSHOP's functionality with the existing components using a new JSHOP parser. However, some of JSHOP's functionality proved more difficult than anticipated. Features of JSHOP like parameters as lists and the *!!assert* and *!!remove* methods have no counterpart in HDDL. Although given enough time these features can be added to the system due to the structure of inheritable classes for component types - such as the *Parameter* class for the inclusion of list parameters. But these features were not feasible given the time constraints on this project. Since JSHOP has no standardised set of features like IPC provides for HDDL, it is difficult to judge what features to support and which to class as extensions. As a result the features of JSHOP supported by the current version of the system are the same as the ones support by HDDL. With the exception for the inclusion of for-all effects, which are a centerpiece of JSHOP problems.

Since JSHOP does not include functionality for types the set of parameters for Tasks, Methods, and Actions was huge when compared to HDDL. This has an enormous detrimental impact on search time. A possible solution to this would be a new parameter selection class that looks ahead to the requirements of Methods and Actions when decomposing Tasks. This has the potential to cut down the amount of options and close in on a result. Unfortunately the time pressures on this project made it unfeasible to spend more time on devising and trialing solutions to this issue. Nonetheless this requirement for a different parameter selection class for JSHOP backs-up the need for interchangeable parameter selectors that are a feature of the developed system.

5.3.6 System Improvements

A focus of this system is on the idea of interchangeable components allowing users to trial their own ideas. The four components identified for interchangeability are parameter selectors, solvers, search queues, and heuristics. Functionality to select these components was added in this stage of development.

The process of making each component interchangeable is the same. First, a parent class is defined which stores all the methods to be shared across all sub-class components. Secondly, the

@abstractmethod is used to decorate the methods which must be defined in sub-classes. Lastly, a parameter is created for the command line for receiving a file path and another for receiving a class name. The *Runner* classes uses these parameters to find and load the desired class.

5.3.7 Partial Order

Supporting partial-orderings is the last remaining requirement of the system. The approach to partial-order taken was to compute all possible valid orderings of subtasks using a variation of Khan's algorithm, as seen in Section 4.2.6.2. This saves computing all possible orderings repeatedly during search as it can be quite expensive. Creating a solver compatible with these orderings was a trivial task, a simple for-loop iterates over the orderings and adds them to a new search model.

Chapter 6

Testing & Evaluation

This section outlines the tests used to validate the system. We also discuss the practices followed for system evaluation as well as the results gathered.

6.1 Testing

When testing this system the aims of tests fall into two categories, ensuring correctness and range of functionality. To assert the range of functionality for HDDL the IPC feature test cases mentioned in Section 5.2.5 are used. These tests check compatibility for the majority of features present in the IPC standard of HDDL. Although these test cases do not exhaust each individual feature in HDDL they do cover key features. Further tests in form of Python *unittests* were devised to inspect other features of HDDL. For the sake of simplicity the IPC tests were ported into *unittests* so they could be autonomously ran at the same time as all other tests.

When testing features of JSHOP all tests needed to be created using *unittest*. This is due to JSHOP not having a standardised set of features like HDDL. To aid in the testing of the system a file to run all unittest test cases was created[1]. In total 124 unit tests can be run using this collective test instigator.

When testing to ensure the correctness of the system, configurations of solving algorithm, parameter selector, heuristic, and problems are run. The output is then checked to match the expected result.

Throughout the tests mentioned both white box and black box testing is used. White box tests step through the execution of a problem checking the stored values after each iteration of the search algorithm. Black box tests check the output of individual methods or in some instances results of problems.

6.2 Evaluation

To evaluate the impact of heuristics performance we consider both the time taken to reach a result and the amount of search models created in the pursuit of doing so, which is a machine-independent metric. The evaluation is split into sections so to consider total-order, partial-order, and JSHOP problems separately. The correlation between solve time and models created is also inspected. All the problems evaluated in this section were run using the *RequirementSelection* parameter selector mentioned in Section 4.2.8. Each of the problems evaluated was timed five times and the average of the five runs was calculated.

[1] https://github.com/C-Milne/4th-Year-Dissertation/blob/main/Tests/UnitTests/All_Tests.py

Problem Name	# Size
Rover/p01.hddl	483
Rover/p02.hddl	486
Rover/p03.hddl	501
Rover/p04.hddl	554
Rover/p05.hddl	663
Rover/p06.hddl	648
Rover/p07.hddl	724
Rover/p08.hddl	808
Rover/p09.hddl	1113
Rover/p10.hddl	1492
Rover/p11.hddl	1573
Rover/p12.hddl	1629
Rover/p13.hddl	1688
Rover/p14.hddl	1918
Rover/p15.hddl	1928

(a) Total-Order Rover Problems

Problem Name	# Size
Depots/p01.hddl	302
Depots/p02.hddl	330
Depots/p03.hddl	358
Depots/p04.hddl	364
Depots/p05.hddl	414
Depots/p06.hddl	440
Depots/p07.hddl	356
Depots/p08.hddl	390
Depots/p09.hddl	471
Depots/p10.hddl	357
Depots/p11.hddl	435
Depots/p12.hddl	461
Depots/p13.hddl	391
Depots/p14.hddl	414
Depots/p15.hddl	517

(b) Total-Order Depots Problems

Problem Name	# Size
Basic/pb1.hddl	46
um-translog01/problem.hddl	1395
Factories/pfile01.hddl	219
Barman/pfile01.hddl	514

(c) Other Total-Order Problems

Problem Name	# Size
Barman/pfile01.hddl	514
Rover/pfile01.hddl	415
Rover/pfile02.hddl	412
Rover/pfile03.hddl	427
Rover/pfile04.hddl	430

(d) Partial-Order Problems

Problem Name	# Size
basic/problem.jshop	43
rover/pb1.jshop	441
rover/pb2.jshop	443
rover/pb3.jshop	463
rover/pb4.jshop	512
rover/pb5.jshop	579

(e) JSHOP Problems

Table 6.1: Calculated Problem Sizes

6.2.1 Calculating Problem Size

Quantifying the size and complexity of a HTN problem is useful for evaluating heuristics as it allows us to examine when a particular heuristic is no longer a viable option. Comparing heuristics for problems varying in difficulty can expose strengths and weaknesses of each heuristic strategy.

To calculate the size of each problem we sum all of the defined components in the domain and problem files. In the domain file we count the total amount of Tasks, Methods, Actions, Types, and Predicate definitions. From the problem file the amount of Objects, Goal Conditions, Initial Subtasks, and Predicates are counted. Table 6.1 shows the problem sizes of the problems used in this evaluation.

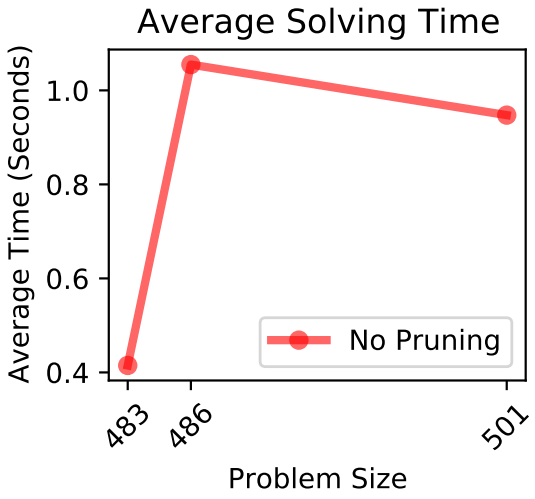
6.2.2 Total-Ordered Problems

The first type of problem we will consider is total-ordered problems. All the problems discussed in this section were run using the total-order solver discussed in Section 4.2.6.1. Each problem was evaluated with *Breadth First Search* and each of the heuristics explored in Section 4.2.7. During the evaluation, each heuristic was paired with each of the three *Search Queues* explained in Section 4.2.5.3. Each combination of heuristic and search queue was used on successive problems until

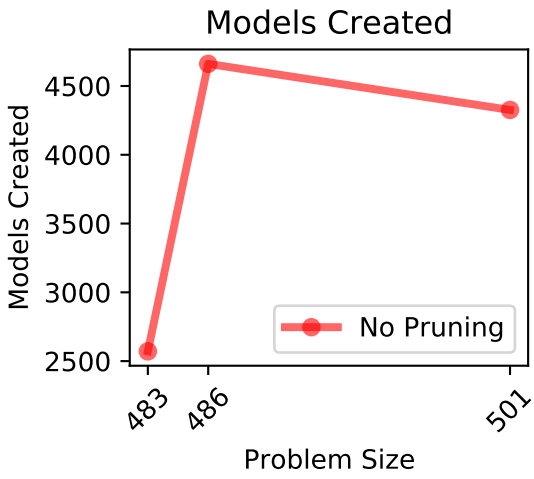
such a time where the time taken to find a result was excessive when compared to other results.

