Adaption, Implementation, and Evaluation of Search Strategies in a HTN Planner

Cael Milne

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

Word Count: 16,605

Department of Computing Science

2023

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:

C. Milne

Date: 2023

Abstract

In this work we research, adapt, implement, and evaluate a *Classical Planning* strategy called *Novelty* for use in *HTN Planning*. *Novelty* is a simple strategy which prioritises nodes that find previously unseen facts. *HTN Planning* is an exciting area of AI planning which has been used in cutting edge technical environments such as operating Mars Rovers in the form of *NASA's Europa Planner*. We contribute our work to the *EPICpy* planner in the form of new components and general optimisations. We evaluate each developed strategy and analyse the gathered data to produce evidence of correlations between attributes and strategies. We use this information to form a basis for creating new search strategies for *HTN Planning*. We show that strategies which are more greedy than others produce solutions quicker and that strategies which prioritise finding previously unseen facts too much can actually produce slower solve times. Instead a high percentage of previously unseen facts within a lower total amount of facts correlates to better solve times. Our additions to *EPICpy* can be further built upon to bring the planner closer to the wider planning community and for the implementation and evaluation of new strategy ideas.

Acknowledgements

This project would not have been possible without the enthusiasm of my supervisor Professor Felipe Meneguzzi and André Grahl Pereira both of whom deserve more thanks and acknowledgement than I can muster in this section. Truly, thank you.

As my time at University draws to its natural end I wish to take this final opportunity to pay tribute to my mam and dad - Heather & Jimmy - for the unwavering support they have showered me in, not only during this project or degree but my entire life. Furthermore, I thank my grandparents, some of whom didn't live long enough to see this degree conclude. Finally, who could forget my troublesome brother with his unique approach to encouragement.

Contents

Chapter 1

Introduction

AI planning comes in many forms each with the same fundamental goal of choosing actions to apply to some environment over time. Some types of AI planning strive to return the optimal plan or plan with the least cost, while others simply aim to find and return a valid plan as quickly as possible.

Most modern planners rely on heuristic search algorithms [25]. Such planners include the *Fast Downward Planner* [24], *PANDA* [35], *PDDL4J* [55], and *PyHiPOP* [46]. Planning problems are often large and complex, this means the effectiveness of heuristic driven planners depends largely on a heuristics guiding search down a path which leads efficiently towards the goal. Using heuristics can come with a trade off, while an informative heuristic can guide the planner to a better path, it can also be expensive to compute which in turn can significantly slow down the progression of a planner.

The two types of AI planning which concern us in this work are *Classical Planning* and *Hierarchical Task Network (HTN) Planning*. In Classical Planning problems a planner applies *Actions* to a given *Initial State*, each of which transforms the *State* into a new *State*. *Actions* are applied successively until a *State* which satisfies specified *Goal Conditions* is reached. In this type of problem, the planner has complete autonomy surrounding which *Action* to apply next - as long as specified preconditions of the *Action* are met. *States* are comprised of a set of *Facts* which describe the environment.

HTN Planning builds upon *Classical Planning* problems with the introduction of *non-primitive Tasks* and *decomposition Methods* [27]. These new operators allow a problem designer to model higher level tasks, which can be decomposed using recipes collected from domain experts. This provides the planner with some more domain specific knowledge. An example of a recipe for a task of traveling between two cities could be *travel by taxi* or *travel by air*. These recipes would involve different steps, for example, *travel by taxi* would have three steps: *get taxi*, *travel to destination*, and *pay driver*. This is different from *travel by air* which would have the steps: *get ticket*, *taxi to airport*, *fly to destination airport*, and *taxi to destination*.

HTN planning problems also add the notion of the *Task Network* which can be described simply as a queue of operations which the planner must execute. Despite the constraints of the *Task Network* HTN planners do have autonomy over a few aspects of the planning process. As we discussed below, the planner handles the choice of which *States* to continue searching from, which decomposition *Methods* to use, which task orderings to assign, and what parameters to select. HTN planning problems differ slightly from Classical planning problems in terms of what is deemed a goal, *Goal Conditions* are optional in HTN planning problems. Instead HTN planning problems require all operations in the *Task Network* to have been executed by the planner.

In this work we investigate heuristic search strategies for HTN planning, evaluate their effectiveness, and adapt an existing heuristic from *Classical Planning* to *HTN Planning*. We use the HTN planner *EPICpy* (Extendable Planner with Interchangeable Components in Python) [52]. We show improvement in planner performance using a simple optimisation and gather information from a range of strategies during search to show the relationship between solve-time attributes and solve time.

1.1 Motivation

AI planning is an exciting area of study which can rival and surpass current fashionable trends in the technology space. An example of this is an AI planner using a strategy called *Novelty* being used to beat a *Deep Q-Network* agent at playing a selection of *Atari* games. When comparing Table 1 from the paper *Classical Planning with Simulators: Results on the Atari Video Games* [49] against Table 2 from paper *Human-level control through deep reinforcement learning* [53] we can see that the AI Planning strategy produced better results than a Deep Q-Network Agent in the majority of games tested.

The *Europa* planner [23] is an impressive planner developed and used by NASA for a "variety of missions, mission-oriented research, and demonstration". A standout application of the planner is its usage in robots and rovers, it has been applied in both land [62] and sea [58] based autonomous machines. Furthermore, it was the central planning system for planning daily activities for robot missions to Mars[1] [2].

There is an obvious discrepancy between the number of heuristics available for *Classical Planners* compared to *HTN Planners*. Classical planning boasts an impressively large collection of works which propose heuristic strategies for both satisficing and optimising planning [60, 47, 17, 33, 40, 43, 41, 26]. HTN planning on the other hand does not have the same backdrop of a large collection of tried and tested heuristics.

1.2 Objectives

In this project we set out to develop a comprehensive knowledge of select search strategies for AI planning and to adapt and implement such strategies for a given HTN Planner. Here we lay forth the key objectives of this project:

1. Research, implement, and evaluate search strategies for HTN Planing. This involves researching strategies used by existing planners and those proposed in literature.

2. Research, adapt, and evaluate the *Classical Planning* strategy *Novelty* for use within the HTN planning space.

3. Achieve improved solve times through the introduction of planner optimisations.

4. Add compatibility with IPC's plan verification module (covered in Section 2.2.1) to the planner and verify the correctness of all plans output.

[1] https://github.com/nasa/europa/wiki/What-Is-Europa

5. If the time constraints of this project permit, implement strategies of existing planners.

Chapter 2

Background on Planning

This chapter provides a brief background on AI planning and formalises planning components while analysing how they co-exist to create a planning problem. We then progress to inspect work done to bring the AI planning community together in a competitive progress-driven spirit.

2.1 Planning

In this section we explore the basis of both *Classical Planning* and *Hierarchical planning*. Before we immerse ourselves in the details of both types of planning, we compare and contrast them. *K. Erol*, *J. Hendler*, and *D.S. Nau* provide us with a high-level difference of the two types of planning in their paper *Complexity Results for HTN Planning* [19]. This paper describes *Classical Planning* as "Given an initial state, STRIPS-style planners search for a sequence of actions that would bring the world to a sate that satisfied certain conditions". The same paper describes *Hierarchical Planning* as "search for plans that accomplish task networks instead", these descriptions give us a very slight insight to the difference between the distinct types of planning.

We require knowledge of the *PDDL* and *HDDL* input languages that describe planning problems. These languages are related and as such, some definitions are exactly the same between the two but, they describe different types of planning problems. *PDDL* typically defines single-agent problems where the environment only changes if the planner executes some *Action* [31], *HDDL* builds upon PDDL to define *HTN* problems following the same principle.

2.1.1 Classical Planning

The Stanford Research Institute Problem Solver (STRIPS) [22] is an early *Classical Planner* developed in 1971. While the software is no longer available, the formalism underpinning its input language still underpins modern input languages such as PDDL [1]. The input language devised for STRIPS was utilised as the initial version of PDDL [20]. PDDL is significant since it is the input language used for the International Planning Competition's (*IPC*) Classical Track which we cover in Section 2.2. A selection of published papers provide information on the composition of *Classical Planning* [22, 1] which we will cover in this section.

Objects are used as parameters to *Facts* and *Actions*. We define *objects* in Definition 1 as tuple with only two components. The name of an *object* is used when mapping to a *Fact* or *Action*. The function of an *object*'s *type* is discussed in Section 2.1.2.1.

Definition 1. PDDL Object

An object in PDDL is defined as a tuple (N, T) where:

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

R. Fikes and *N. Nilsson* [22] write of "world models" as the search space for the STRIPS planner. In this section we discuss and adopt a formal definition to describe a *world model* for *classical planning*. For ease of understanding we adopt the phrase *State* when discussing the composition of a *world model*.

A *State* is comprised of a selection of *Facts* [22]. Each *Fact* represents something that holds true in the environment. Definition 2 provides a formal structure for *Facts* within *Classical Planning*.

Definition 2. Classical Planning Fact

A *fact* is a tuple (N, O) where:

- *N* is the name of the fact
- *O* is an ordered list of objects

Now we have established the concept of *Facts* we can progress to environment *States*. Definition 3 provides the structure of a *State*, which is simply a set of *Facts*. The *Initial State* of a problem is a set of *Facts* given to the planner, this is accompanied by a set of *Objects* which are available to the planner during search.

Definition 3. Classical Planning State

A *State S* is an element of the powerset of F, which is the set of all combinations of all possible *Facts*.

Operators (also known as *Actions*) have the ability to modify a *State* by interacting with its *Facts*. Operators contain three distinct parts as outlined in Definition 5.

Each *parameter* of an *Operator* must be assigned a suitable *Object*. The *Objects* assigned as *parameters* can be use by the *Operator* when interacting with a *State*.

Preconditions are conditions that simply check the *State* for the presence or omission of a specified fact. Only if the *preconditions* hold true can the *Operator* be applied. Definition 4 outlines the structure of preconditions.

Definition 4. Operator Preconditions

The *preconditions* of an *Operator* are split between two types. The first type is a positive condition (pre^{+}) which checks for the presence of a *Fact* in the *State*.

The second type of condition is a negative condition (pre^{-}) which checks that a *Fact* is not in the *State*.

A set of conditions C is determined to be satisfied is all conditions in the set hold true with respect to a specific *State*.

The set of *effects* belonging to an operator specify which *Facts* need to be added and removed from the *State* upon execution. Definition 6 outlines the process of applying an operator to an environment state.

Definition 5. Classical Planning Operator
An *Operator A* can be considered a tuple $A = (P, C, E)$ where:

- P is a list of parameters required by the operator
- C is a set of preconditions which must hold in the environment state before the operator can be applied
- E is a set of effects of types E_+ and E_- where:
    - E_+ signifies a positive effect which introduces new facts to the environment
    - E_- represents a negative effect which removes existing facts from the environment

Definition 6. Classical Planning Application of Operator
A given *State* S is an element of the powerset F — as defined in Definition 3.
S can be transformed into some successor *State* $\bar{S}$ where $\bar{S}$ is also an element of F.
$\bar{S}$ encompasses all of the positive and negative effects that O puts on S.
$S = \bar{S}$ if $O^{effects} = \emptyset$.

STRIPS also introduces the concept of *Goal Conditions* which must be satisfied before a *State* can be considered completed [22]. We refer to *States* which satisfy all *Goal Conditions* as *Goal States*. *Goal Conditions* can assert that a *Fact* must or must not exist in a *Goal State* [1]. Definition 7 outlines the structure of *Goal Conditions* in a *Classical Planning*.

Definition 7. Classical Planning Goal Conditions
The *Goal Conditions* can be defined as the set *GC* where:

- Each element in *GC* is either a positive stipulation S_+ or a negative stipulation S_-
- S_+ denotes a fact F which must be present in a *Goal State*
- S_- denotes a fact which must not be present in a *Goal State*
- *GC* is a non-empty finite set
- Every stipulation S in *GC* must be satisfied by the *Goal State*

A *Classical Planning* problem can be summarised as finding an ordered list of *Operators* which modify the *Initial State* to a *Goal State*. Definition 8 defines this concept.

Definition 8. Classical Planning Output

Given an *Initial State I* and a *State S* which satisfies all *Goal Conditions GC* the output O can be returned where:

- O is an ordered list of *Operators*
- Following each $o \in O$ leads from I to S

2.1.2 PDDL

Planning Domain Definition Language (PDDL) [1] is used in the *Classical Planning Track* of the *IPC* - discussed in Section 2.2. As such it is the *Classical Planning* language that we explore to provide a deeper understanding of such problems.

Understanding the syntax of *PDDL* is essential since the language *HDDL* (Hierarchical Domain Definition Language) follows the same syntax style. In this section we study the syntactical layout of *PDDL* and examine how it refers to the definitions previously put forth in Section 2.1.1. The examples we see are given by the IPC.

PDDL tasks are split into two files, the *domain file* and the *problem file*. The *domain file* defines the components which can be used to construct a problem space, *types* (Section 2.1.2.1), *predicates* (Section 2.1.2.3), and *Actions* (Section 2.1.2.4) are defined in the *domain file*. The *problem file* formulates a problem that the planner needs to solve, this file contains the *Initial State* and *Goal Conditions* (Section 2.1.2.5) and *Objects* (Definition 1). Multiple *problem files* can correspond to a single *domain file*.

2.1.2.1 Type

In this section we discuss *types* in *PDDL* by considering how they are used and their definition. In *PDDL types* play the role of classifying *Objects* while simultaneously acting as a requirement for *parameters* to satisfy in *predicates* and *Actions*. Definition 9 introduces the structure of *types*. When determining if an *Object* satisfies a type we must also check its *parent type*. This cycle can continue until we either find the *type* required or come across a root *type* — a type with no *parent type*.

Definition 9. PDDL Type

A type can be defined as a tuple (T,P) where:

- T is the name of the type
- P is the parent type of the type

Listing 2.1 shows an example of *type* definition in *PDDL*, in this example we have four defined types to consider. The first *type* to consider is the *object* type which is defined as the parent type of *locatable* - *object* is a *root type* as it has no *parent type*. Next we have the types *vehicle* and *package* which both possess the parent type *locatable*. Further considering this example, an *object* of type *vehicle* would be able to satisfy three *types*: *vehicle*, *locatable*, and *object*.

```
    (:types
        locatable - object
        vehicle package - locatable
    )
```

Listing 2.1: PDDL Type Definition

2.1.2.2 Object

Listing 2.2 shows an example of *object* definition in PDDL. In this example two *objects* are being described - *truck-1* which is of the *type vehicle* and *package-1* which is of *type package*.

```
(:objects
  truck-1 - vehicle
  package-1 - package
)
```

Listing 2.2: PDDL Object Definition

2.1.2.3 Predicates and Facts

In this section we discuss the difference between a *predicate* and a *Fact*, then cover how they are related and work together.

Definition 2 puts forth the definition of a *Fact*, this definition states that a *Fact* contains *objects*. *Predicates* do not assign *objects* instead, they act as a formula for facts to be build upon as defined in Definition 10.

Definition 10. PDDL Predicate

A *predicate* can be defined as a tuple (N,P) where:

- N is the name of the predicate
- P is an ordered list of parameters which are required by the predicate
- Each $p \in P$ can be described as a tuple (p_N, p_T) where:
  - p_N is the name of the parameter
  - p_T is the *type* of the parameter

Listing 2.3 gives an example of *predicate* definition in *PDDL*, in this example two *predicates* are defined. The first *predicate* contains the name *road* and two parameters - *?l1* and *?l2* - which share the same *type* of *location*. The other *predicate* detailed has the name *at* and contains two parameters of different *types*.

```
(:predicates
     (road ?l1 ?l2 - location)
     (at ?x - locatable ?v - location)
)
```

Listing 2.3: PDDL Predicate Definition

A *Fact* can be described as a *predicate* which has *Objects* assigned. Listing 2.4 shows an example of a *Fact* definition using the *predicate* provided in Listing 2.3.

```
(at package-1 city-loc-4)
```

Listing 2.4: PDDL Fact Definition

2.1.2.4 Action

In this section we explore how *PDDL* makes use of *Actions* which were defined in Definition 5. Listing 2.5 exhibits an example of an *Action* being defined in *PDDL*. In this example we have three parameters - *?v*, *?l1* and *?l2* - which are split between two types.

The *preconditions* set checks the *State* for two properties. The first property is the assertion that the fact *(at ?v ?l1)* - where *?v* and *?l1* is replaced by the *objects* given as parameters - is present in the *State*. Beyond the conditions shown in this example are conditions which affirm the omission of a fact from the *State*. An example of such a condition is *(not (road ?l1 ?l2))*.

Finally the *effects* of the *Action* shown contains a positive effect and a negative effect. The positive effect *(at ?v ?l2)* adds a new *Fact* to the *State* whereas, the negative effect *(not (at ?v ?l1))* removes a *Fact* from the *State*.

```
(:action drive
    :parameters (?v - vehicle ?l1 ?l2 - location)
    :precondition (and
        (at ?v ?l1)
        (road ?l1 ?l2)
      )
    :effect (and
        (not (at ?v ?l1))
        (at ?v ?l2)
      )
)
```

Listing 2.5: PDDL Action Definition

2.1.2.5 Initial and Goal State

The *Initial State* (*IS*) of a problem follows the structure previously outlined in Definition 3. Listing 2.6 shows an example of an *Initial State* being defined with three *Facts*.

```
(:init
  (road city-loc-1 city-loc-4)
  (at package-1 city-loc-4)
  (at truck-1 city-loc-4)
)
```

Listing 2.6: PDDL Initial State Definition

The attributes of a *Goal State* are defined in Definition 7. In this section we see an example of the definition of *Goal Conditions* in Listing 2.7. This example contains two facts - *(at package-1 city-loc-5)* and *(at package-2 city-loc-2)* - which must be present in a *State* for it to be considered a *Goal State*.

```
(:goal (and
  (at package-1 city-loc-5)
  (at package-2 city-loc-2)
))
```

Listing 2.7: PDDL Goal Condition Definition

2.1.3 Hierarchical Planning

There are multiple types of hierarchical planning - TI-HTN [3], HTN planning with resources and temporal constraints [57], CC-HTN [32]. In this project we operate using the standard HTN planning problems. Some existing works have been produced which blueprint the concepts of HTN planning [27]. In this section we analyse and discuss the components which create HTN-planning problems.

Primitive Tasks are the same as *Actions* in *Classical Planning* which we explored in Section 2.1.2.4. *Primitive Tasks* are the only properties of *HTN Planning* able to interact with a *State*.

Abstract Tasks (also known as *Tasks*) are unique to *HTN Planning*, these tasks do not modify the *State* but instead must be expanded using a *Decomposition Method*. Definition 11 illustrates a framework for *Tasks* containing two attributes, a name and a list of parameters.

Definition 11. Abstract Task

An *Abstract Task* T can be considered a tuple (N,P,M) where:

- N is the name given to the abstract task
- P is the list of parameters required by the abstract task
- M is a mapping to the set of *Methods* which decompose T

Decomposition Methods (also known as *Methods*) are unique to *HTN Planning* and are formalised by Definition 12. This definition contains a few core elements. The *Abstract Task* decomposed by the *Method* along with the parameters inherited from the *Task*. The subtasks of a *Method* can have either a strict total ordering or a partial ordering, a partial ordering sets out some rules which the ordering must follow but the planner can choose other aspects of the ordering. For example *task 1* must come before *task 4* but *tasks 2* and *3* are free to be ordered anywhere. Lastly, the parameter constraints of a decomposition method set out which relationships between parameters are permitted such as, *parameter A* cannot be the same as *parameter B*.

Definition 12. Decomposition Method

A *Method* in HTN planning is a complex component which can be formalised as the tuple (N,P,T,C,S,O,PC) where:

- N is the name assigned to the *Method*
- P is parameters required by the *Method*
- $p \in P$ is a tuple of the form (p_I, p_T) where:
  - p_I is the identifier of the parameter
  - p_T is the type that must be satisfied by a potential parameter
- T is a mapping to the abstract task that the *Method* decomposes
  - T is a tuple of the form (T_{NAME}, T_P)
  - T_{NAME} is the name of the abstract task
  - $T_P \subseteq P$; T_P is the parameters inherited from the abstract task
- C is the preconditions that must be satisfied before the *Method* can be executed
- S is a collection of subtasks to be added to the task network
- Each $s \in S$ is a tuple of the form (s_{NAME}, s_P) where:
  - s_{NAME} is a mapping to an abstract or primitive task
  - $s_P \subseteq P$; s_P is a list of parameters to be passed from the *Method* to the subtask
- O is an ordering over S
- PC is parameter constraints that must be satisfied by parameters provided to the *Method*

HTN planners are constrained by the *Task Network* with regards to what operations can be expanded. Based on an existing formalism [28] Definition 13 outlines the properties of a *Task Network*. We can generalise the behaviour of a *Task Network* to that of a queue although, during search operators can be added to any location of the queue not necessarily the end.

Definition 13. HTN Task Network

A *Task Network* in HTN planning can be represented by the tuple $(O, \prec)$ where:

- O is a list of operators
- $o \in O$ is a tuple (o_N, o_P) where:
  - o_N is a mapping to a *Task*, *Action*, or *Method*
  - o_P is a list of parameters to be passed to o_N
- $\prec$ is a strict ordering over O which dictates which order the operators must be expanded by the planner

Previously in Chapter 1 we discussed that the goal of *HTN Planning* problems builds upon that of *Classical Planning* since the *Task Network* must not contain any not expanded *Tasks*, *Methods*, or *Actions*.

The exact format of output can differ for depending on what the output is to be used for. For example, when verifying the plan output from the planner using the *IPC's* plan verification module the plan verification output format covered in Section 2.2.1 is required.

Definition 14 formalises all of the components stored as part of a *HTN Planning Problem*. All components contained in this definition are discussed previously in this chapter.

Definition 14. HTN Planning Problem

A HTN Planning Problem P can be described as the tuple $(A, M, T, O, IS, OT, TN_I)$ where:

- A is the set of *Actions* defined
- M is the set of *Decomposition Methods* defined
- T is the set of *Tasks* defined
- O is the set of defined *Objects*
- IS is the defined *Initial State*
- OT is the set of defined *Types*
- TN_I is the *Initial Task Network*

2.1.4 HDDL

Hierarchical Domain Definition Language (HDDL) [36, 37] is an extension to PDDL for hierarchical planning problems. *HDDL* includes the definition of the *Initial Task Network* in the *Problem File* mentioned in Section 2.1.2.

HDDL is the input language used for the hierarchical planning tracks of the *International Planning Competition* discussed in Section 2.2. As such *HDDL* is the language we use in this project since it is standardised by the *IPC*. In this section we explore the syntactical definitions used by *HDDL* to define properties which are not present in *PDDL*.

2.1.4.1 Abstract Task

In Definition 11 we defined the role of *Abstract Tasks*, in this section we see the syntactical layout of *Abstract Tasks*. Listing 2.8 lays forth an example of an *Abstract Tasks* being defined in HDDL. In this particular example, the *Abstract Tasks* is assigned the name *get_to* and requires the parameters *?v* and *?l*.

```
(:task get_to
    :parameters (?v - vehicle ?l - location)
)
```

Listing 2.8: HDDL Abstract Task Definition

2.1.4.2 Decomposition Method

In this section we observe the definition of *Decomposition Methods* in *HDDL*, which we previously defined in Definition 12. Listing 2.9 illustrates an example of a *Decomposition Methods* definition. This *Method* is named *m_drive_to_via_ordering_0* and decomposes the *Task get_to* and inherits the parameters *?v* and *?l3* from the *Task*. These inherited parameters are accompanied by another parameter - *?l2* - which is selected by the planner.

The *Method* has two subtasks, *get_to* and *drive* with an imposed ordering that *task0* (*get_to*) must come before *task1* (*drive*). In cases that an ordering is not specifically defined in the method then the ordering is set as the order of definition.

The final aspect of the *Method* shown is the constraints defined for its parameters. These constraints ensure that the parameters *?l2* and *?l3* are not the same object.

```
(:method m_drive_to_via_ordering_0
    :parameters (?l2 - location ?l3 - location ?v - vehicle)
    :task (get_to ?v ?l3)
    :subtasks (and
        (task0 (get_to ?v ?l2))
        (task1 (drive ?v ?l2 ?l3))
    )
    :ordering (and
        (< task0 task1)
    )
    :constraints (and
                    (not (= ?l2 ? l3))
    )
)
```

Listing 2.9: HDDL Decomposition Method Definition

2.1.4.3 Constants

Constants take the role of an *object* which is defined in the *domain file* of a problem. This ensures that the *object* is present in every problem that utilises the domain. Listing 2.10 shows the definition of an constant in HDDL from an IPC test problem. This example simply defines an object *a* which is of type *A*.

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

Listing 2.10: HDDL Constant Definition

2.2 International Planning Competition

In this section we discuss the International Planning Competition (IPC) [64] which is an open competition where planners compete in a variety of different classifications sometimes known as *tracks*. *The First International Planning Competition* was held in 1998 as part of the *AI Planning Systems Conference* and set out with the intention of introducing common problems to allow for a standardised method of evaluating planning systems. The agreed upon standard became known as *PDDL* and is still used today. This agreement resulted in the first *IPC* evaluating five classical planners, despite best intentions hierarchical planning was not part of the first IPC since no planners entered the contest [51].

The first use of hierarchical planning took place in the second IPC in 2000. At the time, hierarchical planning was thought to simply be a means of giving a planner supplementary information in hopes that it find a plan faster. No attempt was made to properly introduce hierarchical planning at an IPC since the third IPC which was held in 2002. A key progression in hierarchical planning has been that it is no longer only seen as providing a planner more information but as a different way of expressing parts of planning domains. The most recent IPC was *The 2020 IPC for Hierarchical Planning* which introduced a standardised track for HTN planners to compete in. This competition also selects *HDDL* as a common input language [4]. This IPC consisted to two tracks, one for totally ordered problems and the other for partially ordered problems [5].

A future IPC — *The International Planning Competition 2023* — is due to be held in June 2023 and is comprised of five classes of track, these are: Classical Tracks, Learning Tracks, Probabilistic Tracks, Numeric Tracks, and HTN Tracks[1]. Within the HTN Tracks six sub tracks are proposed these are: Total-Order Agile, Total-Order Satisficing, Total-Order Optimal, Partial-Order Agile, Partial-Order Satisficing, and Partial-Order Optimal[2]. For classical tracks the IPC uses *PDDL*[1] as a standardised input language, since 2011 *PDDL* version 3.1 has been used in IPC contests. As part of the IPC the details of *PDDL* version 3.1 have been expressed in Backus Normal Form (BNF) [45].

The IPC has published the problems used to evaluate planners for the 2020 competition as well as a set of problems which each test a specific aspect of HTN planning[3]. These problems are what we use throughout this project to evaluate the performance of search strategies.

2.2.1 IPC Verifying Output

The 2020 IPC for Hierarchical Planning utilised a HTN plan verification module to verify the output yielded by planners. This verification module part of the *PANDA* planner's *pandaPIparser* [35] module. This module can be used to verify plans when it is instigated using '-verify [DOMAIN] [PROBLEM] [PLAN]' command line parameters. The plan format requirements for this module have been explicitly laid forth [4].

[1] https://ipc2023-htn.github.io/

[2] https://ipc2023-htn.github.io/

[3] https://ipc2020.hierarchical-task.net/results/status-page

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

The plan verification module is quite lenient with what can be output along side the formatted plan, with the verification module searching for the string ‘==>‘ in the output as a marker of where the plan begins. Any characters prior to this string are ignored by the module.

