Extending SymbolicPlanners with forward propagation landmark extraction

Ka Fui Yang

Supervisors: dr. Sebastijan Dumančić, Issa Hanou

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 28, 2024

Name of the student: Ka Fui Yang
Final project course: CSE3000 Research Project
Thesis committee: dr. Sebastijan Dumančić, Issa Hanou, Luis Miranda da Cruz

Abstract

The Fast Downward planning system is currently mainly used for solving classical problems. Another alternative to Fast Downward is SymbolicPlanners, which sacrifices speed for generality and extensibility. SymbolicPlanners is missing landmark based planners and landmark extraction algorithms. The research question we are trying to answer in this research paper is: What design choices can be made to adapt the forward propagation extraction algorithm into SymbolicPlanners? The forward propagation landmark generation design choices are discussed and implemented in SymbolicPlanners. The runtime performance of the implementation is only about two times slower than the Fast Downward implementation. Another aspect of the implementation is the incorrect amount of landmarks generated in complex problems caused by limitation in the relaxed planning graph from SymbolicPlanners.

1 Introduction

The goal of classical planning is to find a sequence of actions to go from the initial state to the goal state, where the states are defined by predicates. Landmarks are intermediate states that are necessary to achieve the goals state, meaning every planning solution must contain these landmarks. These landmarks can be extracted using the goal state, initial state and domain. A solution plan can be found by using a heuristic function with landmarks in a heuristic search planner.

Multiple algorithms have been proposed for extracting these landmarks and each of them has its own method for extracting these landmarks for heuristic. The majority of them have been implemented in the Fast Downward planning system [3]. Another alternative to the Fast Downward planner is the SymbolicPlanners in Julia. SymbolicPlanners is a compelling alternative to Fast Downward in terms of generality and extensibility while only 3% slower [9].

SymbolicPlanners has multiple planners and heuristics implemented but is missing landmark based planners and landmarks extraction functions. One such landmark extraction algorithm is described by Zhu & Givan[10]. The aforementioned algorithm is also implemented in Fast Downward.

In the following paper we will try to answer the following research question: **What design choices can be made to adapt the forward propagation extraction algorithm into SymbolicPlanners?** We will be comparing our implementation to the Fast Downward implementation. The runtime and the amount of landmarks extracted will be used as metrics. We will also compare the extraction algorithm to the algorithm described by Porteous, Sebastia & Hoffman with the same metrics[7].

In this paper, we will first describe the background. Secondly, we will introduce related works. Thirdly, we will describe the design choices and the results. Furthermore, we will describe the implications of this research in the responsible research section. Afterward, we will discuss the findings and give reasons for them. Lastly, we will summarize the findings and suggest future works.

2 Background

In this section, we present background information, including definitions and concepts essential for understanding the subsequent sections. Firstly, we explain the representation of a STRIPS planning problem. Secondly, we delve into PDDL and its application in SymbolicPlanners. Lastly, we provide an overview of what a planning graph entails.

2.1 Notation and PDDL

For notation, we follow Koehler & Hoffmann[6]. A STRIPS planning problem is defined as a tuple $\langle P, A, I, G\rangle$, where P represents the set of propositions, A stands for the set of actions, $I \subseteq P$ denotes the initial state, and $G \subseteq P$ signifies the goal state. In this context, states signify the subset of propositions currently considered true. An action o in A is a triple, denoted as $o = (PRE(o), ADD(o), DEL(o))$, where $PRE(o)$ represents the precondition—indicating the set of propositions required to be true for execution; $ADD(o)$ represents the add effect—depicting the set of propositions that become true upon action execution; and $DEL(o)$ denotes the delete effect—illustrating the set of propositions that become false once the action is applied. A plan for a classical problem refers to the sequence of actions executed from state I, culminating in state G.

PDDL is used to describe STRIPS planning problems in SymbolicPlanners [9]. A PDDL formulation consists of a domain description and a problem instance. The problem instance includes objects, an initial state, and a goal state. The domain description encompasses predicates and action schemas with free variables. Through the process of grounding, predicates and action schemas are instantiated using objects from the problem instance [4][2]. Free variables within predicates are substituted with various object combinations to derive propositions, while free variables within action schemas are similarly replaced to obtain grounded actions. SymbolicPlanners uses PDDL.jl for the grounding process[9].

