Technische Universiteit Delft

Master of Science Thesis in Computer Science

Macro-Actions for PDDL: A Dynamic Approach

Pallabi Sree Sarker
p.s.p.sarker@student.tudelft.nl

Supervisors:
Dr. Sebastijan Dumančić
Issa Hanou

12th August 2025

Delft University of Technology

Macro-Actions for PDDL: A Dynamic Approach

Master's Thesis in Computer Science

Algorithmics group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Pallabi Sree Sarker

12th August 2025

Author
Pallabi Sree Sarker

Title
Macro-Actions for PDDL: A Dynamic Approach

MSc presentation
August 21st, 2025

Graduation Committee

ir. dr. Sebastijan Dumančić	Delft University of Technology (chair)
ir. dr. Carlos Hernandez Corbato	Delft University of Technology
ir. Issa Hanou	Delft University of Technology

Abstract

Automated Planning, also known as Artificial Intelligence (AI) planning is a branch of AI focused on automated decision-making and scheduling. A sub-problem within AI Planning is domain-independent planning, where we want to develop methods that are generalisable for solving planning problems in many domains. A popular modelling language for domain-independent planning is PDDL. In PDDL we model our problems as having some start state and some goal state; these states are defined by the truth-values of a set of defined predicates applied to a set of objects with corresponding types. In this work we explore the concept of dynamic macro-actions for PDDL, which are macro-actions whose utility are re-evaluated as we solve more problems, and does not require prior training. We find that dynamic macro-actions are a promising method, showing average improvements in the number of nodes explored in the search space of up to 84% depending on the domain.

Contents

Preface

Before you lies my Master's thesis in Computer Science. It was conducted in the PONY lab of the Algorithmics group of Delft University of Technology under the supervision of Ir. dr. Sebastijan Dumančić and Ir. Issa Hanou. I have been working on this thesis from September 2024 to August 2025.

The inspiration for the work comes from a largely unexplored technique for AI Planning: forgetting. We wanted to explore how forgetting could be applied in an offline manner. Brainstorming with this idea in mind, I came up with a slightly different approach which broadly uses the same idea as forgetting but applies it differently: dynamic macro-actions.

I would like to thank both my supervising professor Ir. dr. Sebastijan Dumančić and Ir. Issa Hanou, who have guided me throughout the process. I have learned a lot throughout this past year and I would like to thank them for helping me culminate all of my studies in the past five years at this university into this thesis.

Furthermore I would like to thank all of my colleagues at the PONY lab for helping me overcome challenges in my work and giving me many insightful ideas. Lastly, I would like to thank my dearest friends and family for supporting me throughout my studies, as well as during the process of bringing this thesis to life.

Pallabi Sree Sarker

Delft, The Netherlands
12th August 2025

Chapter 1

Introduction

Automated Planning–also known as Artificial Intelligence (AI) planning–is a branch of AI focused on automated decision-making and scheduling. In essence, automated planning problems are problems in which we want to construct some kind of sequence of actions for a given agent in a given environment. The set of problems this envelops is vast; ranging from constructing sequences of actions for robots, to train routing scheduling problems. The solutions to such problems can be specific to a particular domain–for example, we write an algorithm for a specific robot to plan its course of actions–or they can be generalised to be applicable to many different types of domains (a generic solver).

This is our goal in domain-independent planning; we want to construct a solver that can do automated planning for a vast array of domains. In other words, the specific domain should not matter. When one wants to solve a problem in some domain, all that is required is to construct a model of the domain (including what actions are available to the agent and a model of its environment) and of models of particular problem instances (the starting state, goal state and an enumeration of the particular objects–or even different agents–in the environment). This approach allows us to develop techniques for solving planning problems without limiting our work to a certain domain.

A popular modelling language for domain-independent planning is PDDL [Ghallab et al., 1998]. PDDL has in the past been used to solve a lot of toy problems, but more recently there has been research done on modelling more complex, real world problems, such as personalised medication and activity planning [Alaboud and Coles, 2019], vehicle routing problems [Cheng and Gao, 2014], train dispatching problems [Cardellini et al., 2021], and many more.

In PDDL we model our problems as having some start state and some goal state; these states are defined by the truth-values of a set of instantiated predicates applied to a set of objects with corresponding types. Our starting state is a set of true predicates (all other possible instantiated predicates are implicitly false), and our goal state is a conjunction of literals (see Example 1).

Example 1. Take the following domain: SHAPEBOARD. We informally define this domain as follows. We have a board with any number of spots, in any shape. The board is defined as a set of positions. Furthermore, we have a collection of shapes that can be placed on the board. Importantly, two shapes cannot occupy one position at the same time. We have two actions: we can place shapes on a particular part of the board and we can remove them. Now, take the starting state shown in Figure 1.1 for the domain SHAPEBOARD. We have a circle in the top-right position, and a star in the bottom-left.

Recall that we define a starting state as a collection of true predicates. The starting state shown in our figure would be defined as follows:

$$isOn(\texttt{circle}, \texttt{top} - \texttt{right})$$
$$isOn(\texttt{star}, \texttt{bottom} - \texttt{left})$$

Let us say the goal is that we want the circle to be on the bottom-left and we want the star to be either on the top-right or the bottom-right. We can define our goal state as follows.

$$isOn(\texttt{circle}, \texttt{bottom} - \texttt{left}) \wedge \neg isOn(\texttt{star}, \texttt{top} - \texttt{left})$$

Note that we do not have to state that we do not want the star to be in the bottom-left, as we defined our domain such that two shapes can never overlap. ⌟

The goal of our solver is to find a plan (a path) from the start state to the goal state. Such a plan consists of a sequence of actions. Every action has some parameters, a set of predicates which are its *preconditions* (they must be true in the current state for the action to be allowed), and a set of predicates which are its *effects* (they become true once the action has been performed–they thus define what state we go to next).

The state-space of these problems can grow exponentially, so finding ways to make our search faster is crucial. Earlier research has shown that the use of macro-actions can improve the solving-time of domain-independent problems [Hernádvölgyi, 2001, Botea et al., 2004, 2005]. Macro-actions are concatenated actions inside the domain of a problem. They can be inferred from the domain, or from solutions to the problem, and then added to the domain.

Example 2. Recall our earlier example of the domain SHAPEBOARD (see Example 1), where we can `place` or `remove` shapes onto/from locations on a board. Take the example state shown in Figure 1.1. A possible macro-action could be to swap the shapes on two locations. The sequence of actions would be:

$$\texttt{remove}(\texttt{circle}, \texttt{top} - \texttt{right})$$
$$\texttt{remove}(\texttt{star}, \texttt{bottom} - \texttt{left})$$
$$\texttt{place}(\texttt{star}, \texttt{top} - \texttt{right})$$
$$\texttt{place}(\texttt{circle}, \texttt{bottom} - \texttt{left})$$

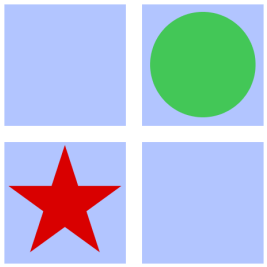


Figure 1.1: Starting state of a problem instance for *shape board*.

To potentially find solutions faster for future problems, we can generalise this sequence of actions and model it as one (macro-)action to add to our domain:

$$\texttt{swap}(\texttt{shape} - \texttt{1}, \texttt{loc} - \texttt{0}, \texttt{shape} - \texttt{2}, \texttt{loc} - \texttt{1})$$

⌟

To find more ways to improve search, we can look at the sister of AI Planning–program synthesis. Program synthesis is the problem where we try to generate a program which satisfies specified constraints[Gulwani et al., 2017]. If one stretches its definition, a program can be seen as a sort of plan. Within program synthesis there exists a technique called refactoring. With refactoring we try to reformulate some grammar by adding new grammar rules that model combinations of pre-existing grammar rules (for example double: $x + x$). It has been shown that refactoring the grammar in program synthesis problems improves accuracy and solving times[Hocquette et al., 2024]. If we use our perspective of seeing programs as a type of plan, we can also view refactorings as a type of macro-action.

It is important to note that this analogy is not perfect, as programs in program synthesis are trees, whereas plans in automated planning are linear sequences of actions. Just as importantly, we should note that this comparison is not just based on a feeling; the comparison between macro-actions and refactoring can be seen in how we change the search space. In both cases, we increase the branching factor inside the search space, but by doing so we create possible shortcuts to the solution. In both refactoring and the addition of macro-actions there lies a delicate balance between these two opposing forces.

An extension of refactoring for program synthesis is the concept of forgetting, where we forget earlier found refactorings. We do this to maintain the delicate balance between the increase in branching factor and the inclusion of useful refactorings. Earlier work has shown that using forgetting for refactoring helps with solving inductive logic programs [Cropper, 2019].