The required plan format can be thought of as containing three sections. The first section details the *Actions* (*Primitive Tasks*) carried out by the planner. These *Actions* must each be on a new line and contain three pieces of information. The first thing that needs to be present is a numeric ID, this is followed by the *Action* name, which is in turn followed by the parameters used by the planner. Definition 15 puts forth a formal definition of the *Action* section of the required plan output format.

Definition 15. IPC Plan Verification Format - Actions (Primitive Tasks)

Each *Action* of the plan takes the form: $[ID][ActionName][Parameters]$ where:

- $[ID]$ is a unique non-negative integer
- $[ActionName]$ is the name of the *Action*
- $[Parameters]$ is a list of size $0 \geq$ consisting of parameters which are the names of objects in the problem that were used as parameters to the specified *Action*
- Each ID must be unique across the entire plan not solely the *Action* section of the plan.
- The length of the *Parameters* list must be the same length as the amount of parameters the specified *Action* accepts.

The second section of the plan output format is the stipulation of which *Tasks* (*Abstract* or *Primitive*) are root tasks. A root task is one which was given to the planner from the problem specification. This section consists of only one line which begins with the string ’root’, followed by a list of id’s which are found in the first of last sections of the plan format. Definition 16 details the formal definition of this section.

Definition 16. IPC Plan Verification Format - Root Tasks

The root tasks of the given problem are specified in the form:
root *[ID’s of Root Tasks]*
where:

- *[ID’s of Root Tasks]* is a list of id’s that are present in either of the other sections of the plan.

The final section of the required plan format is a list of *Tasks* (*Abstract Tasks*) that were decomposed by the planner. Each *Task* needs to be specified on a new line beginning with an ID, followed by the name of the *Task* along with the list of parameters. Following on is the ’->’ string along with the *Method* that was used to decompose the specified *Task* as well as, the IDs of the subtasks added by the *Method*. Definition 17 provides a formal definition of the *Task* section of the required plan output format. To round off the plan, the verification module looks for the ’<==’ string in the output to signify the end of the plan.

Definition 17. IPC Plan Verification Format - Tasks (Abstract Tasks)

Each *Task* of the plan takes the form:
[ID] [Task Name] [Parameters] -> [Method Name]
where:

- *[ID]* is a unique non-negative integer
- *[Task Name]* is a the name of the *Task*
- *[Parameters]* is a list of size $0 \geq$ consisting of parameters which are the names of objects in the problem that were used as parameters to the specified *Task*
- *[Method Name]* is the name of the *Method* used by the planner to decompose the specified *Task*
- Each ID must be unique across the entire plan not solely the *Task* section of the plan.
- The length of the *Parameters* list must be the same length as the number of parameters the specified *Task* accepts.

Figure 2.1 produces an example of the plan verification format on a simple problem from the *Basic* domain.

```
==>
1 drop kiwi
2 pickup banjo
root 0
0 swap banjo kiwi -> have_second 1 2
<==
```

Figure 2.1: Plan Verify Format - Basic Domain Example

2.2.2 IPC Scoring

The 2020 Hierarchical IPC ranked planners following a simple set of rules. During evaluation of planners, each problem is assigned 30 minutes (1800 seconds) to solve the problem otherwise the problem is deemed unsolved. The following structure highlights the scoring principles used [5]:

1. If the problem is solved in less than one second then a score of 1 is assigned to the problem
2. Other wise the minimum of the following is assigned:
   (a) $1 - \frac{log(t)}{log(1800)}$
   (b) 1

This scoring structure provides a way to differentiate between two planners which are able to solve the same problems. Solving the problems quicker yields a higher score and makes it clear which planner is better in terms of performance.

Chapter 3

Background on Search Strategies

3.1 Search Algorithms

In this section we discuss the key search strategies for managing and executing search-driven problems in the context of automated planning. Algorithm 1 shows a standard *Graph Search* algorithm which can be used for searching with the strategies covered in this section.

Algorithm 1 Graph Search

```
function GRAPHSEARCH(problem $p$, strategy $s$)
    $closed \leftarrow \{\}$                              ▷ We now keep a list of explored states
    $frontier$.ADD($newNode(p.initial)$)
    loop
        if $frontier$ is empty then
            return fail
        $n \leftarrow s$.REMOVECHOICE($frontier$)
        $closed$.ADD($n.state$)                          ▷ Where we keep states we already visited
        if P.GOALTEST($n.state$) then
            return GETPATH($n$)
        for all $a \in p$.ACTIONS($n$) do
            $n' \leftarrow a$.RESULT($n.state$)
            if $n'.state \notin closed$ then
                $frontier$.ADD($n'$)                     ▷ And only explore states we haven't visited
```

3.1.1 Greedy Best First Search (GBFS)

Greedy Best First Search (GBFS) is a method of ordering search nodes based upon a heuristic estimate such that the algorithm expands nodes with the lowest heuristic estimate first. This ignores the cost of all predecessor nodes belonging to each node. Optimality is not guaranteed by this search method since the first goal found is usually returned. This strategy has the limitation of being susceptible to being trapped in an endless loop where, one node is continuously expanded but makes no progress.

Figure 3.1 illustrates the search process through a graph using the *Greedy Best First* search strategy. The numbers at the top of each node state which iteration the node was discovered in. The number contained within the dotted box is the heuristic estimate assigned to the node. This diagram omits path costs since this strategy relies solely on the heuristic estimates.

The *Graph Search* procedure shown in Algorithm 1 can search using *GBFS* by setting the behaviour of the function called on Line 7. If this function returns the node with the lowest heuristic estimate then the search will proceed in a GBFS manner.

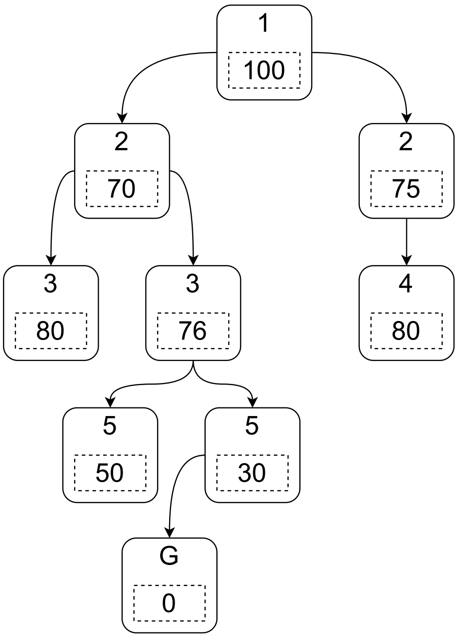


Figure 3.1: Greedy Best First Search Example

3.1.2 A* (A-Star) Search

*A** search builds upon *Greedy Best First Search* seen in Section 3.1.1 by also taking the cost of the node's predecessors into consideration. *GBFS* uses **the estimate of the remaining cost**, whereas *A** uses the known cost so far, plus the estimate of the cost to go, making up an **estimate of the total path cost**. This addition enables this strategy to find optimal solutions [30]. *A** has some performance based drawbacks, the solve time and memory requirments are larger than *GBFS* since more nodes are explored.

When compared to *GBFS* this method has the advantage of not getting stuck in an endless search cycle, which can occur when using *GBFS* when a particular node is always ranked as the node closed to goal but never makes any progress. Again considering Algorithm 1, we can search using *A** if the function called on Line 7 returns the node with the score when calculate using the formula: *Total Predecessor Cost + Heuristic Estimate*.

Figure 3.2 shows an example of search using the *A** strategy following the same graph problem introduced in Section 3.1.1. The nodes in this diagram follow the same design principles laid forth in Section 3.1.1 but with two additions. The first addition is the inclusion of path costs which can been seen on each edge of the graph. Secondly the score assigned to each node within the dotted box is shown as heuristic estimate + path cost = total cost. When we compare this figure to Figure 3.1 we can see that the *A** strategy makes more iterations and needs to store more nodes when compared to the *GBFS* counterpart. *A** is able to find the optimal path, although the goal is first found in iteration *5* search continues to iteration *7* since some nodes possessed a lower path cost than the first solution found.

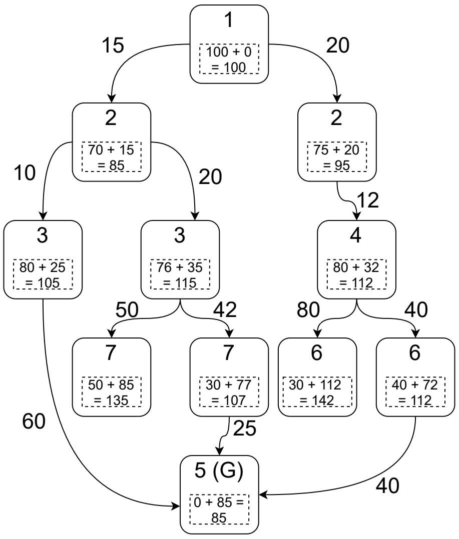


Figure 3.2: A-Star Search Example

3.1.3 WA* (Weighted A-Star) Search

A strategy known as *Weighted A** search ameliorates the performance issues of $A*$ raised in Section 3.1.2. By giving more weighting to the heuristic estimate with the aim of limiting the amount of nodes searched. This removes the promise of optimality from the standard $A*$ strategy but promises bounded suboptimality [16].

Algorithm 1 can achieve this method or ordering if the function called on Line 7 returns the node with the lowest score when nodes are ordered using the formula: *Total Predecessor Cost* + ($W \times HeuristicEstimate$). Where W is a value larger than 1, increasing W gives more priority to the heuristic estimate.

3.1.4 Total-Order Forward Decomposition (TFD) & Partial-Order Forward Decomposition (PFD)

Total-Order Forward Decomposition (TFD) and *Partial-Order Forward Decomposition (PFD)* are algorithms designed for total-ordered and partially-ordered HTN planning problems respectively. These algorithms use recursion to search in a depth-first manner. When selecting options during search, these algorithms only aim to consider only operators which are relevant to what the planner is attempting to achieve [28]. The recursive nature of these algorithms mean that there is no defined data-structure to contain all options during search, instead they are attempted iteratively. Algorithm 2 shows the pseudocode of this strategy.

3.2 Classical Planning Heuristics

In this section we discuss two prominent *Classical Planning* heuristics and analyse ways in which they can be adapted for HTN planning problems.

3.2.1 Delete Relaxed

Delete Relaxed heuristics relax a planning problem by creating an alternate problem with the delete effects of every *Action* ignored. Within this relaxed problem the cost of adding a specific *Fact* F to the *Initial State* S can be computed by adding the cost of an *Action* A which adds F, to the sum of adding all *Facts* in the preconditions of A. When done recursively across all required

Algorithm 2 Total-Order Forward Decomposition

```
function TFD($s$, $\langle t_1,\ldots,t_k\rangle$, $O$, $M$)
    if $k = 0$ then return $\langle\rangle$                ▷ (i.e. the empty plan)
    if $t_1$ is primitive then
        $active \leftarrow \{\langle a,\sigma\rangle | a \in O, a = t_1 \cdot \sigma,$ and $s \models pre(a)\}$
                ▷ Match applicable ground instances of actions to the task
        if $active = \emptyset$ then return $failure$
        nondeterministically choose any $\langle a,\sigma\rangle \in$ active
        $\pi \leftarrow$ TFD($\gamma(s,a), \langle t_2,\ldots,t_k\rangle\sigma, O, M$)
        if $\pi = failure$ then return $failure$
        else return $a \cdot \pi$
    else if $t_1$ is nonprimitive then
        $active \leftarrow \{(m,\sigma) | m \in M, m = t_1\sigma,$ and $s \models pre(m)\}$
                ▷ Match applicable ground instances of methods to the task
        if $active = \emptyset$ then return $failure$
        nondeterministically choose any $\langle m,\sigma\rangle \in active$
        $w = tn(m) \cdot \langle t_2,\ldots,t_k\rangle\sigma$
        return TFD($s, w, O, M$)
```

Actions, this provides a path from S to a *State* with F present. Where there is multiple paths for adding F to the *State* all but the shortest are discard [13]. The following formula formalises this heuristic, where O is the set of operators which add F to the *State* and, Q is a precondition of action A.

$$h(F,S) = \begin{cases} 0 & \text{if } F \in S \\ \min\limits_{A \in O}\{cost(A) + \sum\limits_{Q \in pre(A)} h(Q,S)\} & \text{otherwise} \end{cases} \tag{3.1}$$

Using a relaxed problem yields two heuristics - h_{add} and h_{max}. h_{add} is an additive heuristic where the cost of all *Facts* in the *Goal Conditions* being added to the state is accumulated. The h_{max} heuristic returns only the maximum estimate as the heuristic estimate for the entire set of *Facts* [10].

Work has been established on the basis of adapting *Delete Relaxed* heuristics for HTN planning problems. Within the scope of HTN planning the *Delete and Ordering Relaxed* heuristic estimates the distance until a task network is fully decomposed using the relaxed problem. A Relaxed problem also ignores task orderings [39].

3.2.2 Landmarks

Landmarks heuristics guide search by identifying key elements which every *Goal State* must have or encounter. In *Classical Planning*, landmarks are *Actions* and *Facts* which must be executed or present in the *State* for any solution to a problem.

In *HTN Planning* problems the types of landmarks generated are *Tasks*, *Methods*, and *Facts*. Since goal conditions are not guaranteed to exist in HTN problems, the landmarks must be mainly derived from operators.

The first step of finding landmarks is to generate an *AND / OR* tree where, *Tasks* and *Facts* are *OR* nodes, and *Methods* and *Actions* are *AND* nodes. Landmarks for *Fact* nodes are defined as the *Action* that is able to add the *Fact* to the *State*. For all other nodes, landmarks are inherited from the downward stream of the *AND / OR* tree. The heuristic estimate assigned to a *Model* is the amount of landmarks which are not satisfied [38].

We can see an example of this concept in action in Figure 3.3 which is a modified version of an example provided in a previous work [38]. In this example *T* and *S* are *Tasks*, *m1*, *m2*, and *m3* are *Methods*, and *a* and *b* are *Actions*. Lastly, *x* and *z* are *Facts* with *x* being defined as part of the *Initial State*. This visualisation is accompanied by Figure 3.4 which details the landmark sets for the aforementioned example. This figure also shows the process followed to produce each of the landmark sets.

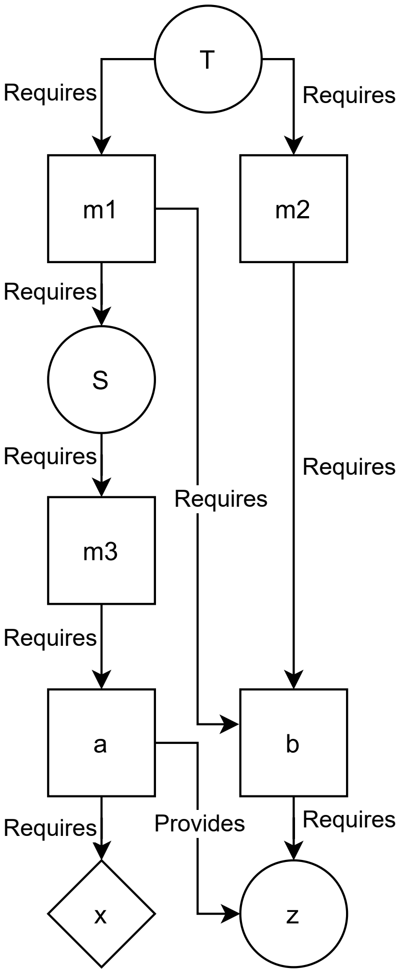


Figure 3.3: Example of Landmark AND/OR Graph

3.3 HTN Planning Heuristics

Task Decomposition Graph (TDG) strategies generate an *AND / OR* graph modelling of the task hierarchy. Each *Task* and *Method* is represented as a vertex. Vertices are connected by edges which denote which *Methods* decompose a *Task*, and the subtasks of each *Method*. A requirement of this structure is that the *Initial Task* is the root of this *AND / OR* [18].

Definition 18 puts forth a calculation for assigning cost estimates to each node in the TDG. TDG_C known as the cost-aware TDG heuristic which works by simply computing the sum of the estimate costs of all *Abstract Tasks* in the *Task Network*. Definition 19 formalises this approach [8].

```
LM(x)   = {x}
LM(a)   = LM(x) ∪ {a}
        = {a, x}
LM(z)   = LM(a) ∪ {z}
        = {a, x, z}
LM(b)   = LM(z) ∪ {b}
        = {a, x, z, b}
LM(m3)  = LM(a) ∪ {m3}
        = {a, x, m3}
LM(S)   = LM(m3) ∪ {S}
        = {a, x, m3, S}
LM(m1)  = LM(S) ∪ LM(b) ∪ {m3}
        = {a, x, m3, S, z, b, m1}
LM(m2)  = LM(b) ∪ {m2}
        = {a, x, z, b, m2}
LM(T)   = LM(m1) ∩ LM(m2) ∪ {T}
        = ({a, x, m3, S, z, b, m1} ∩ {a, x, z, b, m2}) ∪ {T}
        = {a, x, z, b, T}
```

Figure 3.4: Landmark Sets for graph shown in figure 3.3

Definition 18. TDG Estimate Cost Calculation

Considering a *TDG* where:

- *o* is an *Action*, *Task*, or *Method*
- *t* represents a *Task*
- *m* represents a *Method*
- $E_{t\to M}$ signifies the set of edges which connect a *Task t* to any *Method* which decomposes it $E_{m\to T}$ is the set of edges which connect a *Method* to its subtasks

$$C(o) = \begin{cases} cost(o) & \text{if } o \text{ is a primitive task} \\ \min_{m\in E_{t\to M}} C(m) & \text{if } o \text{ is an abstract task} \\ \sum_{t\in E_{m\to T}} C(t) & \text{if } o \text{ is a decomposition method} \end{cases} \tag{3.2}$$

Definition 19. TDG_C Heuristic Calculation

Considering each abstract task (t) in a task network TN.

$$h_{TDG_C}(TN) = \sum_{t\in TN} C(o) \tag{3.3}$$

3.4 Multiple Heuristics

The idea of using more than one heuristic to solve a problem has been explored in conjunction with the use of a *Novelty* heuristic [48]. A second heuristic can be used to break ties between *States* which are assigned in the same *Novelty* score. This can be extended to other heuristics since more than one *State* can be assigned the same heuristic estimate.

3.5 Novelty

The *Novelty* strategy was first brought to the forefront of attention when applied in a planner capable of playing Atari video games [49] using the *Arcade Learning Environment* (ALE) which provides a "platform and methodology for evaluating the development of general domain-independent AI technology" [7]. When introduced the strategy was named *Iterated Width* (IW) or as we will refer to it *Novelty* [47].

The *Novelty* strategy works by pruning states which do not contain a previously unseen fact or combination of facts [49]. Checking for a new *Fact* in the *State* is a trivial operation however, the combination of *Facts* is not. When checking for a combination of *Facts* we need to assign a maximum *level of Novelty* to tell the planner the size for each unique combinations of *Facts*. For example if the maximum *level of Novelty* was set to three then the planner would check for a new *Fact* (Novelty level 1), a new pairing of *Facts* (Novelty level 2), and a new trio of *Facts* (Novelty level 3). *States* are ordered based on their computed novelty scores which lowest being put first, and any *States* which do not satisfy the assigned maximum level of *Novelty* are pruned [49].

Progressing onwards from its introductory application *Novelty* has been applied in standard *Classical Planning* [42]. In this application *States* are separated based on their level of *Novelty* then ordered using a heuristic. In this case non-novel *States* are not pruned but are instead sent to the end of the queue.

Novelty has also shown impressive results when being used to bring an end to a heuristic plateaus, which occurs when a large number of expansions using a Greedy Best First (GBFS) search strategy does not yield a lower heuristic estimate than already seen. Compared with standard GBFS, GBFS-LS (where the default heuristic is used on the *State* as if it were an *Initial State* for a set amount of iterations). GBFS-W (*Novelty*) proves to be a valuable inclusion, since the number of problems solved increases [48].

3.5.1 Planners for Atari Games

All the work which we will analyse in this section took place using the *Arcade Learning Environment* (ALE) mentioned in Section 3.5. The ALE provides groundwork for two types of autonomous agent, known as *online planning* and *learning*. The *online planning* aspect allows for autonomous agents to "look ahead" [49] in the game space whereas, the *learning* mode does not allow this but does allow for the agent to have one long training period with the system [61]. The planners we will discuss in this section make use of the *online planning* setting of the ALE.

Traditionally AI planners operate in static environments as mentioned in Section 2.1. In the application of playing *Atari* games this is not true, since the game changes in ways outwith the player's control. Typically planning problems continue until a goal is found however, in *Atari* games this is not possible. Instead the planner searches for a predefined length of time or number of *Actions*, after this period the best plan found is selected. At this stage the first *Action* of the plan is executed on the game then the planner begins to search again [61]. This adaption allows a planner to operate in a setting which it does not have full control over.

Modifications to the traditional ordering and expansions of *States* has been trialed through the use of *Novelty*. One interesting method is the idea of having two queues to store the states in opposing orders - ascending and descending. Iterations of the planner then alternate between which queue the returned state is taken from [61].

Now that we have discussed how planners undertake the challenge of playing Atari games we can explore the results achieved. We can compare the results between a planner and a *Deep Q-Network* agent, when comparing *Table 1* from the paper *Classical Planning with Simulators: Results on the Atari Video Games* [49] against *Table 2* from paper *Human-level control through deep reinforcement learning* [53] we can see that the *AI Planning* strategy produced better results than a *Deep Q-Network Agent* in the majority of games tested.

A comparison between a planner using the *Novelty* strategy and UCT (Upper Confidence Bounds) [44] strategy are also shown in *Table 1* of the paper *Classical Planning with Simulators: Results on the Atari Video Games* [49]. Comparing these results we find a significantly reduced difference between two than our previous comparison against a deep Q-Network agent. In this experiment the *Novelty* strategy produced better results in 31 of the 54 games tested [49].

Chapter 4

Background & Related Work - Existing Planners

In this chapter we explore the inner workings of a selection of planners and consider the ways in which they solve planning problems.

4.1 EPICpy

The planner which we build upon in this project is called *Extendable Planner with Interchangeable Components in Python* (*EPICpy*) and is detailed in the dissertation *A Multi-Language Heuristic Driven HTN Planner in Python* [52]. *EPICpy* is able to parse and solve problems of the languages, HDDL and JSHOP. In the subsequent subsections we explore some of the key components of *EPICpy*.

4.1.1 EPICpy - Model

The *Model* component of *EPICpy* is used to illustrate the information required and used by the planner during search. The *Task Network* and *State* are both stored along with the steps taken previously during search. During search we can have a huge amount of options to consider, whether it be several distinct parameters or different *Methods*. Each of these unique options is stored in a unique *Model* object, with all *Models* stored in a *Search Queue* as explained in Section 4.1.2.

4.1.2 EPICpy - Interchangeable Components

EPICpy comes with existing functionality for a selection of interchangeable components. These components are designed to be compatible with each other in custom configurations set by the user. *EPICpy* allows for four component types to be interchangeable.

The first type of interchangeable component is the *Solving Algorithm* components. These are is used by the planner when processing what needs to be added to a *Task Network* during execution of a *Method*, the options for decomposing a *Task*, and the interaction between *Actions* the *State*. *EPICpy* contains two such components, a total-order solver and a partial-order solver. In this project we will work solely with the partial-order solver as it has the ability to solve both types of problem.

The second type of interchangeable component is *Heuristics* that are used to estimate the distance a given model is from the goal, we discuss these further in Section 4.1.3.

Next we have *Search Queues* which store and order all *Models* for use by the solving algorithm. The first search queue orders *Models* as *A-Star search* [11] - covered in Section 3.1.2. The second type of search queue is *Greedy Best First* [34] - covered in Section 3.1.1. The final search queue ordering is the *Greedy Cost* ordering which is a hybrid of the first two search queues where, models are ordered based on the cost they have incurred already divided by five and the heuristic estimate which can be formally written as $(Cost\,so\,Far/5) + Heuristic\,Estimate$.

Finally, *Parameter Selection* components are used for choosing objects to be used as parameters. Two such components are provided by *EPICpy*, *AllParameters* and *RequirementSelection*. The *AllParameters* component simply finds all possible combination of objects that satisfy the type of each parameter. Meanwhile, the *RequirementSelection* component returns combinations of objects which satisfy the preconditions of the *Method* as well as the type constraints on parameters.

4.1.3 Heuristics

As mentioned in Section 4.1.2 *EPICpy* comes with three heuristics as standard - Delete Relaxed, Tree Distance, and Hamming Distance. In this section we will take a closer examination of these heuristics.

Delete Relaxed counts the amount of iterations required for the all of the *Tasks* in the *Task Network* to be added to a relaxed planning problem using a bottom up reachability search. This concept is discussed further in Section 3.2.1.

Tree Distance assigns each *Task*, *Method*, and *Action* a score which represents the lowest possible number of expansions until full decomposition of the *Task Network*. The cost of expanding an *Action* is 1. The cost of expanding a *Method* is 1 plus the cost of expanding all of its subtasks. The cost of expanding a *Task* is 1 plus the lowest cost of all *Methods* which can be used to decompose it. This follows the *TDG* concept previously introduced in Section 3.3.

Lastly, Hamming Distance simply evaluates a *State* by counting the amount of *Goal Conditions* that are not satisfied.

4.2 FAPE

Flexible Acting and Planning Environment (FAPE) [15] is an HTN planner specifically designed to operate in the robotics space. This planner follows a recursive search procure. When provided with a search node the planner begins by identifying the ‘flaws‘ of the node, these are defined as the *Goal Conditions* which are yet to be satisfied and undecomposed *Actions*.

When attempting to resolve a *flaw* of an unsatisfied *Goal Condition*, the planner looks for some *Actions* which will satisfy the condition. Resolving an undecomposed *Action* is simple, the planner follows the traditional HTN methodology of applying a *Method*. When a move to resolve a *flaw* is carried out by the planner, the resulting search node is used as part of a recursive call. If the applied resolving technique is unable to solve the problem, then a failure response is returned by the recursive call, at which stage the planner tries another resolving technique.

4.3 SIADEX

SIADEX does not use heuristics to inform its search, instead it infers advice from the problem. *SIADEX* uses *HPDL* [29] as opposed to *HDDL*, to accommodate this change SIADEX translates a *HDDL* problem into a *HPDL* problem using a translator component. *SIADEX* has shown poor results in domains which feature recursive tasks, it is hypothesised that the planner is unable to effectively infer advice from such domains [21].

SIADEX makes use of a pair of inference methodologies to reason with the environment. The first is the use of 'deductive inference' which allows the planner to assert or retract *Facts* of the *State*. Such inferences take the form of a two element tuple where, the first element sets out some condition of the *State*, and the second provides some consequence for when the condition holds. The consequences outline what *Facts* should be added or removed from the *State*. The second type of inference methodologies is that of 'abductive inference' which provides the planner with functionality to deem a condition true in the current *State* although not explicitly evaluated as true. This is done by defining an additional expression which when true infers that the original condition should also be true [12].