6.2.2.1 Rover Problems

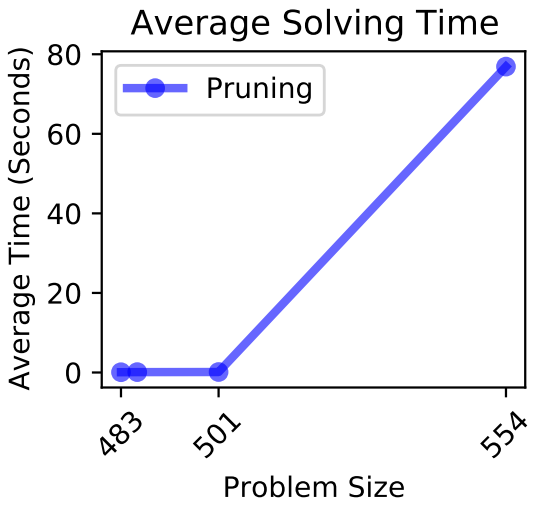
Figures 6.1, 6.2, and 6.3 visualise the results found when evaluating Total-Order Rover problems with all search strategies. The results show a trend of the *Greedy Best First Search* strategy besting the *Total Cost* and *Greedy Cost* strategies with all heuristics. The benefit of the pruning technique put forth in Section 4.2.7.1 is immediately visible when we compare Figures 6.1a and 6.1c. Without pruning the *Breadth First Search* strategy was only able to yield plans for the first three Rover problems. The introduction of pruning was able to push *Breadth First Search* to find a plan for Rover problem four.



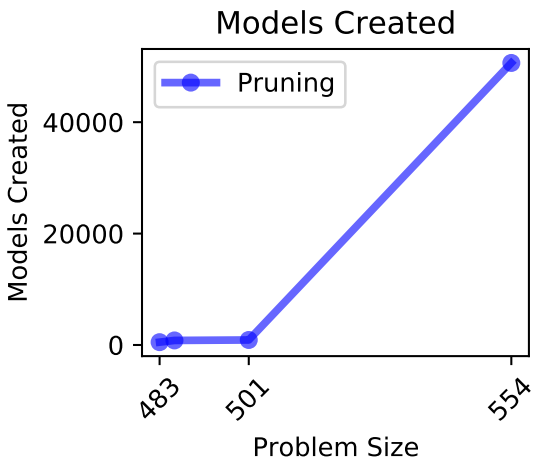
(a) Breadth First - Average Solve Time



(b) Breadth First - Models Created

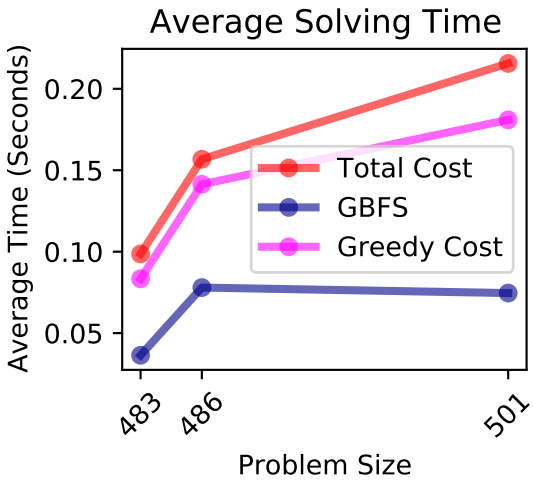


(c) Breadth First (Pruning) - Average Solve Time

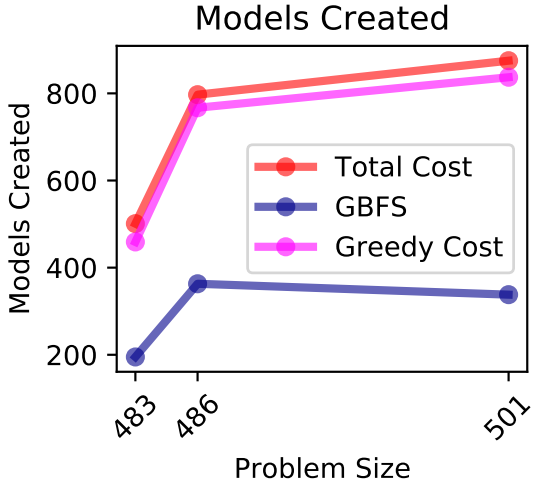


(d) Breadth First (Pruning) - Models Created

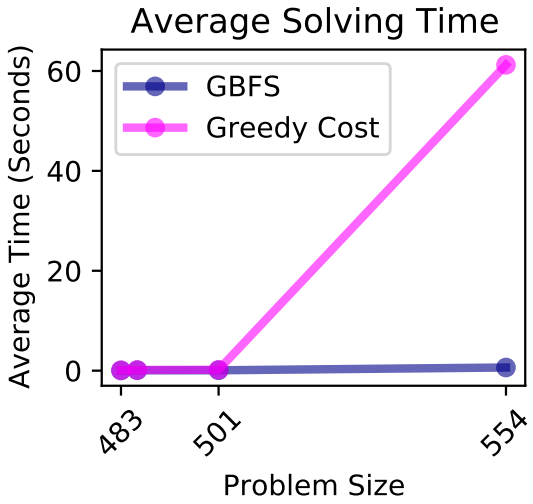
Figure 6.1: Total-Order Rover Problems with Breadth First (No Pruning), Breadth First (Pruning), and Delete Relaxed



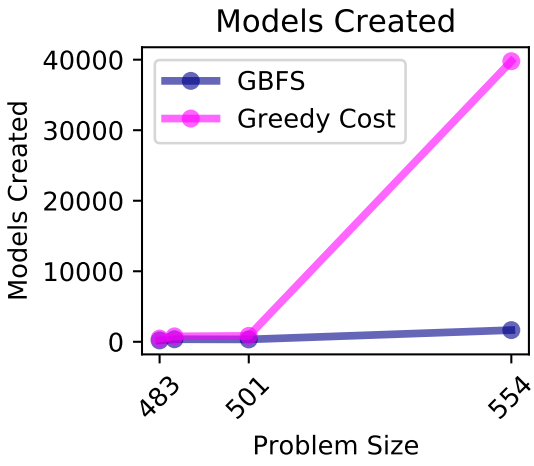
(a) Hamming Distance - Average Solve Time



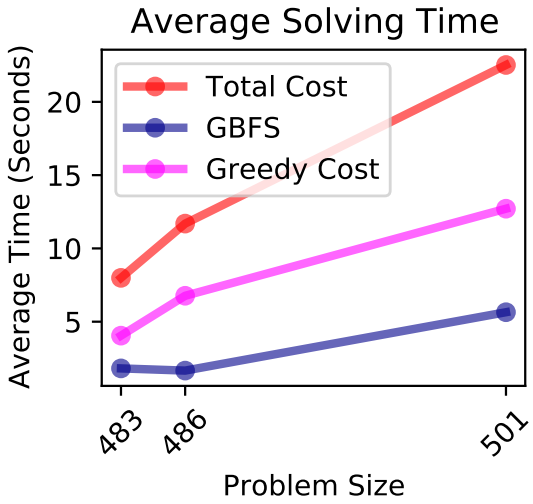
(b) Hamming Distance - Models Created



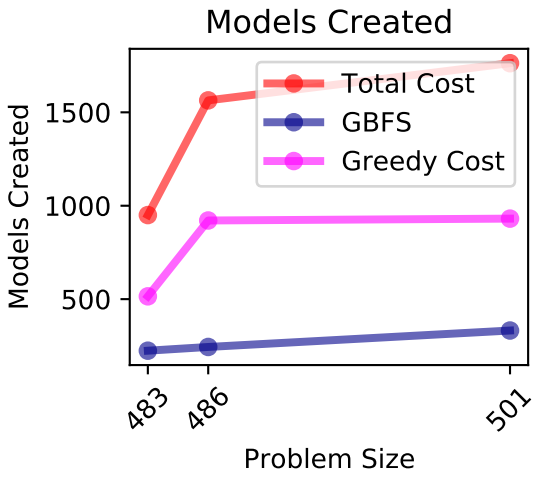
(c) Hamming Distance - Average Solve Time