2.2 Planning graph

For definition of planning graph we follow Blum & Furst[1]. A planning graph is a directed, leveled graph where the levels alternate between action nodes and predicate nodes. The initial level consists of all the predicates in the initial state. For level i the precondition predicates are connected via edge to the action node and the action nodes are connected to effect predicates in level $i + 1$. Delete and add effects are both present in the planning graph. No operations(no-op) are present for capturing conditions that do not change. Additionally, mutex exclusions must be marked since adding and deleting the same condition does not correspond to a legal state.

A relaxed planning graph is a planning graph with the delete effects omitted. As the relaxed planning graph lacks delete effects, there is no need to verify whether the delete effect and the add effect of the same condition are in the same

layer. The downside is that landmark verification is essential for the extracted landmarks on a relaxed planning graph[7].

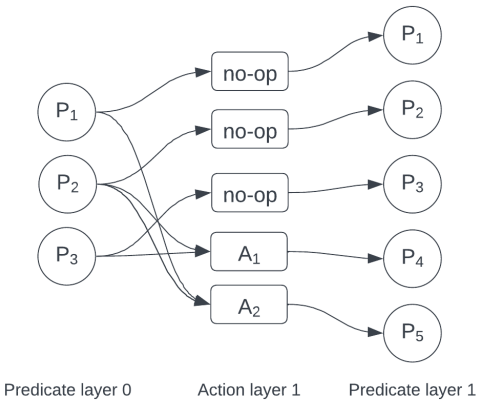

Figure 1: Planning graph for a simple example. Predicate layer 0 here is the initial layer.

SymbolicPlanners provides a method that builds a relaxed planning graph using the domain and the problem instance. It is not a relaxed planning graph in definition since it lacks the layer property but is helpful for the implementation. The planning graph includes various fields, with the key ones being:

- **conditions**: Contains all the ground conditions.
- **act_parents**: Contains the effects of the actions.
- **act_children**: Contains the preconditions of the actions.
- **cond_children**: Contains the action where the conditions are precondition.

3 Related Work

In this section, we introduce related work that serves as a reference or point of comparison throughout the paper. Firstly, we outline the workings of the Zhu & Givan extraction algorithm and highlight key data structures essential for its implementation. Secondly, we will explore the Porteous, Sebastia & Hoffmann extraction algorithm. Thirdly, we discuss the Fast Downward planning system. Lastly, we will explore the Keyder, Richter & Helmert extraction algorithm.

3.1 Forward propagation algorithm

The forward propagation of labels described by Zhu & Givan works as follows[10]. In the initial layer, conditions from the initial state are labeled with themselves, while labels for other conditions remain empty. An action can only occur if the labels from the preconditions are not empty. The label of the effect condition in the next layer contains the union of all precondition labels and the effect itself. The new label for conditions with multiple actions that add the same conditions is the intersection of precondition labels of the actions. The labeling process continues until there are no changes in the labels of subsequent layers. The landmarks are then identified as the conditions labeled in the goal state in the last layer. Subsequently, a landmark graph is created from these conditions and the noncausal landmarks are removed from the graph.

3.2 Backward search algorithm

Another extraction algorithm similar to forward propagation is the extraction algorithm described by Porteous, Sebastia & Hoffmann [7]. The algorithm uses a backward search on the relaxed planning graph. It uses a landmark candidates set LC initially containing the goal conditions. For condition c in LC, actions with add effect c are added to a new set. The intersection of the preconditions of every action in this new set is added to the LC set. The process gets repeated till the initial state is reached. The forward propagation and backward search algorithms employ the intersection of conditions, meaning that this is an essential component of extracting landmarks.

Each condition in LC except the conditions in the initial state and the goal state is verified in the verification process, where candidates that are not landmarks are removed. Actions that add candidates c are removed from the relaxed planning graph. Candidate c is removed from LC whenever the goal state is reachable in the new relaxed planning graph. A landmark-generation tree LGT is also used in the process of generating and ordering landmarks in the landmark graph.

