Working-sets in automated planning

Finding better performing sets of actions in established domains

Søren Drejer Jakobsgaard

Software HCI Masters, cs-24-sv-10-04, 2024

AALBORG
UNIVERSITY

Software
Aalborg University
https://www.aau.dk/

Title:

Working-sets in automated planning

Project:

Master Thesis

Project period:

February 2024 - June 2024

Project group:

cs-24-sv-10-04

Participants:

Søren Drejer Jakobsgaard

Supervisor:

Alvaro Torralba

Page Numbers: 15

Date of Completion:

June 9, 2024

Abstract

Automated planning is a complex issue and multiple avenues are researched to help solve planning problems. Composite actions like meta-actions are one such method. These meta-actions functions as shortcuts making planners skip steps they would otherwise be deciding the next action. Adding these extra actions through, increases branching factor so meta-actions can't be added haphazardly.
This paper explores the idea of removing actions that are made redundant from meta-actions, thereby decreasing the overall branching factor and hopefully improve performance of planners. First some preliminaries actions are taken to ensure that removing the actions does not result in sets of actions that can not solve problems. Afterwards, an AI tool called SMAC is used to select sets of actions based on different metrics. These selected sets are then sent to a cluster and tested alongside other sets to compare results. The results show there is broadly a gain in planning speed and coverage when removing redundant actions across domains. But each planning domain is very distinct, so no completely common pattern to what feature makes an action more or less suited to be included/excluded could be determined.

I. Intro

Automated planning is the topic of finding sequences of actions that solve a given problem in some specific domain. Domains describe what type of problem is to be solved, which could be everything from how the drone in a warehouse moves around packages to how to mix drinks at a bar. Problems describe a specific scenario in a domain, both the initial state and the goal state.

Solving problems in automated planning can quickly become very complex. As a very naive example, something as simple as a problem involving 10 boolean elements would have 2^{10} states possible. Of course planners, the engines used to solve planning problems, do not naively explore every possible state. Using heuristics is the way to give planners some direction on what path to take towards the goal [1]. Using composite actions like macro-action or meta-actions is another way to help planners solve problems[2]. Whenever planners have performed an action they have to consider the next step to perform, however in a lot of domains, the same sequence of actions shows up multiple times, if say a robot picks up a package the most common next step for it to perform might be to move the package somewhere else. Composite actions are attempts to find these reoccurring sequences of actions and give them their own actions instead. This essentially makes planners skip steps where it has to consider its options. A problem with adding these composite actions is that it increase branching factor, simply said, more options for planners to consider slows planners down. Therefore, care must be taken when selecting composite actions to compliment the base-actions of a domain, in order to not negate the gain of using them in the first place. In some cases, these composite actions might even be able to fully replace the base-actions, which creates the possibility of removing these base-actions to lower the branching factor.

The paper *"Can I Really Do That? Verification of Meta-Operators via Stackelberg Planning"* [3] created a number of meta-actions for some domains. Of those created they selected 2 meta-actions in the end to add to each explored domain and called them domain_best (for each domain that is). The meta-actions were chosen depending on the coverage (problems solved) and the time to solve the problems. A drawback of this was that the meta-actions were tested separately, so any synergy between the two meta-actions is unknown. In my previous work I took this domain_best and attempted to remove all possible base-actions that were made "redundant" by the meta-actions. The number of domains tested were few, but the results from that work showed that across most of the domains, a very small gain in search time was achieved, but in one domain the performance was terrible. It was speculated that removing all possible base-actions was too strict of a method, and instead the best performing set of actions were very likely in between the 2 extremes (removing all possible or removing none).

This project is a continuation in this process. First off more meta-actions are included in the domains to be explored. This obviously significantly increases branching factor, which means that unlike the previous work which only considered base-actions for removal, this time, the meta-actions themselves are to be considered for removal. The number of possible sets of actions grows by 2^n with n being the number of actions. Therefore steps are taken to skip processing sets of actions that can quickly be deemed unable to solve problems.

The paper *"On the Effective Configuration of Planning Domain Models"*[4] studies the effect, the ordering of actions in domain files has on planning results. To do this the paper is using an ML tool named SMAC[5]. SMAC is particularly convenient for this, as it has numerous presets for optimizing problems beyond just machine learning problems. Similarly, this project will use SMAC to find sets of actions that are promising to have a better performance than the domain_best set of actions. These sets are then sent to an external cluster and tested to judge the merits of these sets.

II. Background

To understand the work that has been done, first an explanation of what planning is, and the concept of what a Meta-action is, is required.