(d) Hamming Distance - Models Created

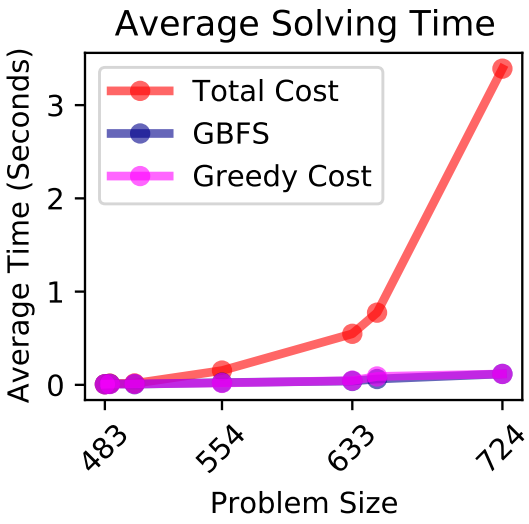


(e) Delete Relaxed - Average Solve Time

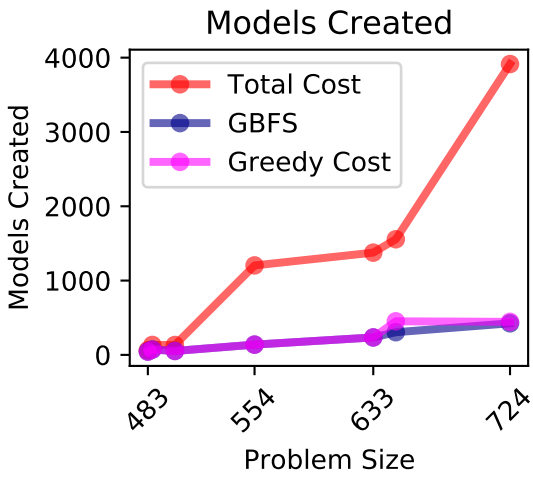


(f) Delete Relaxed - Models Created

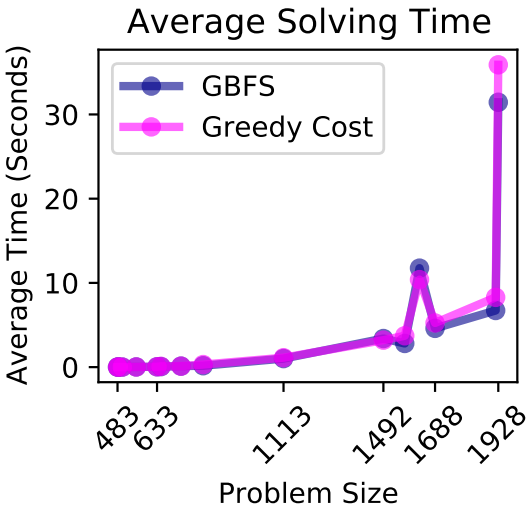
Figure 6.2: Total-Order Rover Problems with Hamming Distance and Delete Relaxed



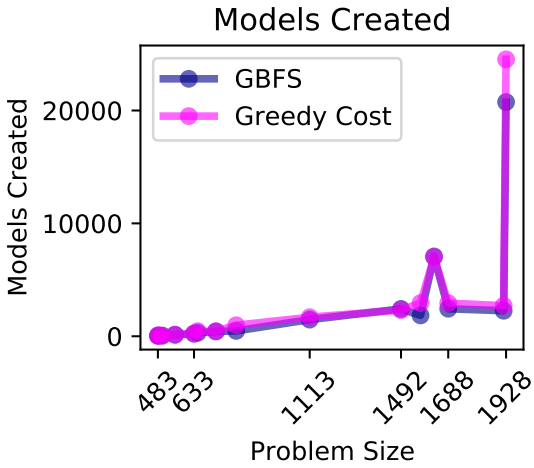
(a) Average Solve Time



(b) Models Created



(c) Average Solve Time

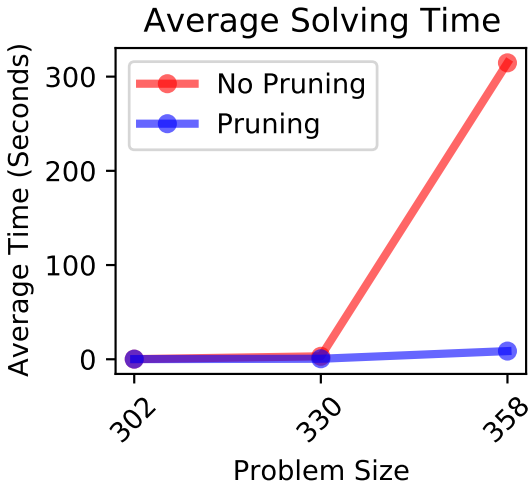


(d) Models Created

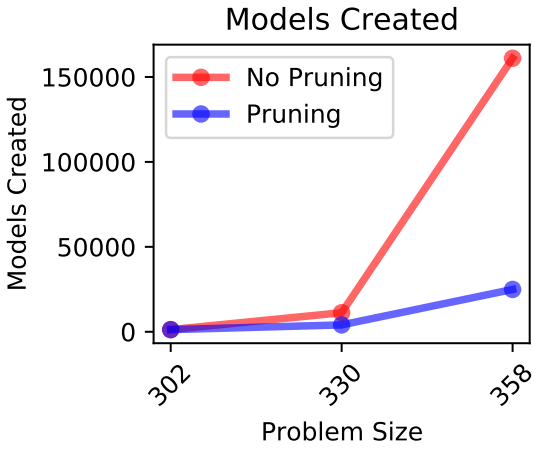
Figure 6.3: Total-Order Rover Problems with Tree Distance

6.2.2.2 Depots Problems

Figures 6.4 and 6.5 display the results recorded when evaluating total-order Depots problems. Table 6.4c shows the average recorded time to solve the first Depots problem using the *Delete and Ordering Relaxed* heuristic. When we compare this performance to that of other strategies, it is plainly clear how ineffective this heuristic is. The *Delete and Ordering Relaxed* heuristic is incredibly expensive to compute, which in this case has resulted in solving time taking longer than all other strategies by a considerable margin.



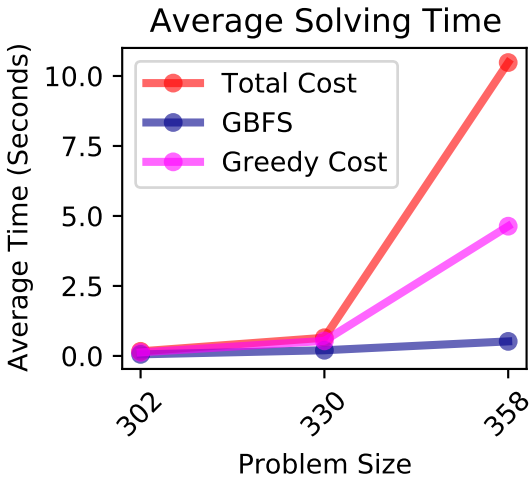
(a) Average Solve Time



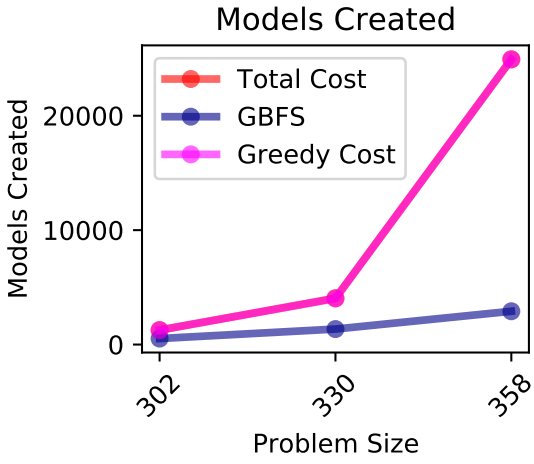
(b) Models Created

Strategy	Av. Solve Time (Seconds)	Models Created
Total Cost	12.808	1347
Greedy Best First	12.485	567
Greedy Cost	12.71	1290

(c) Table of Delete Relaxed Depot pb1 Times

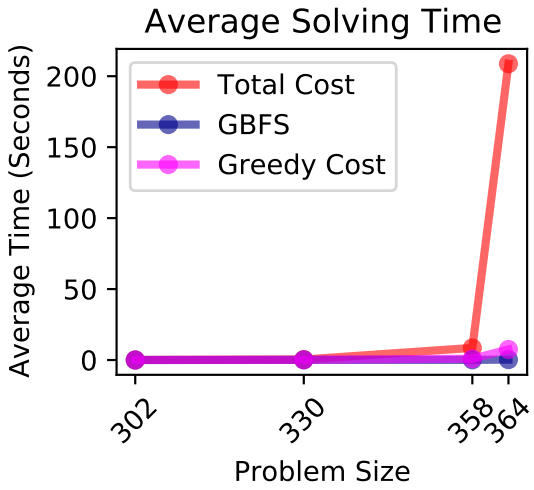


(d) Hamming Distance - Average Solve Times

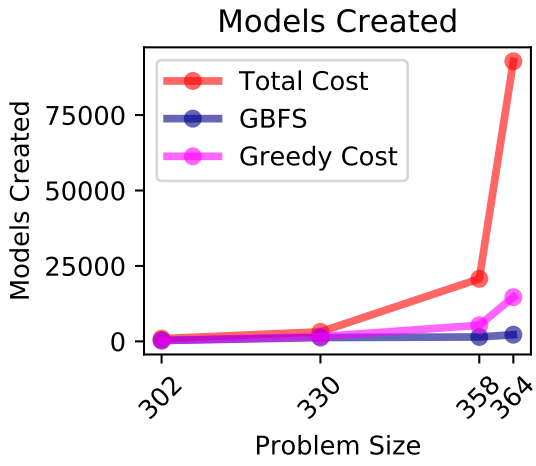


(e) Hamming Distance - Models Created

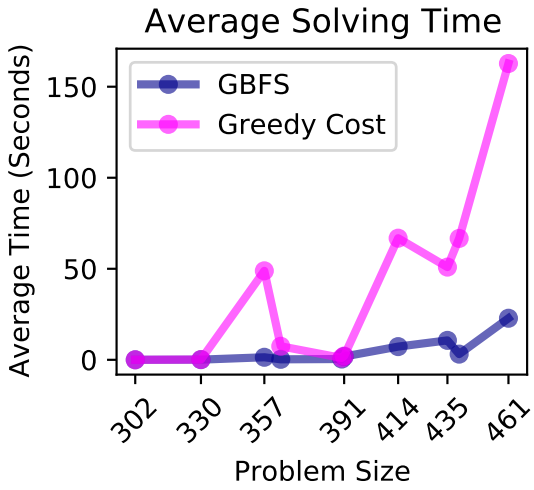
Figure 6.4: Total-Order Depots Problems with Breadth First, Delete Relaxed, and Hamming Distance