3.3 The Fast Downward System

The Fast Downward consists of three components: translation, knowledge compilation, and search [3]. The translation component transforms the PDDL domain and problem into a finite-domain representation using the method outlined in the paper [4]. While there may not be an apparent advantage to this finite-domain representation for planning algorithms that do not encounter any infeasible states within this representation, other algorithms do derive benefits from it.

The finite-domain representation translation involves multiple steps, with one key step being invariant synthesis, particularly the identification of mutual exclusion (mutex) invariants. Mutually exclusive conditions can be encoded into a single state variable, where the value specifies which of the conditions is true.

The forward propagation extraction algorithm is located in the search component and uses the finite-domain representation as input to extract the landmarks and create a landmark graph as result. The noncausal landmarks are removed from the graph. Solving planning problems using a planner, a heuristic and a landmark graph also happens in the search component.

3.4 AND/OR graphs algorithm

Keyder, Richter & Helmert describes that many problems related to delete relaxation can be understood as computation on the AND/OR graph[5]. The forward propagation process can be understood as performing the update rules on the AND/OR graph according to the order in which the nodes are generated in the relaxed planning graph. Their approach finds

strictly more causal landmarks than the forward propagation algorithm.

According to Zhu & Givan, "We call a propositional landmark causal when every successful plan contains an action that requires the landmark as a pre-condition"[10]. They also explain that since the landmarks extracted from the forward propagation are causal, the extracted landmarks will survive the verification process from subsection 3.2.

4 Methodology

In this section, we define the design choices made and provide a detailed description of the algorithm implementation. Lastly, we elaborate on the chosen domain and the criteria used to evaluate performance.

4.1 Design choices

As discussed in Section 3, two crucial design choices can impact the extraction algorithm. The first choice relates to how the planning graph is represented and utilized in the propagation phase. The second design choice involves how the extracted labels are used to construct a landmark graph.

The data structure for each layer is a vector of labels, whereas a label is a set of integers. The vector's size corresponds to the size of the 'conditions' from subsection 2.2. The labels of the conditions from the initial state contain the index of themself and other conditions are initialized as empty.

Another data structure created using 'cond_children' is used for adding actions in the queue. The data structure is a vector of vectors of integers. The size of the outer vector corresponds to the size of the 'conditions.' The inner vector contains the index of the actions where the precondition index is equal to the index of the outer vector.

Before propagation, initialize both structures. Add actions from initial state condition indexes to queue and propagate using algorithm 1.

Algorithm 1 Forward propagation

```
while queue not empty do
    create new queue
    copy old layer
    for action in queue do
        if action is applicable then
            label the effect of action in layer
            if Check for difference old and new layer then
                Add action of difference in new queue
            end
        end
    end
    queue = new queue
end
```

We take the intersection of the label for conditions with multiple actions that add the same condition.

The landmark graph creation works as follows: Firstly, a new empty Landmark Graph is initialized. Secondly, extract the goal conditions using the PlanningGraph. Thirdly, check the labels of the goal condition for empty. An empty label means the goal condition is not reached during the forward propagation. A goal landmark is created if the label is not empty. Fourthly, create a Landmark for each integer in the label. The landmark connects to the goal landmark with a natural ordering edge. The noncausal landmarks are removed during the verification process.

4.2 Domains and performance criteria

We will use the domains from Zhu & Givan experiment and Porteous,Sebastia & Hoffman experiment[10][7]. These will be Blocksworld, Logistics, Tireworld, Grid, Gripper and Freecell. The Sokoban domain before 2003 does not exist in the PDDL-instances repository which means we will not use it in the experiment.

The domains above are very different from each other. Blocksworld, Logistics, Gripper, and Grid are simple domains with actions that contain two or three free variables. Freecell is a complex domain with multiple actions containing five or six free variables. Tireworld lies in the middle. We will run every problem instances in the domain.