As mentioned prior, refactoring for program synthesis and macro-actions for automated planning are similar in the sense that they both aim to balance an increase in the branching factor to create a shorter path in our state-space to our desired solution. It is thus in our interest to explore similar techniques for both domains, such

as forgetting. In this work we want to explore the usefulness for macro-actions in PDDL problems of a key insight that forgetting uses–that our estimation of the utility of found knowledge can change over time. We want to construct a method that can learn macro-actions and can dynamically reassess their usefulness macro-actions.

The main contribution of this work is a dynamic offline approach to the learning of macro-actions. This entails that the system can be used with any solver to improve over as we solve more problems, rather than training it once before use. Furthermore, we evaluate the system and analyse its strengths and weaknesses.

Lastly, we assess how different different hyper-parameters change the performance of our system. Our main goal is to discover how we can best estimate and (re)asses the utility of a potential macro-action.

This brings us to the following main research question: 'How can we use dynamically learn macro-actions to improve the solving time of AI planning problems modelled in PDDL?', with the following sub-questions:

1. How does the performance of dynamic macro-actions change as we train on more problems?
2. At what point do we observe diminishing returns as we train on more problems for dynamic macro-actions? That is, (when) do we observe a convergence of the performance?
3. How can we best estimate the utility of macro-actions?

Broadly, in our method we store information of all macro-actions it finds with each problem we solve (each iteration). Then, for each iteration we select macro-actions based on some utility function we define. This approach ensures that no problem-instances are required for training; we train as we solve problems we want to solve and learn over time.

We benchmark our method using various domains: Baking and Logistics from the PDDL-Gym problem set, and Satellite, Pipesworld-no-tankage and Pipesworld-tankage from the IPC-4 competition[1].

We find that the most important metric for estimating the utility of a macro-action depends on the domain we are trying to solve. Specifically, we see that the best performing formulae for the estimated utility are the number of historic uses for the Baking domain with an average decrease in nodes explored in the search space of 84% (after solving 5 problems prior), the number of historic uses weighted by the number of unique actions (by name) in the macro-action for the Logistics domain with an average decrease of 75%, and the number of uses weighted by the size of the macro-action for the Satellite domain, with a 33% average decrease.

We found that our method does not improve performance on the Pipesworld domains and hypothesise that this is due to the large amount of parameters in their actions. but does improve performance for the other domains. We do not see a very clear trend–however–in any domain, for how the performance changes; the performance does not seem to converge.

[1] https://ipc04.icaps-conference.org/deterministic/

Chapter 2

Background

The following chapter describes all the background knowledge that is required to understand the coming chapters. We start by introducing AI planning in general. Then, we introduce basic domain-independent planning in PDDL. Lastly, we introduce what macro-actions are and what forgetting them entails.

2.1 Automated Planning

Automated Planning, or AI Planning, is the field in which we aim to automatically solve planning problems. Such problems are problems where we want to plan out *what* an agent is/multiple agents are going to do in a given world/with a given set of resources; we want to create a plan of action for some agent(s). We want to generate a plan that achieves some sort of goal. A problem consists (see Definition 4) of a set of objects, a starting state and a goal state.

The solution to such a problem consists out of a list of *actions* (a plan) which bring us from the initial state to the goal state.

2.1.1 Domain-Independent Planning

While AI Planning can be done in a domain *dependent* manner–that is, catering your planner to some specific domain–it can also be done in a domain *independent* manner. This means that we design a planner that can solve problems regardless of their domain. For such planners, the domain is given as a model to the planner alongside the specific problem(s) that need(s) to be solved. A popular example of a domain-independent modelling language is PDDL.

Definition 1 (PDDL Domain)**.** A PDDL Domain $D : (T, \mathscr{P}, \mathscr{A})$ consists of:

- T: A type tree where the root note is the `object` type.
- $\mathscr{P}$: A set of predicates, each consisting of a name, a set of argument names A, and their corresponding types $\{(p_1, A_1, \Gamma_1), \ldots, (p_n, A_n, \Gamma_n)\}$, where p_i is the predicate name of predicate i, A_i an ordered tuple of the argument names

of predicate i, and Γ_i is an ordered tuple of the corresponding types of the arguments in A_i.

- $\mathcal{A}$: A set of actions, each with a name, a set of parameter names with their corresponding types, a set of preconditions, and a set of effects–where both the preconditions and effects are in conjunctive normal form (a conjunction of disjunctions).

$$\{(\text{name}_1,\ \text{params}_1\ \text{pre}_1,\ \text{eff}_1), \ldots, (\text{name}_n,\ \text{params}_1,\ \text{pre}_n,\ \text{eff}_n)\}$$

⌟

Definition 2 (Predicate Instance)**.** For any type tree T, predicate $p(p_1 : t_1, \cdots, p_n : t_n) \in \mathcal{P}, t_i \in T$ and set of object literals O, a predicate instance $p(o_1, \cdots, o_n)$ is the instantiation of predicate $p \in \mathcal{P}$ with a tuple of object literals $o_1, \cdots, o_n, o_i \in O$, where $n \leq |O|$ such that the types of $o_1, \cdots, o_n$ are $t_1, \cdots, t_n$. A predicate instance can have a truth or false value. ⌟

Definition 3 (State)**.** For any PDDL domain (see Definition 1) and a set of object literals O, a state S is a truth-value assignment for every possible instantiation of predicates in $\mathcal{P}$ for the objects in O. ⌟

Definition 4 (PDDL Problem Instance)**.** A PDDL Problem Instance $P : (O, S_0, S_g)$ corresponding to some domain $D : (T, \mathcal{P}, \mathcal{A})$ consists of:

- O: A set of objects and their corresponding types

$$\{(\text{name}_1, \gamma_1), \ldots, (\text{name}_n, \gamma_n)\}$$

- S_0: The starting state expressed as a set of predicates in $\mathcal{P}$ and their corresponding arguments (objects) $\{(p_1, A_1), \ldots, (p_n, A_n)\}$ where A_i is an ordered tuple of objects in O containing the arguments for p_i.
- S_g: A goal state expressed as a set of predicates in $\mathcal{P}$ and their corresponding arguments (objects) $\{(p_1, A_1), \ldots, (p_n, A_n)\}$ where A_i is an ordered tuple of objects in O containing the arguments for predicate p_i.

⌟

A PDDL domain (see Definition 1) models the *type* of problem we want to solve. An example of such a domain can be found in Figure 2.1. In this domain we are modelling a type of problem involving a teapot and a teacup; we can have a set of teacups and teapots and we want to pour into some amount of teacups. A domain only models the type of problem we want to solve; it defines the types, predicates, and actions in that type of problem. Specifically, we see one singular action: pour. This action requires the teapot to be full and the cup to be empty to be able to pour a given teacup with a given teapot. We see that the effect is that the cup is no longer empty. Notably, this domain is only using two requirements: `strips` and `typing`. PDDL

knows many other requirements, and different planners support different requirements[Ghallab et al., 1998].

For each domain there exist many problem *instances* (see Definition **??**). An example of a problem instance for the tea domain can be found in Figure 2.2. We see here that we have one singular cup and one singular teapot. Our starting state is that our cup is empty and our goal is for it to not be empty.

```
(define (domain tea)
        (:requirements :strips :typing)
        (:types cup teapot)
        (:predicates    (cupisempty ?x - object)
                        (teapotisempty ?y - teapot))
        (:action pour
            :parameters (?x - cup ?y - teapot)
            :precondition (and
                (cupisempty ?x)
                (not (teapotisempty) ?y)
            )
            :effect (
                (not (cupisempty ?x))
            ))
)
```

Figure 2.1: An example of a PDDL domain modelling pouring a tea cup.

```
(define (problem teaproblem)
        (:objects
            cup0 - cup
            pot0 - teapot
        )
        (:init
            (isempty cup0)
        )
        (:goal (and
                (not (isempty cup0))
            )
        )
)
```

Figure 2.2: Example of a simple PDDL problem where we have to fill a teacup.

2.2 Macro-Actions

In this section we give a brief introduction to macro-actions. They are formally defined in Chapter 4. Macro-actions are sequences of actions that are modelled as one. Take, for example, the following sequence of actions from a `baking` problem[1]:

$$\texttt{putflourinpan}(\texttt{flour0}, \texttt{pan0})$$
$$\texttt{mix}(\texttt{egg0}, \texttt{flour0}, \texttt{pan0})$$
$$\texttt{putpaninoven}(\texttt{pan0}, \texttt{oven0})$$

If we deem this a useful, or 'smart' sequence of actions, we can decide to store it as knowledge for our solver as a macro-action, so it is easier to find. Our resulting action would be:

$$\texttt{combined}(\texttt{flour0}, \texttt{egg0}, \texttt{pan0}, \texttt{oven0})$$

Note that the name of our macro-action does not matter, as long as it models the correct logic. That is, a macro-action $m : a_1, \ldots, a_n$ is valid if and only if it can only be used if the sequence of actions would be allowed to be used, and the effects of the macro-actions are the same as if the actions were performed one-by-one.