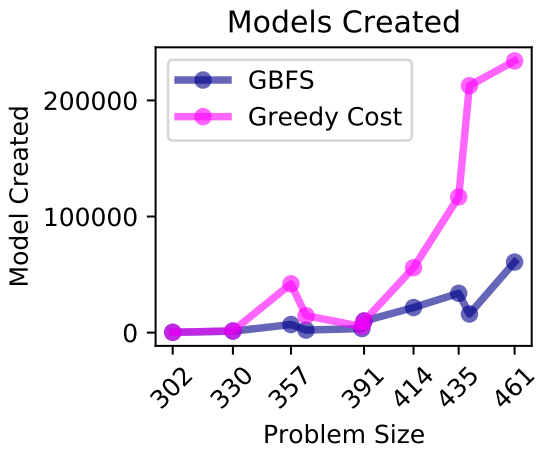
(a) Tree Distance - Average Solve Times



(b) Tree Distance - Models Created



(c) Tree Distance - Average Solve Times



(d) Tree Distance - Models Created

Figure 6.5: Total-Order Depots Problems with Tree Distance

6.2.2.3 Other Problems

Table 6.2 exhibits the results found when evaluating a cocktail of other total-ordered problems. Table 6.2a shows two instances of the *Greedy Best First Search* strategy being unable to find a result when other strategies find a result quite quickly. This is due to the *Greedy Best First Search* strategy falling victim to infinite search loops. This exemplifies the benefit of the *Greedy Cost* strategy in combating this issue.

Table 6.2c illustrates a great example of the *Tree Distance* and *Greedy Best First* strategy honing in on a result quickly. This strategy only created 372 models during search whereas, *Breadth First Search* created 21,845. This is a hugely significant improvement and demonstrates the effectiveness of the *Tree Distance* and *Greedy Best First* strategy in choosing promising models to search with.

Strategy	Av. Solve Time	Models Created
Breadth First	7.096	14249
Breadth First (Pruning)	7.475	14249
Delete Relaxed (Total Cost)	28.12	9029
Delete Relaxed (GBFS)	N/A	N/A
Delete Relaxed (Greedy Cost)	4.937	1245
Hamming Distance (Total Cost)	7.443	14249
Hamming Distance (GBFS)	7.710	14249
Hamming Distance (Greedy Cost)	7.682	14249
Tree Distance (Total Cost)	0.228	1717
Tree Distance (GBFS)	N/A	N/A
Tree Distance (Greedy Cost)	1.237	7315

(a) Factories/pfile01.hddl Problem

Strategy	Av. Solve Time	Models Created
Breadth First	0.137	1289
Breadth First (Pruning)	0.1	1289
Delete Relaxed (Total Cost)	4.687	1395
Delete Relaxed (GBFS)	4.103	1113
Delete Relaxed (Greedy Cost)	4.735	1371
Hamming Distance (Total Cost)	0.123	1289
Hamming Distance (GBFS)	0.131	1289
Hamming Distance (Greedy Cost)	0.109	1289
Tree Distance (Total Cost)	0.099	1289
Tree Distance (GBFS)	0.017	218
Tree Distance (Greedy Cost)	0.013	218

(b) Um-Translog01 Problem

Strategy	Av. Solve Time	Models Created
Breadth First	5.575	21845
Breadth First (Pruning)	5.703	21845
Delete Relaxed (Total Cost)	14.695	5074
Delete Relaxed (GBFS)	7.970	1751
Delete Relaxed (Greedy Cost)	14.784	4961
Hamming Distance (Total Cost)	5.715	21845
Hamming Distance (GBFS)	5.805	21845
Hamming Distance (Greedy Cost)	6.153	21845
Tree Distance (Total Cost)	1.835	11433
Tree Distance (GBFS)	0.041	372
Tree Distance (Greedy Cost)	0.466	3747

(c) Barman/pfile01.hddl Problem

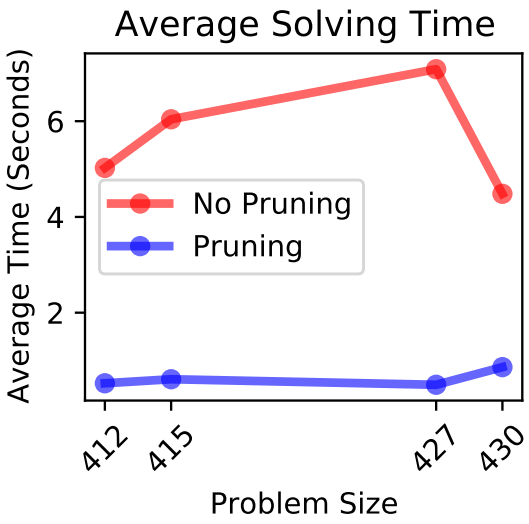
Table 6.2: Problem Evaluation Results

6.2.3 Partial-Ordered Problems

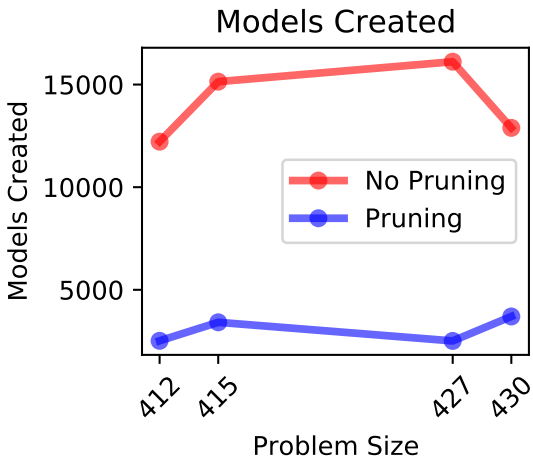
Partial-order problems were evaluated using the same methods as total-order problems as discussed in Section 6.2.2. The only difference being that the heuristics used the Partial-Order pruning technique explained in Section 4.2.7.1.

Figures 6.6 and 6.7 display the results collected for partial-order Rover problems. These results further back-up the observations discussed in Section 6.2.2. The partial-order pruning technique has no noticeable adverse implications in performance compared to the total-order pruning technique.

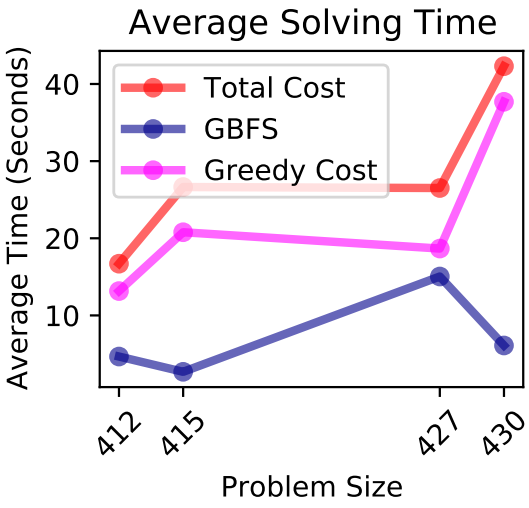
Table 6.3 displays the recorded results when evaluating strategies using the first Barman problem. The results shown highlight the superiority of the *Tree Distance* and *Greedy Best First Search* combination.