Beyond the scope of academic research and testing *SIADEX* is used for planning how to fight forest fires. When used for this purpose *SIADEX* assess' the environment and returns a plan, it then continues to observe the environment before making a revision for an updated plan as the fire progresses. Plans returned by the planner for this purpose contain actions which are paired with timestamps indicating when the corresponding action should take place [14].

4.4 LiloTane

Lilotane (Lifted Logic for Task Networks) is a HTN planner which operates using 'layers'. Beginning from the initial layer which is derived from the problem specification. The planner constructs the successor of a layer by applying a *Method* or *Action* to a node in the layer. Each application of an operation yields one or more nodes in the successor layer. When returning a plan the planner selects one node from the collection of successors to return as part of the plan.

Lilotane aims to only to only consider operators that are required to satisfy a subtask as opposed to all possible operations. To achieve this the planner calculates a set of positive and negative *Facts* which are possible based upon 'adding any direct or indirect effects'. This sets are then used to prune operations which cannot have their precondition satisfied. The planner also extends the precondition of operations to include that of the subtasks which aids in pruning operations which are not currently viable or not viable at a future *State*. A similar exercise is carried out on the effects of operations [59].

4.5 PyHiPOP

pyHiPOP (Hierarchical Partial-Order Planner) is a HTN planner written in Python. The planner operates with the use of heuristics, the planner calculates a few pieces of information which can be used to inform the planner during search. All possible *Facts*, *Facts* which can not be modified by any operator, operator with parameters that can not be achieved, minimum cost of decomposing a *Task*, and the maximum cost of a decomposition [46].

PyHiPOP uses the h^{add} [65] covered in Section 3.2.1. This estimate in conjunction with the maximum possible value of h^{add} (h^{max}_{add}) along with an estimate of the cost of *Actions* in the plan is used to guide the planner [46].

4.6 PDDL4J

PDDL4J (Planning Domain Description Library for Java) is a library of tools for planning in *PDDL* and *HDDL*. *PDDL4J* contains heuristics belonging to four families, *Delete Relaxation*, *Critical Path*, *Abstraction*, and *Landmarks*. Critical path heuristics estimate the cost of deriving propositions using the most costly subset of propositions [55]. Delete relaxation heuristics are covered in Section 3.2.1.

PDDL4J contains two *HTN* planners which have entered the IPC, *TFD* (Totally Ordered Fast Downward) and *PFD* (Partially Ordered Fast Downward). These planners operate in the same fashion but TFD assumes that tasks are totally ordered while PFD handles partial ordering of tasks. The planning algorithm handles *abstract* and *primitive tasks* differently. If the task is primitive the planner attempts each combination of parameters individually until a valid one is found, then the algorithm is called recursively on the new *State*. This continues until a *Goal State* is found or no progression is available. *Abstract tasks* are tackled by attempting each valid decomposition individually in the same recursive fashion as *primitive tasks* [56].

4.7 The PANDA Planner

PANDA (*Planning and Acting in a Network Decomposition Architecture*) is a famed *HTN* planner. External to solving HTN problems, *PANDA* able to verify plans output for HTN planning problems as covered in Section 2.2.1.

PANDA grounds problems using its *pandaPI* component which utilises a three step procedure. First the problem is simplified, then a delete-relaxed reachability function is called, followed by a hierarchical reachability function. In order to produce the smallest possible grounding set the two reachability functions are called repeatedly until the size of the grounding remains the same [6].

PANDA relies on heuristics to guide its search, with early versions of the planner making use of landmark based heuristics which are explored in Section 3.2.2. More recent heuristics added to *PANDA* utilise the *Task Decomposition Graph* (TDG) covered in Section 3.3 [35].

Chapter 5

Design & Requirements

In this section we cover the design and requirements of the software written for this project. We begin by outlining and discussing the requirements of this project's scope then progress to exploring software design.

5.1 Requirements

In this section we introduce the requirements of the software we developed for this project. We begin with the functional requirements then proceed to the non-functional requirements.

5.1.1 Functional Requirements

1. **Implement Support for the IPC's Plan Verify Functionality**

   The *IPC* has developed capabilities for HTN plans to be verified, we discuss the required format in Section 2.2.1. This project should include functionality that extends *EPICpy* to yield output in the format required for plan verification.

2. **Optimise the planner to solve problems quicker**

   Refactoring the existing code base to introduce optimisations which result in quicker solving times.

3. **Implement the Novelty search strategy**

   In Section 3.5 we learned of the *Novelty* search strategy, adapting this to suit HTN planning problems is the flagship piece of technical work to be carried out in this project.

4. **Create code to autonomously evaluate search strategies for IPC problems**

   Since we aim to improve solve times for HTN planning problems through the use of optimisations or search strategies. We need a way to autonomously run problems and report results for all search strategies.

5.1.2 Non-Functional Requirements

1. **Clear and Well Documented Code**

   Developed code should be written with future maintenance and modifications in mind. Code should be neatly structured and consist of comments to document the features implemented.

2. **Handle Errors Appropriately**

At some stages of execution some errors may arise in situations where it is possible for the system to recover, we should endeavour to provide such recovery methods where possible. In the event of an error which we cannot recover from we should provide clear error messaging.

3. **Build upon EPICpy in the manner it was intended for**

   EPICpy is built upon the principle of interchangeable components, in this work we should keep this in mind and only implement components which can be interchanged or create new frameworks for other component types to be interchangeable. Ideally all added software should refrain from modifying the architecture of the planner unless necessary.

4. **Document the findings of using the Novelty search strategy for HTN planning problems**

   We should provide detailed analysis on the decisions made during the adaption of the *Novelty* search strategy to HTN planning problems and detail the results found when using the strategy.

5. **Verify the correctness of plans**

   We should verify the correctness of the planner when using each search strategy.

5.2 Risk Assessment

The goals previously outlined for this project have the natural constriction of the project deadline (May 5th 2023). We lay forth the risks and mitigating factors applicable to this project in this section.

This project has ambitions to implement a *Classical Planning* strategy in an HTN planner as well as some existing HTN planning strategies. Most of the work done will focus on adaption, which raises a few risks that need to be considered, namely what if the adapted strategy does not work for *HTN* problems. To mitigate this, we would present the knowledge gained during research and discuss the issues faced when bringing the strategy to HTN problems and analyse why the strategy is not suitable for HTN planning problems. This would be followed by an exploration of possible substitutes.

Experimenting with existing HTN planning search strategies is one of this projects goals. If any issue is encountered, then we will scale back this ambition to more primitive HTN strategies which are easier to implement.

We also aim to achieve quicker solve times for HTN planning problems using an existing planner. A risk relating to this is the scenario in which no improvements are made. In this case we would analyse the run time process of the planner using Python's profiling capabilities (cProfile[1]) to develop an understanding of the root causes of poor results.

We intend to produce compatibility with the IPC's plan verification module. If we are unable to develop such functionality or use the functionality in conjunction with any search strategy, then manual verification of a selection of plans output by each search strategy would be required. The

[1] https://docs.python.org/3/library/profile.html#module-cProfile

problems to verify would include all of the IPC's test cases for individual HDDL features, as well as at least one IPC problem.

A risk of any project with a hard deadline is the risk of the deadline marching closer even if the goals of the project are taking longer to accomplish than expected. In the unlikely event that this becomes an issue for this project we will prioritise time and select a subset of the goals, which are more likely to be accomplished in time remaining and write about the issues that encountered and how future work could continue working on uncompleted goals.

If there is any issue with *Macleod* not working when or how required, we instead use a personal PC to run all evaluation code intended for *Macleod*

5.3 Design

In this section we consider the design options for implementing the requirements described in Section 5.1.

5.3.1 Plan Output in Format of IPC Verification

In this section we design a solution for is the requirement of being compatible with the HTN plan verification module used by the *IPC* which is detailed in Section 2.2.1.

The *IPC* plan verification format requires the planner to store more information regarding the progress of search. As standard *EPICpy* only tracks the order which *Tasks*, *Methods*, and *Actions* are applied to a *Model*. We extend this functionality to include knowledge of which *Method* decomposed each *Tasks* and which subtasks were added to the *Task Network* as a result. Definition 20 formalises this relationship. Sections A.1 and A.2 of the appendix further explores the software engineering implementation of this concept.

Definition 20. Structure for Storing Planner Progress - Decomposing Abstract Tasks

Each *Task* (T) which is decomposed by a *Method* (M) is recorded by the tuple (I,T,M,S) where:

- I is the ID of the *Task*
- T is the *Task* along with the assigned parameters
- M is the *Method* used to decompose T
- S is a tuple of subtasks where each $s \in S$ is an integer which correlates to the ID of an *Action* or *Task* added to the *Task Network* upon application of M

As mentioned each *Task* and *Action* needs to be assigned a unique ID. To do this we simply begin counting at 0 and increment by 1 each time a *Task* or *Action* is assigned an ID value. The first ID's are assigned to the *root tasks* and other ID's are distributed when the *Task Network* receives additions. Lastly, we must identify the *root tasks* before the planner commences solving, which is a trivial addition.

5.3.2 Pruning Seen States

In this section we construct a design for a simple search optimisation for *EPICpy*. We can assert that if we come across a previously seen *State* and *Task Network* pairing then we do not need to add the *Model* to the queue. This is because all future children of the *Model* have already been considered.

Figure 5.1 visualises the structure of hashing which must be undertaken for this functionality. The classes *Object*, *ProblemPredicate*, *State*, and *Subtask*, are defined within *EPICpy*. A *Subtask* is defined as a ground *Task*, *Method*, or *Action* from the planner to consider i.e. *Objects* have been assigned as parameters.

When hashing a *State* we must be mind-full that the ordering of *Facts* within the *State* must not impact the produced hash value. The contrary holds when we consider hashing the *Task Network*, in this case the ordering of *Subtasks* is influential and unique hash values should be generated for different orderings of the same *Subtasks*.

Section A.3 in the appendix discusses how this structure can be implemented using Python in more detail.

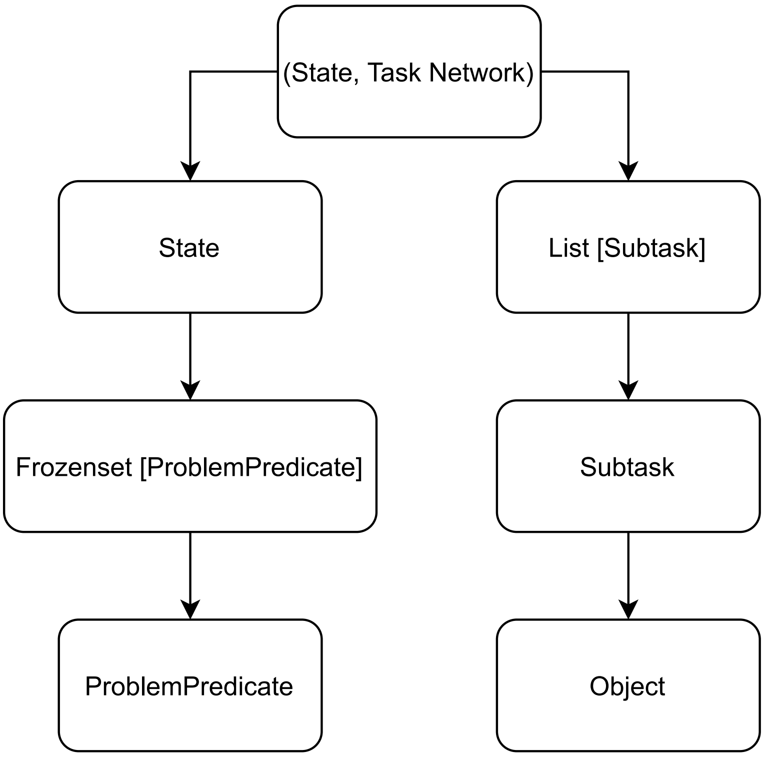


Figure 5.1: Structure of Hashing for Set of Seen States

Algorithm 3 puts forth pseudocode of the search algorithm required to make use of hashing pairings of *States* and *Task Networks*. In this listing, each pairing is hashed and checked against the set of all previously seen pairings before the *Model* is added to the queue. Pairings which have already been seen are ignored.

5.3.3 Novelty

In Section 3.5 we discussed the methodology of the *Novelty* strategy as well as its effectiveness in a variety of settings. In this section we discuss the restrictions which impact the *Novelty* strategy in HTN planning and subsequently present a design compatible with HTN planning problems.

Algorithm 3 Pruning Seen States Procedure Pseudocode

```
function SEARCH(Problem $\langle A, M, T, O, IS, OT, TN_I \rangle$, Strategy $s$)
    $closed \leftarrow \{\}$        ▷ We now keep a list of explored state and task-network pairings
    $frontier$.ADD($Model(IS, TN_I)$)
    loop
        if $frontier$ is empty then
            return fail
        $model \leftarrow s$.REMOVECHOICE($frontier$)
        if PROBLEM.GOALTEST($model$) then
            return GETPATH($model$)
        $subtask \leftarrow model$.POP-NEXT-SUBTASK()
        if $subtask \in T$ then
            EXPAND_TASK($problem, model, subtask$)
        if $subtask \in M$ then
            EXPAND_METHOD($problem, model, subtask$)
        if $subtask \in A$ then
            EXPAND_ACTION($problem, model, subtask$)
function EXPAND_TASK(Problem p, Model model, Task $\langle N, P, M \rangle$)
    for all $method \in M$ do
        $newModel \leftarrow Model(model.State, method + model.Task\_Network)$
        $frontier$.ADD($newModel$) ▷ The model is only added to the frontier if its hashed components are unique
function EXPAND_METHOD(Problem p, Model model, Method $\langle N, P, T, C, S, O, PC \rangle$)
    for all $subtask\_grounding \in S$ do
        $newModel \leftarrow Model(model.State, subtask\_grounding + model.Task\_Network)$
        $frontier$.ADD($newModel$) ▷ The model is only added to the frontier if its hashed components are unique
function EXPAND_ACTION(Problem p, Model model, Action $\langle P, C, E \rangle$)
    for all $effect \in E$ do
        $model.State$.APPLY($effect$)
    $frontier$.ADD($model$) ▷ The model is only added to the frontier if its hashed components are unique
```

5.3.3.1 State Novelty

As mentioned in Section 3.5 the concept of *Novelty* was first defined as a pruning technique. This is not compatible with HTN planning since the planner cannot move off the course set in the *Task Network*. Pruning *non-novel States* will likely result in no *Models* remaining in the queue. Instead we propose a method in which both *novel* and *non-novel States* can co-exist together, where *novel States* are sent to the front of the queue with *non-novel States* following behind as proposed in Section 3.5.

When we consider the *Novelty* strategy in *Classical Planning* we can identify how it helps improve performance. If *States* with new *Facts* are prioritised, we expect to encounter *States* with more *Facts*. This helps in *Classical Planning* since we keep searching until a *State* which satisfies the *Goal Conditions* is found. This is not the case with HTN planning as mentioned in Section 2.1.3. This raises some considerations for *Novelty* in *HTN Planning*. How long should a *State* be considered *novel*? Until it changes? Until a *Task* or *Method* is expanded?

We begin by considering a *Model novel* after an *Action* adds a previously unseen *Fact*, up until the point where the planner next considers the *Model* when expanding a *Task*, *Method*, or *Action*. This process is illustrated by the pseudocode in Algorithm 4, we name this strategy *Novelty with Reset*. This listing builds upon that seen previously in Algorithm 3 by introducing a global set of *seen facts*. Calculating the *Novelty* value for the *Model* is simple, if the effect being applied to the *State* is a positive effect then we check to see if the *Fact* added has been previously seen, if not we assign a *Novelty* value of -1.

Algorithm 4 Novelty with Reset Procedure Pseudocode

```
function SEARCH(Problem $\langle A, M, T, O, IS, OT, TN_I \rangle$, Strategy $s$)
    $Seen\_Facts \leftarrow \{\}$                                ▷ We now keep a list seen Facts
    ...                                              ▷ The remainder of this function is unchanged
function EXPAND_TASK(Problem p, Model model, Task $\langle N, P, M \rangle$)
    for all $method \in M$ do
        $newModel \leftarrow Model(model.State, method + model.Task\_Network)$
        $frontier.$ADD$(newModel, 0)$▷ The 0 here is changing the novelty score of the model back to 0
function EXPAND_METHOD(Problem p, Model model, Method $\langle N, P, T, C, S, O, PC \rangle$)
    for all $subtask\_grounding \in S$ do
        $newModel \leftarrow Model(model.State, subtask\_grounding + model.Task\_Network)$
        $frontier.$ADD$(newModel, 0)$ ▷ The 0 here is changing the novelty score of the model back to 0
function EXPAND_ACTION(Problem p, Model model, Action $\langle P, C, E \rangle$)
    $novel \leftarrow 0$
    for all $effect \in E$ do
        if $effect \in E^+$ then
            if $effect \notin Seen_Facts$ then
                $Seen\_Facts.$ADD$(effect)$
                $novel \leftarrow -1$
        $model.State.$APPLY$(effect)$
    $frontier.$ADD$(model, novel)$                          ▷ We also pass the novelty score to the queue
```

Building upon this strategy we can devise another named *Novelty with No Reset*. This entails a simple expansion of the algorithm previously seen in Algorithm 4 which is laid forth in Algorithm 5. This defines a new process for handling the *Novelty* of a *Model* when expanding a *Task* or *Method*, in this case we keep the same *Novelty* value assigned from the most recent *Action* applied.

Algorithm 5 Novelty with No Reset Procedure Pseudocode

```
function EXPAND_TASK(Problem p, Model model, Task $\langle N, P, M \rangle$)
    for all $method \in M$ do
        $newModel \leftarrow Model(model.State, method + model.Task\_Network)$
        $frontier.\text{ADD}(model, model.NoveltyScore)$ ▷ Here the keep the novelty score of the model
function EXPAND_METHOD(Problem p, Model model, Method $\langle N, P, T, C, S, O, PC \rangle$)
    for all $subtask\_grounding \in S$ do
        $newModel \leftarrow Model(model.State, subtask\_grounding + model.Task\_Network)$
        $frontier.\text{ADD}(model, model.NoveltyScore)$ ▷ Here the keep the novelty score of the model
```

In this section we have discussed two ways we can consider the *Novelty* of *Facts* in the planning procedure. Here we have seen *Novelty Level 1* which means only checking for single *Facts*. In Section 5.3.3.3 we build upon this concept for greater levels of *Novelty*. The software engineering design of this concept can be explored in Section A.4 of the appendix.

5.3.3.2 Method Novelty

In this section we build upon the design covered in Section 5.3.3.1 by discussing the role of *Tasks* and *Methods* in a *Novelty* strategy.

When we use a *Method* during search we can consider it as a *Fact* since we have the *Method* name and all of the objects being used as parameters. With this information we can determine if an instance of a *Method* has been used before by creating a set of used *Methods*. Algorithm 6 build upon Algorithm 4 to provide the pseudocode for this strategy.

Algorithm 6 Novelty with Methods Reset Procedure Pseudocode

```
function SEARCH(Problem $\langle A, M, T, O, IS, OT, TN_I \rangle$, Strategy $s$)
    $Seen\_Methods \leftarrow \{\}$                    ▷ We now keep a list seen Methods
    ...                                          ▷ The remainder of this function is unchanged
function EXPAND_METHOD(Problem p, Model model, Method $\langle N, P, T, C, S, O, PC \rangle$)
    $novel \leftarrow 0$
    $Ground\_Name \leftarrow N + \forall p_N \in P$
    if $Ground\_Name \notin Seen\_Methods$ then           ▷ Check if the Method is Novel
        $Seen\_Methods.\text{ADD}(Ground\_Name)$
        $novel \leftarrow -1$
    for all $subtask\_grounding \in S$ do
        $newModel \leftarrow Model(model.State, subtask\_grounding + model.Task\_Network)$
        $frontier.\text{ADD}(model, novel)$
```

This strategy of *Novelty* with *Methods* can be easily modified to remove the resetting of the *Novelty* value when expanding a *Task* using the same method explained previously in Section 5.3.3.1. We can also extend this strategy to include the calculation of *Novelty* values for expanded *Tasks*.

5.3.3.3 State Novelty - Greater Levels

In Section 5.3.3.1 we discussed the process for calculating *Novelty Level 1*, in this section we construct a process for calculating levels of *Novelty* which are greater than 1. Algorithm 7 builds upon the process for calculating *Novelty* presented in Algorithm 4. This strategy revolves around collecting the possible combinations of *Facts* from the *State* and combining them with the new *Fact* to check for a new combination. For example for *Novelty* level 2, the combinations of facts (line 7) would be a list of each individual *Facts* which we conjoin with the added *Fact* to create a pair. We then check if this pair has been seen previously. A new pair would be assigned the *Novelty* value of -1 since -2 would signify a unique *Fact*. This means that the search queue is ordered by *Models with novel facts*, *Models with novel fact pairings* and so forth.

Algorithm 7 Novelty Level 2+ Procedure Pseudocode

```
function EXPAND_ACTION(Problem p, Model model, Action $\langle P, C, E \rangle$)
    $novel \leftarrow 0$
    for all $effect \in E$ do
        if $effect \in E^+$ then
            $level \leftarrow 1$
            while $level \leq Max-Novelty-Level$ do
                $fact-combos \leftarrow model.State.\text{COMBINATIONS-OF-LENGTH}(level-1)$
                for all $FC \in fact-combos$ do
                    if $\{effect, FC\} \notin Seen\_Facts$ then
                        $Seen\_Facts.\text{ADD}(\{effect, FC\})$
                        $new\_novel \leftarrow (Max-Novelty-Level-level+1) \times -1$
                        if $new\_novel < novel$ then:
                            $novel \leftarrow new\_novel$
                $level \leftarrow level+1$
            $model.State.\text{APPLY}(effect)$
    $frontier.\text{ADD}(model, novel)$
```

5.3.4 Search Queue

Currently, *EPICpy* allows for *Models* in the search queue to be assigned a heuristic estimate with ties broken with the *Model* already in the queue taking priority. We modify this to allow for more customisable tie breakings. Instead each *Model* will be assigned three values, a *heuristic estimate*, *secondary heuristic estimate*, and a *queue location*.

The *secondary heuristic* is default to *0* in which case it will not impact the ordering of *Models*. The *queue location* value will then control whether the oldest or newest *Model* takes priority. To prioritise the oldest the value set for *queue location* will be incremented by 1 for each *Model* added, and -1 for prioritising newest.

5.3.5 Landmarks

In this section we review the design process for generating *Landmarks* using an *AND / OR Tree* discussed in Section 3.2.2. Algorithm 8 shows a recursive algorithm which constructs an *AND / OR Tree*.

The *Landmarks* of a problem can be extracted from the developed tree structure by following the procedure in shown in Algorithm 9. This algorithm puts the set manipulation process shown in Figure 3.4 into action.

Algorithm 8 Generate AND / OR Tree

```
$problem\langle A,M,T,O,IS,OT,TN_I\rangle$
$node\_list \leftarrow \{\}$
$root\_node \leftarrow AndNode('Root')$
for all $initial\_task \in TN_I$ do
    ADD_TO_NODE$(initial\_task, root\_node)$
function ADD_TO_NODE(task, Node parent_node)
    if $task \in T$ then
        $node \leftarrow OR\_NODE(task, parent\_node)$
    else
        $node \leftarrow AND\_NODE(task, parent\_node)$
    $parent\_node.requires.$ADD$(node)$
    $node.required\_by.$ADD$(parent\_node)$
    $already\_seen \leftarrow node \in node\_list$
    if $NOTalready\_seen$ then
        if $task \in T$ then
            for all $method \in task_M$ do
                ADD_TO_NODE$(method, node)$
        if $task \in M$ then
            for all $subtask \in task_S$ do
                ADD_TO_NODE$(subtask, node)$
        if $task \in A$ then
            for all $effect \in task_E$ do
                $node.provides.$ADD$(effect)$
                $effect.provided\_by.$ADD$(node)$
            for all $precondition \in task_C$ do
                $node.requires.$ADD$(precondition)$
                $precondition.required\_by.$ADD$(node)$
```

Algorithm 9 Extracting Landmarks from AND / OR Tree

```
function EXTRACT_LANDMARKS(node_list)
    $leaf\_nodes \leftarrow node \in node\_list$ WHERE node.required IS $\emptyset$
    while $leaf\_nodes$ IS NOT Empty do
        $node \leftarrow leaf\_nodes$.POP()
        CALCULATE_LANDMARKS($node$)
        for all $r \in node.required\_by + node.provides$ do
            if $r$ IS $OR\_NODE$ then
                if ANY(prov.landmarks $\neq \emptyset \forall$ prov $\in$ r.provided_by) then
                    $leaf\_nodes$.ADD($r$)
            else
                if ALL(req.landmarks $\neq \emptyset \forall$ req $\in$ r.requires) then
                    $leaf\_nodes$.ADD($r$)
function CALCULATE_LANDMARKS(node)
    if $node$ IS $AND\_NODE$ then
        if $node$ IS NOT $FACT$ then
            $node.landmarks \leftarrow UNION(req.landmarks \forall$ req $\in node.requires) + node.name$
        else
            $node.landmarks \leftarrow UNION(prov.landmarks \forall$ prov $\in node.provided\_by) +$
            $node.name$
    if $node$ IS $OR\_NODE$ then
        $node.landmarks \leftarrow (INTERSECTION(req.landmarks \forall$ req $\in node.requres)) +$
        $node.name$
```

The process of extracting landmarks can be very expensive for large problems. In an attempt to combat this we suggest the optimisation of limiting the amount of *AND Nodes* which relate to *Methods*. By using the *Delete Relaxed* heuristic mentioned in Section 3.2.1 to determine which *Methods* should be part of the *AND / OR Tree*. Thus decreasing the amount of nodes required to be considered.

5.3.6 Hamming Novelty

In Section 5.3.3.1 we introduced a design for *Novelty Level 1*, in this section we will expand this design to create a new strategy in combination with *Hamming Distance*. *Novelty Level 1* checks if a *Fact* has previously be seen by any *State*, we modify this to check if a *Fact* has been seen by any *State* with the same *Hamming Distance* heuristic estimate. This enables a *State* to be considered *Novel* even if it is not the first to encounter a particular *Fact* so long as it is closer – or further away – to the *Goal Conditions* than any *State* which previously saw the same *Fact*.

5.3.7 Collecting Results

In this section we discuss the evaluation of strategies and collection of results. We evaluate *34* strategies as listed in Table A.1 of the appendix.

To evaluate each strategy we devise a design for a module which runs and reports the results of problems to a csv file. Each strategy has its own csv file for easy comparison. Figure 5.2 outlines the steps required for this module. The problems we use are those which were used to evaluate planners in the 2020 *IPC* as discussed in Section 2.2.

Table A.2 of the appendix lists the attributes collected for each strategy.