Furthermore, note that we did not simply concatenate the parameters. We collapsed them based on how they were called (in our combined actions, we do not have the parameter `flour0` twice). Our method for doing so, and combining macro-actions in general, can be found in Chapter 5. Notably, PDDL does not offer native support for macro-actions, so the merging of actions must be deduced based on their parameters, preconditions and effects.

[1] `https://github.com/tomsilver/pddlgym/blob/master/pddlgym/pddl/baking.pddl`

Chapter 3

Related work

There exists a considerable amount of research surrounding macro-actions. Most of the research on macro-actions are for *basic* PDDL, where the logic of combining actions is quite simple. This changed in 2023, when a paper [Bortoli et al., 2023] is published that expands macro-actions to also support temporal PDDL problems, paving the way for being able to apply macro-actions to more complex, real-world problems.

The main shortcomings of the related works is that they are either static, use an online approach, or both. In this work we give a dynamic approach which is offline, and can thus be plugged into any solver.

3.1 Static Macro-Actions

We start with macro-actions. They are a very widely discussed topic in the literature, with not only papers showing their effectiveness, but also exploring different ways of finding them and filtering through them. Many papers find that the addition of macro-actions improves performance, but also find that having too many has a counterproductive effect [Hernádvölgyi, 2001, Botea et al., 2004, 2005].

In the paper *Macro-FF: Improving AI Planning with Automatically Learned Macro Operators*, the authors build two systems. CA-ED, which interacts with a planner that does not require support for macro-actions, and SOL-EP, which interacts with a planner that *does* require support for macro-actions. They do not provide source code for their solutions, nor an algorithm for merging macro-actions for solvers that do not support them [Botea et al., 2005]. Importantly, they do collapse the parameters of macro-actions. Another planner called *Marvin* (***M**acro **A**ctions from **R**educed **V**ersions of the **IN**stance)* is described in a different paper [Coles and Coles, 2004]. This is a forward-chaining domain-independent planner that uses a relaxed-plan heuristic for its search. In this work they do not discuss the collapse of parameters for macro-actions. Furthermore, as it is a stand-alone planner, their implementation cannot be used with external planners. In this work we describe a method that can be used with *any* solver, making the approach more versatile.

Other papers explore how to generate / find good macro-actions. One paper uses

automatically generated heuristics for a given problem to generate macro-actions [Hernádvölgyi, 2001]. Another work finds a 44% improvement in solving time for the average solution [Hernádvölgyi, 2001]. However, their approach is about state-space search in general and not about PDDL problems. They thus do not discuss how macro-actions are combined in PDDL, nor do they discuss the collapsing of parameters in macro-actions. Another paper measures the utility of plan fragments to find suitable macro-actions. Though they do not address the problem that adding too many macro-actions is counterproductive and do not explicate what they do to prevent that from happening[Minton, 1985]. Their paper applies macro-actions to STRIPS, but also does not discuss the collapsing of parameters. Another method of generating macro-actions, which is also used by Macro-FF[Botea et al., 2005] is by generating abstractions of the problem. This method is explored in the paper *Automatically Generating Abstractions for Planning*[Knoblock, 1994]. They run experiments and give empirical results, as well as formal proofs. Lastly, in the paper *Applying Inductive Program Synthesis to Macro Learning* [Schmid and Wysotzki, 2000], the authors apply inductive program synthesis to find macro operators. A problem in their paper is it lacks empirical results, meaning that the effectiveness of their method can not properly be assessed. Furthermore, they use a small problem and extrapolate, potentially leading to false conclusions.

A paper published in 2023[Bortoli et al., 2023] applies macro-actions to *temporal* PDDL problems. This is noteworthy because the difficulty of merging actions to form a macro-action in PDDL–especially for temporal problems in PDDL–is non-trivial; PDDL does not have built-in functionality for macro-actions.

All the aforementioned papers have a static approach to macro-actions. That is, once they are learned, they are learned. After that we simply stop learning and never reassess the utility of the macro-actions. In the literature there does exist an approach where we do re-assess utility of learned knowledge called forgetting.

3.2 Forgetting

In the literature of macro-actions, most papers focus on *filtering* macro-actions, rather than unlearning already learned macro-actions. One paper [Coles et al., 2007] does discuss the 'pruning'–or forgetting–of macro-actions. They do this on an *online basis*–it is built into the solver.

They find that this leads to an improvement. They do not provide the code to their solutions but do provide good research into how to identify the quality of macro-actions. A limit of their approach is that because they focus on the solving of one problem, it cannot reason about knowledge gathered by different problems about the usefulness of macro-actions. An advantage of this approach is that it does not require a big set of problems to learn from. In this work we focus on finding macro-actions that are good for a range of problems.

In the domain of Inductive Logic Programming (ILP) there exists a paper which explores the concept of forgetting *background knowledge*[Cropper, 2019] and finds that it is promising for ILP.

Chapter 4

Problem Statement

In this chapter we describe and formally define the problem we are trying to solve–the macro-action extension problem. We first provide all necessary definitions required for understanding the problem and then give a formal definition of it.

4.1 Preliminaries

We give the necessary definitions for describing the domain extension problem. Going forward, 'PDDL domain' (see Definition 1) is shortened to 'domain' and similarly 'PDDL problem instance' (see Definition 4) is shortened to 'problem instance'.

4.1.1 Solving a Problem Instance

We first want to define the problem of solving problem instances modelled in PDDL. A solution to such a problem instance is a *plan*, or a path through the state space from our starting state S_0 to our goal state S_g. Each edge on this path is an *action call*, which we define as follows.

Definition 5 (Action call)**.** For any problem instance $P : (O, S_0, S_g)$, domain $D : (T, \mathscr{P}, \mathscr{A})$, an action call A is a pair (a, o), where $a \in \mathscr{A}$ and o is an ordered list of object literals in O. An action call can be applied to a certain state S, resulting in $S' = A(S)$. ⌟

We furthermore define a path as follows:

Definition 6 (Path)**.** For any problem instance $P : (O, S_0, S_g)$, domain $D : (T, \mathscr{P}, \mathscr{A})$, a path p is an ordered list of action applications $A_1, \ldots, A_n$ starting from some state S_a and ending in some state S_b, where $S_b = A_n(\cdots(A_1(S_a)))$. ⌟

The task of a PDDL solver is then to, given a domain D and a problem instance P, to find the shortest path p from our starting state S_0 to our goal state S_g.

Definition 7 (Shortest-plan problem)**.** Given a problem instance P and a domain D, find the shortest path p from S_0 to S_g. ⌟

4.1.2 Macro-actions

Macro-actions are actions which are logically equivalent to particular action-paths, or, parametrised sequences of actions.

Definition 8. (Action path) An action path $p(x, \ldots, z) : a_1 \circ \cdots \circ a_n(x, \cdots, z)$ is a parametrised ordered list of actions where each action uses a specified tuple from the parameters $x, \ldots, z$. ⌟

Definition 9 (Equivalence of action paths)**.** Any two action paths $p_a(x, \ldots, z)$, $p_b(x, \ldots, z)$ are logically equivalent $p_a(x, \ldots, z) \simeq p_b(x, \ldots, z)$ iff for any state S_0, taking $p_a(x, \ldots, z)$ takes us to the same state S_n as taking $p_b(x, \ldots, z)$. ⌟

Definition 10 (Macro-action)**.** For any domain D, a macro-action m is an action which is equivalent to some action path p in D, $[m] \simeq p$. ⌟

4.1.3 The Macro-Action Extension Problem

With these definitions for the Shortest-plan problem and macro-actions, we can define the domain extension problem (see Problem 11).

Definition 11 (Macro-Action Extension Problem)**.** Given a domain $D : (T, \mathscr{P}, \mathscr{A})$, we want to find some domain $D' : (T, P, \mathscr{A}')$ such that $\mathscr{A}' \supset \mathscr{A}$ to minimise the average amount of visited nodes in the search tree when solving the shortest-plan problem for problem instances of D, where every $a \in \mathscr{A}' \setminus \mathscr{A}$ is a macro-action.

⌟

Chapter 5

Methods

The following chapter will detail the main contribution of this work: dynamic macro-actions (how they are constructed, kept track of and used). First we contrast dynamic macro-actions with static macro-actions and compare compare our approach to forgetting. We follow by explaining what dynamic actions are. We motivate our choice for the overall structure and flow of our method and give some notes on how it can be used and adjusted. This is followed by an overview of the whole system, including description of each step. Lastly, we give details of our implementation.

Our main contribution is the concept of *dynamic macro-actions*. Rather than training and returning a new domain as with static macro-actions, we want to improve over time as we solve more problems; with each problem we solve, we want to gain more information about the domain and update our knowledge base accordingly. This method is similar to *forgetting*–where we learn some set of macro-actions and forget them if they are not useful. We differ in that we store information on all found potential macro-actions and pick the best ones with each problem instance we solve.