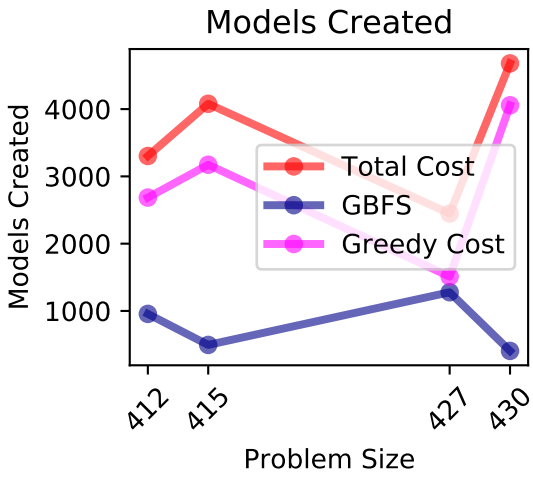
(a) Breadth First - Average Solve Times



(b) Breadth First - Models Created

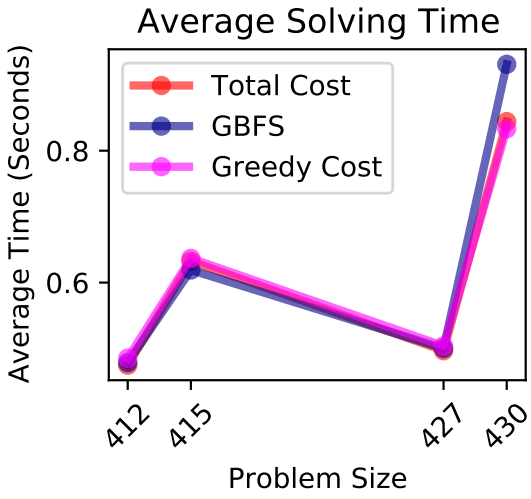


(c) Delete Relaxed - Average Solve Times

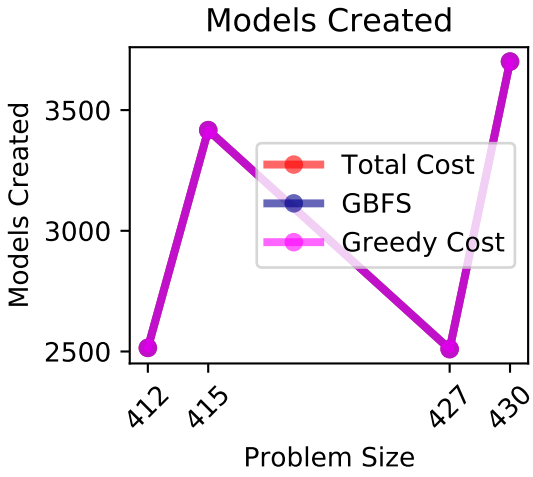


(d) Delete Relaxed - Models Created

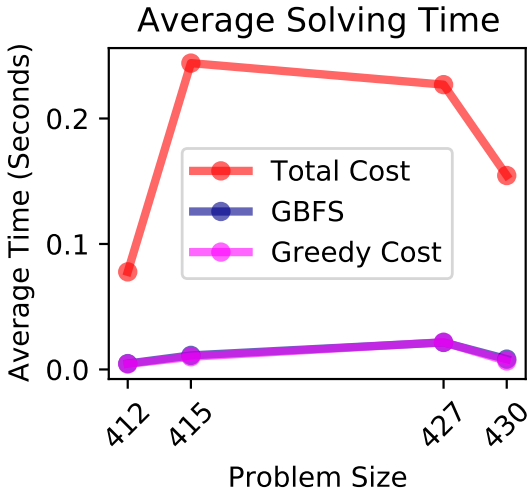
Figure 6.6: Partial-Order Rover Problems with Breadth First and Delete Relaxed



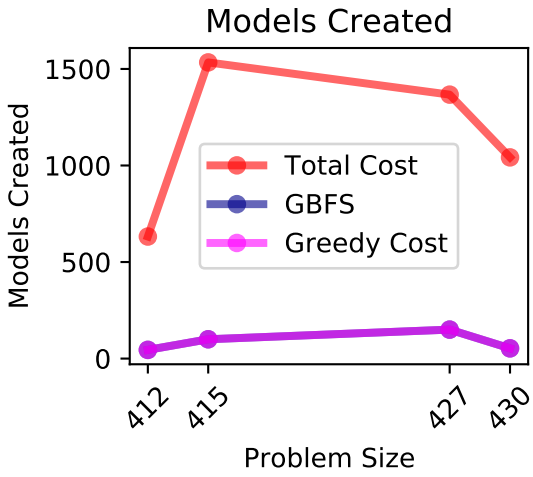
(a) Hamming Distance - Average Solve Times



(b) Hamming Distance - Models Created



(c) Tree Distance - Average Solve Times



(d) Tree Distance - Models Created

Figure 6.7: Partial-Order Rover Problems with Hamming Distance and Tree Distance

Strategy	Av. Solve Time	Models Created
Breadth First	5.584	16280
Breadth First (Pruning)	5.689	16280
Delete Relaxed (Total Cost)	14.574	4030
Delete Relaxed (GBFS)	7.774	1386
Delete Relaxed (Greedy Cost)	14.089	3917
Hamming Distance (Total Cost)	5.566	16280
Hamming Distance (GBFS)	5.412	16280
Hamming Distance (Greedy Cost)	5.483	16280
Tree Distance (Total Cost)	1.757	8497
Tree Distance (GBFS)	0.045	309
Tree Distance (Greedy Cost)	0.452	318

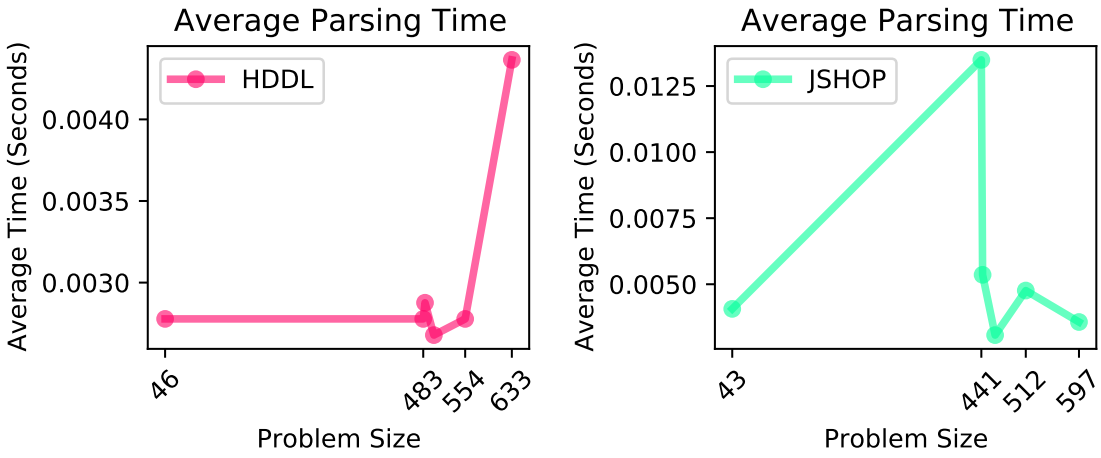
Table 6.3: Partial-Order Barman/pfile01.hddl Results

6.2.4 JSHOP

Now we have discussed both total and partial order HDDL problems, we can compare the performance of HDDL to JSHOP. To do this a few common problems were used, the first is the small Basic problem which is shown in Listings 5.1 and 5.2. HDDL and JSHOP both have a Rover domain which is very similar and provides the same functionality. To create comparable problems, Rover problems 1 to 4 were adapted from HDDL to JSHOP. Search performance is not the only measurable attribute between JSHOP and HDDL, parsing times is also comparable.

6.2.4.1 Parsing Time

Figures 6.8a and 6.8b show the average times for parsing HDDL and JSHOP problems respectively. Upon comparison of these figures it is evident that there is no huge difference between the times. A collection of Independent T-Tests concluded that there was no statistically significant difference between the parsing times recorded.



(a) HDDL Parsing Time

(b) JSHOP Parsing Time

Figure 6.8: Problem Parsing Time

6.2.4.2 Solving Results

When evaluating the solving of JSHOP problems we only consider the *Tree Distance* and *Hamming Distance* heuristics along with *Breadth First Search* since the *Delete and Ordering Relaxed* heuristic is not compatible with JSHOP problems.

Table 6.4 displays the times taken to solve two problems using Breath First (No Pruning) in HDDL and JSHOP. When considering the Basic problems shown in this table, both HDDL and JSHOP recorded very small times for this problem, an Independent T-Test confirmed that there is no significant difference between the two results. The Rover problems also shown in this table show a huge difference between the HDDL and JSHOP results. This shows the added complexity of solving JSHOP problems.

Figure 6.9 illustrates the performance of all search strategies for JSHOP problems. When we compare Figures 6.1c and 6.9a a stark difference in solve time is obvious. The JSHOP problem takes much longer to solve than the HDDL equivalent. When we consider the remaining results a similar picture appears, the JSHOP problems take considerably longer to solve than their HDDL counterparts.

Hamming distance is set apart from its competing heuristics since it was the only one unable to return a plan for JSHOP's Rover problem four.