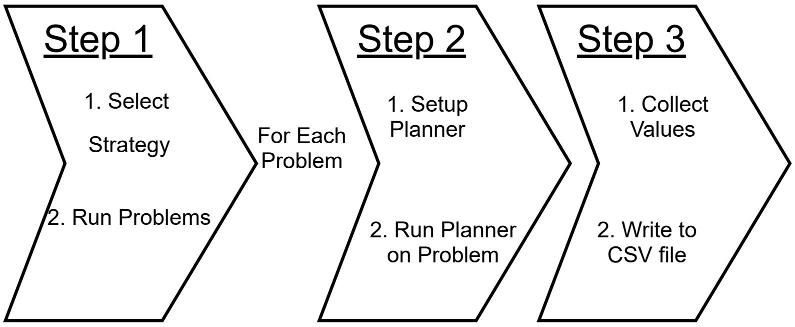


Figure 5.2: Collection of Results Structure

5.4 Methodology

The aim of this project is to adapt, implement, and evaluate the *Novelty* search strategy using the *EPICpy* planner. Building upon the planner with the inclusion of optimisations and strategies is also within the scope of this project. Below is the activities required for this project:

1. Researching existing literature relating to search strategies for HTN planners.
2. Researching existing literature relating to search strategies for Classical planners.
3. Considering methods of adaptation for the *Novelty* strategy.
4. Devise design options for adding *Novelty* to *EPICpy*.
5. Implement search strategies and optimisations.
6. Construct software unit tests to ensure the correctness of software developed.
7. Evaluate performance of implemented strategies on HTN planning problems.
8. Verify the correctness of plans generated by the planner.

5.5 Technology Choice

For this project the choice of technology was somewhat limited since we are expanding existing software. In this section we describe the different technologies utilised and their usages.

5.5.1 Python

We use Python [2] for software implementation in this project since it is the language *EPICpy* is implemented in. Despite this restriction Python is still a great choice as it is compatible across lots of operating systems. This helpful since the the development of this project will be split between two different operating systems as mentioned in Section 5.5.2.

Python is not cemented as the programming language of choice from HTN planners, popular planners are implemented in C++[3] [35, 59, 21] or Java[4] [55]. Few are implemented using interpreted languages such as Ruby [50] and Python [46].

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

[3] https://isocpp.org

[4] https://openjdk.java.net

Testing software is a central part of every large software project, the capability to ensure the correctness of developed code even following the addition of new code is an indispensable capability. For the purposes of this project we will use the *unittest*[5] package which is built-in to Python.

5.5.2 Operating System

For this project we make use of two operating systems - Windows and Linux. Windows since it is common place within desktop and laptop computers. On the other hand Linux or more specifically Ubuntu is required at a technical level for some aspects of this project. For example the plan verification module mentioned in Section 2.2.1 requires a Linux based system to run.

5.5.3 MacLeod

To evaluate all strategies concurrently we use the University of Aberdeen's *MacLeod* computing cluster. This provides much more memory capabilities and processing power than can be found on any personal machine.

[5] https://docs.python.org/3/library/unittest.html

Chapter 6

Testing & Evaluation

In this chapter we review the results gathered when solving the selection of IPC problems discussed in Section 2.2. We begin by discussing the implementation strategy, then follow on by deliberating why we can trust the *EPICpy* planner in its standard form. Finally, we view the results gathered when evaluating strategies.

6.1 Implementation

The implementation of software for this project was done in a task by task approach. Work to achieve each of the designs laid forth in Chapter 5 was kept separate from existing code using *Git Branches*. Instead of having a *'main'* branch we have a *'mastersProject'* branch, and every branch we create for implementing components and functionality take the form *masters/<branch name>*.

The implementation of *Novelty* search strategies was split into sections each of which tackles one of the following: *Novelty with reset*, *Novelty with no reset*, *Novelty level 2*, *Novelty with Tie Breakers*, and *Novelty for Methods*. During implementation a staggered approach was adopted where each strategy was developed one at time, this resulted in a smooth implementation experience where each strategy built upon the code of the previous.

An early contribution to the code base during this project was the implementation of the HTN plan verification module designed Section 5.3.1. This required a lot of new code and modifications to existing code, making the role of unit tests invaluable to ensure existing functionality has not been unexpectedly altered. Adding this large amount of code caused unforeseen issues, the main being that *EPICpy*'s parser would convert all text from the domain and problem files to lowercase. While this does not impact the planners ability to find correct plans, it did cause issues with the *pandaPIparser* plan verification module which expected to see object names with capital letters. Removing the lowercase conversion caused some quite substantial issues for *JSHOP* parsing since both *EPICpy*'s *HDDL* and *JSHOP* parsers use the same method to read the contents of input files. To fix this issue an implementation of the function used to read files was added to the *JSHOP* parser, keeping the lower case presentation styling.

The implementation of *Landmarks* strategies followed a trial by fire approach where several approaches were considered at each step with only the most promising taken forward. The inclusion of a reachability calculation discussed in Section 5.3.5 came to being as a result of this approach. This calculation required a substantial chunk of existing *Delete Relaxed* code within *EPICpy* to be entirely re-written, which absorbed a lot of development time.

Implementation of a software structure to evaluate strategies took place relatively early in the cycle of this project as we required data to inform design choices. A lot of files are required for this since each strategy required an *SBATCH* job file along with a file calling the main testing framework for each problem.

6.2 Verifying the Correctness of Existing EPICpy Components

Before we examine the results gathered we first state why we are able to trust the existing components of the *EPICpy* planner to return correct results. To do this we run the planner with its standard components on a selection of *IPC* test problems some of which are designed to test specific edge cases of domain definition. Each of the returned plans is in the format discussed in Section 2.2.1 and evaluated using the *PANDA* plan verification tool. Section B.1 of the appendix hosts three tables, each of which show the the problems evaluated for two *EPICpy* configurations. These tables show that none of the configurations returned an incorrect result and the only omissions from the table are for problems with took too long to return a result.

6.3 Evaluation

Section 5.3.7 lays forth the process followed to gather results. In line with the IPC's evaluation environment covered in Section 2.2.2, each strategy is allocated 30 minutes to solve each problem. For the purposes of this project, the memory allocation limit was scrapped since the aim of this project is to achieve quicker solving times, not better memory efficiency. For all strategies *EPICpy* is configured to use a *Greedy Best First* approach since we are not interested in result optimality. All problems were evaluated under the same conditions using *MacLeod*.

We also compare the performance of *EPICpy* with each of its strategies to the *PANDA* planner introduced in Section 4.7. In the interest of fairness we ran *PANDA* under the same conditions are *EPICpy* using *MacLeod*. *PANDA* was configured with its default settings with the exception of *optimal plans* and the *heuristic weighting* since we are not concerned with optimal plans.

Throughout this section we will see graphs which include data gathered from a variety of the strategies previously mentioned. We only consider the results gathered from Landmarks based strategies in Section 6.3.7 since the range of problems ran using these strategies is considerably less that those attempted by all other strategies. Section B.3 of the appendix contains a selection of graphs showing the solve time for each strategy in each domain.

In some instances some strategies did not attempt all problems on a domain due to tight time constraints on this project. Each result gathering iteration took approximately two weeks, and although the time limit of *1800* seconds is set for each problem the planner can only terminate execution at intervals between the consideration of *Tasks, Methods,* and *Actions*. In some cases problems would remain in a cycle of paralysis for considerably longer than the allotted time. If a significant breach of the allotment occurred for a particular problem in a domain, the rest of the domain would be skipped by the strategy if there was no evidence of the strategies ability to solve any problems of the domain. Multiple iterations of data gathering were required since the range of data collected was increased each iteration to further explore trends.

6.3.1 Correlations

In this section we discuss correlations found within the results gathered when following the evaluation design discussed in Section 5.3.7. To investigate the presence of a correlation between attributes we apply *Spearman's rank correlation coefficient*. This is a statistical test used to determine if a positive or negative correlation exists between two groups and the *p-value* certainty that the correlation is not due to noise in the data. The results of this test are within the range -1 to 1. Scores of 1 or -1 indicate a perfect monotonic correlation between the groups given.

The correlations we are checking for are those between *solve time* and each of *number of expansions*, *amount of Novel States*, *percentage of Novel States*, *amount of Novel Methods*, *percentage of Novel Methods*, and *percentage of possible facts in the final State*. To calculate the correlations for each problem we consider the results gathered for strategies able to solve the problem, if more than 1 strategy was able to solve it. The results for each correlation are ordered in ascending order.

In some instances we were unable to run a *Spearman's* test for the attributes recorded for a problem since each of the strategies which solved the problem produced the same value. In this case the *Spearman's* test cannot produce a result. This occurred in 32 instances when testing for a correlation with *amount of Novel States*, 4 instances of *percentage of Novel States*, 10 instances of *amount of Novel Methods*, 27 instances of *percentage of Novel Methods*, 10 instances of *number of expansions*, and 26 instances of *percentage of total facts in final State*. These occurrences are interesting since they highlight the possibility of unique search strategies recording exactly the same attributes during search, this could prove to be interesting area for extra research beyond the scope of this project.

6.3.1.1 Visualising Relationship

Before we progress to the statistical results found, we can look at the relationship between attributes from a high-level for *Hamming Distance* based strategies solving the *Rover* domain. Figure 6.1 shows the *solve time*, *percentage of novel Methods*, and *percentage of novel States* for each strategy solving *Rover* problems. From the figure we are able to identify the strategy which solves the least amount of problems is *Hamming-Oldest* which also records the lowest percentages of *Novelty*. While the strategies which solve the most problems -*Hamming-TreeDis* and *Hamming-Novelty-Reset* - record higher percentages of *Novelty*.

6.3.1.2 Correlation Across All Problems

Figure 6.2 visualises the correlations found within the results gathered across *EPICpy* strategies. Along the x-axis is the number of problems while the y-axis shows the result of a *Spearman's* test. This figure only shows the correlations which had a *p-value* of less than *0.05*, this means we can be confident that the results shown are not due to noise. This is also accountable for the differences in line length visible in the figure. Figure B.1 of the appendix shows the same graph but with the filtering of *p-values* removed.

6.3.1.3 Correlation between Expansions and Solve Time

When we consider the possibility of a correlation between number of expansions and solve time, intuition tells us that the more expansions that are done the longer solving will take. From the results plotted in Figure 6.2 we can see that this correlation does in fact exist, with all of the problems plotted showing a positive correlation.

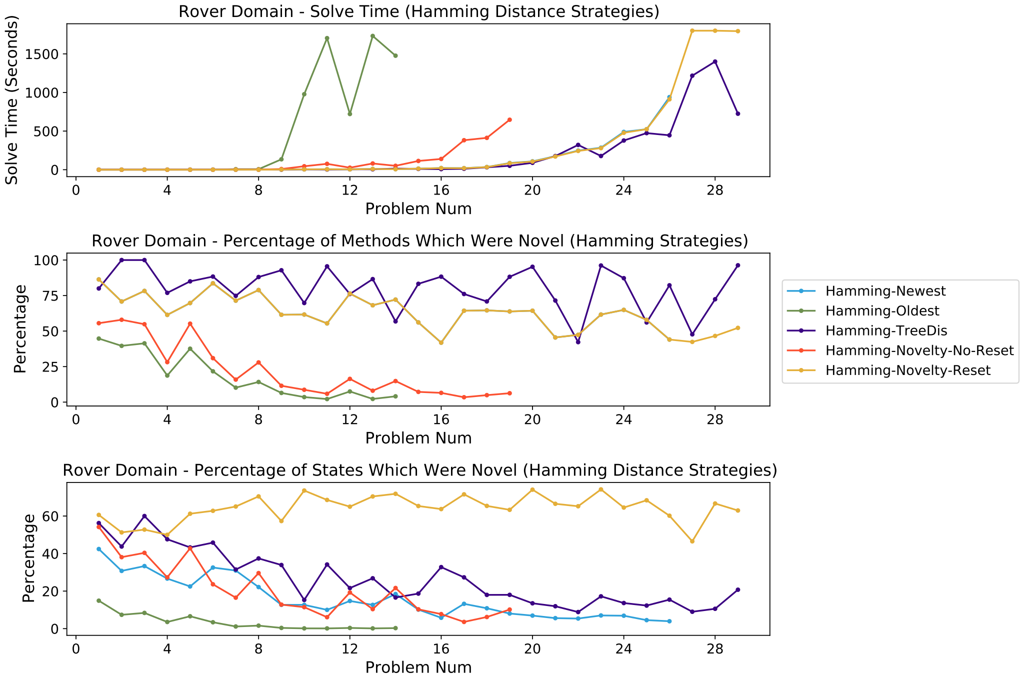


Figure 6.1: Attributes Recorded for Hamming Distance Strategies Solving Rover Domain Problems

6.3.1.4 Correlation between Novelty and Solve Time

Since the an objective of this project is to determine the effectiveness of *Novelty* for *HTN* planning, the correlation between *Novelty* attributes and *solve time* is of importance. From Figure 6.2 we can see that as the amount of *Novel States* increases as does the *solve time* but, as the *percentage of Novel States* increases the *solve time* tends to decrease. From this we can hypothesise that strategies which see a larger amount of *Novel States* may be prioritising finding *Novel States* ahead of finding the goal which could be producing more *States* in general. We can also say that strategies which find more *Novel States* as a percentage of all *States* tend to solve problems quicker.

When we assess the distribution of correlations between *amount of Novel Methods* and *percentage of Novel Methods* against *Solve Time* in Figure 6.2 we can see a similar distribution to that previously discussed between *States* and *Solve Time*. From this we can state that strategies with a high percentage of *Novel Method* usage are able to achieve quicker *Solve Times* than strategies which prioritise finding larger quantities of *Novel Methods*.

Finally, we can address the possibility of a correlation between the *percentage of possible facts in the final State* and *Solve Time*. The points plotted in Figure 6.2 appear to be split between positive and negative correlations. From this split we can determine that this attribute is not a major contributor to quicker or slower *Solve Times* due to the ambiguous nature of the distribution.

To round off this section we test for a correlation between the *percentage of novel States* and *percentage of novel Methods*. A *Spearman's* test shows a correlation score of *0.6883* with a

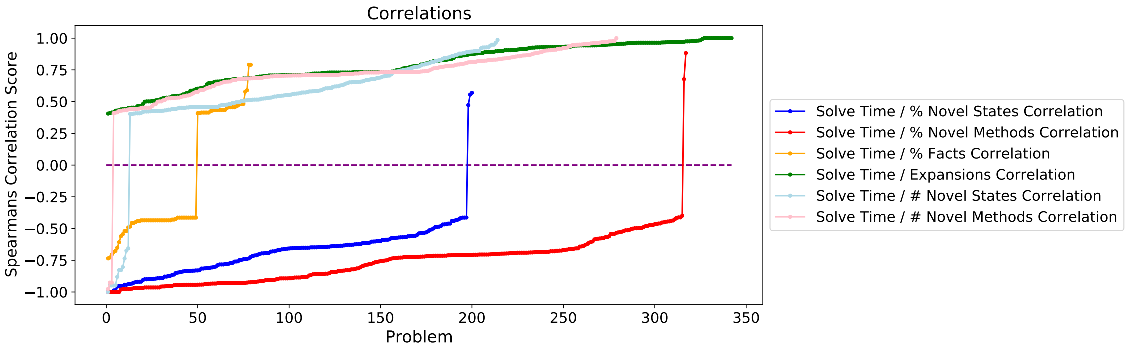


Figure 6.2: Correlation (Spearmans Test) between Solve Time and Attributes

p-value of *0.00014*, this informs us of a likely relationship between the two attributes such that as one increases as does the other.

6.3.2 IPC Score

Figure 6.3 shows the distribution of *IPC Scores* across all strategies and *PANDA* following the calculation formula provided in Section 2.2.2. Glaringly clear from this graph is that when considering all domains tested the *PANDA* planner beats all strategies of *EPICpy*. Figure B.4 of the appendix showcases the *IPC Scores* of only the *EPICpy* strategies.

We can identify the top performing strategies as *Hamming-Newest*, *Hamming-TreeDis*, and *Tree-Distance-HamDis*, all of which are new strategies added as part of the work in this project. Disappointingly none of the *Novelty* based strategies were part of the top three performers. The best *Novelty* based strategies were *Hamming-Novelty-Reset* and *Novelty-Level1-HamDis* ranked fourth and fifth respectively, both of these strategies made use of the *Hamming Distance* functionality of *EPICpy*.

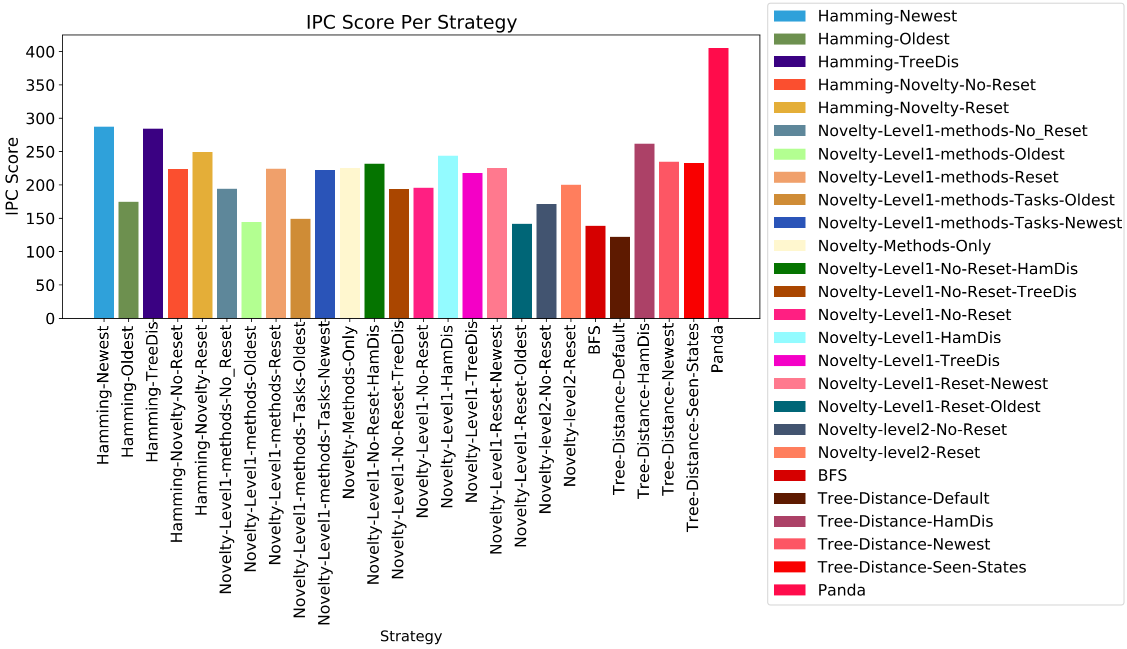


Figure 6.3: Total IPC Score of Each Strategy

Similarly to the correlation tests carried out in Section 6.3.1 we can test for a correlation between percentages of *Novelty* and *IPC Scores*. Figure 6.4 shows a scatter plot of each strategy's average percentage of *Novel States* in the problems which were solved by all strategies. At a glance we see an upwards trend that as the percentage of *States* which were *Novel* increases as does the *IPC Score*. The results of a *Spearman's* test show a positive correlation of *0.6234* with a *p-value* of *0.00087*. This means we can say with confidence that increasing the percentage of *States* which are *Novel* tends to increase the *IPC Score* achieved.

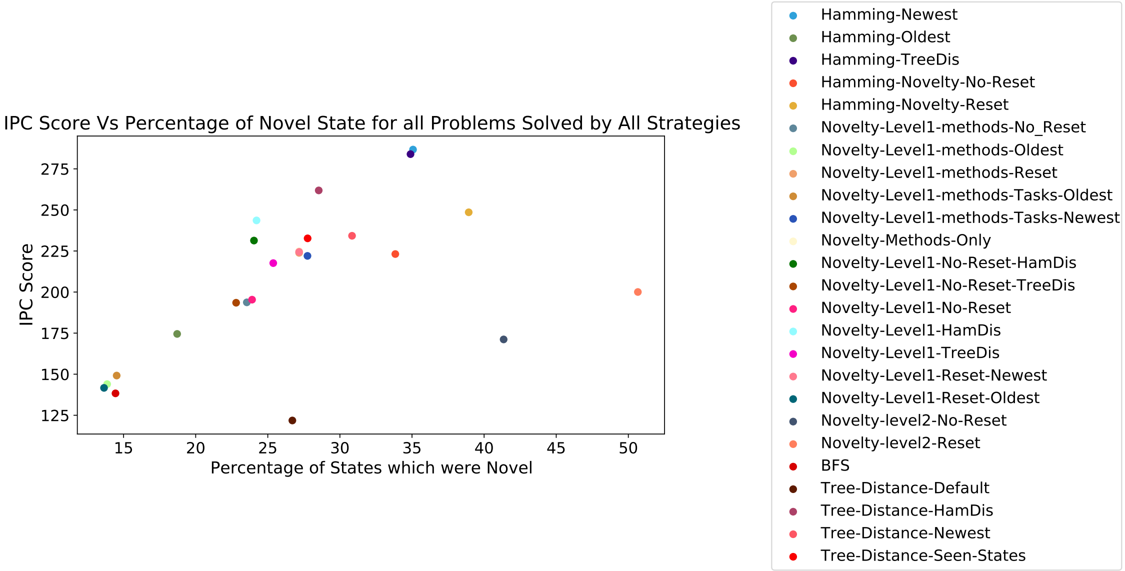


Figure 6.4: Scatter Plot of IPC Score against % Novel States

Figure 6.5 shows a scatter plot of the relationship between *IPC Score* and the percentage of *Methods* which were *Novel*. Like with the previous graph we can see what appears to be an positive correlation between the two. A *Spearman's* test confirms this observation with a correlation score of *0.7649* and a *p-value* of 8.456×10^{-06}. This confirms that the relationship between the two is likely to be significant.

6.3.3 Comparison to PANDA

In Section 6.3.2 we established *PANDA's* superiority when we compare the overall *IPC Scores* achieved by each strategy, in this section we explore these scores further by investigating *IPC Scores* on a domain-specific basis. Section B.5 of the appendix contains graphs showing the *IPC Score* for each strategy across each domain.

From the aforementioned graphs we are able to identify that *PANDA* is beaten by a *EPICpy* strategy in the following domains: *Rover*, *Barman*, *Hiking*, *Multiarm-Blocksworld*, and *Transport*. These graphs also provide a basis for identifying which domains all *EPICpy* strategies do poorly such as the *Robot* and *Monroe-Partially-Observable* domains. Performance in these domains could prove to be good future work for the *EPICpy* planner.

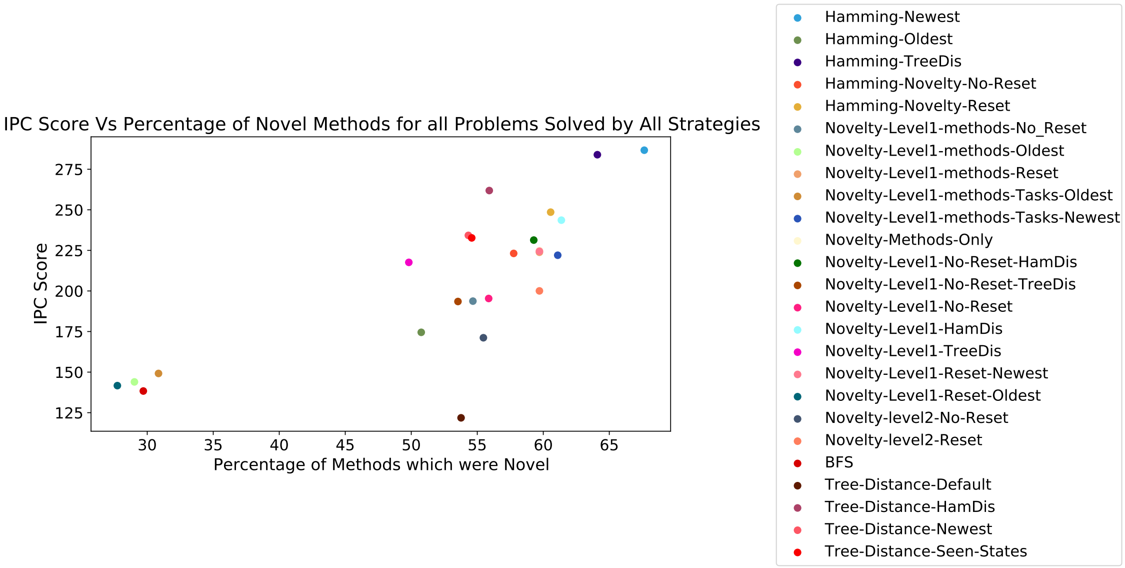


Figure 6.5: Scatter Plot of IPC Score against % Novel Methods

6.3.4 Domain Coverage

In this section we discuss the differences in coverage across *EPICpy* strategies for the domains tested. Section B.6 of the appendix provides a collection of tables which detail the amount of problems solved by each strategy in each domain. The strategy with the most problems solved is *Tree-Distance-HamDis* which boasted the third best *IPC Score* in Section 6.3.2.

Strategies developed for this project had differing tie breaking options, with some prioritising newest *Models* first while others prioritise oldest first. When we compare these strategies we can see that prioritising the newest *Models* first enables the planner to solve more problems overall across all strategies. This is particularly highlighted when we compare the *431* problems solved by *Hamming-Newest* against the *307* problems solved by *Hamming-Oldest*.

An interesting observation of this data is that strategies using *Tree Distance* are able to solve problems of the *Transport* and *Entertainment* domains while all others fail to solve any - with the exception of an scarce outlier. This highlights a key challenge faced by domain-independent planners since planners need to be able to solve all problems given but different strategies can dominate in particular problem types.

In Section 5.3.2 we introduced a process for pruning pairings of *States* and *Task Networks* which were previously seen by the planner. We can analyse the direct impact of that optimisation by comparing the results of *Tree Distance* with and without the optimisation. Without the pruning optimisation *Tree-Distance-Default* seen in Table B.13 was able to solve *179* problems, which is even less than *Breadth First Search* with the pruning optimisation which was able to solve *252* problems. Table B.14 shows that *Tree-Distance-Seen-States* was able to increase the amount of problems solved to *388* in a meteoric difference.

6.3.5 Cumulative Solved Problems

Figure 6.6 visualises the amount of problems solved by each strategy over time. From this format we can see that *Hamming-TreeDis* and *Hamming-Newest* solve more problems within the first *250* seconds of search than other strategies. The top five best performing strategies discussed in

Section 6.3.2 are the strategies with the most problems solved at the full *1800* seconds.