For each problem instance in the domains, the number of landmarks is extracted using backward search and forward propagation. For the backward search implementation, we will use the implementation described by Tervoort[8]. From Tervoort we will also use the landmark verification method. For the forward propagation, we will run one without verification and one with verification. We will also run the compiled and non compiled domains for each instance. We will run the instance three times and calculate the average runtime and amount of landmarks. For each problem instance in the domains, we will also run the Fast Downward implementation of the forward propagation. We will run the instance twice and calculate the average runtime and amount of landmarks.

For performance criteria, we will use the runtime of the extraction algorithm on the problem instances in the domain mentioned above versus the number of objects in the problem instance. Also we will compare the amount of landmark extracted to the Fast Downward implementation. The forward propagation implementation should extract the same amount of landmark to the Fast Downward implementation for every problem instances in every domain.

5 Experimentation result

The experiment has been runned on a laptop running Manjaro Linux with a 2.80 GHz 11th Gen Intel and 15.3 Gb main memory. The benchmark code, implementation and data are available in the GitHub repository.

Figure 2 shows the implementation without verification using a compiled and non-compiled Blocksworld domain. Figure 3 shows the runtime of three different implementations in the non-compiled Blocksworld domain. The first data point for implementation without verification and backward search is significantly higher compared to the data point afterward. Figure 4 shows the runtime between the forward propagation algorithm in SymbolicPlanners and Fast Downward in Blocksworld domain. The reason for high and low data point is because there is multiple problem instances with the same problem size.

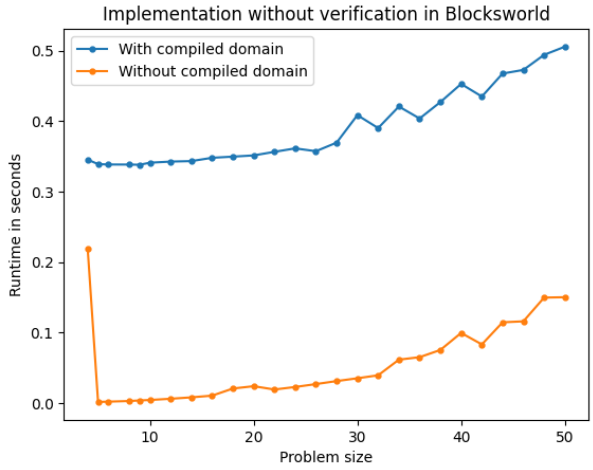

Figure 2: The runtime for the implementation without verification in Blocksworld domain. Runtime lower is better.

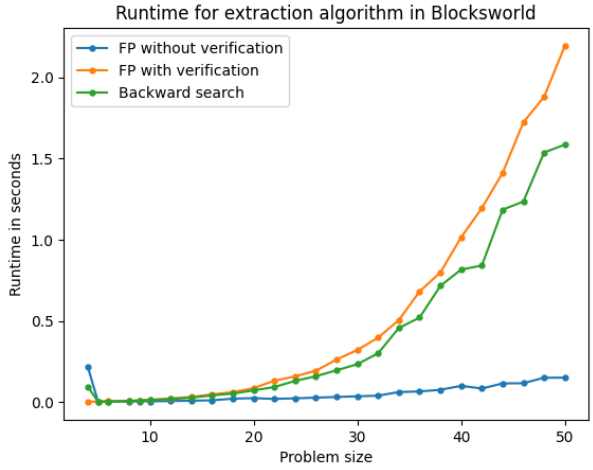

Figure 3: The runtime for three different implementation versus the problem size in non-compiled Blocksworld domain. Lower runtime is better.

Table 1: Amount of instances where our implementation extracted an equal amount of LM to Fast downward implementation. Higher ratio is better.

Domain	Problem instance with same amount LM extracted
Blocksworld	102/102
Logistics	82/82
Tireworld	16/30
Grid	0/5
Freecell	0/60
Gripper	20/20

Table 1 shows the number of problem instances where the implementation extracted the same number of landmarks as the Fast Downward implementation. The implementation does not perform well at Freecell and Grid but well at Blocksworld, Logistics and Gripper. For the Tireworld domain, half of the problem instances have the same extracted landmarks. In the discussion, we will use the Tireworld domain to explain the faults in the implementation.