A. Classical planning

STRIPS (Stanford Research Institute Problem Solver)[3] is a way of expressing automated planning tasks, by specifying facts, actions, initial state, and goal state. STRIPS assumes a closed world ie the entire state is known at the start. A planning task in STRIPS is a tuple $\Pi = (F, O, I, G)$. Where F is a set of facts, O is a set of operators, $I \subseteq F$ is the initial state set and $G \subseteq F$ is the goal set. Each action $o \in O$ has a precondition($pre(o)$), an add effect($add(o)$) and a deletion effect($del(o)$). These 3 sets are all subsets of F. A state s is a set of facts which describe the current state, where the presence of said facts means for the fact to be **True**. An operator o is only applicable in state s if $pre(o) \subseteq s$. Whenever an operator o is performed, the state is changed by removing from s the set of $del(o)$ and adding to s the set of $add(o)$. The solution to a planning task in STRIPS is a sequence of actions that goes from the initial state to a state s such that $G \subseteq s$.

To help explain an example in STRIPS, a simple domain, gripper is presented. The gripper domain models a world where there are balls that need to be moved between rooms and there is 1 robot arm with 2 grippers, each capable of carrying one ball. The domain has 3 base-actions. **Pick** which picks up a ball from a room and occupies one of the grippers. **Drop** which places a ball in a room and frees the relevant gripper. And **Move** which moves the arm with all grippers from their present room to another one. All rooms are connected and there are no restraints in regard to moving between rooms.

In an automated planning task using STRIPS in the gripper domain, F would contain a fact like *"ball1-is-in-roomA"* O would contain an operator *"drop-ball1-in-roomB"* (to model the **Drop** action being performed in roomB on ball1) with the precondition set being *"ball1-in-hand"*,*"arm-in-roomB"*. The add effect set being *"ball1-is-in-roomB"*,*"hand-free"*. The deletion set being *"hand-free"*,*"ball1-in-hand"*. It contains *"ball1-is-in-roomA"* and G would be *"ball-is-in-roomB"*. As can be inferred from this example, it can be quite cumbersome to create big planning tasks. If say in the gripper domain, a ball is directly involved in 10 facts, each ball added to the plan would necessitate 10 facts being added.

B. PDDL

PDDL was designed in 1998 to make the International planning competition possible [6]. PDDL is a language specification that builds on STRIPS to become more expressive. While in STRIPS facts had to have grounded facts for each object in a plan, in PDDL the specification of a planning task is separated into a lifted Domain specification and a grounded problem specification. A PDDL planning task is a tuple $\Pi = (Domain, Problem)$. Where $Domain = (P, A)$ and $Problem = (C, I, G)$, so the full PDDL planning tuple becomes $\Pi = (P, A, C, I, G)$. P is a set of predicates, A is a set of action-schemas, C is a set of object constants, I is the initial state and G is the goal state. Objects are typed and the action-schemas and predicates are parameterised by these types, allowing reuse like a regular programming language. Actions-schemas are similar to STRIPS operators in having pre,add,del sets, but they also have a parameter list which tells variable names and their types. The P and A set are yet to be grounded, and this means that the domain is independent of the problem (unlike STRIPS). The domain specifies what type of objects can exist in the domain, and what actions exist to change the state, while the problem specifies the actual objects in the current task, the initial state and the goal state. Action-schemas become grounded when they are tied to a specific instance of an object. Similar to STRIPS a state s in PDDL is a conjugation of predicates where their existence in the set s indicates **true**. Like STRIPS the solution to a PDDL task is a sequence of actions from the initial state to a state s such that $G \subseteq s$.

Again a gripper example to explain. P would contain a predicate $(room?r)$ room being the type and $?r$ being a placeholder for whatever element it is eventually grounded with. A would have an action-schema **Drop** with the preconditions $(ball?obj)(room?room)(gripper?gripper)$- $(carry?obj?gripper)(at-robby?room)$ and the add effects set being $(at?obj?room)(free?gripper)$ and finally the del effects set $(carry?obj?gripper)$. Unlike the STRIPS example where the example of the Drop action was very specific for what ball and what room, here the Drop action is generic with the use of the placeholders and can be performed with any ball, room and gripper, provided the preconditions are satisfied.