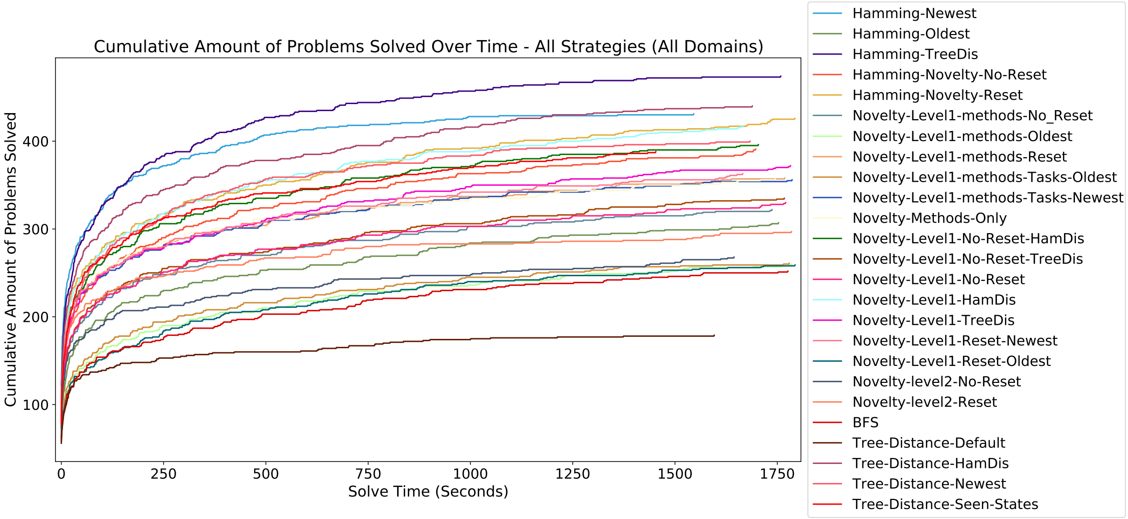


Figure 6.6: Cumulative Amount of Problems Solved Over Time

6.3.6 Analysis of Novelty Strategies

In this section we compare all *Novelty* based strategies to the top performing strategy discussed in Section 6.3.2 - *Hamming-Newest*. Figure 6.7 shows the average percentage of *novel States* and *novel Methods* found by each strategy when tackling problems solved by all *Novelty* strategies and *Hamming-Newest*. From the graph we can identify that *Hamming-Newest* encountered a higher percentage of *novel Methods* and *States* on average than any of the *Novelty* based strategies. This is interesting since the strategies which set out to find *novel States* and *Methods* are beaten at this objective by a strategy which does not consider this at all. When we focus on the strategy *Novelty-Methods-Only* which bases its strategy around finding *novel Methods*, we see that it finds a very similar percentage to the other *Novelty* based strategies while maintaining a similar percentage of *novel States*. The only *Novelty* based strategies which show a substantial increase in the percentage of *novel States* is the *Novelty-level2* strategies, this could simply be because the criteria placed upon a *State* being *novel* is widened from finding a unique *Fact* to finding a unique pairing of *Facts*.

We can consider the information shown in the bar graph deeper by analysing the percentage of *Novel States* and *Methods* found for the *Rover* domain. Figure 6.8 shows the *percentage of Novel States* found by a limited selection of strategies, in this graph we show only the percentages achieved on solved problems. We choose to show the strategies *Novelty-level2-Reset* and *Novelty-level1-TreeDis* since they boast the best *Percentage of Novel States* of their respective *Novelty* levels. From this graph we see that *Novelty-level2-Reset* has a much higher percentage than the other two strategies but interestingly fails to solve as many problems. Figure B.7 of the appendix shows the same graph but including all *Novelty* based strategies.

We can now combine the findings of the discussed *Percentage of Novel States* graph with Figure 6.9 which shows the *Percentage of Novel Methods* for the same strategies. From this graph we can interesting see that *Novelty-level2-Reset* records exactly the same percentage as *Hamming-Newest* despite a huge difference in the *Percentage of Novel States*. Figure B.8 of the appendix

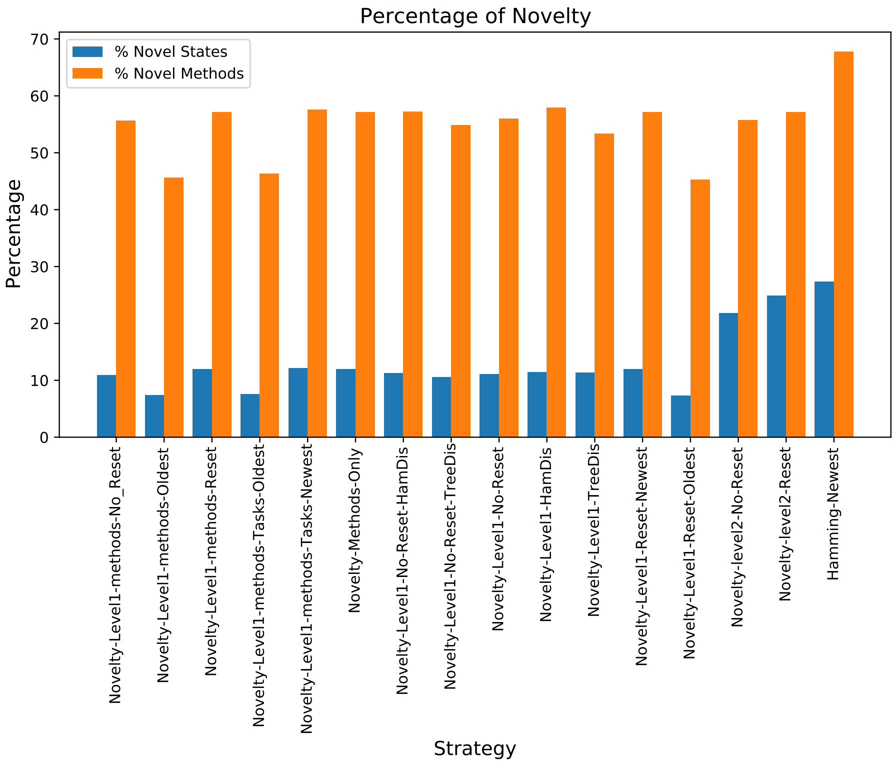


Figure 6.7: Percentage of Novelty

shows the same graph but including all *Novelty* based strategies.

Figure 6.10 shows the amount of *States* and *Methods* found by each strategy, this graph is clear that *Hamming-Newest* finds significantly less *novel Methods* compared to the other strategies, but finds a very similar amount of *novel States*. These results suggest that the reason *Hamming-Newest* records a higher percentage of *novel States* & *Methods* is because it finds less in total. Figures B.9 and B.10 of the appendix provide a closer look at these figures compared with the total amount of *States* and *Methods* found.

6.3.7 Landmarks Results

Section 5.3.5 introduced a design for adding a reachability calculation in an attempt to improve the time taken to calculate *Landmarks*. Figure 6.11 shows a comparison of the time taken to calculate landmarks using a reachability method and without one, including the time to calculate the reachability. From these graphs there is no doubt over the effectiveness of a reachabilty method when calculating *Landmarks* for a problem since the difference illustrated is monumental.

Figure 6.12 visualises the time taken to solve problems using select *Landmarks* based strategies and compares them to the strategy *Hamming-Newest*. From these graphs we can see how competitive the *Landmarks* could be if the time taken to calculate landmarks we significantly reduced. In the *Factories* domain a *Landmarks* based strategy is able to solve a problem left unsolved by *Hamming-Newest*. Figure B.11 in the appendix shows the same graphs but with all *Landmarks* strategies shown.

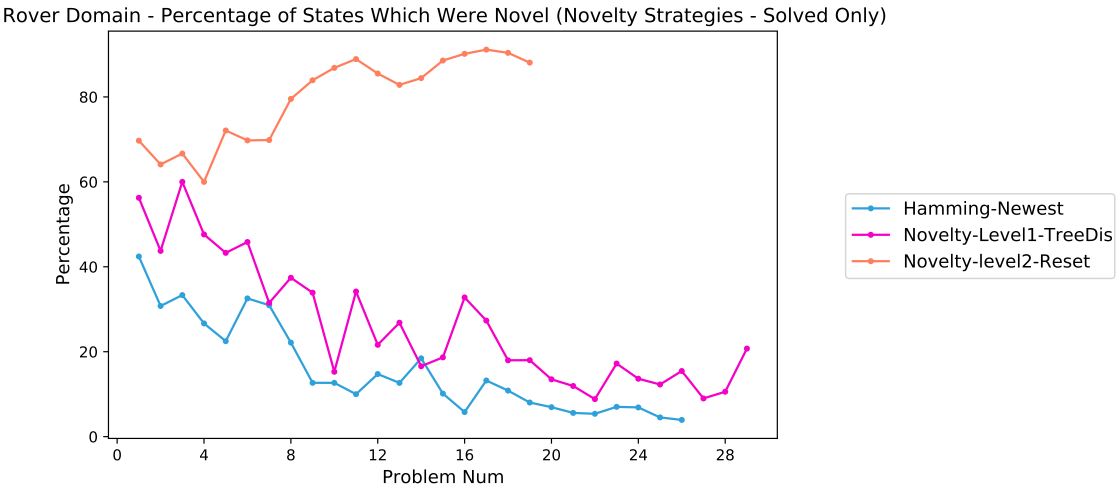


Figure 6.8: Percentage of Novel States - Rover Domain (Select Strategies)

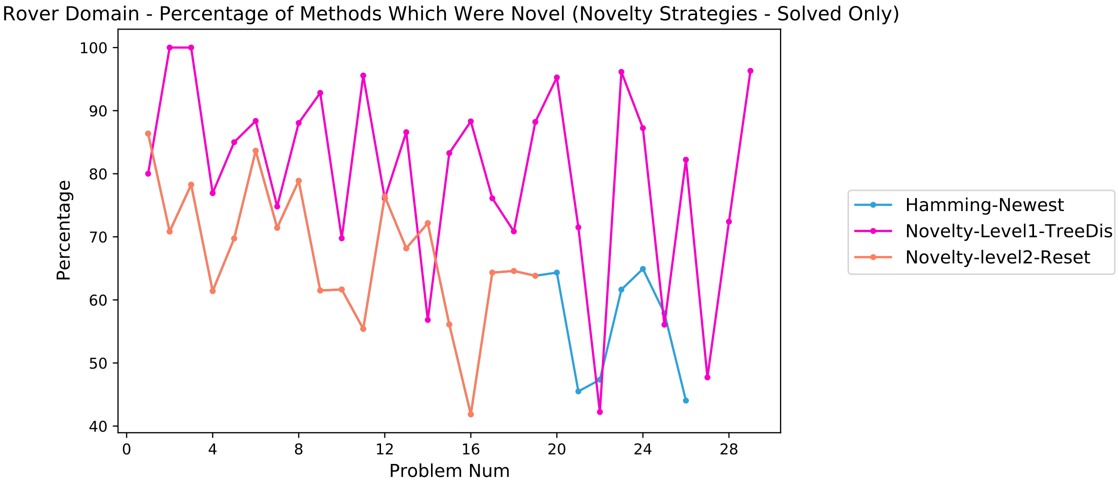


Figure 6.9: Percentage of Novel Methods - Rover Domain (Select Strategies)

6.4 Verifying Plans

To verify the output for each problem which returned a solution we run the evaluating suite for each strategy on the problems solved using the plan verification tools outlined in Section 5.3.1. We do this separately since the plan verification components slow down the performance of the planner an unbelievable amount. The memory required for these components is also significantly more than required by default.

Due to this very poor performance we were unable to verify plans for all solved problems due to the time constraints of this project. A solution was not returned for a small selection of problems and as such could not be verified. Given the huge amount of problems verified for lots of other strategies, we have no reason to suspect that there is any issue with the unverified solutions. With more time the verification of the problems could be achieved.

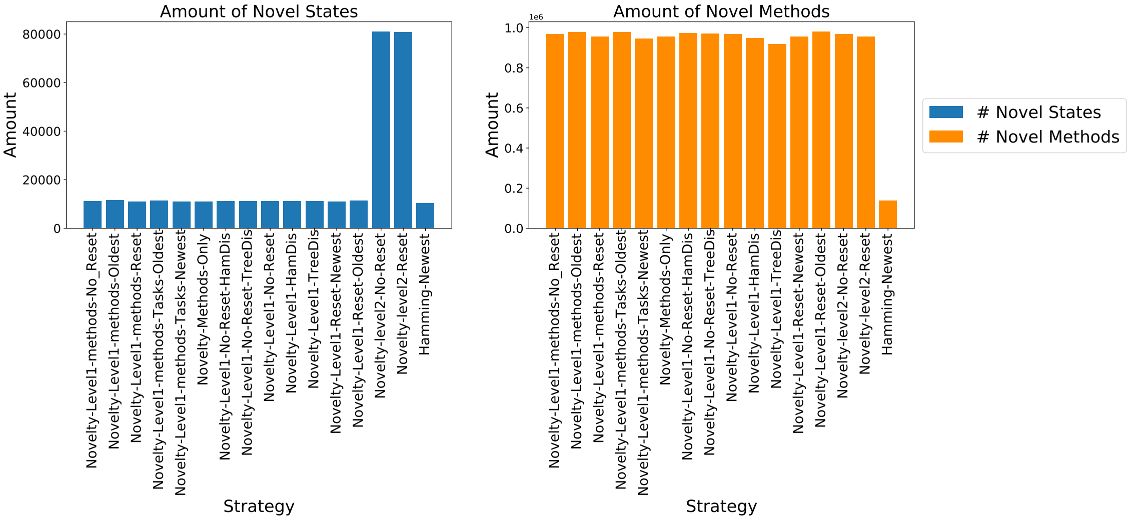


Figure 6.10: Amount of Novel States & Methods (Novelty Strategies)

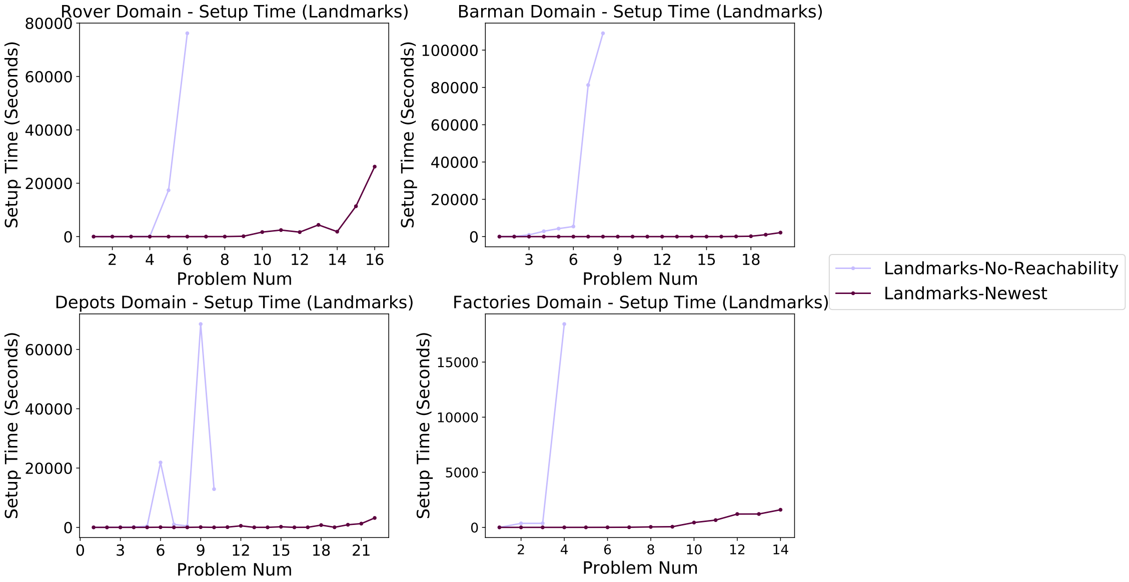


Figure 6.11: Landmarks Setup Time

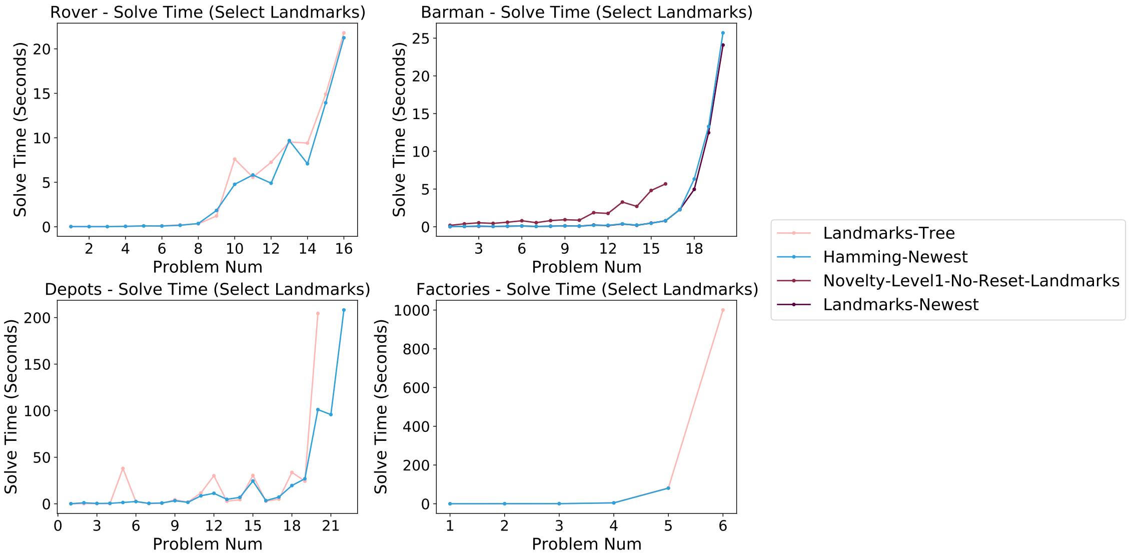


Figure 6.12: Select Landmarks Solve Times

Chapter 7

Discussion, Conclusion & Future Work

7.1 Discussion

In Section 6.3.2 we established that the *PANDA* planner clocks a higher *IPC Score* than all of *EPICpy*'s strategies. This is not very surprising when we consider that *PANDA* is developed in *C++* compared to *EPICpy*'s *Python*. Furthermore the amount of work spent by multiple collaborators to produce the current version of *PANDA* dwarfs the modest short-term work of an individual developer of *EPICpy*. By comparing *EPICpy* to *PANDA* we were not aiming to compete in terms of *IPC Score* but to analyse the results of each domain to establish if *EPICpy* is able to produce competitive results in any circumstances. In Section 6.3.3 we discussed domains which *EPICpy* is able to produce better results than *PANDA*. These results are very encouraging given the difference in work contributed between the planners.

In Section 6.3.2 we identified *Hamming-Newest* as the top performing *EPICpy* strategy, when we consider the simple nature of the *Hamming-Distance* strategy outlined in Section 4.1.3 we can hypothesise to what is happening during solving. If a problem has defined *Goal Conditions* then the first *Model* to find a *State* which satisfies any of the *Goal Conditions* is prioritised by the planner while all other *Models* remain in the queue. This makes the search process extremely greedy since no other *Model* has an opportunity to find any *Goal Conditions*. In Section 6.3.4 we discussed the improvements found when ordering *Models* in the queue based on newest *Models* first. This ordering makes the search more greedy by its self since if all *Models* have the same ranking then the same *Model* will be in the front of the queue indefinitely. When we consider the greedy nature of the *Hamming-Newest* pairing we can suggest that greedy strategies could posses a benefit to solving *HTN Planning* problems.

Section 6.3.1 brought evidence of a correlation between the percentages of *novel States* and *Methods* relating to *solve time*. While a simple increase in the *amount of novel States* and *Methods* proved to be less influential. When we consider this in combination with the breakdown of *Novelty* attributes shown in Section 6.3.6 we see that *Hamming-Newest*'s high percentage of *novel Methods* is due to the fact that it uses less *Methods* during search. This enables us a identify a balance between the amount of *Novelty* we prioritise. Higher quantities *Novelty* are not necessarily better but, a high *Percentage of Novelty* within a smaller collection of used *Methods* and *States* appears to be beneficial based upon the evidence gathered.

7.2 Applications of Work Done

AI planning systems are becoming more common place in every aspect of modern life, with systems able to devise plans in a huge range of applications from autonomously operating vehicles safely [54] to planning tasks for robots in smart warehouses [9], and even operating rovers on planet Mars [2]. Published studies suggest that the trend of autonomous smart agents operating in new environments is likely to continue in the future [63].

In this project we focused on one specific type of AI planning - *HTN Planning* - and explored how the incredibly successful *Novelty* strategy [47] from *Classical Planning* can be adapted and utilised for *HTN* planning problems. From the results discussed in Chapter 6 we have shown the importance of ensuring that a high percentage of *States* and *Methods* considered by the planner are previously unseen to give the best possible chance of finding a solution quickly.

Using an autonomous AI agent in any real world scenario brings lots of risk which must be carefully navigated. For example a system planning operations of an autonomous car may produce a plan which is unsafe to other road users, pedestrians, or property in its vicinity. To combat this extensive testing must be done to establish the safety of the system. A similar principle applies to the types of planning system we utilised in this project. The correctness of any planning agent using a new strategy must be verified in a manner similar to that covered in Section 2.2.1. The domain defined for the planner to operate in must also be constructed in an effective and valid manner which ensures the safety of humans, assets, and information. For example a system planning the path of a robot should not assume that any humans in the way will move in time for an oncoming robot.

7.3 Conclusion

Section 5.1 details both the functional and non-functional requirements of this project. When we consider the functional requirements laid forth we can explain the ways in which each requirement is satisfied. To satisfy functional requirement *1* we designed, implemented, and utilised the plan verification format presented in Section 2.2.1. Satisfaction of requirement *2* is evident through the development of the pruning method discussed in Section 5.3.2. Requirement *3* is satisfied in Chapter 5 and Section 6.1 where we discussed the design and implementation of the *Novelty* strategies. Lastly, requirement *4* is tackled by the autonomous process outlined in Section 5.3.7.

The main non-functional requirements of *Document the findings of using the Novelty search strategy for HTN planning problems* and *Verify the correctness of plans*. Were tackled in Chapter 6 as concrete evidence of satisfaction. Pointing to such evidence is trickier for the remaining non-functional requirements. But we can discuss steps taken to respect the intentions of each requirement. Throughout this report we have referred to the appendices for extra content on software engineering designs which are not central to this report, ensuring that we *Document Additions Well*. Across the developed code base *Errors are Handled* with meaningful warnings and error messages which aim to effectively inform any issues encountered. Overall, we can be satisfied that the work developed *Builds Upon EPICpy in the manner it was intended for* since all features are developed using interchangeable components which are a key cornerstone to the design of *EPICpy*.

In conclusion, in this work we have presented a new wave of functionality to the *EPICpy* planner and have discussed gathered results thoroughly with the purpose of revealing correlations in the data. The consideration of these findings are useful when considering the design of new strategies for *HTN* planning problems. Some of the technical work implemented in the scope of this project provides a starting point for future work, particularly the performance of the *Plan Verification Output* components discussed in Section 6.4.

HTN planning is an area of research which is lucky to have an proactive community and defined methods of evaluating, comparing, and testing work produced in the form of the *IPC*. We hope the work done in this project makes a start in bringing *EPICpy* closer to the planning community as well as providing insight for the development of future planning agents for competing in the *IPC* or addressing any real world application.

7.4 Future Work

In Section 6.3.3 we identified some domains where all of the strategies developed for *EPICpy* record poor results. Deeper analysis of these domains and the issues they cause for *EPICpy* could prove some interesting information and inform future optimisations of the planner.

In Chapter 6 we presented results for the effectiveness of *Novelty*. The presented information provides a firm basis for the qualities future developed strategies should posses.

Section 6.3.1 highlighted interesting problems where all strategies recorded the same values for a selection of attributes. This would suggest that in such problems, the search strategy being used by the planner is not making a large — if any — difference to the outcome of search. A deeper analysis and understanding of this would make for a good future addition to the work done in this project.

Section 6.3 described the issue of strategies taking considerably longer than the allotted solve time for some problems. This is because the planner would only terminate at specified intervals opposed to being interrupted by the kernel. This was necessary since for the purposes of this project since, the recorded attributes for problems which were not solved within the time limit were sought after. In future work an approach to terminate the process from the kernel could be beneficial if all data for unsolved problems is not required.

Appendix A

Design Appendix

A.1 Progress Tracker Structure

Since EPICpy is designed to be extendable and have interchangeable components we can begin by designing some components that we can switch from the defaults for the purposes of verifying plans with the plan verification module. For this requirement we propose a modification to the existing code. The existing code has the *Model* class (discussed in Section 4.1.1) monitoring the state of the environment as well as keeping track of the decompositions taken during search. As such we propose to separate these responsibilities into two components, the *Model* will remain in control of representing the state of the environment during search but it will delegate the responsibility of tracking the progress of search to a new type of component.

We call the new type of component a *Progress Tracker*. Similarly to that of *Heuristics* and *Parameter Selectors* (discussed in Section 4.1.2) we want the *Progress Tracker* component to be interchangeable. To do so we follow the design pattern used by the existing interchangeable components of EPICpy. This pattern consists of defining a skeleton class which outlines the methods that must be implemented by any child classes using Python's @*abstractmethod* decorator which is included in the built-in *abc*[1] library. This decorator forces sub-classes of the skeleton class to have an implementation of the decorated methods.

Figure A.1 illustrates the structure of the *Progress Tracker* components. The class *ProgressTracker*, is the skeleton class that all progress trackers to be used with the planner must be a subclass of. All of the methods in this class are decorated with the @*abstractmethod* decorator, as they are the methods which are required by the solving algorithm and heuristics.

The second class we see in Figure A.1 is the *SequentialTracker* class which inherits the *ProgressTracker* class. The behaviour of this class is designed to replicate that of the *Model* class, which simply tracks which Tasks, Methods, and Action have been used by the planner and in what order.

The final tracker class which is shown in Figure A.1 is the *PandaVerifyFormatTracker* class. This is the class which contains the new functionality required for the plan verification module. From the figure, it is glaringly obvious that this class contains more methods than the previously discussed class. This is because this model works in a significantly different way to the *SequentialTracker* class. Whereby the *SequentialTracker* class simply tracks what operations the planner has carried out, the *PandaVerifyFormatTracker* class must be aware of how additions to the task

[1] https://docs.python.org/3/library/abc.html

network are related to previous operations. Due to the nature of this difference, the *PandaVerifyFormatTracker* is only compatible with a specific Model class - the *PandaVerifyModel* class explored in Section A.2.