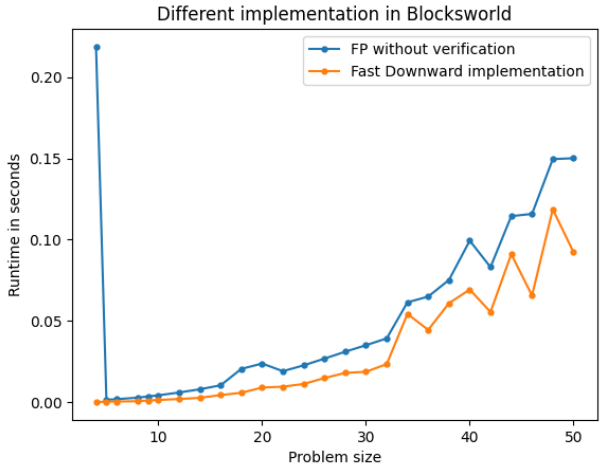

Figure 4: The runtime for Fast Downward and SymbolicPlanners implementation in non-compiled Blocksworld. Lower runtime is better.

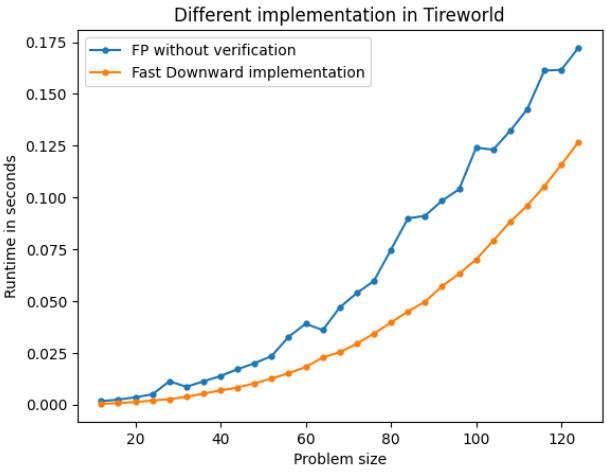

Figure 5: Runtime of Fast Downward and SymbolicPlanners implementation with the first entry omitted in non-compiled Tireworld. Lower runtime is better.

Figure 5 shows the runtime difference Fast Downward and SymbolicPlanners implementation in non-compiled Tireworld domain. The runtime for the implementation is higher in places where it should be low, like the data point with problem size 60. Figure 6 shows the amount of landmarks extracted for both implementations in Tireworld. Our implementation stops extracting more landmarks at problems with sizes bigger than 68. Figure 7 shows that the backward search and forward propagation with verification stop running at sizes bigger than 68 while the forward propagation keeps running.

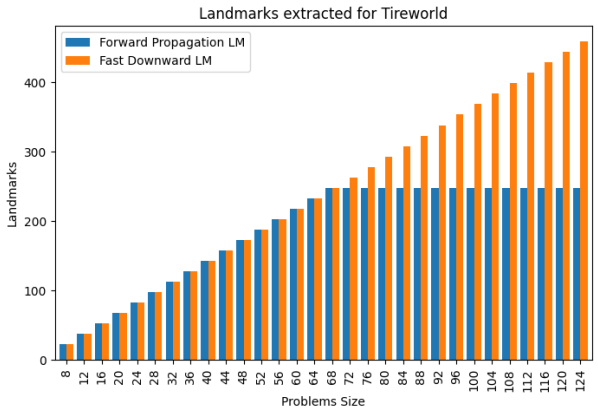

Figure 6: The amount of landmarks extracted for Fast Downward and SymbolicPlanners implementation in Tireworld domain.