When solving a new problem, we look at our knowledge base and select the best n (a given parameter) macro-actions that we know of according to some estimated utility value. An overview of the whole system can be found in Figure 5.1.

5.1 Static macro-actions and forgetting

In previous work (see chapter 3) on macro-actions, a set of problems of a particular domain and their solutions are analysed to get some set of macro-actions, which are either used by some solver or are appended to a domain.

The main motivation of this work is to further investigate macro-actions by trying to apply a crucial insight from forgetting to their use to see its effectiveness; the insight that when encountering new problem instances, we might learn that certain macro-actions are not (as) useful any more. To be able to use this insight we need to have some kind of knowledge database that we can update whilst solving new problem instances.

Importantly forgetting been implemented in an *online* manner–that is, embedded into some solver. In this work we will approach implementing a dynamic approach

similar to forgetting.

5.2 Dynamic Macro-actions

We take the main insight of forgetting–namely that we might change our assessment of the usefulness of certain macro-actions over time–but use it in a different manner. Instead of having one core of macro-actions we remember and forget in favour of new ones, we choose to store information of all macro-actions the system ever encounters in an external knowledge base $\mathcal{K}$. We update this external knowledge base $\mathcal{K}$ with each problem instance that is solved. For each next problem we solve, we choose the best macro-actions to add to our domain D given the knowledge in $\mathcal{K}$ we have accumulated. This ensures we do not lose earlier found knowledge, while still keeping our 'real' knowledge-base (the macro-actions that are actually added to the domain D) small. In short, our knowledge stored in $\mathcal{K}$ is ever-growing, while our knowledge we store in D is continually reset and recreated based on our continually updated knowledge-base $\mathcal{K}$.

The first main benefit of our method are that it is extremely versatile; we can use it with any solver, because it is offline. The second main benefit is that it is easily adjustable, especially in the way that we estimate the utility of macro-actions. In this work we use basic metrics such as the number of times a macro-actions has historically been used in solutions, the size of the macro-action and the number of unique actions in it–but an extension of this work could easily swap this out for more complex metrics or use

5.3 Method overview

An overview of our method can be found in Figure 5.1. We see that we can use our method with any generic PDDL solver. Each problem we solve goes through the pipeline. First we retrieve the top (according to some estimated utility function) n (which is a given parameter) macro-actions (encoded as sequences of actions with arguments)–if they exist–from our macro-action database (see Section 5.3.2). We then merge those sequences of actions into singular actions (see Section 5.3.3) and add them to a copy of our original domain. We solve our problem using any PDDL solver and use the solution to continue. We extract macro-actions from the solution we found and update our macro-action database for the next problem(s) we want to solve (see Section 5.3.1). We continue this cycle with every problem we solve, so we continually improve our knowledge of our domain. This means there is no training required; we improve as we go.

To maintain our knowledge database $\mathcal{K}$, we must have some way of extracting macro-actions from solutions and storing them to $\mathcal{K}$ or updating $\mathcal{K}$.

5.3.1 Extracting and Storing Macro-Actions

Given a solution $\mathcal{S}$ which is an ordered list of actions and corresponding arguments, we extract all sublists $|\mathcal{S}| \geq 2$ of $\mathcal{S}$. It is noteworthy that this component of the system will differ per solver that is used, as different solvers will output the solution differently; we call this component the macro-action extractor.

We then add these to $\mathcal{K}$ if they do not yet exist, or update them if they do with usage data. We store the number of times this sequence of actions (or a macro thereof) has been used, the size of the macro-action, the number of unique actions within the macro-action and the number of cycles it has been since it was last picked.

Each macro-action gets a row in this database with the following attributes: `sub_actions`–which is a plain-text encoding of the macro-action, `size`–which is the length of the sequence of actions in the macro-action, `num_unique_actions` –which is the unique number of actions in the sequence, `num_uses`–which is the total number of times that sequence of actions has been used throughout the different solutions the system has encountered (as macro or otherwise), and `cycles_last_included`, which keeps track of how many problems the system has solved since this sequence was last included as a macro-action. These were chosen for simplicity's sake. Other considerations which were not included are discussed in chapter 8.

Encoding Macro-Actions

As stated earlier, we use a plain-text encoding to store the sequences of the macro-actions. We want to store information of both the names of the actions and with what variables they were used, as this information is crucial to understanding sequences of actions found in solutions. We illustrate this using the following example:

```
action1--var1-var2 action2--var2-var3
```

We then need some way to pick the 'best' macro-actions from $\mathcal{K}$–that is–the macro-actions that will lead to the largest decrease in the visited nodes in our search tree.

5.3.2 Picking the Best Macro-Actions

When we want to solve a new problem, we choose the macro-actions we want to add as follows. Given that we want to use n macro-actions, we first sort them by their estimated utility. Then we simply pick the top n macro-actions according to this sorting. For the next iteration of our system, we will only use these n macro-actions.

Estimating the Utility of a Macro-Action

To be able to implement our method, we need some kind of measure of the usefulness, or utility, of a macro-action m. We denote this as $\hat{U}(m)$. In this work we want to investigate what a good estimation of utility is, so we experiment on the following. For any macro-action m with an associated action path p, we have:

- The number of uses n–that is, how many times p has occurred in previous iterations.
- The size of the macro-action $|m|$–that is, how many sub-actions does m contain, or $|p|$.
- The number of unique actions $u(m)$–that is, how many unique (by name) actions are in p.
- The number of uses n weighted by the size of the macro-action, $n \cdot |m|$.
- The number of uses n weighted by the number of unique actions in the macro-action, $n \cdot u(m)$.
- As a control we also want to see how randomly assigning

Overlapping macro-actions

It is possible that the macro-actions we pick when simply sorting are subsequences of one another; we call these overlapping macro-actions. We hypothesise adding several overlapping macro-actions is not optimal, and therefore we want some way to pick which macro-actions to include, and which to includes. We discuss three methods to do so.

The first method is a control; we simply allow overlapping macro-actions to be added to our domain simultaneously. We can use this control method to test our hypothesis.

For the other two methods we introduce some notation. We have a set of macro-actions we want to add $\mathcal{M}$ and we are given a sorted sequence M of macro-actions we add, sorted by their estimated utility. Furthermore, let $m|n$ denote that m is a subsequence of n.

Our second method called LARGEST goes as follows. We iterate through M. If the next $m \in M$, is a subsequence for any $p \in \mathcal{M}$–that is, $m|p$–then we discard it. Otherwise, we take the largest subset $\mathcal{P} \subseteq \mathcal{M}$, such that $\forall p \in \mathcal{P}$, $p|m$ we update $\mathcal{M} := (M \setminus \mathcal{P}) \cup \{m\}$. We continue until $|\mathcal{M}| = n$.

The benefit of this method is that we add larger macro-actions which could bring us closer to the solutions. The drawback is that these larger macro-actions could be of lower *quality*, as they can be lower in our ranking than the smaller macro-actions we discard.

Our third method called BEST goes as follows. We iterate through M. If for the next $m \in M$, there exists any $p \in \mathcal{M}$ such that $m|p$, we discard it. Otherwise, we update $\mathcal{M} := \mathcal{M} \cup \{m\}$. We continue until $|\mathcal{M}| = n$.

The benefit of this method is that we always use the *best* macro-actions according to their estimated utility. A drawback is that we are potentially missing out on macro-actions which could bring us closer to the solution. It is a priori unclear which method is optimal.

Once we have our macro-actions we cannot just add them to the domain, as PDDL does not natively support macro-actions. We need to model a new action in such a way that it is equivalent to its corresponding sequence.

5.3.3 Macro-Actions in PDDL

PDDL does not have native support for macro-actions. That is, we cannot tell PDDL we want a macro-actions by referring to existing actions. Instead, we need to create new actions which model a sequence of actions. We do this in the component called the macro-action merger.

When given an action path p as found in some solution, we want to to create a new macro-action $m \simeq p$ (see Definition 9). We broadly do this in two steps (also see Algorithm 1).

Firstly we have to rename all the parameter references in the clauses of our preconditions and effects, such that we have one parameter for each of the unique arguments found in the solution where this sequence was found. We illustrate this in Example 3.

Example 3. Take the sequence of actions:

$$\texttt{putegginpan}(\texttt{egg\_0}, \texttt{pan\_1})$$
$$\texttt{putpaninoven}(\texttt{pan\_1}, \texttt{oven\_1})$$
$$\texttt{bakesouffle}(\texttt{oven\_1}, \texttt{egg\_0}, \texttt{pan\_1}, \texttt{new\_0})$$

where the action signature of `putegginpan` is $\texttt{putegginpan}(?egg{-}egg\ ?pan{-}pan)$, the action signature of `putpaninoven` is $\texttt{putpaninoven}(?pan{-}pan\ ?oven{-}oven)$ and the action signature of `bakesouffle` is $\texttt{bakesouffle}(?oven{-}oven\ ?egg{-}ingredient\ ?pan - pan\ ?new - ingredient)$.