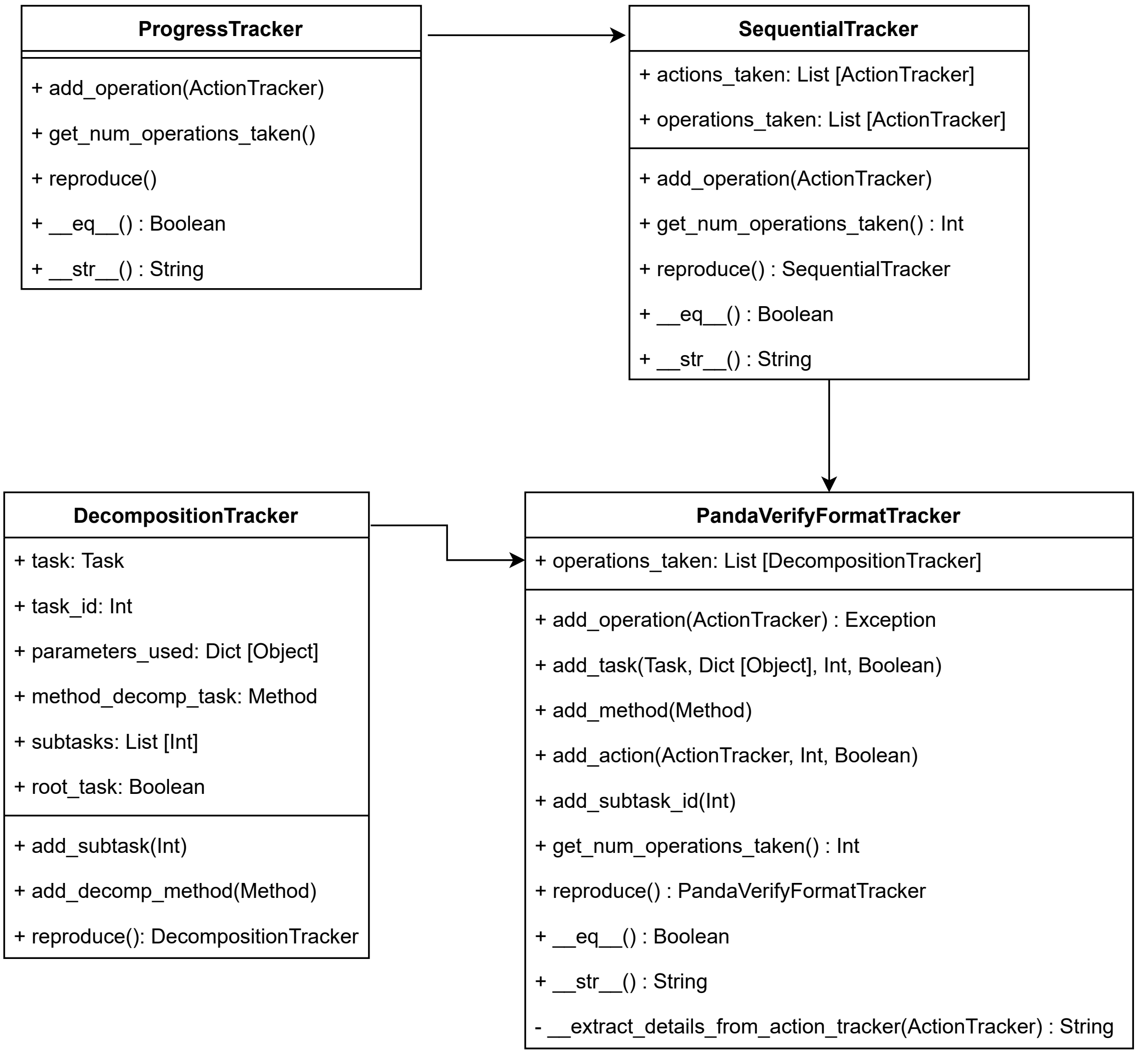

ProgressTracker
+ add_operation(ActionTracker)
+ get_num_operations_taken()
+ reproduce()
+ __eq__() : Boolean
+ __str__() : String
SequentialTracker
+ actions_taken: List [ActionTracker]
+ operations_taken: List [ActionTracker]
+ add_operation(ActionTracker)
+ get_num_operations_taken() : Int
+ reproduce() : SequentialTracker
+ __eq__() : Boolean
+ __str__() : String
DecompositionTracker
+ task: Task
+ task_id: Int
+ parameters_used: Dict [Object]
+ method_decomp_task: Method
+ subtasks: List [Int]
+ root_task: Boolean
+ add_subtask(Int)
+ add_decomp_method(Method)
+ reproduce(): DecompositionTracker
PandaVerifyFormatTracker
+ operations_taken: List [DecompositionTracker]
+ add_operation(ActionTracker) : Exception
+ add_task(Task, Dict [Object], Int, Boolean)
+ add_method(Method)
+ add_action(ActionTracker, Int, Boolean)
+ add_subtask_id(Int)
+ get_num_operations_taken() : Int
+ reproduce() : PandaVerifyFormatTracker
+ __eq__() : Boolean
+ __str__() : String
- __extract_details_from_action_tracker(ActionTracker) : String

Figure A.1: Structure of Progress Trackers

The final consideration we need to give to the structure of *Progress Trackers* is how we can make them interchangeable within the planner. To do this we will follow the existing pattern for interchangeable components adopted by EPICpy. This will require adding some additional arguments to the *Runner* class to allow for the selection of *Progress Trackers* classes.

A.2 Modifying Model Structure

In Section A.1 we discussed the needs of a new method of tracking the progress of the planner for a given model. We also formulated a new class with the ability to track the planners progress appropriately given the requirements of the plan verification module. To compliment this new *Progress Tracker* we need to devise a new model class. The new model class must encompass all the functionality of the existing model class as well as having knowledge of the Task, Method, or Action previously yielded to the solving algorithm during search. On top of this, knowledge of what *Method* added Tasks and Actions to the task network is essential. Lastly, the new model needs the ability to assign unique id's to tasks and actions during search. These are all of the functionality that needs to be encompassed by the new *PandaVerifyModel* class.

The first addition we will discuss is that of assigning every Task and Action a unique id value. We propose to accomplish this by devising a new structure for the *PandaVerifyModel* to use when storing subtasks in the task network. Under the current *Model* class the task network is represented by an ordered list of subtasks. Since we want to add id's to this list, there is two clear choices. The first is to use a tuple for each item in the list, the second is to create a small nested class to handle this addition. For the purposes of this project we will opt to create a small nested class to handle this requirement.

The aforementioned changes pose some issues since existing heuristics interact with the model and changing the inner-workings of the model with a new edition would pose some integration issues. To mitigate these issues we propose a new structure to dictate which functionality must be implemented by a model. This structure follows the same pattern as used by other interchangeable components in EPICpy as previously outlined in Section A.1. Figure A.2 illustrates the newly proposed structure for the *Model* class. From this figure we can see a class named *Model* this is the skeleton class that all further model classes must inherit from. The methods decorated with the *@abstractmethod* decorator in this class are as follows: insert_modifier, get_next_modifier, add_operation, promote_waiting_subtask, get_search_modifier, reproduce, and get_names_of_task_network_modifiers. The other methods defined have set implementations but, these implementations can be overwritten by sub classes.

To provide the functionality currently offered by the existing *Model* class, we have the *DefaultModel* class visible in Figure A.2. This class operates as the existing model in that, it has no knowledge of which method added which tasks and actions to the task network.

The final model class drawn out in Figure A.2 is the new *PandaVerifyModel*, this class will operate with the *PandaVerifyFormatTracker* component unveiled in Section A.1. This class has some new attributes, the first being the *id_counter* which simply begins at 0 and issues its value as the id for tasks and actions. After it issues an id value, the variable increments by one. The other new attribute is the *last_dispense* attribute which is what gives this model knowledge over which method added tasks and actions to the task network. When a task or action is added to the task network, we know that the method which is stored in the attribute *last_dispense* is the one that added it to the task network. This forms an essential part of the plan verification output format. Unlike other model classes, we do not allow for the user to choose which *progress tracker* component to use in conjunction with this model class instead, we force the use of the *PandaVerifyFormatTracker* class. The final difference with this class compared to the previously mentioned *DefaultModel* class is that the tasks and actions in the task network (*search_modifiers*) are stored in the type *PandaVerifyTaskNetworkNode*. This new class is a simple one that stores the *Subtask* to be handled by the planner along with an assigned *ID*.

Similarly to the *Progress Tracker* structure laid out in Section A.1 we will need to add some arguments to the *Runner* class to handle the selection of *Model* classes.

A.3 Pruning Seen States

Designing this functionality is quite simple since we can utilise Python's built-in *hash* [2] functionality. At the high-level of design we need to hash the pairings of state and task network, this is simple as we can hash the two components in the form of a tuple. When we consider the design in more detail we unveil four components which require hashing for this functionality. Hashing user defined classes can be done by setting the magic method *__hash__*. These hash values can be stored in a *set* [3] which can then be used to check for future occurances of the same pairing.

We must consider the method we use to gather components which need to be hashed together. This is because when we hash a list or a tuple the ordering of elements matters whereas, ordering does not matter when hashing a *frozenset* [3]. When hashing the facts in a state we do not want ordering to make an impact whereas, we do want ordering to matter when hashing the objects given to facts within the state.

A.4 Novelty

When we consider the options for how long a state should stay novel, we conclude that we require an interchangeable component in order to efficiently evaluate the search procedure provided by each option. To accomplish this we propose a design for a new solving algorithm which can work in conjunction with a modified *State* (*StateNovelty*) class to determine when a state is novel - and to which novelty level. The modified solving algorithm class (*PartialOrderNoveltySolver*) receives a novelty score from the *StateNovelty* class and passes it on to a new search queue class. This new search queue receiving the novelty level of and treats it like a heuristic estimate. As default the *PartialOrderNoveltySolver* class resets a states novelty score to 0 after a task of method is decomposed. To compliment this we design another class *PartialOrderNoveltyNoResetSolver* which maintains the novelty score of a state until an action is executed, this simply inherits *PartialOrderNoveltySolver* and overwrites the *add_model_to_search_queue* method to maintain the novelty level already assigned to a state. Figure A.3 illustrates the structure of these additions.

By default the maximum level of novelty searched with is 1 which only checks if a fact is new. To enable search of a higher maximum level we will need to introduce new solvers which inherit the aforementioned *PartialOrderNoveltySolver* and set the *max_novelty_level* attribute to the desired value.

[2] https://docs.python.org/3/library/functions.html#hash

[3] https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

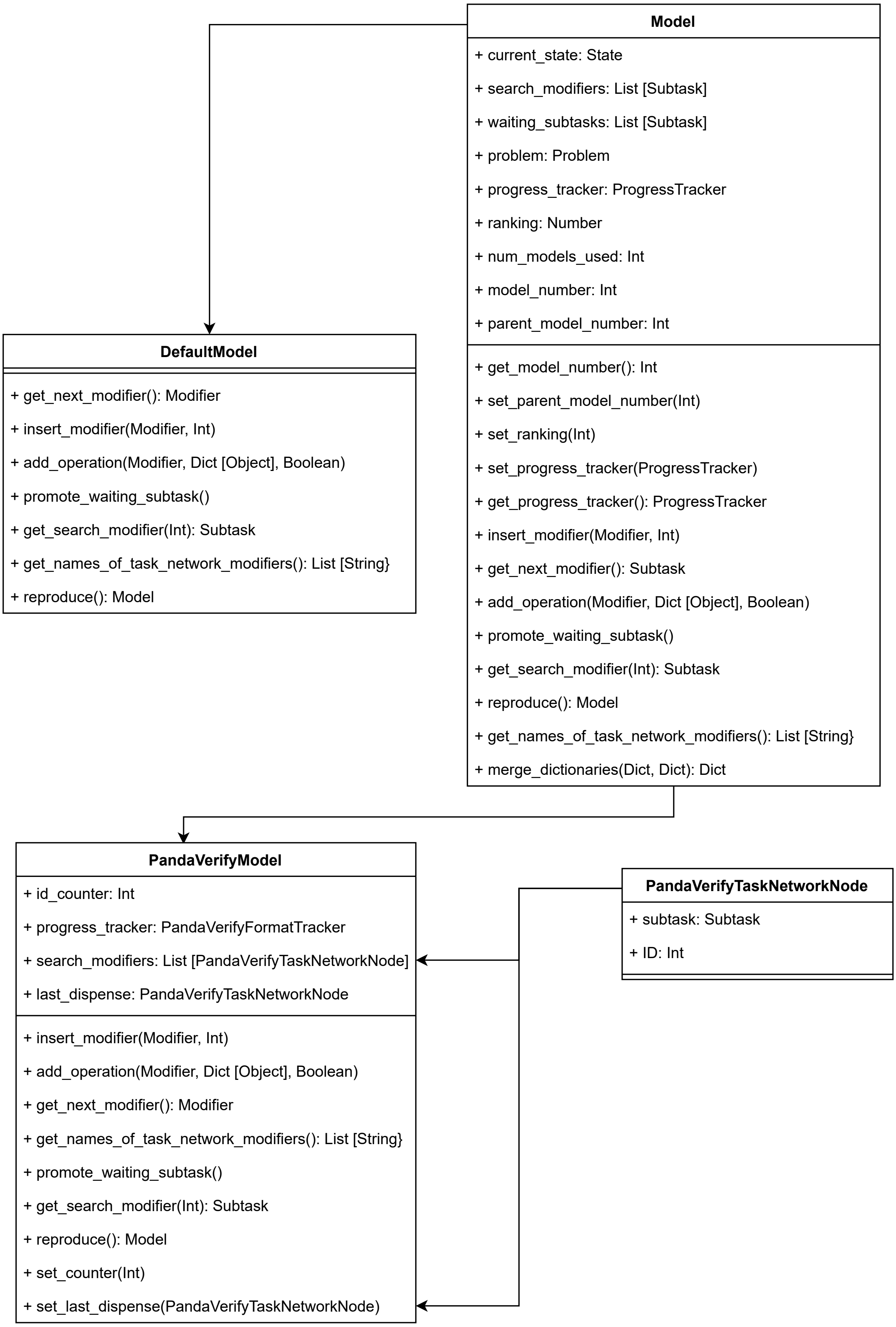

Model
+ current_state: State
+ search_modifiers: List [Subtask]
+ waiting_subtasks: List [Subtask]
+ problem: Problem
+ progress_tracker: ProgressTracker
+ ranking: Number
+ num_models_used: Int
+ model_number: Int
+ parent_model_number: Int
+ get_model_number(): Int
+ set_parent_model_number(Int)
+ set_ranking(Int)
+ set_progress_tracker(ProgressTracker)
+ get_progress_tracker(): ProgressTracker
+ insert_modifier(Modifier, Int)
+ get_next_modifier(): Subtask
+ add_operation(Modifier, Dict [Object], Boolean)
+ promote_waiting_subtask()
+ get_search_modifier(Int): Subtask
+ reproduce(): Model
+ get_names_of_task_network_modifiers(): List [String}
+ merge_dictionaries(Dict, Dict): Dict
DefaultModel
+ get_next_modifier(): Modifier
+ insert_modifier(Modifier, Int)
+ add_operation(Modifier, Dict [Object], Boolean)
+ promote_waiting_subtask()
+ get_search_modifier(Int): Subtask
+ get_names_of_task_network_modifiers(): List [String}
+ reproduce(): Model
PandaVerifyModel
+ id_counter: Int
+ progress_tracker: PandaVerifyFormatTracker
+ search_modifiers: List [PandaVerifyTaskNetworkNode]
+ last_dispense: PandaVerifyTaskNetworkNode
+ insert_modifier(Modifier, Int)
+ add_operation(Modifier, Dict [Object], Boolean)
+ get_next_modifier(): Modifier
+ get_names_of_task_network_modifiers(): List [String}
+ promote_waiting_subtask()
+ get_search_modifier(Int): Subtask
+ reproduce(): Model
+ set_counter(Int)
+ set_last_dispense(PandaVerifyTaskNetworkNode)
PandaVerifyTaskNetworkNode
+ subtask: Subtask
+ ID: Int

Figure A.2: Structure of Models

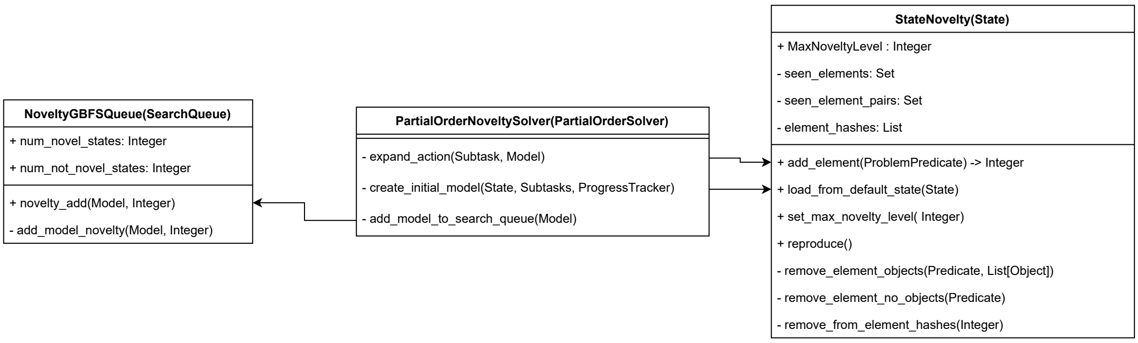

NoveltyGBFSQueue(SearchQueue)
+ num_novel_states: Integer
+ num_not_novel_states: Integer
+ novelty_add(Model, Integer)
- add_model_novelty(Model, Integer)
PartialOrderNoveltySolver(PartialOrderSolver)
- expand_action(Subtask, Model)
- create_initial_model(State, Subtasks, ProgressTracker)
- add_model_to_search_queue(Model)
StateNovelty(State)
+ MaxNoveltyLevel : Integer
- seen_elements: Set
- seen_element_pairs: Set
- element_hashes: List
+ add_element(ProblemPredicate) -> Integer
+ load_from_default_state(State)
+ set_max_novelty_level(Integer)
+ reproduce()
- remove_element_objects(Predicate, List[Object])
- remove_element_no_objects(Predicate)
- remove_from_element_hashes(Integer)

Figure A.3: Novelty Architecture

A.5 Collecting Results

Strategy	**Description**
Hamming-Newest	Hamming Distance with Newest First Ordering
Hamming-Oldest	Hamming Distance with Oldest First Ordering
Hamming-TreeDis	Hamming Distance With Tree Distance Tie Breaking
Hamming-Novelty-No-Reset	Hamming Novelty with no reset and Newest First Ordering
Hamming-Novelty-Reset	Hamming Novelty with reset and Newest First Ordering
Novelty-Level1-methods-No_Reset	Novelty Level 1 with Novelty Methods with No Reset and Newest First Ordering
Novelty-Level1-methods-Oldest	Novelty Level 1 with Novelty Methods with Reset and Oldest First Ordering
Novelty-Level1-methods-Reset	Novelty Level 1 with Novelty Methods with Reset and Newest First Ordering
Novelty-Level1-methods-Tasks-Oldest	Novelty Level 1 with Novelty Methods and Tasks with Reset and Oldest First Ordering
Novelty-Level1-methods-Tasks-Newest	Novelty Level 1 with Novelty Methods and Tasks with Reset and Newest First Ordering
Novelty-Methods-Only	Novelty Methods with Newest First Ordering
Novelty-Level1-No-Reset-HamDis	Novelty Level 1 with No Reset with Hamming Distance Tie Breaker
Novelty-Level1-No-Reset-TreeDis	Novelty Level 1 with No Reset with Tree Distance Tie Breaker
Novelty-Level1-No-Reset	Novelty Level 1 with No Reset with Newest First Ordering
Novelty-Level1-HamDis	Novelty Level 1 with Reset with Hamming Distance Tie Breaker
Novelty-Level1-TreeDis	Novelty Level 1 with Reset with Tree Distance Tie Breaker
Novelty-Level1-Reset-Newest	Novelty Level 1 with Reset with Newest First Ordering
Novelty-Level1-Reset-Oldest	Novelty Level 1 with Reset with Oldest First Ordering
Novelty-level2-No-Reset	Novelty Level 2 with No Reset and Newest First Ordering
Novelty-level2-Reset	Novelty Level 2 with Reset and Newest First Ordering
BFS	Breadth First Search (with seen states pruning) and Oldest First Ordering
Tree-Distance-Default	Tree Distance (with NO seen states pruning) and Oldest First Ordering
Tree-Distance-HamDis	Tree Distance with Hamming Distance Tie Breaker with Oldest First Ordering
Tree-Distance-Newest	Tree Distance and Newest First Ordering
Tree-Distance-Seen-States	Tree Distance (with seen states pruning) and Oldest First Ordering
Hamming-Landmarks	Hamming Distance with Landmarks Tie Breaker
Landmarks-Hamming	Landmarks with Hamming Distance Tie Breaker and Oldest First Ordering
Landmarks-Newest	Landmarks with Newest First Ordering
Landmarks-Oldest	Landmarks with Oldest First Ordering
Landmarks-Tree	Landmarks with Tree Distance Tie Breaker
Landmarks-No-Reachability	Landmarks with NO Method Reachability Oldest First Ordering
Novelty-Level1-No-Reset-Landmarks	Novelty Level 1 with No Reset and Landmarks Tie Breaker
Novelty-Level1-Landmarks	Novelty Level 1 with Reset and Landmarks Tie Breaker
Tree-Distance-Landmarks	Tree Distance with Landmarks Tie Breaker

Table A.1: List of Strategies and Descriptions

Attribute
number of expansions
solve time
setup time
amount of facts possible
amount of facts in final state
percentage of possible facts in final state
amount of possible fact pairings
amount of fact parings in final state
percentage of possible fact pairings in the final state
number of novel states
number of not novel states
percentage of states which were novel
number of unique facts seen
number of novel methods
number of not novel methods
if the solution is verified
if the problem was solved
FOR SELECT STRATEGIES WE ALSO COLLECT THE ATTRIBUTES BELOW:
number of novel methods belonging to not novel states
number of novel methods belonging to novel states

Table A.2: List of Attributes Collected

Appendix B

Testing and Evaluation Appendix

B.1 Verifying the Correctness of Existing EPICpy Components

Domain	Solver Heuristic Parameter Selection Search Queue	Partial Order No Pruning All Parameters A Star	Partial Order No Pruning Requirement Parameters A Star
Basic	pb1	TRUE	TRUE
Rover (IPC TEST)	pfile01	Took Too Long To Find Plan	TRUE
satellite01	1obs-1sat-1mod	TRUE	TRUE
transport01	pfile01	Took Too Long To Find Plan	Took Too Long To Find Plan
Um-translog01	problem	TRUE	TRUE
IPC Tests	test01_empty_method	TRUE	TRUE
	test02_forall	TRUE	TRUE
	test03_forall1	TRUE	TRUE
	test04_no_abstracts	TRUE	TRUE
	test05_constants_in_domain	TRUE	TRUE
	test06_synonymes	TRUE	TRUE
	test07_arguments	TRUE	TRUE
Rover	p01	Took Too Long To Find Plan	TRUE
	p02		TRUE
	p03		TRUE
	p04		Took Too Long To Find Plan
	p05		
	p06		
	p07		
	p08		
	p09		
	p10		

Table B.1: EPICpy Existing Components Output Verification - Part 1

Domain	Solver Heuristic Parameter Selection Search Queue	Partial Order Pruning Requirement Parameters A Star	Partial Order Tree Distance Requirement Parameters Greedy Cost
Basic	pb1	TRUE	TRUE
Rover (IPC TEST)	pfile01	TRUE	TRUE
satellite01	1obs-1sat-1mod	TRUE	TRUE
transport01	pfile01	Took Too Long To Find Plan	TRUE
Um-translog01	problem	TRUE	TRUE
IPC Tests	test01_empty_method	TRUE	TRUE
	test02_forall	TRUE	TRUE
	test03_forall1	TRUE	TRUE
	test04_no_abstracts	TRUE	TRUE
	test05_constants_in_domain	TRUE	TRUE
	test06_synonymes	TRUE	TRUE
	test07_arguments	TRUE	TRUE
Rover	p01	TRUE	TRUE
	p02	TRUE	TRUE
	p03	TRUE	TRUE
	p04	TRUE	TRUE
	p05	Took Too Long To Find Plan	TRUE
	p06		TRUE
	p07		TRUE
	p08		TRUE
	p09		TRUE
	p10		TRUE

Table B.2: EPICpy Existing Components Output Verification - Part 2

Domain	Solver Heuristic Parameter Selection Search Queue	Partial Order Tree Distance Partial Order Requirement Parameters Greedy Best First	Partial Order Hamming Distance Requirement Parameters Greedy Cost
Basic	pb1	TRUE	TRUE
Rover (IPC TEST)	pfile01	TRUE	TRUE
satellite01	1obs-1sat-1mod	TRUE	TRUE
transport01	pfile01	TRUE	Took Too Long To Find Plan
Um-translog01	problem	TRUE	TRUE
IPC Tests	test01_empty_method	TRUE	TRUE
	test02_forall	TRUE	TRUE
	test03_forall1	TRUE	TRUE
	test04_no_abstracts	TRUE	TRUE
	test05_constants_in_domain	TRUE	TRUE
	test06_synonymes	TRUE	TRUE
	test07_arguments	TRUE	TRUE
Rover	p01	TRUE	TRUE
	p02	TRUE	TRUE
	p03	TRUE	TRUE
	p04	TRUE	TRUE
	p05	TRUE	TRUE
	p06	TRUE	TRUE
	p07	TRUE	TRUE
	p08	TRUE	Took Too Long To Find Plan
	p09	TRUE	
	p10	TRUE	

Table B.3: EPICpy Existing Components Output Verification - Part 3

B.2 Correlations

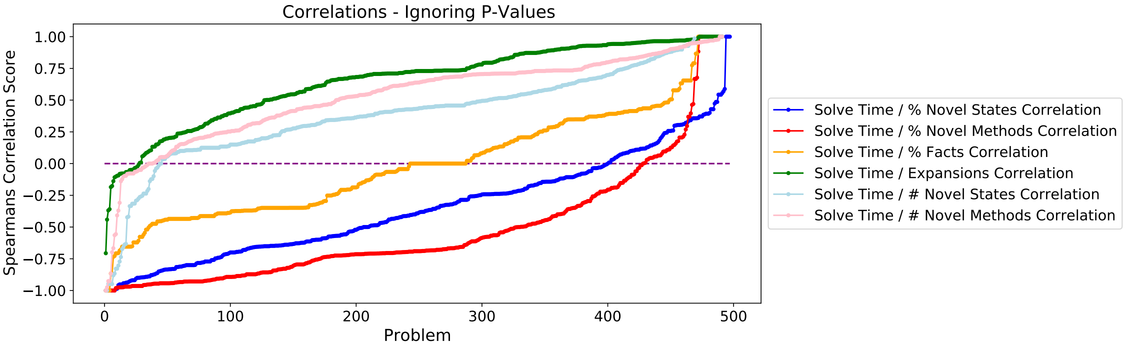


Figure B.1: Correlation between Solve Time and Attributes

B.3 Solve Time Graphs

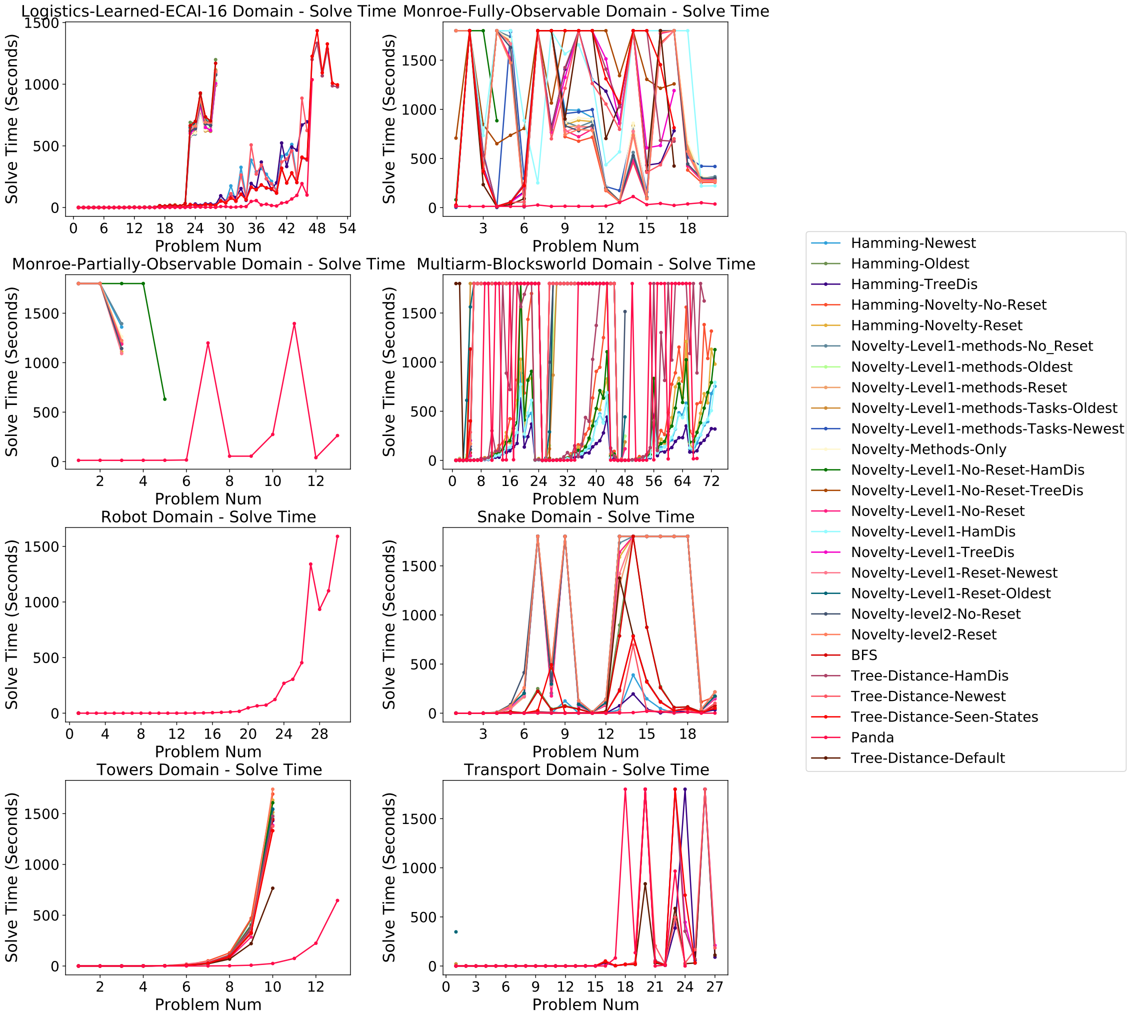


Figure B.2: Solve Time for Domains - 'Logistics-Learned-ECAI-16', 'Monroe-Fully-Observable', 'Monroe-Partially-Observable', 'Multiarm-Blocksworld', 'Robot', 'Snake', 'Towers', 'Transport'