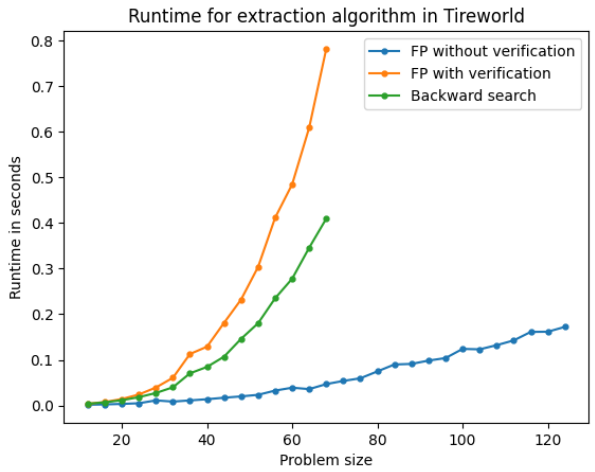

Figure 7: The runtime for three different implementation in non-compiled Tireworld domain with the first entry omitted. Lower runtime is better.

6 Responsible Research

In the following subsection, we discuss why the experiment's reproducibility is necessary and the actions taken to make it reproducible. We also discuss the context in which the implementation is used and the limitations.

Research must be reproducible such that each generation of scientists can build on the previous generation's achievements. The Internet and new technology are constantly changing, making experiments more challenging to reproduce. The following remedy has been taken to make the following research reproducible. Firstly, the source code and data used for generating the results are published. Both are available in the GitHub repository. The result is located in the 'forward-propagation-results' folder while the data are in the 'experiments/logical' folder. Secondly, a README.md file is included in the repository, describing how to reproduce the experiment with the source code and data. The file includes instructions on running the experiment, sources of the data used, and problem instances used for the experiment.

The International Planning Competition is a yearly competition to evaluate planning systems on different benchmark problems. The data used for the experiment comes from the old IPC competitions, ranging from 1998 to 2008, in a GitHub repository [1]. The used domains and problem instances are copied into the experiment repository to prevent data change. Change to the data should not happen since we do not own the IPC data repository and the data we used are from 1998 to 2003.

Another option for data was the official repository of the Planning Domains website [2]. The reason for not using some of their data was because they included a disclaimer that says that data is being fixed, problems set are being replaced, and bugs are being corrected; therefore, it was not recommended for use in academics.

The implementation is used in the context of solving classical planning problems. Classical planning problems are extensive; many types of problems exists ranging from easy to complex problems. The implementation does not solve classical problem but extracts the landmarks. From the result, the implementation extracts the correct number of landmarks in simple domains such as Blocksworld and Logistics but not for others. We hope that users are aware of the limitations of the implementations and refrain from using them in real-life complex problems.

7 Discussion

For simpler domains such as Logistics and Blocksworld, the implementation extracts the same amount of landmarks as the Fast Downward implementation. Using the Tireworld domain, we can derive that forward propagation keeps running for problem sizes bigger than 68 but extracts the same landmark amount to a problem with size 68. After thorough debugging and testing, we have found that the relaxed planning graph provided by SymbolicPlanners does not add ground action after a specific amount of actions. Complex domains have actions that contain multiple free variables and these problem instances also contain a lot of objects. It is understandable that there is a limit to the number of actions being added.

Using the result of Freecell, we can see that the extracted landmark is always four. Further inspection of the relaxed planning graph code shows that no action that adds the goal condition as an effect exists. The relaxed planning graph also includes illegal actions such as On(A, A) for Blocksworld domains.

The runtime for the first data point is always higher compared to the point afterward. Figure 5 and figure 8 show that our implementation does take more than one second in the first problem instance to extract the landmarks. We assume Julia causes this. Julia takes some time when the implementation is compiled for the first time. Subsequent calls within the same session use the fast compiled function. This also explains figure 3, because the backward search uses the verification method first and then the forward propagation.

[1] https://github.com/potassco/pddl-instances

[2] https://github.com/AI-Planning/classical-domains

The runtime for forward propagation without verification is two times slower than Fast Downward. A reason for the difference can be explained by either the finite-domain representation from Fast Downward or the time is timed when the propagation and landmark generation happens. In contrast, our implementation runtime includes the relaxed planning graph creation, forward propagation and landmark generation.

The forward propagation runtime is much faster than the backward search extraction algorithm but the forward propagation with verification is slower than the backward search. The backward search algorithm without verification could resemble the forward propagation in runtime but verification is necessary for backward search. Backward search also extracts more landmarks than forward propagation.