We collapse the parameters found in our sequence and get the following list:

- `egg_0` : `ingredient`
- `pan_1` : `pan`
- `oven_0` : `oven`
- `new_0` : `ingredient`

We replace all references of egg and pan in putegginpan with egg_0 and pan_0, all references of pan and oven in putpaninoven with pan_0 and oven_0, etc. ⌟

Secondly, we need to merge the preconditions and effects of all the actions into one set of preconditions and one set of effects. The preconditions pre and effects eff of two subsequent actions a_1, a_2 can be merged as follows. Concerning the preconditions, we want the preconditions of a_1, plus the preconditions of a_2 which are not satisfied by the effects of a_1, so $\text{pre}_1 \cup (\text{pre}_2 \setminus \text{eff}_1)$. For the effects we want the effects of a_2, plus the effects of a_1 that are not negated or repeated in the effects of a_2, so $\text{eff}_2 \cup (\text{eff}_1 \setminus \{\neg c \mid c \in \text{eff}_2\})$. The preconditions and effects of a sequence of actions can be merged using the aforementioned property by using a fold operation.

Algorithm 1: Merging a sequence of actions

Input: A sequence s of n actions actions $[a_i : (\langle P_i, T_i \rangle,\ \text{pre}_i,\ \text{eff}_i)]_i^n$, an ordered list L of tuples $[l_1, \ldots, l_n]$ of the arguments (objects) they were called with, and a mapping $t : O \to T$ mapping objects to their corresponding types.

Output: A new macro-action $m \simeq s$ (see Definition **??**).

$\mathscr{P} \leftarrow \{\langle o, t(o) \rangle \mid o \in l, l \in L\}$;

Replace all parameters and parameter calls in actions $a_1, \ldots, a_n$ with the corresponding argument names in $l_1, \ldots, l_n$.;

Merge all the preconditions and effects in order, where those of two actions (in order) (a_x, a_y) are merged as follows:

$\text{pre} \leftarrow \text{pre}_x \cup (\text{pre}_y \setminus \text{eff}_x)$;

$\text{eff} \leftarrow \text{eff}_y \cup (\text{eff}_\text{x} \setminus \{\neg c \mid c \in \text{eff}_y\})$;

return $m : (\mathscr{P},\ \text{pre},\ \text{eff})$;

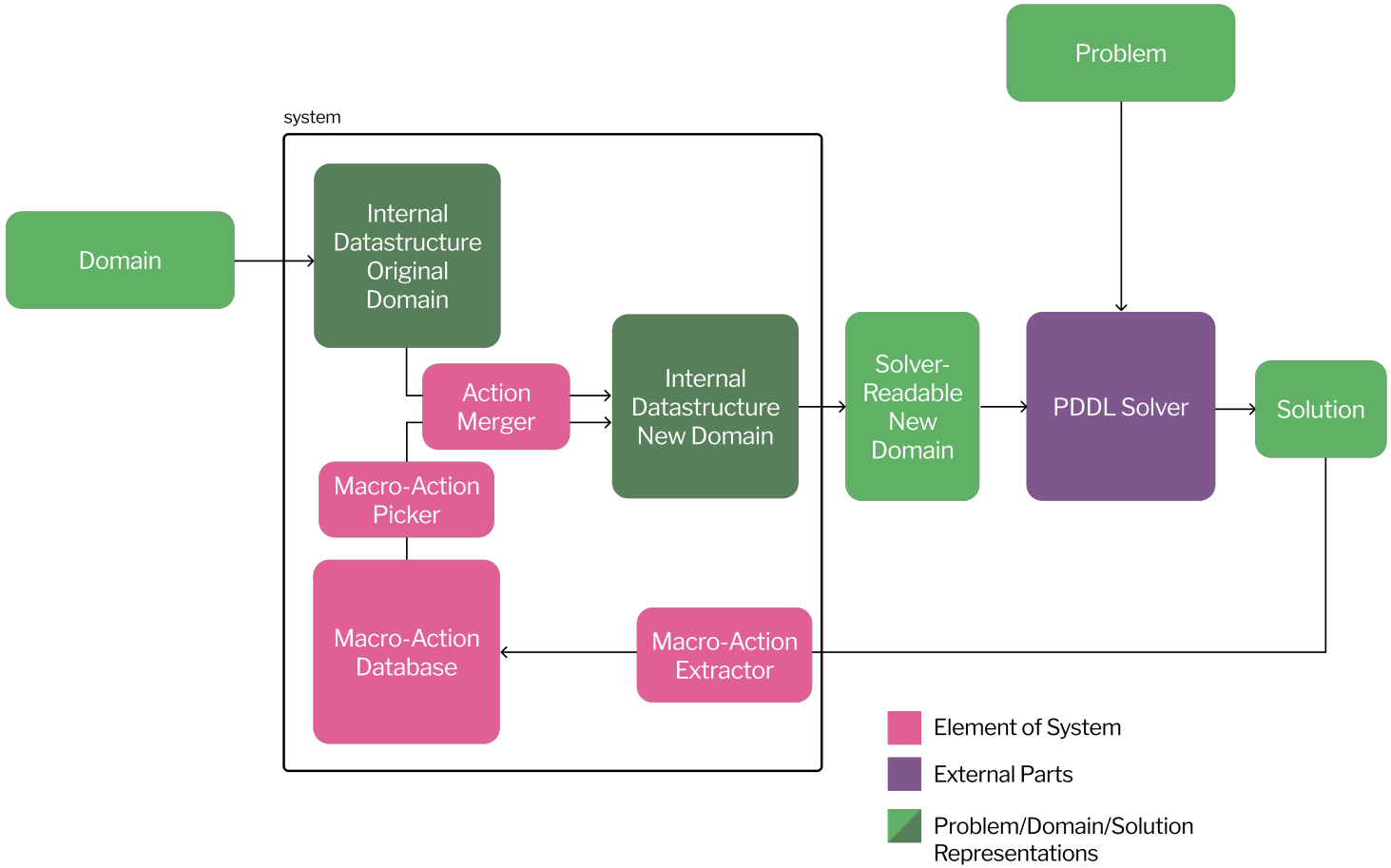


Figure 5.1: Diagram showing the design of the system and its different components.

5.4 Implementation

We implement the system in Julia[1], using the PDDL.jl library[2]. The macro-actions are stored in an SQLite database, which we use for its simplicity and speed. It is important that the database is fast because the amount of macro-actions stored can become massive if solutions are large and/or the solutions differ a lot.

[1] https://julialang.org/
[2] https://github.com/JuliaPlanners/PDDL.jl

Chapter 6

Results

To evaluate our method, we want to answer three main questions:

1. How does the performance of dynamic macro-actions change as we train on more problems?
2. At what point do we observe diminishing returns as we train on more problems for dynamic macro-actions?
3. How can we best estimate the utility of macro-actions?

We run our experiments on five domains. Two are from the PDDL gym problem set: the BAKING domain and the LOGISTICS domain. Furthermore, we take three domains from the IPC-4 benchmarks[1]: Satellite Non-temporal STRIPS, Pipesworld Non-temporal STRIPS No-tankage and Pipesworld Non-temporal STRIPS tankage, hereafter referred to as SATELLITE, PIPESWORLD-NO-TANKAGE and PIPESWORLD-TANKAGE. We will answer the questions for both of these domains separately.

We chose these domains because our implementation requires a model that has at most the requirements STRIPS and TYPING, and requires there to be one domain for many different problem-instances.

6.1 Experimental Setup

To run our experiments we use the A^* planner from the Symbolic Planners.jl[2] library using the `HAdd` heuristic. We generated the problem instances for BAKING and LOGISTICS using the parameters shown in Tables A.2 and A.1. The scripts that we used to run our experiments can be found on the Github repository[3].

For the IPC-4 problem sets, we took the first 20 problems because of time-constraints.

[1] `https://ipc04.icaps-conference.org/deterministic/`
[2] `https://github.com/JuliaPlanners/SymbolicPlanners.jl`
[3] `https://github.com/pujiii/thesis_implementation`

6.1.1 Machines

Due to time constraints, each experiment was run on one of three machines, hereafter referred to as MACHINE-1, MACHINE-2, and MACHINE-3. MACHINE-2 uses a AMD Ryzen 7 5800X 8-Core Processor 3.80 GHz processor, and MACHINE-3 uses a Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz processor.

6.1.2 Performance metric

Across all our experiments we use the same performance metric, from here on called "the performance metric", which is the ratio $\frac{\text{\#expanded nodes using macro-actions}}{\text{\#expanded nodes without using macro-actions}}$ (using a certain configuration), where 'expanded nodes' refers to nodes in our search tree that have been expanded upon by our solver, as provided by the solver. We use this metric to prevent hardware from having an effect on how we evaluate the performance of our method.