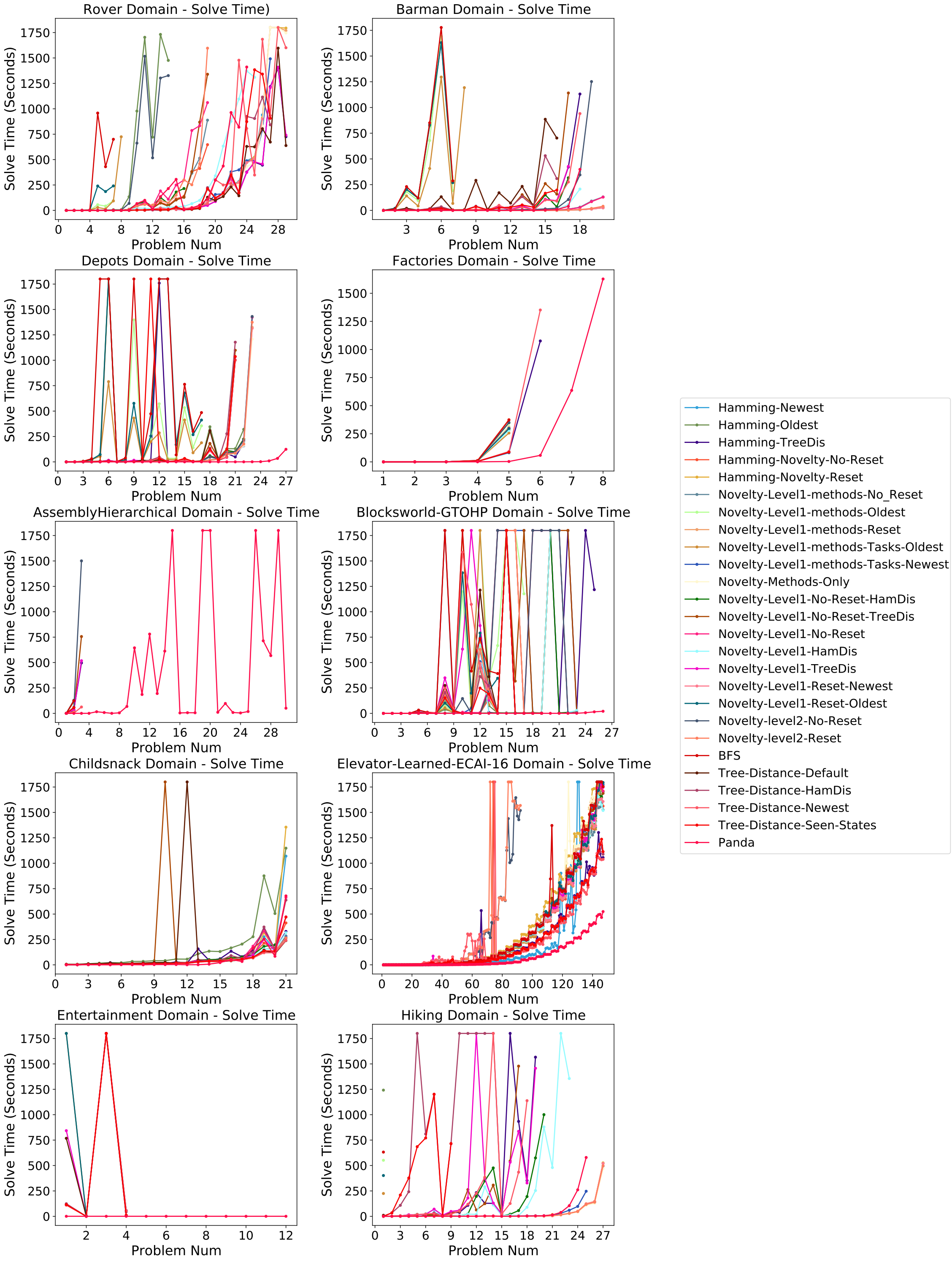


Figure B.3: Score Time for Domains - 'Rover', 'Barman', 'Depots', 'Factories', 'Assembly-Hierarchical', 'Blocksworld-GTOHP', 'Childsnack', 'Elevator-Learned-ECAI-16', 'Entertainment', 'Hiking'

B.4 IPC Score

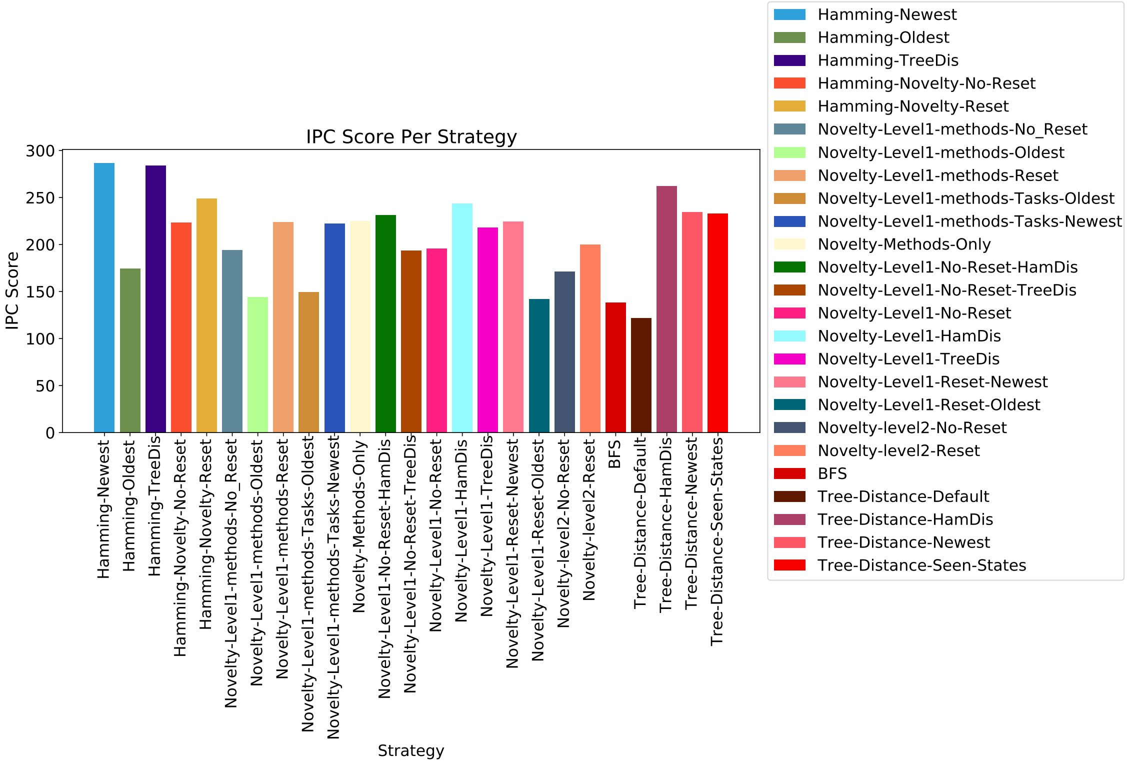
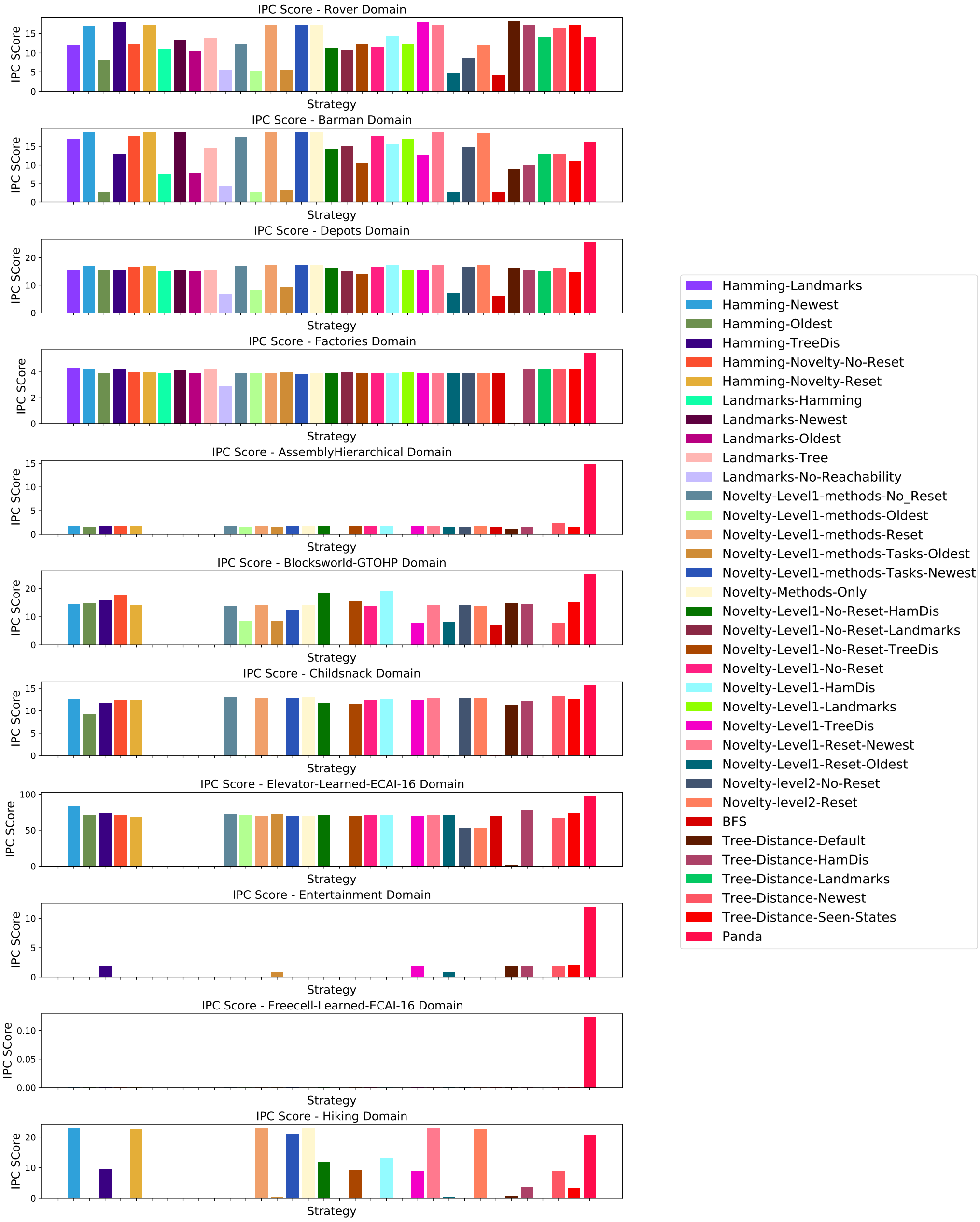


Figure B.4: Total IPC Score of Each Strategy - Excluding PANDA

B.5 Domain Specific IPC Scores



IPC Score - Rover Domain

IPC Score - Barman Domain

IPC Score - Depots Domain

IPC Score - Factories Domain

IPC Score - AssemblyHierarchical Domain

IPC Score - Blocksworld-GTOHP Domain

IPC Score - Childsnack Domain

IPC Score - Elevator-Learned-ECAI-16 Domain

IPC Score - Entertainment Domain

IPC Score - Freecell-Learned-ECAI-16 Domain

IPC Score - Hiking Domain

Figure B.5: IPC Score for Domains - ’Rover’, ’Barman’, ’Depots’, ’Factories’, ’AssemblyHierarchical’, ’Blocksworld-GTOHP’, ’Childsnack’, ’Elevator-Learned-ECAI-16’, ’Entertainment’, ’Freecell-Learned-ECAI-16’, ’Hiking’

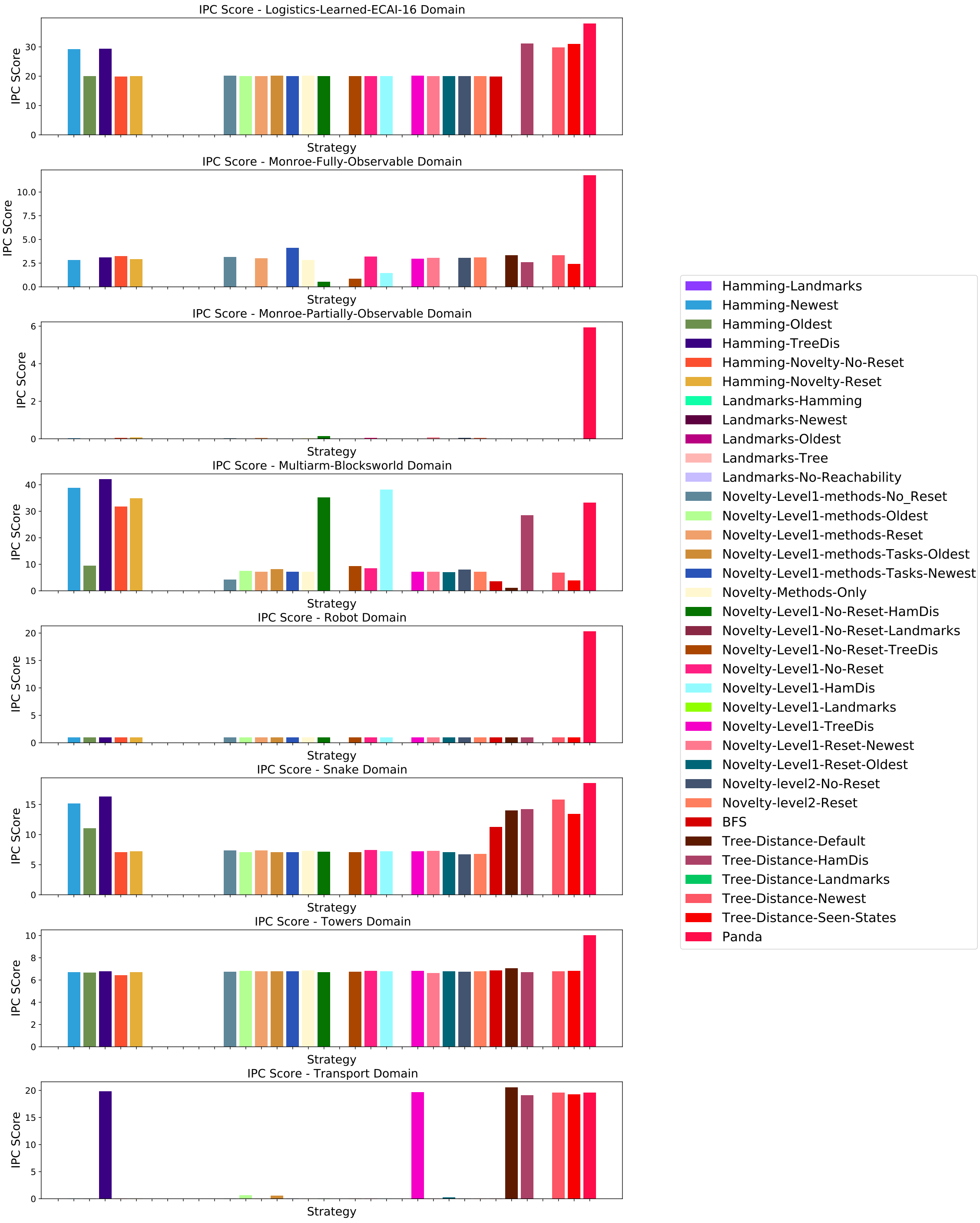


Figure B.6: IPC Score for Domains - ’Logistics-Learned-ECAI-16’, ’Monroe-Fully-Observable’, ’Monroe-Partially-Observable’, ’Multiarm-Blocksworld’, ’Robot’, ’Snake’, ’Towers’, ’Transport’

B.6 Domain Coverage

Domain	Hamming-Newest	Hamming-Oldest	Hamming-TreeDis
AssemblyHierarchical	2	2	3
Barman	20	7	18
Blocksworld-GTOHP	17	18	20
Childsnack	21	21	21
Depots	22	22	22
Elevator-Learned-ECAI-16	130	147	147
Factories	5	5	6
Hiking	27	1	18
Logistics-Learned-ECAI-16	44	28	46
Monroe-Fully-Observable	12	0	13
Monroe-Partially-Observable	1	0	0
Multiarm-Blocksworld	73	13	73
Robot	1	1	1
Rover	26	14	29
Snake	20	19	20
Towers	10	9	10
Entertainment	0	0	3
Freecell-Learned-ECAI-16	0	0	0
Transport	0	0	24
TOTAL-SOLVED	431	307	474

Table B.4: Domain Coverage - Hamming-Newest, Hamming-Oldest, Hamming-TreeDis

Domain	Hamming-Novelty-No-Reset	Hamming-Novelty-Reset
AssemblyHierarchical	2	2
Barman	20	20
Blocksworld-GTOHP	20	17
Childsnack	21	21
Depots	22	22
Elevator-Learned-ECAI-16	146	146
Factories	5	5
Hiking	0	27
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	14	13
Monroe-Partially-Observable	1	1
Multiarm-Blocksworld	70	73
Robot	1	1
Rover	19	27
Snake	12	13
Towers	10	10
Entertainment	0	0
Freecell-Learned-ECAI-16	0	0
Transport	0	0
TOTAL-SOLVED	391	426

Table B.5: Domain Coverage - Hamming-Novelty-No-Reset, Hamming-Novelty-Reset

Domain	Novelty-Level1-methods-No_Reset	Novelty-Level1-methods-Oldest
AssemblyHierarchical	2	2
Barman	20	7
Blocksworld-GTOHP	15	14
Childsnack	21	0
Depots	23	16
Elevator-Learned-ECAI-16	147	145
Factories	5	5
Hiking	0	1
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	13	0
Monroe-Partially-Observable	1	0
Multiarm-Blocksworld	5	11
Robot	1	1
Rover	19	7
Snake	13	12
Towers	9	10
Entertainment	0	0
Freecell-Learned-ECAI-16	0	0
Transport	0	1
TOTAL-SOLVED	322	260

Table B.6: Domain Coverage - Novelty-Level1-methods-No_Reset, Novelty-Level1-methods-Oldest

Domain	Novelty-Level1-methods-Reset	Novelty-Level1-methods-Tasks-Oldest
AssemblyHierarchical	2	2
Barman	20	8
Blocksworld-GTOHP	17	11
Childsnack	21	0
Depots	23	17
Elevator-Learned-ECAI-16	142	145
Factories	5	5
Hiking	27	1
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	13	0
Monroe-Partially-Observable	1	0
Multiarm-Blocksworld	9	11
Robot	1	1
Rover	27	8
Snake	13	12
Towers	9	10
Entertainment	0	1
Freecell-Learned-ECAI-16	0	0
Transport	0	1
TOTAL-SOLVED	358	261

Table B.7: Domain Coverage - Novelty-Level1-methods-Reset, Novelty-Level1-methods-Tasks-Oldest

Domain	Novelty-Level1-methods-Tasks-Newest	Novelty-Methods-Only
AssemblyHierarchical	2	2
Barman	20	20
Blocksworld-GTOHP	16	17
Childsnack	21	21
Depots	23	23
Elevator-Learned-ECAI-16	143	141
Factories	5	5
Hiking	25	27
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	15	13
Monroe-Partially-Observable	0	1
Multiarm-Blocksworld	9	9
Robot	1	1
Rover	27	27
Snake	12	13
Towers	9	9
Entertainment	0	0
Freecell-Learned-ECAI-16	0	0
Transport	0	0
TOTAL-SOLVED	356	357

Table B.8: Domain Coverage - Novelty-Level1-methods-Tasks-Newest, Novelty-Methods-Only

Domain	Novelty-Level1-No-Reset-HamDis	Novelty-Level1-No-Reset-TreeDis
AssemblyHierarchical	2	3
Barman	17	17
Blocksworld-GTOHP	21	18
Childsnack	20	19
Depots	22	20
Elevator-Learned-ECAI-16	147	145
Factories	5	5
Hiking	20	17
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	2	10
Monroe-Partially-Observable	1	0
Multiarm-Blocksworld	72	11
Robot	1	1
Rover	16	19
Snake	12	12
Towers	10	10
Entertainment	0	0
Freecell-Learned-ECAI-16	0	0
Transport	0	0
TOTAL-SOLVED	396	335

Table B.9: Domain Coverage - Novelty-Level1-No-Reset-HamDis, Novelty-Level1-No-Reset-TreeDis

Domain	Novelty-Level1-No-Reset	Novelty-Level1-HamDis
AssemblyHierarchical	2	2
Barman	20	18
Blocksworld-GTOHP	17	21
Childsnack	21	21
Depots	23	22
Elevator-Learned-ECAI-16	147	147
Factories	5	5
Hiking	0	22
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	14	10
Monroe-Partially-Observable	1	0
Multiarm-Blocksworld	10	73
Robot	1	1
Rover	19	25
Snake	13	12
Towers	9	10
Entertainment	0	0
Freecell-Learned-ECAI-16	0	0
Transport	0	0
TOTAL-SOLVED	330	417

Table B.10: Domain Coverage - Novelty-Level1-No-Reset, Novelty-Level1-HamDis

Domain	Novelty-Level1-TreeDis	Novelty-Level1-Reset-Newest
AssemblyHierarchical	3	2
Barman	17	20
Blocksworld-GTOHP	12	17
Childsnack	21	21
Depots	21	23
Elevator-Learned-ECAI-16	147	147
Factories	5	5
Hiking	18	27
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	12	14
Monroe-Partially-Observable	0	1
Multiarm-Blocksworld	9	9
Robot	1	1
Rover	29	26
Snake	12	13
Towers	10	9
Entertainment	3	0
Freecell-Learned-ECAI-16	0	0
Transport	24	0
TOTAL-SOLVED	372	363

Table B.11: Domain Coverage - Novelty-Level1-TreeDis, Novelty-Level1-Reset-Newest

Domain	Novelty-Level1-Reset-Oldest	Novelty-level2-No-Reset
AssemblyHierarchical	2	3
Barman	7	19
Blocksworld-GTOHP	14	16
Childsnack	0	21
Depots	14	23
Elevator-Learned-ECAI-16	145	92
Factories	5	5
Hiking	1	0
Logistics-Learned-ECAI-16	28	28
Monroe-Fully-Observable	0	13
Monroe-Partially-Observable	0	1
Multiarm-Blocksworld	11	11
Robot	1	1
Rover	7	14
Snake	12	12
Towers	10	9
Entertainment	1	0
Freecell-Learned-ECAI-16	0	0
Transport	1	0
TOTAL-SOLVED	259	268

Table B.12: Domain Coverage - Novelty-Level1-Reset-Oldest, Novelty-level2-No-Reset

Domain	Novelty-level2-Reset	BFS	Tree-Distance-Default
AssemblyHierarchical	2	2	1
Barman	20	7	16
Blocksworld-GTOHP	17	12	18
Childsnack	21	0	20
Depots	23	12	21
Elevator-Learned-ECAI-16	88	143	2
Factories	5	5	0
Hiking	27	1	1
Logistics-Learned-ECAI-16	28	28	0
Monroe-Fully-Observable	14	0	10
Monroe-Partially-Observable	1	0	0
Multiarm-Blocksworld	9	5	1
Robot	1	1	1
Rover	19	7	29
Snake	12	19	20
Towers	10	10	10
Entertainment	0	0	3
Freecell-Learned-ECAI-16	0	0	0
Transport	0	0	26
TOTAL-SOLVED	297	252	179

Table B.13: Domain Coverage - Novelty-level2-Reset, BFS, Tree-Distance-Default

Domain	Tree-Distance-HamDis	Tree-Distance-Newest	Tree-Distance-Seen-States
AssemblyHierarchical	2	3	2
Barman	16	18	16
Blocksworld-GTOHP	18	13	18
Childsnack	21	21	21
Depots	21	21	20
Elevator-Learned-ECAI-16	147	146	147
Factories	5	6	5
Hiking	9	17	9
Logistics-Learned-ECAI-16	52	46	52
Monroe-Fully-Observable	11	13	9
Monroe-Partially-Observable	0	0	0
Multiarm-Blocksworld	54	9	5
Robot	1	1	1
Rover	27	28	27
Snake	20	20	20
Towers	10	10	10
Entertainment	3	3	3
Freecell-Learned-ECAI-16	0	0	0
Transport	23	25	23
TOTAL-SOLVED	440	400	388

Table B.14: Domain Coverage - Tree-Distance-HamDis, Tree-Distance-Newest, Tree-Distance-Seen-States

B.7 Analysis of Novelty Strategies

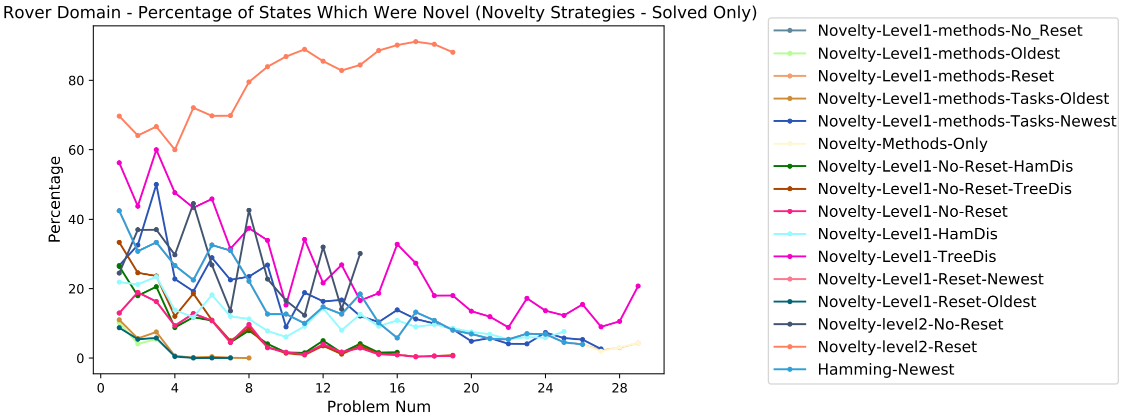


Figure B.7: Percentage of Novel States - Rover Domain (Novelty Strategies)