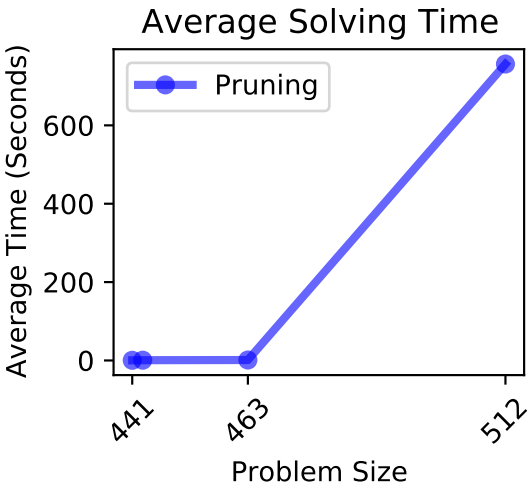
Problem	Language	Av. Solve Time	Models Created
Basic	HDDL	0.0000992	3
Basic	JSHOP	0.0005948	3
Rover 1	HDDL	0.415	2571
Rover 1	JSHOP	49.117	36086

Table 6.4: HDDL & JSHOP - Breadth First (No Pruning) Results

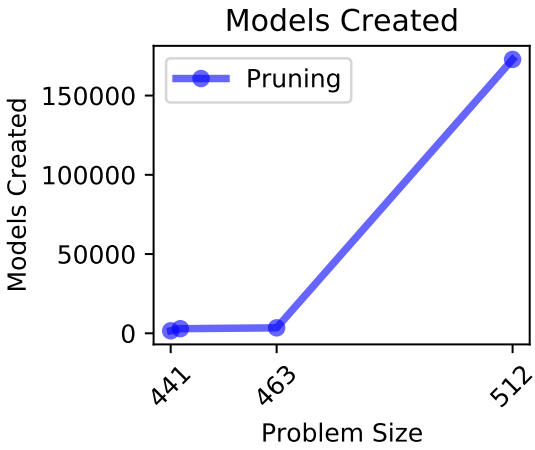
6.2.5 Correlation

To test for a correlation between the time taken to return a result and the amount of models created during search, Spearman's test was employed. Spearman's rank correlation coefficient determines if there is a positive, negative, or no correlation between two sets of numbers. The results of Spearman's test are bound between 1 and -1. Scores of 1 or -1 indicate a monotonic relationship between the sets tested.

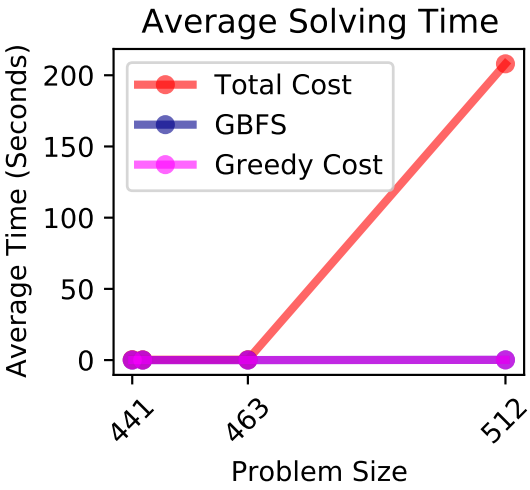
We can use Spearman's test to determine if there is a correlation between the time taken to solve a problem and the amount of models created during solving. A Spearman's test using the amount of models created and solve times for all problems evaluated returned a result of 0.795. This result indicates a strong correlation between solve times and models created, specifically that when the amount of models increases as does the time to solve.



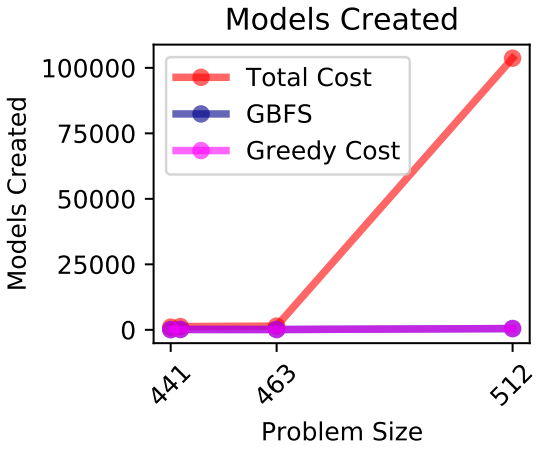
(a) Breadth First - Average Solve Times



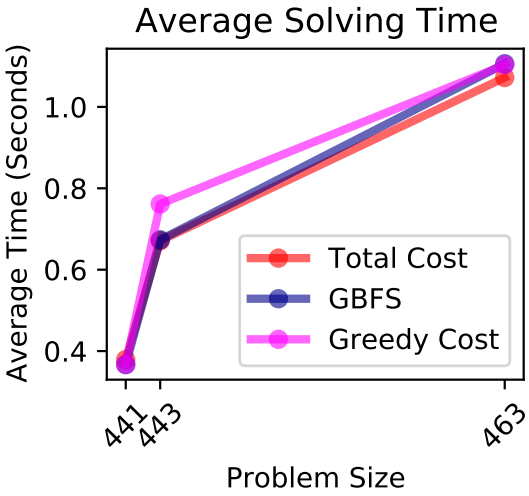
(b) Breadth First - Models Created



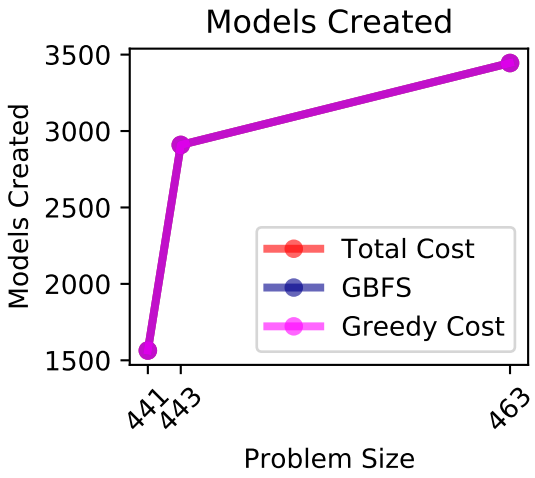
(c) Tree Distance - Average Solve Times



(d) Tree Distance - Models Created



(e) Hamming Distance - Average Solve Times



(f) Hamming Distance - Models Created

Figure 6.9: JSHOP Rover Problems with Breadth First, Tree Distance, and Hamming Distance

Chapter 7

Discussion, Conclusion & Future Work

7.1 Discussion

HTN Planning is an extensive area of study, with a plethora of optimisers and features. Due to the time constraints on this project prioritising specific aspects of the system was essential. A key prioritisation came when considering the non-core functionality of JSHOP compared to heuristic development. Ultimately, the extra JSHOP functionality was shelved to allow for work on heuristics.

For this project a lot of knowledge of new concepts was required, AI Planning was a new topic of study at the off set of the project. Learning and utilising concepts concurrently was at times a great challenge.

In Section 6.2 we discussed the impact different search strategies have on the performance of the system for differing problem types. The drastic improvement from standard *Breadth First Search* to much more efficient search strategies is a hugely motivational aspect of this project. These advancements leave much to ponder about new strategies and opimisers that can provide more impressive results.

A known limitation of the *Delete and Ordering Relaxed* Heuristic is that it does not find valid plans for some problems. Since the intention of this heuristic in this project was to act as a proof of concept to the compatibility for Classical Planning heuristics the limitations of this prototype are not monumental. The limitations of the heuristic can be overcome during any future work to the system.

A key feature of the system is its extendability, future users are encouraged to develop new components to make use of the capability of interchangeable components covered in Section 4.2.2.1. The components currently in the system can be used as guidelines for future users when experimenting with their own ideas.

7.2 Conclusion

Section 4.1 defines the functional and non-functional requirements put forth for this project. An achievement of the developed system is that all of the defined functional requirements are satisfied. The developed system satisfies all but one of the non-functional requirements. The requirement for *Effective Error Handling and Reporting* is only partly satisfied in the systems current form. Error handling is widespread across the system and its effectiveness was put to use during later implementation phases. The reporting of errors is admittedly not great in the current system, finding the cause of an error almost always requires looking at the source code. This omittance

was a conscious one; as mentioned in Section 7.1 prioritising aspects of the system was essential, which led to a choice between error reporting and core functional requirements.

HTN planning is an active area of research with new concepts being proposed and trialled. This planner provides the opportunity to trial ideas and components without the need for a new planner to be constructed. The inclusion of interchangeable components is a future-looking feature with the intention of being used for prototyping and concept proving.

7.3 Future Work

In Section 6.2.2.3 we saw the adverse impact of infinite loops when using a *Greedy Best First* search strategy. A future addition to the system could be a mechanism of recognising when a model is in an endless loop.

Furthermore, since in Section 4.2.7.1 we saw the benefit of the simple model pruning technique, a more advanced pruning method could prove beneficial to the system. Pruning already seen state and task network pairs has the potential to severely improve search times.

Hopefully, future work uses this systems key principles; to provide a base point for developing new search strategies. Work to add functionality for new input languages or extend the functionality of the already established *HDDL* and *JSHOP* languages would be very welcome. In this project we only explored one Classical Planning based heuristic, expanding on this with more complex Classical Planning heuristics could improve the performance of the planner.

The system developed for this project follows a ground planning approach, other approaches such as lifted planning would provide a new challenge relevant to the knowledge gained from this project. Other planning concepts such as Satisfiability (SAT) planning could also be used to further develop the knowledge of AI planing gained from this project.