8 Conclusions and Future Work

Our implementation does extract the correct amount of causal landmarks with simple domains such as Blocksworld, Gripper, and Logistics. However, our implementation fails to extract the landmarks for complex and large problem size instances. For Freecell and Grid, zero problem were correctly extracted. For Tireworld, the algorithms stop extracting landmarks with problem sizes bigger than 68. The problem originated from the relaxed planning graph that we use, which can only handle small problem instances.

The design choices discussed in section 4 were significant because the runtime between ours and the Fast Downward implementation is close. Our implementation is about two times slower compared to the Fast Downward. The runtime of the forward propagation algorithm starts to increase if verification is used.

The forward propagation is much faster than the backward search but extracts fewer landmarks. The drawback of forward propagation is that the landmark graphs do not have weakly reasonable ordering. Landmark heuristics that need reasonable ordering do not work with our landmark graph.

The next couple of things to do in the future are as follows. Firstly, improve the relaxed planning graph to remove redundant actions and handle large problem size instances. Secondly, improve the forward propagation to work with domains with axioms. Thirdly, Zhu & Givan describe the landmark counting heuristic that can be used with the generated landmark graph[10]. The heuristic can be implemented in SymbolicPlanners. Using the heuristic and the landmark graph we generated, we can solve classical planning problems.

References

[1] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[2] Daniel Bryce and Subbarao Kambhampati. A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28(1):47–47, 2007.

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

[4] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[5] Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for and/or graphs. In *ECAI*, volume 215, pages 335–340, 2010.

[6] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.

[7] Julie Porteous, Laura Sebastia, and Jorg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *6th European Conference on Planning*, 2001.

[8] Paul Tervoort. *Reproducing the concept of ordered landmarks in planning*. Bachelor's thesis, Delft University of Technology, 2024.

[9] Tan Zhi-Xuan. *PDDL. jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning*. PhD thesis, Massachusetts Institute of Technology, 2022.

[10] Lin Zhu and Robert Givan. Landmark extraction via planning graph propagation. *ICAPS Doctoral Consortium*, pages 156–160, 2003.

A Appendix

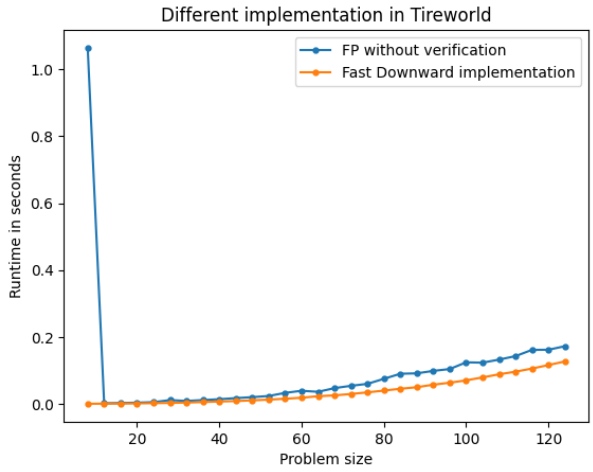

Figure 8: The comparison between Fast Downward and Symbolic-Planners implementation without the first entry omitted.

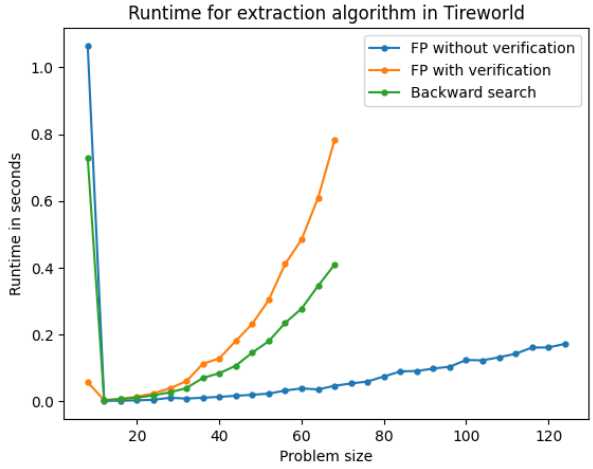

Figure 9: The runtime for three different implementation in Tireworld domain without the first entry omitted.