Importantly, a value of 1 for this method means that there is no difference in performance, a value < 1 means that we perform better using macro-actions, and a value > 1 means that we are performing worse using macro-actions.

6.2 Configuration experiments

The design of our system is flexible and allows us to run it using different configurations. To answer our questions we would want to use the best configuration to do so, so we can accurately assess the performance of our method.

To do so, we first run some experiments to determine what configuration to use. Namely, we will do this to determine the mode by which we deal with overlap, and the ideal number of macros to use.

6.2.1 The Ideal Overlap mode

To determine what overlap mode we should use for our main experiment, we split up our problem set into two parts: a training set and a test set. We train our system on the training set and then evaluate the performance of macro-actions using each overlap mode. We use problems 1-16 as our training set and 17-20 as our test set (see Table A.2)

The results can be found in Figure 6.1. We find that BEST vastly outperforms the other two overlap modes. Therefore we will use this for our future experiments. We also observe that allowing overlap performs worse than not using macro-actions at all and that choosing the largest macro-action in case of overlap also performs better, but not as well as BEST.

6.2.2 The Ideal Number of Macro-Actions to Choose

To determine the best number of macro-actions to pick, we perform a similar test, but only on the large problems. We split our training and test set such that our test set are

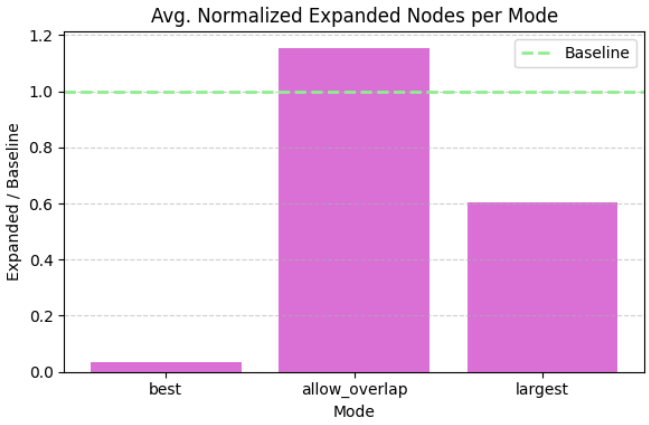


Figure 6.1: Barchart showing how different overlap modes perform for large problems (17-20) in the BAKING domain (see Table A.2). The green line shows the baseline of not using macro-actions. This experiment was run on MACHINE-1.

problems 17-20 for the BAKING domain.

The results can be found in Figure 6.2. We see that our system performs the best with ≥ 4 macro-actions. We pick 4 to run the rest of our experiments with because it has the best improvement.

6.3 Final Experiment

Now we can set up our main experiment to answer our earlier posed questions. We run this experiment with num_macros $= 5$ by picking the 'best' macro in case of overlap. For this we do the following:

1. We take our test set and measure the performance without using our system at all. That is, we do not pick macro-actions and we also do not add any knowledge to our database.
2. We iteratively run all of our problems using our system, learning as we go.
3. We do this for every method of estimating utility (see Section 5.3.2) to pick our macro-actions.

We can see the graphs showing the results of this experiment in Figure 6.3 and Figure **??**. Tables 6.1, 6.2 and 6.3 show the detailed results for the best-performing sorting criteria per domain.

Baking For the baking domain, we observe that selecting on the number of uses works the best by far. This is followed by the number of uses weighted by the number of unique actions.

We observe a significant increase in performance after 5 problems. The trend does not seem to converge in this experiment, so we cannot speak of diminishing returns.

Our best performing sorting criterium–the number of uses–has an average decrease in the number of explored nodes of 84% after solving 5 problems (see Table 6.1).

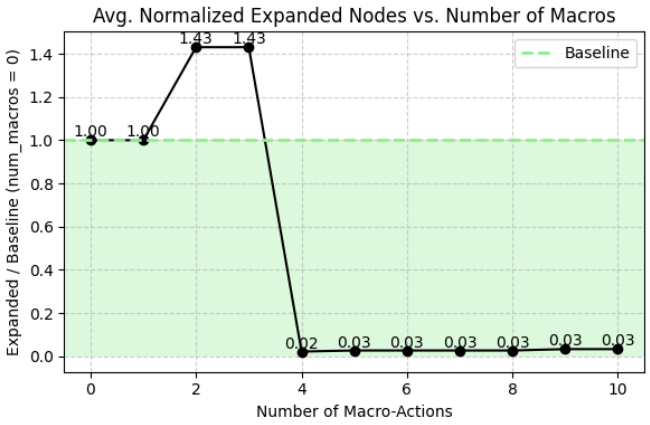


Figure 6.2: Line-graph showing the performance for large problems (17-20) in the BAKING domain (see Table A.2)of macro-actions for various numbers of macro-actions added. The green area shows the improvement area. This experiment was run on MACHINE-1.

problem no.	#expanded nodes		percent-wise
	without macro-actions	with macro-actions	performance increase
1	5	5	0.0%
2	6	5	17.0%
3	5	4	20.0%
4	6	5	17.0%
5	11	11	0.0%
6	15	4	73.0%
7	11	4	64.0%
8	15	4	73.0%
9	234	10	96.0%
10	723	12	98.0%
11	300	10	97.0%
12	897	12	99.0%
13	132	62	53.0%
14	372	134	64.0%
15	192	78	59.0%
16	552	29	95.0%
17	1466	54	96.0%
18	11726	107	99.0%
19	2346	86	96.0%
20	19250	204	99.0%

Table 6.1: Table showing the detailed results for the sorting criterium `num_uses` of the experiment shown in Figure 6.3 in the BAKING domain. The percent-wise performance increase is $(1 - \frac{\text{\#expanded nodes with macro-actions}}{\text{\#expanded nodes without macro-actions}}) \cdot 100\%$.

problem no.	#expanded nodes		percent-wise
	without macro-actions	with macro-actions	performance increase
1	16	16	0.0%
2	17	12	29.4%
3	21	14	33.3%
4	80	138	-72.5%
5	22	35	-59.1%
6	84	29	65.5%
7	264	33	87.5%
8	100	17	83.0%
9	410	33	92.0%
10	7205	3725	48.3%

Table 6.2: Table showing the detailed results for the sorting criterium `num_uses · num_unique_actions` of the experiment shown in Figure 6.3 in the LOGISTICS domain. The percent-wise performance increase is $(1 - \frac{\text{\#expanded nodes with macro-actions}}{\text{\#expanded nodes without macro-actions}}) \cdot 100\%$.

Logistics For the LOGISTICS domain we observe that the criterium of the number of uses weighted by the number of unique actions works the best by far.

We observe a decently steady performance increase boost from problem 6. However, the trend does not seem to converge yet by problem 10. Therefore, we cannot speak of diminishing returns in this experiment yet.

For our best performing sorting criterium–the number of uses weighted by the number of unique actions–we observe a 75% decrease in the number of nodes expanded in the search tree after solving 5 problems (see Table 6.2).

Satellite For the LOGISTICS domain we observe that the the number of uses weighted by the size performs the best, solving all problems (25% more problems than without using macro-actions). This is quickly followed by the unweighted number of uses, which can solve 15% more problems.

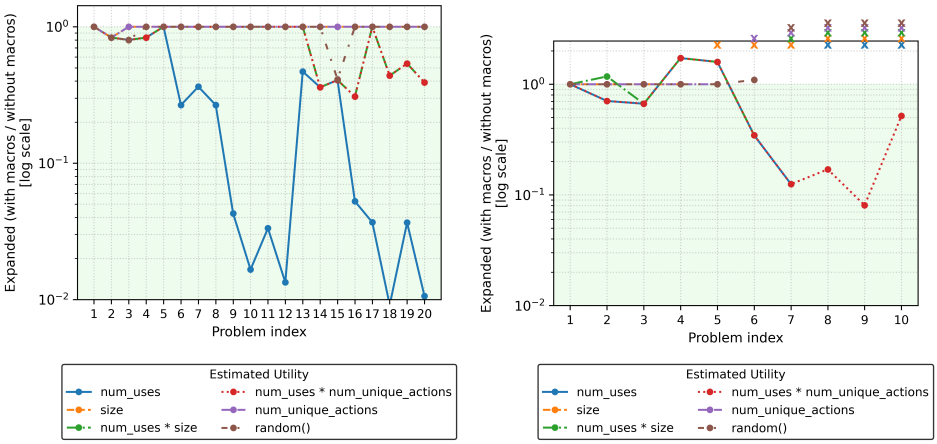
We observe that the improvement in performance occurs rapidly for this domain, starting at the first problem where we add macro-actions (problem 2). We cannot judge any diminishing returns, as because of the timeout we do not know the exact performance boost for many of the problems.

For our best performing sorting criterium–the number of uses weighted by the size– we observe a 33% decrease in the number of nodes expanded in the search tree after solving 5 problems.