The inclusion of meaningful error messages during parsing and search could also be a front for future work. Since the aim of this project is to encourage people to experiment with HTN planning, errors which effectively inform users of issues would be a great addition.

Appendix A

User Manual

A.1 Running the planner

There are two methods of running the planner, from the command line and by using the Runner class.

A.1.1 Running from Command line

From the main project directory the planner can be initiated using the command:

python ./Runner.py <Domain File Path> <Problem File Path>

Additional help can be found via the following command:

python ./Runner –help

A.1.2 Running Via Runner Class

The Runner class can be imported to a program and used to control the planner. The following example shows how the operations of the class can be used:

```
1  from runner import Runner
2
3  controller = Runner(domain_path, problem_path)
4  controller.parse_problem()
5  controller.parse_problem()
6  controller.set_solver(Solver)
7  controller.set_search\_queue(SearchQueue)
8  controller.set_parameter_selector(ParameterSelector)
9  controller.set_heuristic(Heuristic)
10 result = controller.solve()
```

A.2 Example Problems

Example problems can be found in the Tests/Examples directory.

A.3 Component Selection

Interchangeable components can be set from the command line or via the Runner class as previously seen.

Four types of component can be interchanged; Heuristics, Search Queues, Parameter Selectors, and Solvers. When setting each of these components from the command line two pieces of information need to be set - class names and file paths.

The following tables show the class name and file path pairs used to set components.

A.3.1 Total-Order Heuristics

Heuristic	-heuModName	-heuPath
Delete & Ordering Relaxed	DeleteRelaxed	Solver/Heuristics/delete_relaxed.py
Tree Distance	TreeDistance	Solver/Heuristics/tree_distance.py
Hamming Distance	HammingDistance	Solver/Heuristics/hamming_distance.py

A.3.2 Partial-Order Heuristics

Heuristic	-heuModName	-heuPath
Delete & Ordering Relaxed	DeleteRelaxedPartialOrder	Solver/Heuristics/ delete_relaxed_partial_order.py
Tree Distance	TreeDistancePartialOrder	Solver/Heuristics/ tree_distance_partial_order.py
Hamming Distance	HammingDistancePartialOrder	Solver/Heuristics/ hamming_distance_partial_order.py

A.3.3 Search Queue

Queue Type	-searchQueueName	-searchQueuePath
Total Cost	SearchQueue	Solver/Search_Queues/search_queue.py
Greedy Best First	GBFSSearchQueue	Solver/Search_Queues/ Greedy_Best_First_Search_Queue.py
Greedy Cost	GreedyCostSearchQueue	Solver/Search_Queues/ Greedy_Cost_So_Far_Search_Queue.py

A.3.4 Parameter Selectors

Selector Name	-paramSelectName	-paramSelectPath
All Parameters	AllParameters	Solver/Parameter_Selection/All_Parameters.py
Requirement Selector	RequirementSelection	Solver/Parameter_Selection/ Requirement_Selection.py

A.3.5 Solvers

Solver Name	-solverModName	-solverPath
Total-Order Solver	TotalOrderSolver	Solver/Solving_Algorithms/total_order.py
Partial-Order Solver	PartialOrderSolver	Solver/Solving_Algorithms/partial_order.py

A.4 Output

From the command line the plan found from a problem will be displayed on screen. Using the Runner class a plan can be printed using the following method:

Runner.output_result(Result)

The plan found for a problem can also be written to a Pickle file. From the command line the file path for output can be set using the following argument:

python ./Runner <Domain File> <Problem File> -w <Output File Path>

Using the Runner class the same output can be achieved using the following method:

Runner.output_result_file(Result, file_path)

An example of a plan being displayed is shown below in Figure A.1:

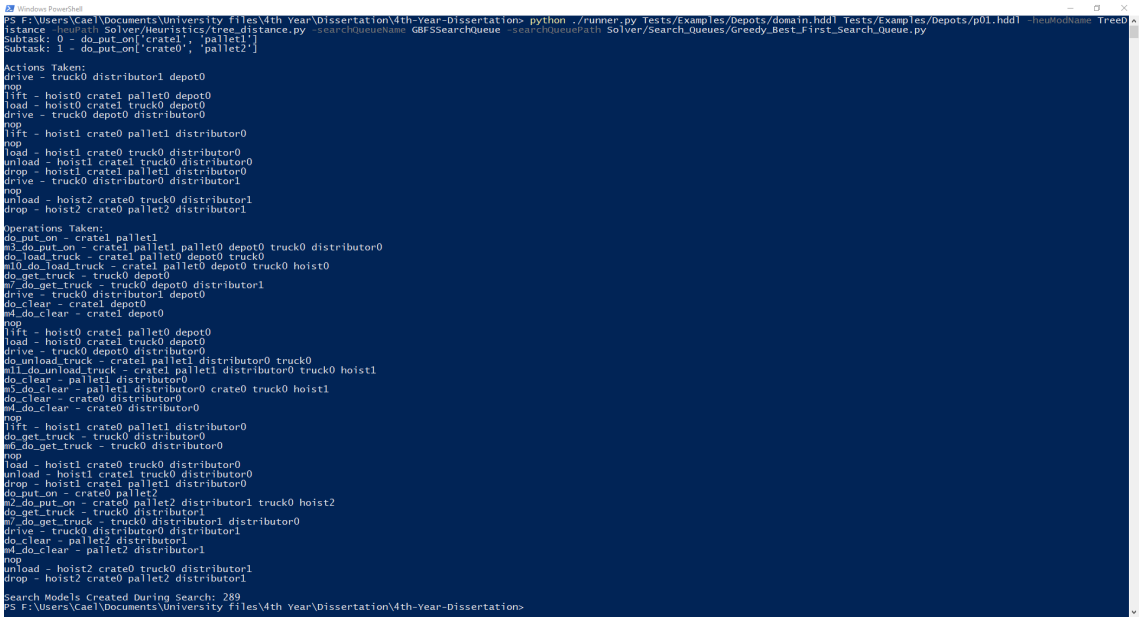


Figure A.1: Example of Plan Being Displayed

A.4.1 Output Plan Reader

Plans saved to Pickle files can be displayed on screen by using the plan reader tool. The plan reader can be found in the Tests/Evaluation directory.

python ./output_plan_reader.py <Output File>

A.5 Running Unittests

All the unittests can be run from the Tests/UnitTests directory using the following command:

python ./All_Tests.py

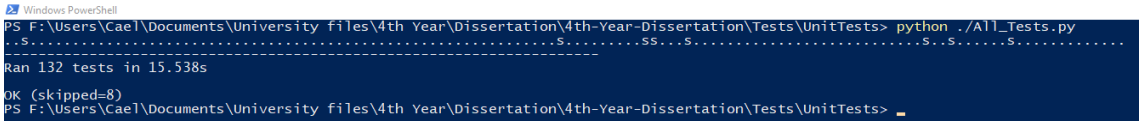


Figure A.2: All Unit Tests Being Run

A.6 System Evaluation

System evaluation is composed in a similar manner to the unit tests. All evaluation files are contained in the Tests/Evaluation/Heuristic_Evaluation directory. From here all evaluation tests can be executed using the following command: *python ./AllEvaluationTests.py*

The Tests/Evaluation/Heuristic_Evaluation/Archive directory is where all the results of the evaluation tests are stored. Within this directory results are sorted by problem domain.

The *Heuristic_Evaluation_Data_Processing.ipynb* notebook visualises the results collected using graphs.

A.7 Demo Running Configurations

1. Tests/Examples/Basic/basic.hddl Tests/Examples/Basic/pb1.hddl

2. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p01.hddl

3. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p02.hddl
   -heuModName DeleteRelaxed -heuPath Solver/Heuristics/delete_relaxed.py

4. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p02.hddl
   -heuModName TreeDistance -heuPath Solver/Heuristics/tree_distance.py

5. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p05.hddl
   -heuModName TreeDistance -heuPath Solver/Heuristics/tree_distance.py
   -searchQueueName GBFSSearchQueue
   -searchQueuePath Solver/Search_Queues/Greedy_Best_First_Search_Queue.py

6. Tests/Examples/Partial_Order/Rover/domain.hddl
   Tests/Examples/Partial_Order/Rover/pfile01.hddl -heuModName HammingDistance
   -heuPath Solver/Heuristics/hamming_distance.py
   -searchQueueName GreedyCostSearchQueue
   -searchQueuePath Solver/Search_Queues/Greedy_Cost_So_Far_Search_Queue.py

7. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p01.hddl
   -solverModName TotalOrderSolver -solverPath Solver/Solving_Algorithms/total_order.py

8. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p01.hddl -paramSelectName
   AllParameters -paramSelectPath Solver/Parameter_Selection/All_Parameters.py

9. Tests/Examples/JShop/rover/rover.jshop Tests/Examples/JShop/rover/pb1.jshop

Appendix B

Maintenance Manual

B.1 Installing the system

This system can be downloaded from GitHub - https://github.com/C-Milne/4th-Year-Dissertation

B.2 Dependencies

This system requires Python to be installed. Python downloads can be found here - https://www.python.org/

All other packages used by the system are included with Python as standard.