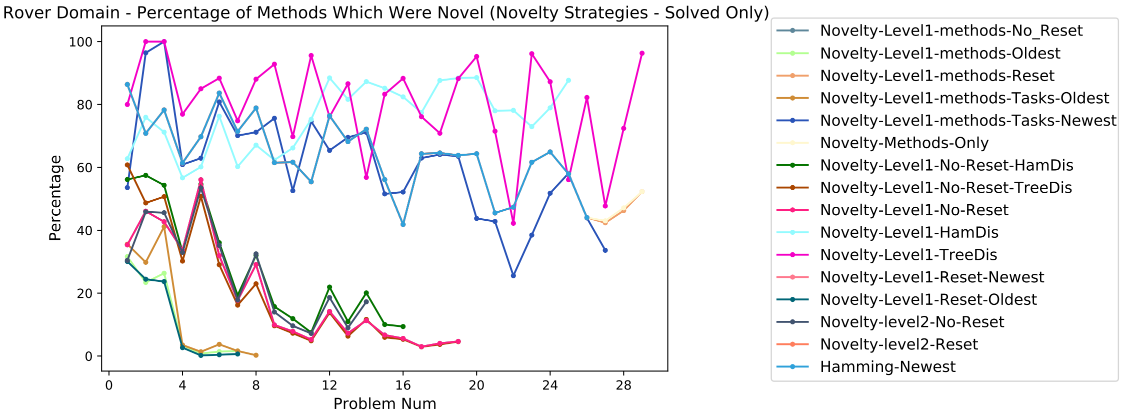


Figure B.8: Percentage of Novel Methods - Rover Domain (Novelty Strategies)

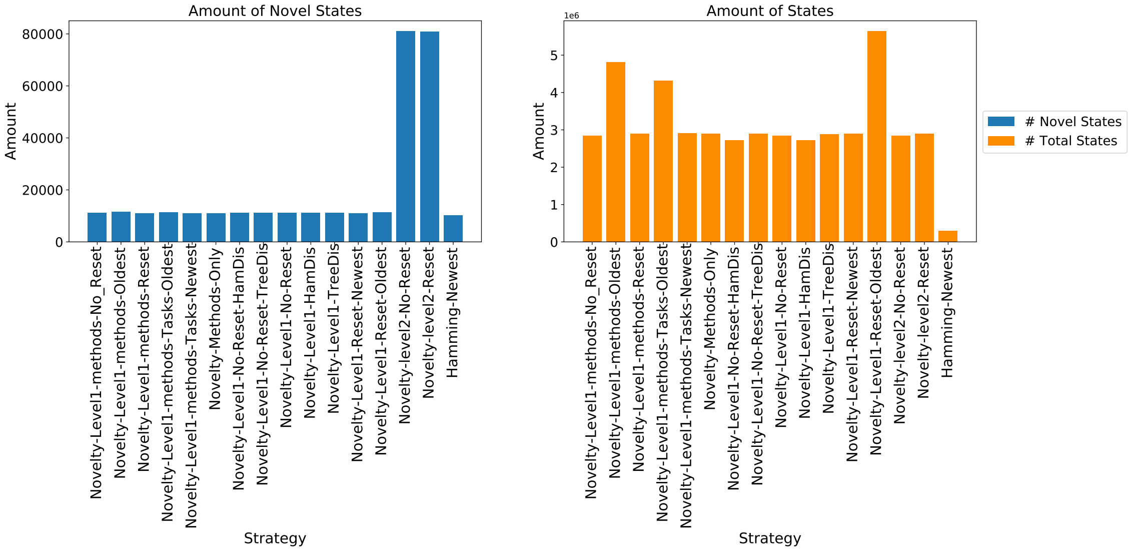


Figure B.9: Amount of Novel States and Total Amount of States (Novelty Strategies)

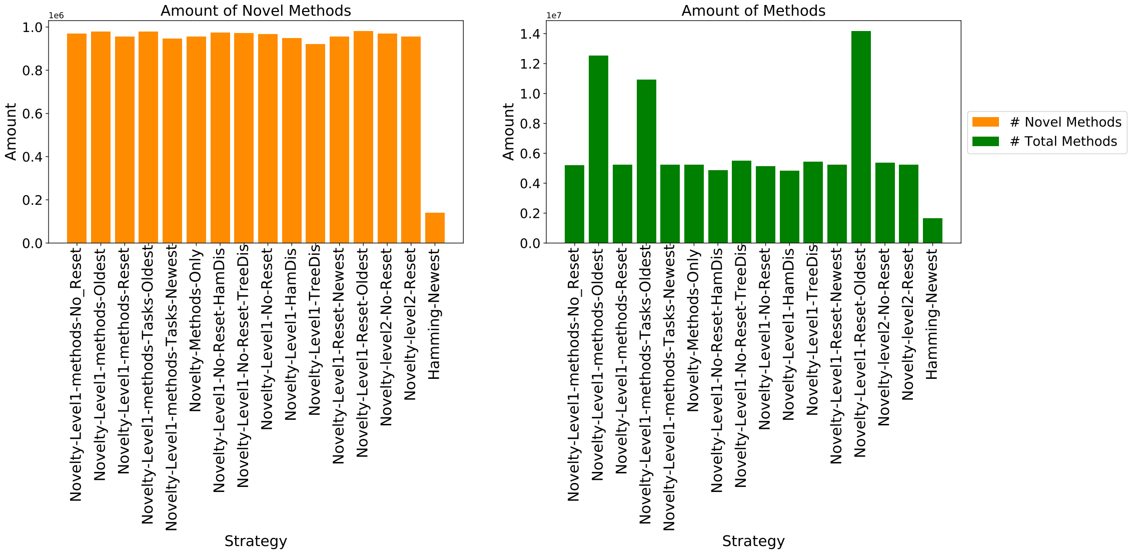


Figure B.10: Amount of Novel States and Total Amount of Methods (Novelty Strategies)

B.8 Landmarks Results

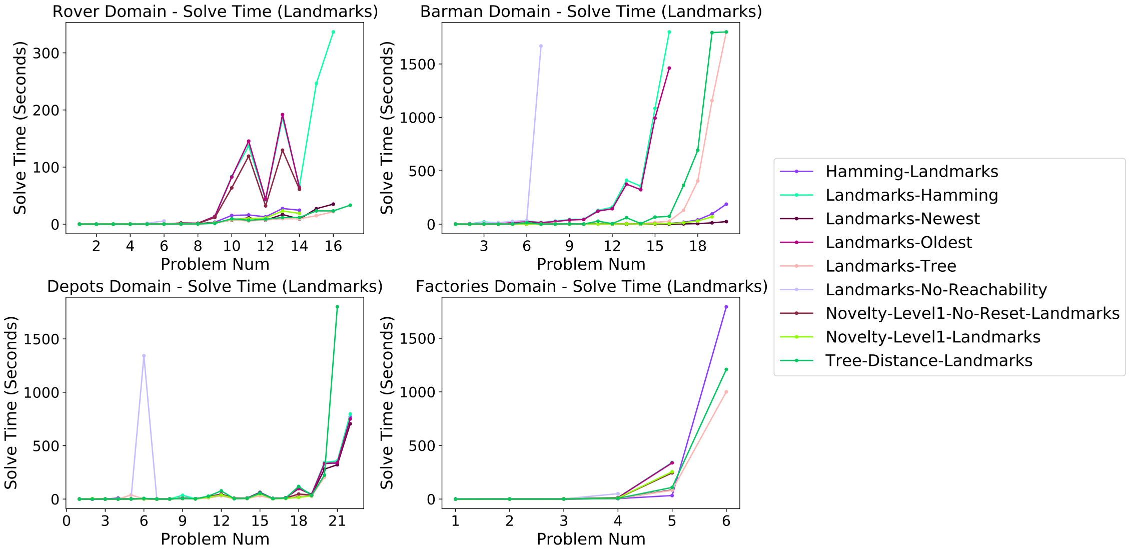


Figure B.11: All Landmarks Solve Times

Appendix C

User Manual

C.1 Using the Planner

The planner can be set up and used with one of two methods.

C.1.1 Command Line Usage

The first way to use the planner is from the command line using the command:

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

A help menu is available using the prompt:

python ./Runner –help

C.1.2 Integrating with other software

The second way to operate the planner is in conjunction with another Python program. The planner can be imported following this example:

```
from runner import Runner

controller = Runner(domain_path, problem_path)
controller.parse_problem()
controller.parse_problem()

result = controller.solve()
```

C.2 Selecting Custom Components

EPICpy has six types of interchangeable components these are: Solving Algorithms, Heuristic, Parameter Selection, Search Queue, Progress Tracker, and Model.

The following sections show the details of each interchangeable component which is available in EPICpy. We then show how to select these components using the two methods of interacting with EPICpy previously shown.

C.2.1 Solving Algorithms

-solverModName	-solverPath
TotalOrderSolver	Solver/Solving_Algorithms/total_order.py
PartialOrderSolver	Solver/Solving_Algorithms/partial_order.py
PartialOrderHammingNoveltySolver	Solver/Solving_Algorithms/ partial_order_hamming_novelty.py
PartialOrderHammingNoveltyNoResetSolver	Solver/Solving_Algorithms/ partial_order_hamming_novelty_no_reset.py
PartialOrderNoveltySolver	Solver/Solving_Algorithms/ partial_order_novelty.py
PartialOrderNoveltyLevelTwoSolver	Solver/Solving_Algorithms/ partial_order_novelty_level_2.py
PartialOrderNoveltyLevelTwoNoResetSolver	Solver/Solving_Algorithms/ partial_order_novelty_level_2_no_reset.py
PartialOrderNoveltyLightSolver	Solver/Solving_Algorithms/ partial_order_novelty_light.py
PartialOrderNoveltyMethodsSolver	Solver/Solving_Algorithms/ partial_order_novelty_methods.py
PartialOrderNoveltyMethodsNoResetSolver	Solver/Solving_Algorithms/ partial_order_novelty_methods_no_reset.py
PartialOrderNoveltyMethodsOnlySolver	Solver/Solving_Algorithms/ partial_order_novelty_methods_only.py
PartialOrderNoveltyMethodsTasksSolver	Solver/Solving_Algorithms/ partial_order_novelty_methods_tasks.py
PartialOrderNoveltyNoResetSolver	Solver/Solving_Algorithms/ partial_order_novelty_no_reset.py

C.2.2 Heuristics

-heuModName	-heuPath
DeleteRelaxed	Solver/Heuristics/delete_relaxed.py
DeleteRelaxedPartialOrder	Solver/Heuristics/delete_relaxed_partial_order.py
HammingDistance	Solver/Heuristics/hamming_distance.py
HammingDistancePartialOrder	Solver/Heuristics/hamming_distance_partial_order.py
HammingDistanceSeenStatesPruning	Solver/Heuristics/hamming_distance_seen_states.py
Landmarks	Solver/Heuristics/landmarks.py
LandmarksNoReachability	Solver/Heuristics/landmarks_no_reachability.py
NoPruning	Solver/Heuristics/no_pruning.py
PartialOrderPruning	Solver/Heuristics/partial_order_pruning.py
Pruning	Solver/Heuristics/pruning.py
SeenStatesPruning	Solver/Heuristics/seen_states_pruning.py
TreeDistance	Solver/Heuristics/tree_distance.py
TreeDistancePartialOrder	Solver/Heuristics/tree_distance_partial_order.py
TreeDistanceSeenStatesPruning	Solver/Heuristics/tree_distance_seen_states.py

C.2.3 Parameter Selectors

-paramSelectName	-paramSelectPath
AllParameters	Solver/Parameter_Selection/All_Parameters.py
RequirementSelection	Solver/Parameter_Selection/Requirement_Selection.py
StateSelector	Solver/Parameter_Selection/StateSelector.py

C.2.4 Search Queues

-searchQueueName	-searchQueuePath
GBFSSearchQueue	Solver/Search_Queues/ Greedy_Best_First_Search_Queue.py
GBFSSearchQueueNewestFirst	Solver/Search_Queues/ Greedy_Best_First_Search_Queue_Newest_First.py
GreedyCostSearchQueue	Solver/Search_Queues/ Greedy_Cost_So_Far_Search_Queue.py
HeuNoveltyGBFSQueue	Solver/Search_Queues/ Heu_Novelty_GBFS_Queue.py
NoveltyGBFSQueue	Solver/Search_Queues/ Novelty_GBFS_Search_Queue.py
NoveltyGBFSOldestFirstQueue	Solver/Search_Queues/ Novelty_GBFS_Search_Queue_Oldest_First.py
NoveltyTreeDistanceGBFSSearchQueue	Solver/Search_Queues/ Novelty_TreeDistance_GBFS_Search_Queue.py
SearchQueueGBFSDualHammingDistance	Solver/Search_Queues/ search_queue_dual_heuristic_HammingDistance.py
SearchQueueGBFSDualLandmarks	Solver/Search_Queues/ search_queue_dual_heuristic_Landmarks.py
SearchQueueGBFSDualTreeDistance	Solver/Search_Queues/ search_queue_dual_heuristic_TreeDistance.py
SearchQueueNewestFirst	Solver/Search_Queues/ search_queue_newest_first.py

C.2.5 Progress Trackers

-paramSelectName	-paramSelectPath
SequentialTracker	Solver/Parameter_Selection/sequential_progress_tracker.py
PandaVerifyFormatTracker	Solver/Parameter_Selection/panda_verify_format.py

C.2.6 Models

-modelModName	-modelPath
DefaultModel	Solver/Models/default_model.py
PandaVerifyModel	Solver/Models/PandaVerifyModel.py

C.2.7 Selecting From Command Line

The above listed components can be selected from the command line using the following command line format:

python ./Runner.py <Domain File Path> <Problem File Path> -searchQueueName <Search Queue Class Name> - searchQueuePath <Search Queue File Path>

In this example only one component is being selected but more can be added by simply adding more arguments in the same style. The argument flags for each component type are given in the column headers of the tables above.

C.2.8 Selecting From Import Module

The following example shows how to select components when using the planner within another Python program.

```
controller.set_solver(<Solving Algorithm Class>)
controller.set_heuristic(<Heuristic Class>)
controller.set_parameter_selector(<Parameter Selection Class>)
controller.set_search_queue(<Search Queue Class>)
controller.set_progress_tracker(<Progress Tracker Class>)
contreoller.set_model(<Model Class>)
```

C.3 Writing to Output File

The plan found for a problem can be written to a file using a command line argument as seen below:

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

When using the planner as part of another Python program the plan can be written to a file using the following function:

Runner.output_result_file(Result, file_path)

The plan can be printed using the function:

Runner.output_result(Result)

C.4 Evaluating Strategies

From the path *Tests/Evaluation/Experiment* a strategy's evaluation can be initiated using the command:

python ./ER<X>.py

Where <*X*> is an integer corresponding to a strategy. Details of what strategy each integer value relates to can be found in the Maintenance manual.

When evaluating the strategy using a computing cluster using an *SBATCH (SLURM)* queue use the following command to submit the strategy for evaluation:

sbatch ExperimentRunner<X>.sh

C.5 Verifying Plans

As part of the software developed for this project, a repository was developed with the sole intention of verifying plans output from EPICpy. This repository is called *EPICpy-Panda-Verify* and must be stored within the same folder as EPICpy. Thie code base can be executed using the command:

python ./verifier.py

Simple modifications to the verifier.py file can be made to change which strategy EPICpy adopts and which problems run and verified.

C.6 Viewing Analysis of Result

The results gathered for this project are stored within the repository *EPICpy-Analysis*. The repository contains a lot of files and folders, we will now describe at a high level what each contains. The file *analytics.ipynb* contains analysis that looks at the over reaching picture illustrated with all problems are considered. Files of the form *<Domain Name>-Graphs.ipynb* hold analysis for a particular problem domain. The files *Domain-Graphs.ipynb* and *Domain-IPC-Scores.ipynb* store analysis for domain overviews and the IPC score of each domain respectively. The file *Landmarks-Analysis.ipynb* analyses only the results of landmarks based strategies. *Novelty-Graphs.ipynb* holds analysis of only novelty based strategies. All of the aforementioned files make of code within *commonCode.py* for generating graphs and loading data.

calculateResultsStats.py is a handy program to calculate the IPC scores for each domain for a results file. The IPC scores are then inserted at the top of the file as a summary. The files being summarised can be changed by modifying the files paths set at the bottom of the file's source code.

Finally, *Plan-Verification-Confirmation.ipynb* compares the results collected to the plans which have been verified.

This repository contains a selection of folders, the main two are *results2* and *verified-results* which store the complete set of results and results of verified plans respectively. Other folders contain partial collections of results from earlier in the project while the folder *output* stores output files and graphs.

C.7 Unit Tests

The unit tests of EPICpy can be run from the directory *Tests/UnitTests* using the command:

```
python ./All_Tests.py
```

C.8 Working with the Profiler

EPICpy provides functionality to be used in conjunction with Python's *cProfile* capabilities. The the *runner-cprofiler.py* contains the code for setting up the planner in a specific strategy and setting an output file path. The following command can be used to run the planner with the profiler:

```
python ./runner-cprofiler.py
```

This can also be used in a SBATCH system using the command:

```
sbatch runCProfiler.sh
```

C.9 Demo 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/p01.hddl
   -progressTrackerName PandaVerifyFormatTracker
   -progressTrackerPath Solver/Progress_Tracking/panda_verify_format.py
   -modelModName PandaVerifyModel -modelPath Solver/Models/PandaVerifyModel.py

4. Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl

5.
   ```
   Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl
   -heuModName SeenStatesPruning
   -heuPath Solver/Heuristics/seen_states_pruning.py
   ```

6.
   ```
   Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl
   -heuModName HammingDistance
   -heuPath Solver/Heuristics/hamming_distance.py
   -searchQueueName GBFSSearchQueue
   -searchQueuePath Solver/Search_Queues/Greedy_Best_First_Search_Queue.py
   ```

7.
   ```
   Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl
   -heuModName HammingDistance
   -heuPath Solver/Heuristics/hamming_distance.py
   -searchQueueName GBFSSearchQueueNewestFirst
   -searchQueuePath Solver/Search_Queues/
   Greedy_Best_First_Search_Queue_Newest_First.py
   ```

8.
   ```
   Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl
   -solverModName PartialOrderNoveltyNoResetSolver
   -solverPath Solver/Solving_Algorithms/partial_order_novelty_no_reset.py
   ```

9.
   ```
   Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl
   -solverModName PartialOrderNoveltySolver
   -solverPath Solver/Solving_Algorithms/partial_order_novelty.py
   ```

10.
    ```
    Tests/Examples/Rover/domain.hddl Tests/Examples/Rover/p04.hddl
    -solverModName PartialOrderHammingNoveltySolver
    -solverPath Solver/Solving_Algorithms/partial_order_hamming_novelty.py
    ```

Appendix D

Maintenance Manual

D.1 Installation and Dependencies

EPICpy's github is currently private due to some of the work being carried out in this project being considered for a publication. As such the only way to set up the planner is by the accompanying code zip file given alongside this report.

The planner requires Python to be installed, this system has been tested with version 3.9.

D.2 Storage and Memory requirements

Requirements on the memory of a system running the planner differs based on the problem being solved and the components selected for EPICpy to use during search.

D.3 Future Additions

Any further components developed for EPICpy should follow the planners current design principles. This means components of specific types need to be inherited by a specific class or any of its direct or indirect sub-classes. *Solving Algorithms* must inherit the *Solver* class, *Heuristics* the *Heuristic* class, *Parameter Selectors* the *ParameterSelector* class, *Search Queues* the *SearchQueue* class, *Progress Trackers* the *ProgressTracker* class, and *Models* the *Model* class.

Each strategy addition should be added to the *ExperimentRunner.py* file to enable evaluation and data collection.

D.4 Evaluating Strategies

Each strategy has a number, the strategy correlating to each number can be easily seen from the large *if-else* control structure within the *ExperimentRunner.py* file.

The problems attempted for a strategy can be set by modifying the contents of the correlating *ER<x>.py* file to add or remove some tests.

Strategies are automatically verified if the plan verification model and progress tracker are set, and the system running the evaluation is linux-based.

D.5 Code Purposes

As part of this project, four code repositories were submitted.

1. EPICpy

   This is the main planner code

2. EPICpy-Analysis

   This is the code to investigate the planner with graphs in the form of notebooks

3. EPICpy Panda Verify Master

   This is an experiment repository which autonomously verifies plans output by EPICpy

4. Panda Planner Evaluation

   This is used to gather results of the panda planner which were used in the results section of this project.

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] Mitch Ai-Chang, John Bresina, Leonard Charest, Jennifer Hsu, Ari K Jónsson, Bob Kanefsky, Pierre Maldague, Paul Morris, Kanna Rajan, and Jeffrey Yglesias. Mapgen planner: Mixedinitiative activity planning for the mars exploration rover mission. *Printed Notes of ICAPS*, 3, 2003.

[3] Ron Alford, Pascal Bercher, and David W Aha. Tight bounds for htn planning with task insertion. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[4] 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.

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

[6] Gregor Behnke, Daniel Höller, Alexander Schmid, Pascal Bercher, and Susanne Biundo. On succinct groundings of htn planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9775–9784, 2020.

[7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[8] Pascal Bercher, Gregor Behnke, Daniel Höller, and Susanne Biundo. An admissible htn planning heuristic. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 480–488, 2017.

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

[10] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[11] Ade Candra, Mohammad Andri Budiman, and Kevin Hartanto. Dijkstra's and a-star in finding the shortest path: a tutorial. In *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*, pages 28–32, 2020.

[12] Luis A Castillo, Juan Fernández-Olivares, Oscar Garcia-Perez, and Francisco Palao. Efficiently handling temporal knowledge in an htn planner. In *ICAPS*, pages 63–72, 2006.

[13] Augusto B. Corrêa, Guillem Francès, Florian Pommerening, and Malte Helmert. Delete-relaxation heuristics for lifted classical planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):94–102, May 2021.

[14] Marc de la Asunción, Luis Castillo, Juan Fdez-Olivares, Óscar García-Pérez, Antonio González, and Francisco Palao. Siadex: An interactive knowledge-based planner for decision support in forest fire fighting. *Ai Communications*, 18(4):257–268, 2005.

[15] Filip Dvorak, Arthur Bit-Monnot, Félix Ingrand, and Malik Ghallab. A Flexible ANML Actor and Planner in Robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS)*, Portsmouth, United States, June 2014.

[16] Rüdiger Ebendt and Rolf Drechsler. Weighted a* search – unifying view and application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[17] Stefan Edelkamp. Planning with pattern databases. In *Proc. ECP*, volume 1, pages 13–24. Citeseer, 2001.

[18] Mohamed Elkawkagy, Pascal Bercher, Bernd Schattenberg, and Susanne Biundo. Improving hierarchical planning performance by the use of landmarks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):1763–1769, Sep. 2021.

[19] Kutluhan Erol, James Hendler, and Dana S. Nau. Complexity results for htn planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, Mar 1996.

[20] Patrick Eyerich, Bernhard Nebel, Gerhard Lakemeyer, and Jens Claßen. Golog and pddl: What is the relative expressiveness? In *Proceedings of the 2006 International Symposium on Practical Cognitive Agents and Robots*, PCAR '06, page 93–104, New York, NY, USA, 2006. Association for Computing Machinery.

[21] 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.

[22] 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.

[23] Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8:339–364, 2003.

[24] Ángel García-Olaya, Sergio Jiménez, and Carlos Linares López. The 2011 international planning competition, 2011.

[25] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*, volume 7 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Springer, 2013.

[26] P Haslum H Geffner and Patrik Haslum. Admissible heuristics for optimal planning. In *Proceedings of the 5th Internat. Conf. of AI Planning Systems (AIPS 2000)*, pages 140–149, 2000.

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

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

[29] Arturo González-Ferrer, Annette ten Teije, Juan Fdez-Olivares, and Krystyna Milian. Automated generation of patient-tailored electronic care pathways by translating computer-interpretable guidelines into hierarchical task networks. *Artificial Intelligence in Medicine*, 57(2):91–109, 2013.

[30] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[31] 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.

[32] Bradley Hayes and Brian Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5469–5476, 2016.

[33] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM (JACM)*, 61(3):1–63, 2014.

[34] Manuel Heusner, Thomas Keller, and Malte Helmert. Understanding the search behaviour of greedy best-first search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 8, pages 47–55, 2017.

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

[36] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. Hddl–a language to describe hierarchical planning problems. *arXiv preprint arXiv:1911.05499*, 2019.

[37] 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.

[38] Daniel Höller and Pascal Bercher. Landmark extraction in htn planning. *HPlan 2020*, page 9, 2020.

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

[40] Michael Katz and Carmel Domshlak. Implicit abstraction heuristics. *Journal of Artificial Intelligence Research*, 39:51–126, 2010.

[41] Michael Katz, Jörg Hoffmann, and Carmel Domshlak. Who said we need to relax all variables? In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23, pages 126–134, 2013.

[42] Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov. Adapting novelty to classical planning as heuristic search. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017.

[43] Emil Keyder and Héctor Geffner. Heuristics for planning with action costs revisited. In *ECAI 2008*, pages 588–592. IOS Press, 2008.

[44] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*, pages 282–293. Springer, 2006.

[45] Daniel L Kovacs. Bnf definition of pddl 3.1. *Unpublished manuscript from the IPC-2011 website*, 15, 2011.

[46] 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.

[47] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI 2012*, pages 540–545. IOS Press, 2012.

[48] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[49] Nir Lipovetzky, Miquel Ramirez, and Hector Geffner. Classical planning with simulators: Results on the atari video games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[50] 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.

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

[52] Cael Milne. A multi-language heuristic driven htn planner in python. Unpublished Dissertation, 2022.

[53] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[54] 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.

[55] D. Pellier and H. Fiorino. Pddl4j: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176, 2018.

[56] 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.

[57] Chao Qi, Dan Wang, Héctor Muñoz-Avila, Peng Zhao, and Hongwei Wang. Hierarchical task network planning with resources and temporal constraints. *Knowledge-Based Systems*, 133:17–32, 2017.

[58] Kanna Rajan, Frédéric Py, and Javier Barreiro. Towards deliberative control in marine robotics. In *Marine Robot Autonomy*, pages 91–175. Springer, 2012.

[59] 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.

[60] Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[61] Alexander Shleyfman, Alexander Tuisov, and Carmel Domshlak. Blind search for atarilike online planning revisited. *Heuristics and Search for Domain-independent Planning (HS-DIP)*, page 85, 2016.

[62] Tristan B Smith, Javier Barreiro, David E Smith, Vytas SunSpiral, and Daniel Chavez-Clemente. Athlete's feet: Mu1ti-resolution planning for a hexapod robot. In *International Conference on Automated Planning and Scheduling*, 2008.

[63] 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.

[64] Mauro Vallati, Lukás Chrpa, Marek Grzes, Thomas Leo McCluskey, Mark Roberts, and Scott Sanner. The 2014 international planning competition - progress and trends. *AI Magazine*, 2015.

[65] Vincent Vidal. Yahsp2: Keep it simple, stupid. *Proceedings of the 7th International Planning Competition (IPC-2011)*, pages 83–90, 2011.