Pipesworld For both the PIPESWORLD-NO-TANKAGE and the PIPESWORLD-TANKAGE domain we see that the macro-actions do not improve our performance. Rather, it decreases our performance. We do see that the number of unique actions works as well as not using macro-actions for the most problems for both domains.

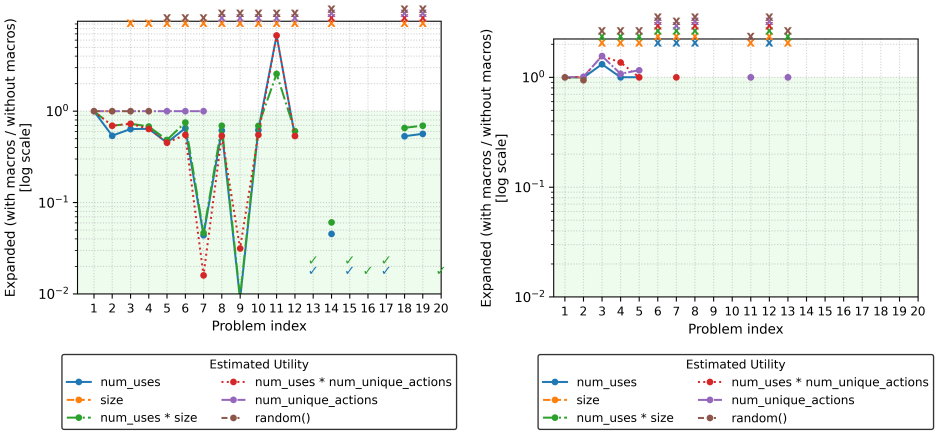
problem no.	#expanded nodes		percent-wise
	without macro-actions	with macro-actions	performance increase
1	9	9	0%
2	13	9	31%
3	11	8	27%
4	25	17	32%
5	31	15	52%
6	20	15	25%
7	751	35	95%
8	26	18	31%
9	3558	33	99%
10	29	20	31%
11	34	87	-156%
12	43	26	40%
13	Timed Out	82	x
14	1367	83	94%
15	Timed Out	48	x
16	Timed Out	109	x
17	Timed Out	116	x
18	32	21	34%
19	62	43	31%
20	Timed Out	166	x

Table 6.3: Table showing the detailed results for the sorting criterium `num_uses` · `size` of the experiment shown in Figure 6.3 in the SATELLITE. The percent-wise performance increase is $(1 - \frac{\text{\#expanded nodes with macro-actions}}{\text{\#expanded nodes without macro-actions}}) \cdot 100\%$.



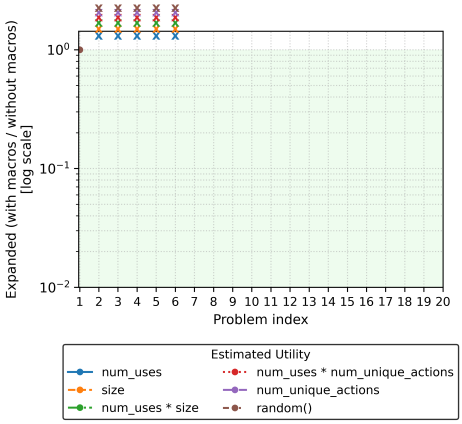
(a) Baking domain

(b) Logistics domain



(c) Satellite domain

(d) Pipesworld-no-tankage domain



(e) Pipesworld-tankage domain

Figure 6.3: Graphs showing the effect on the performance of our solver of our method over the course of training many instances.[4]

[4]Each line represents a different sorting criterium that is used. A cross at the top means that the problem can be solved within a timeout of 10 minutes without using macro-actions but not with macro-actions, and a check-mark at the bottom means that we could not solve it without macro-actions but we could solve it with. Each experiment uses 4 macro-actions and picks the macro-action with the highest estimated utility in the case of overlapping macro-actions. The the experiments for the BAKING, LOGISTICS, and SATELLITE domains were run on MACHINE-3 and the experiments for the PIPESWORLD domains were run on MACHINE-2. Each problem has a time-out of 10 minutes.

Chapter 7

Discussion

In the following chapter we discuss the results we have found in chapter 6. Firstly, we discuss the results of the configuration experiments and secondly we discuss the results of our final experiment.

7.1 Overlap Mode

We observed in Section 6.2.1 that the overlap mode BEST outperformed the other two overlap modes by far. This is expected, as our estimated utility of the macro-actions we add is the highest in this mode.

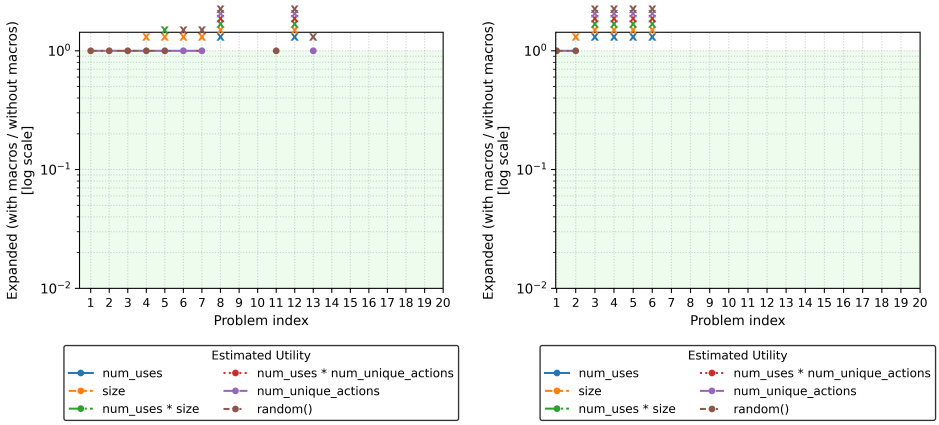
We also observe that LARGEST performs better than not using macro-actions, but performs considerably worse than BEST. This could be explained by the fact that we are eliminating overlapping macro-actions, but we do not pick the ones with the highest estimated utility.

Lastly, we observe that allowing overlaps performs worse than not using macro-actions. This could be because we are adding several macro-actions which essentially do the same so do not bring us closer to the solution necessarily, but do increase the branching factor.

7.2 Number of Macros

We observed in Section 6.2.2 that picking ≥ 4 macro-actions gives us a stark performance increase. We hypothesise that this is because it is a good balance between having picked enough macro-actions to have one that is useful for our problem, while not increasing the branching factor by too much, which increases the search-space exponentially as we add more macro-actions.

In Macro-FF[Botea et al., 2005] they use two macro-actions instead of four for PIPESWORLD. As the PIPESWORLD domains do not perform well on our domain with four macro-actions, we run a second experiment where we select only two macro-actions (see Figure 7.1).



(a) PIPESWORLD-NO-TANKAGE domain (b) PIPESWORLD-TANKAGE domain

Figure 7.1: Graphs showing the effect on the performance of our solver of our method over the course of training many instances. Each line represents a different sorting criterium that is used. A cross at the top means that the problem can be solved within a timeout of 10 minutes without using macro-actions but not with macro-actions, and a check-mark at the bottom means that we could not solve it without macro-actions but we could solve it with. Each experiment uses 2 macro-actions and picks the 'best' macro-action when given the choice of two overlapping macro-actions. All the experiments were run on MACHINE-2 and each problem has a time-out of 10 minutes.

We observe that for regular PIPESWORLD, we perform slightly better than with 4 macro-actions, but we still never have a perform increase w.r.t. not using macro-actions. For PIPESWORLD-NO-TANKAGE, we observe that it performs slightly worse compared to using 4 macro-actions. We can therefore not conclude that the reason that the PIPESWORLD domains do not work well on our domain is because we add too many macro-actions. Macro-FF did find that their method of using macro-actions and the macro-actions they did add improved performance with two macro-actions.

7.3 What Macro-actions are we using?

We analyse the macro-actions that we pick to solve problem 20 in the SATELLITE domain, so we can compare it to the macro-actions that were picked in Macro-FF[Botea et al., 2005] for said domain.

We will carefully observe what macro-actions our system gives for the sorting criterium `size` · `num_uses` in the Satellite domain. We do this because it is able to solve the largest problems out of all the sorting criteria. Here, we pick the following macro-actions:

- TURN-TO(A, B, C) + CALIBRATE(A, E, B)
- TURN-TO(A, B, C) + TAKE-IMAGE(D, E, B, A) + TURN-TO(D, F, E)
- TURN-TO(A, B, C) + TAKE-IMAGE(D, C, A, E)

- TURN-TO(A, B, C) + TAKE-IMAGE(D, E, B, A)

We observe that we have macro-actions that comprise of the same actions–in the same order–but that they have different parameter structures. Future work should investigate whether this is useful, or whether we need to be more strict in the filtering out of overlapping macro-actions.