This system has been tested with Python versions 3.9 & 3.10

B.3 Running the system

From the command line a problem can be given to the planner to solve using the command:

python ./Runner.py <Domain File Path> <Problem File Path>

For more information on running the system please refer to the User Manual - https://github.com/C-Milne/4th-Year-Dissertation/blob/main/README.md

B.4 Space and memory requirements

Memory requirements depend entirely on the size of the problem being solved. Larger problems require more options to be searched which in turn uses more memory.

B.5 Key File Paths

1. Heuristics : /Solver/Heuristics
2. Parameter Selectors : /Solver/Parameter_Selection
3. Search Queues : /Solver/Search_Queues
4. Solving Algorithms : /Solver/Solving_Algorithms
5. Parsers : /Parsers
6. Representation Objects : /Internal_Representation
7. Example Problems : /Tests/Examples
8. Evaluation methods : /Tests/Evaluation/Heuristic_Evaluation
9. Unit Tests : /Tests/UnitTests

B.6 Directions for Future Improvements

When developing new interchangeable components specific classes need to be inherited by any developed component.

B.6.1 Heuristic

Developed Heuristics need to inherit the Heuristic class found in the Solver/Heuristics/Heuristic.py file. There are some alternatives to inheriting the Heuristic class directly, the Pruning and NoPruning classes found in files Solver/Heuristics/pruning.py and Solver/Heuristics/no_pruning.py respectively. Both of these classes inherit from the Heuristic class. The Pruning Class contains functionality for basic model pruning that can be inherited by other heuristics. The PartialOrderPruning Class from file Solver/Heuristics/partial_order_pruning.py provides the same functionality but for partial-order problems.

B.6.2 Parameter Selector

Developed Parameter Selectors need to inherit the ParameterSelector Class found in file Solver/-Parameter_Selection/ParameterSelector.py.

B.6.3 Search Queue

Developed Search Queues need to inherit the SearchQueue class found in Solver/Search_Queues/search_queue.py.

B.6.4 Solver

Developed Solvers need to inherit the Solver class within the Solver/Solving_Algorithms/solver.py file.

B.7 Running Unittests

All the unittests can be run from the Tests/UnitTests directory using the following command:

python ./All_Tests.py

B.8 Bug Solving

When attempting to debug the system it is recommended that a debugger with breakpoints is used. To aid in the debugging process the search procedure can be manually controlled using a script. Below in Listing B.1 is a snippet of a test case from the file Tests/UnitTest/JSHOP_Solving_Tests.py:

```
def test_rover_execution_part_guided(self):
    domain, problem, parser, solver = env_setup(False)
    parser.parse_domain(self.rover_test_path + "rover.jshop")
    parser.parse_problem(self.rover_test_path + "problem.jshop")

    execution_prep(problem, solver)
    solver.parameter_selector.presolving_processing(domain, problem)
    # res = solver.solve()

    solver._search(True)
```

Listing B.1: Example of using search step functionality

In this example the search produce is completely controlled by the script shown. Notice that instead of using *solver.solve()*, *solver._search(True)* is being used. This acts as a step control for search. Each functional call of *solver._search(True)* will only decompose 1 Task, Method, or Action. This is an effective way to debug and track search procedure.

Bibliography

[1] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl| the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[2] Diego Aineto, Sergio Jiménez, and Eva Onaindia. Learning strips action models with classical planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 28(1):399–407, Jun. 2018.

[3] G Behnke, D Höller, P Bercher, S Biundo, Damien Pellier, H Fiorino, and R Alford. Hierarchical Planning in the IPC. In *Workshop on HTN Planning (ICAPS)*, Berkeley, United States, July 2019.

[4] Gregor Behnke, Daniel Höller, and Pascal Bercher, editors. *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, 2021.

[5] Gregor Behnke, Daniel Höller, Alexander Schmid, Pascal Bercher, and Susanne Biundo. On succinct groundings of HTN planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*. AAAI Press, 2020.

[6] Gregor Behnke, Daniel Höller, and Susanne Biundo. totsat - totally-ordered hierarchical planning through sat. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[7] Ali Bolu and Ömer Korçak. Adaptive task planning for multi-robot smart warehouse. *IEEE Access*, 9:27346–27358, 2021.

[8] Stephen Cass. The 2021 top programming languages. *IEEE Spectrum*, 2021.

[9] Amanda Coles, Andrew Coles, Alvaro Torralba, and Florian Pommerening. An overview of the international planning competition part 1: Classical tracks, 2019.

[10] Lavindra De Silva, Raphaël Lallement, and Rachid Alami. The hatp hierarchical planner: Formalisation and an initial study of its usability and practicality. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6465–6472, 2015.

[11] George W Ernst and Allen Newell. Generality and gps. 1967.

[12] Kutluhan Erol, James A Hendler, and Dana S Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Aips*, volume 94, pages 249–254, 1994.

[13] Juan Fernandez-Olivares, Ignacio Vellido, and Luis Castillo. Addressing HTN planning with blind depth first search. In *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, pages 1–4, 2021.

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

[15] Ilce Georgievski and Marco Aiello. An overview of hierarchical task network planning. *CoRR*, abs/1403.7426, 2014.

[16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[17] Arturo González-Ferrer, Juan Fernández-Olivares, Luis Castillo, et al. Jabbah: a java application framework for the translation between business process models and htn. *Proceedings of the 3rd International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS'09)*, 2009.

[18] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019.

[19] James A. Hendler, Austin Tate, and Mark Drummond. Ai planning: Systems and techniques. *AI Magazine*, 11(2):61, Jun. 1990.

[20] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. The panda framework for hierarchical planning. *KI-Künstliche Intelligenz*, pages 1–6, 2021.

[21] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. Hddl: An extension to pddl for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9883–9891, 2020.

[22] Daniel Höller, Pascal Bercher, and Gregor Behnke. Delete-and ordering-relaxation heuristics for htn planning. In *IJCAI*, pages 4076–4083, 2020.

[23] Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. A generic method to guide htn progression search with classical heuristics. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[24] Okhtay Ilghami and Dana S Nau. A general approach to synthesize problem-specific planners, 2003.

[25] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[26] Charles Lesire and Alexandre Albore. pyHiPOP – Hierarchical partial-order planner. In *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, pages 13–16, 2021.

[27] Tomas Lozano-Perez. A simple motion-planning algorithm for general robot manipulators. *IEEE Journal on Robotics and Automation*, 3(3):224–238, 1987.

[28] Maurício Cecílio Magnaguagno, Felipe Meneguzzi, and Lavindra de Silva. HyperTensioN: A three-stage compiler for planning. In *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, pages 5–8, 2021.

[29] Drew M. McDermott. The 1998 ai planning systems competition. *AI Magazine*, 21(2):35, Jun. 2000.

[30] Samuel P Morgan. Hamming, richard w. In *Encyclopedia of Computer Science*, pages 765–766. 2003.

[31] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. *Proceedings of the 16th international joint conference on Artificial intelligence*, 2:968–973, 1999.

[32] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop and m-shop: Planning with ordered task decomposition. Technical report, MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE, 2006.

[33] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2 - an htn planning system. *Journal of Artificial Intelligence Research*, 2003.

[34] Dana S Nau, Stephen JJ Smith, Kutluhan Erol, et al. Control strategies in htn planning: Theory versus practice. In *AAAI/IAAI*, pages 1127–1133, 1998.

[35] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55, 2016.

[36] Damien Pellier and Humbert Fiorino. Totally and partially ordered hierarchical planners in PDDL4J library. In *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, pages 17–18, 2021.

[37] Abdeldjalil Ramoul, Damien Pellier, Humbert Fiorino, and Sylvie Pesty. Grounding of htn planning domain. *International Journal on Artificial Intelligence Tools*, 26(05):1760021, 2017.

[38] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Pearson education limited, 2022.

[39] Earl D Sacerdoti. A structure for plans and behavior. Technical report, SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1975.

[40] Dominik Schreiber. Lifted logic for task networks: TOHTN planner lilotane in the IPC 2020. In *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, pages 9–12, 2021.

[41] Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Toma´s Balyo˘. Tree-rex: Sat-based tree exploration for efficient and high-quality htn planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):382–390, May 2021.

[42] Vikas Shivashankar, Ugur Kuter, and Dana S Nau. Hierarchical goal network planning: Initial results. Technical report, MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE, 2011.

[43] David E Smith, Jeremy Frank, and William Cushing. The anml language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, page 31, 2008.

[44] Lakshmisri Surya. An exploratory study of ai and big data, and it's future in the united states. *International Journal of Creative Research Thoughts (IJCRT), ISSN*, pages 2320–2882, 2015.

[45] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples

using weighted max-sat. *Artificial Intelligence*, 171(2):107–143, 2007.