7.4 Macro Sorting Criteria

In Section 6.3 we evaluated different utility estimations for macro-actions. We saw that for each domain, a different utility estimation worked best: for Baking–the unweighted number of uses, for Logistics–the number of uses weighted by the number of unique actions, for Satellite–the number of uses weighted by the size.

For the Pipesworld domains they all perform worse or as well as not using macro-actions. The reason why the number of unique actions solves the most problems in Pipesworld-no-tankage is unclear.

7.5 Diminishing Returns

We find that the rate of improvement varies vastly from the domain to domain. It is thus difficult to make a general statement that encompasses our results for all our domains–other than that the rate of improvement depends on the domain. Furthermore, we did not observe a convergence of the performance for any of the domains, which means we cannot determine a point of diminishing returns.

7.6 What Domains Work the Best?

We observe that our system works the best on the Baking, Logistics and Satellite domains. We hypothesise that this is because their actions have few parameters compared to the Pipesworld domains, which means that the branching factor does not increase as rapidly when constructing macro-actions.

7.7 Limits of Experiments

The system has an infinite amount of configurations. Because of this, we have chosen a small subset of configurations to test, but this means that a large amount of configurations are not tested. Most notably we did not run our configuration experiments (see Section 6.2) on all macro-sorting criteria, while that could have an effect. Furthermore, we ran them on only one domain. This makes us blind to configurations that may perform better than the ones tested, or ones that are perform better on more domains.

Another limit of our experiment is that we did not inform the solver to treat the cost of macro-actions as the sum of the number of actions, and instead we left it

unweighted. This means that the solver is potentially biased towards using certain macro-actions (see Example 4), and this effect can snowball (particularly in the number of uses logged by our method) as we solve more problems, which can affect our results.

Example 4. Take an optimal solution $s^* : a_1, a_2$ to some problem instance P in domain $D : (T, \mathscr{P}, \mathscr{A})$ where there are three actions $a_1, a_2, a_3 \in \mathscr{A}$. We add the macro-action $a_1(x, y) \circ a_3(y, z) \circ a_3(y, z)$ to our domain, such that $D' = D \cup \{a_1(x, y) \circ a_3(y, z) \circ a_3(y, z)\}$.

If we do not inform the solver that our macro-action is actually of length 2, it might find that the optimal solution is, for example, $a_1(x, y) \circ a_3(y, z) \circ a_3(x, z)$, rather than $a_1(x, y), a_2(x, y, z)$, whose true length is shorter, but seen as the same length as $a_1(x, y) \circ a_3(y, z) \circ a_3(y, z)$ to our solver. ⌟

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this work we explored the concept of dynamic macro-actions, which does not require prior training as it learns on-the-go and constantly re-evaluates the utility of macro-actions as it encounters more problems. This is contrasted with static macro-actions (which are up until now the most widely used form of macro-actions), where the estimated utility of macro-actions is never re-evaluated. The method is versatile as it can be used with any solver and for any domain. Lastly, the method is by its nature easy to tweak (primarily in the way we evaluate the utility of macro-actions). This not only paves the way for further research to improve the system, but also makes it very domain specialise-able.

To evaluate our method, we wanted to (a) see how our performance changes as we solve more problems, (b) investigate whether we find a point of dimishing returns– where solving more problems does not significantly improve our performance, (c) investigate what criteria we should use to find promising macro-actions.

We have seen that dynamic macro-actions show promising results for the BAKING domain (with an 84% avg. improvement in performance after solving our fifth problem), LOGISTICS (with a 75% avg. improvement) and SATELLITE (with a 33% avg. improvement) domains, but not for the PIPESWORLD domains (where we saw a decrease in performance). We have not been able to see a point of diminishing returns in our experiments, as the performance did not converge in our experiments. Lastly, we have not been able to pin-point one sorting criterium that works the best across domains; we find different sorting criteria perform well on different domains. Specifically, we saw that the number of uses works the best for the BAKING domain, the number of uses weighted by the number of unique actions works the best for the LOGISTICS domain, and the number of uses weighted by the size works the best for the SATELLITE domain.

We can conclude that the method of using dynamic macro-actions is a promising method, but requires more research to be reliably used for a vast range of domains.

8.2 Future Work

To improve the current method and more thoroughly investigate its strengths and weaknesses, more research is needed. Particularly, research can be done on more domains and with larger time-outs. Furthermore, research can be done to improve the estimated utility of macro-actions. A possible path to set forth for this question is to explore a more complex utility function, such as a weighted sum of the different attributes (such as size, number of uses, and number of unique actions).

Moreover, as we have noted in Section 7.7, our experiments have a bias towards already used macro-actions, ending in a potential snowball effect biasing our results. This can be combatted in two ways: by changing the cost of the actions according to their true size, or by adding exploration to our system in general. By adding exploration, we give lesser-used macro-actions a 'fair chance' to be explored. It must be noted that the results of exploration would only be visible after solving many problems, as our macro-action knowledge base $\mathcal{K}$ contains a substantial amount of macro-actions to explore. Unless, of course, the method is altered to limit how many macro-actions we store in our knowledge base $\mathcal{K}$.

Another extension to our experiments could include testing more configurations for our method, such as running our main experiment for the 'larger' overlap mode and testing our system for more domains on various numbers of macro-actions used. Different overlap modes can also be developed.

Once the method is more polished and examined, future work should include expanding the system to support more features of PDDL, such as durative actions, to be able to use this technique for more complex, real-world problems.

Bibliography

F. K. Alaboud and A. Coles. Personalized medication and activity planning in pddl+. In *Proceedings of the international conference on automated planning and scheduling*, volume 29, pages 492–500, 2019.

M. D. Bortoli, L. Chrpa, M. Gebser, and G. Steinbauer-Wagner. Enhancing temporal planning domains by sequential macro-actions (extended version), 2023. URL `https://arxiv.org/abs/2307.12081`.

A. Botea, M. Müller, and J. Schaeffer. Using component abstraction for automatic generation of macro-actions. pages 181–190, 01 2004.

A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res. (JAIR)*, 24: 581–621, 07 2005. doi: 10.1613/jair.1696.

M. Cardellini, M. Maratea, M. Vallati, G. Boleto, and L. Oneto. In-station train dispatching: a pddl+ planning approach. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 450–458, 2021.

W. Cheng and Y. Gao. Using pddl to solve vehicle routing problems. In *International Conference on Intelligent Information Processing*, pages 207–215. Springer, 2014.

A. Coles and A. Coles. Marvin: Macro actions from reduced versions of the instance. 01 2004.

A. Coles, M. Fox, and A. Smith. Online identification of useful macro-actions for planning. In *ICAPS*, pages 97–104, 2007.

A. Cropper. Forgetting to learn logic programs. *CoRR*, abs/1911.06643, 2019. URL `http://arxiv.org/abs/1911.06643`.

M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld. Pddl - the planning domain definition language. 08 1998.

S. Gulwani, O. Polozov, R. Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

I. Hernádvölgyi. Searching for macro operators with automatically generated heuristics. volume 2056, pages 194–203, 06 2001. ISBN 978-3-540-42144-3. doi: 10.1007/3-540-45153-6_19.

C. Hocquette, S. Dumančić, and A. Cropper. Learning logic programs by discovering higher-order abstractions, 2024. URL https://arxiv.org/abs/2308.08334.

C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(94)90069-8. URL https://www.sciencedirect.com/science/article/pii/0004370294900698.

S. Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, page 596–599, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc. ISBN 0934613028.

U. Schmid and F. Wysotzki. Applying inductive program synthesis to macro learning. 02 2000.

Appendix A

Problem Parameters Baking & Logistics

problem no.	n_1	n_2	m_1	m_2	o
1	2	1	1	2	2
2	2	1	1	2	3
3	2	1	2	2	4
4	3	1	2	3	4
5	3	1	3	3	3
6	2	2	4	4	5
7	5	1	2	5	5
8	5	1	4	5	5
9	5	1	5	5	5
10	5	2	5	5	5

Table A.1: Table showing the parameters for all the problem numbers for the problem set used for the experiments for the LOGISTICS domain. Here n_1 is the number of cities, n_2 is the number of locations, m_1 is the number of airplanes, m_2 is the number of trucks and o is the number of objects.

problem no.	n_1	n_2	m	o
1	1	0	2	2
2	1	0	3	2
3	1	0	2	3
4	1	0	3	3
5	0	1	2	2
6	0	1	3	2
7	0	1	2	3
8	0	1	3	3
9	1	1	2	2
10	1	1	3	2
11	1	1	2	3
12	1	1	3	3
13	2	0	2	2
14	2	0	3	2
15	2	0	2	3
16	2	0	3	3
17	0	2	2	2
18	0	2	3	2
19	0	2	2	3
20	0	2	3	3

Table A.2: Table showing the parameters for all the problem numbers for the problem set used for the experiments for the BAKING domain. Here n_1 is the number of souffles, n_2 is the number of cakes, m is the number of pans and o is the number of ovens.