C. Meta-actions

Meta-actions [7] are as mentioned earlier composite actions. They are an extension of macro-actions which very briefly explained, are sequences of actions put together to one action and the effects of a macro being the same effects as performing its composite actions in sequence one at a time. For an example, imagine in the gripper domain a macro-action that would be **pick-move-drop**, which have exactly the same effect as performing **pick,move,drop** in sequence. However, Macro-actions have a lot of side effects and preconditions, like the aforementioned macro-action having the side effect of the gripper-arm being in a different room, and the precondition of the gripper being free, and the arm already being in the first room. The equivalent meta-action would disregard the preconditions mentioned and would try to leave the state of the planning-task the same as when it started to act, except for the effect of the ball now being in the desired room. Creating a Meta-action based on an existing Macro-action the first step is to remove parameters that are absent from the effects of the macro. If a parameter is removed, all predicates containing it are removed from the precondition set as well. Next, any parameter that is considered a side effect of the macro action is removed from the effects and precondition set. This creates meta-actions with no side effects, and fewer preconditions, allowing them to be performed more freely than a Macro-action. When a plan has been found with meta-actions, a reconstruction phase has to be performed. This reconstruction phase finds a sequence of actions which when performed achieves the same state as the meta-action did, this way meta-actions essentially becomes a way of partitioning the problem into smaller problems. This reconstruction phase is not included in the timings in this paper, and instead Meta-actions will be treated as fully functioning actions.

An example of meta-actions being more versatile than macro-actions. For argument's sake, let's change the gripper domain, it instead of as usually having only 2 connected rooms, it now has a series of rooms, where the rooms have bidirectional connections $room_1 \leftrightarrow room_2 \leftrightarrow room_3 ... \leftrightarrow room_n$. The **pick-move-drop** macro operation would be unsuitable for this scenario, as moving a ball from $room_1$ to $room_5$ via this macro, the planner would perform multiple redundant pick and drop actions. Adding more macros to match the distance of farthest distance of the rooms is also unfeasible, as if there are n rooms, then n-1 number of macros would need to be added, significantly increasing the branching factor. Using the meta-action **pick-move-drop**, the planner could use this meta-action to immediately place the ball in the desired room, only caring about the method of how to achieve this later when reconstructing the plan.

III. Problem

The problem when adding more actions to domains is that the branching factor of the planners increases. Simply said, the planners have to consider more options when more options are available. While in most cases the gain of having good meta-actions are still overall a net positive, it is still something to consider. To solve this problem, then, the hope is that by removing actions that meta-actions makes redundant, that runtime could further be increased.

Working-sets is coined to describe sets that have a reduced number of actions compared to

the original domain's actions, but that can still solve the same problems. The removed actions must be "safe" to remove from the domain.

Definition 1. *Let $\mathcal{D}$ be a domain, $\mathcal{A}$ be $\mathcal{D}$'s actions and $\mathcal{S}$ be a set of actions. We say that $\mathcal{S}$ is safe to remove as long as any problem solvable in $\mathcal{D}(\mathcal{A})$ is still solvable in $\mathcal{D}(\mathcal{A} \setminus \mathcal{S})$*

The goal of this project is to find these working-sets which has the best performance. The work continues of my previous project. That project had very strict method to find these "better" performing working-set. It assumed that a smaller number of actions is always superior, it removed all possible base-actions while still being able to solve problems previously solvable. The results showed that generally a very small gain would be achieved in most of the tested domains, but in some the results would show terrible performance comparatively to the original versions. In order to broaden the search for these patterns, some changes were made. In the previous project, at most 2 meta-action per domain was explored. They were chosen because they had been deemed the best by the creators of the meta-actions Github. The meta-actions were sourced from[7]. In this project all the defined meta-actions from the GitHub are included (initially). In the previous project it was also assumed that those meta-actions had to be used, and were thus never removed when exploring for working-sets, but with the increased number of meta-actions these actions has to be considered for removal as well.
This brings us to defining the input and overall output of the problem The input of the problem is

- A Domain D
- A set of actions A belonging to D. This set of actions contains both all the base actions and all the meta-actions
- A set containing only the meta-actions M
- Multiple problems P belonging to D

The output by the end is a working-set that optimizes the coverage and totaltime when solving a testing set of instances.

To find this working-set the work is divided into 2 steps. First, the frontier of what working-sets are possible is found by looking for minimal working-sets, sets that still solves problems, but have removed as many actions as possible. In the second step, working-sets are selected based on different metrics. Here, the previously found frontier is used in order to skip considering working-sets that lies beyond minimal-sets.

IV. Finding minimal working-sets

The minimal working-sets are found by exploring possible sets on a single problem, and later these possible working-sets are tested on a number of known solvable problems. If a single problem fails, the possible set is thrown out. If it passes all the problems, then it is assumed to be a working-set. As long as the set of problems is representative of the infinite class of problems in the related domain, then this check is considered good enough.
Working-sets also have two important properties, that the algorithm 1 is making use of when exploring and confirming working-sets

Proposition 1. *Let $\mathcal{S}$ be a set of actions and $\mathcal{D}$ be a domain. If $\mathcal{S}$ can safely be removed from $\mathcal{D}$, then any subset $\int \subseteq \mathcal{S}$ can safely be removed too.*

Proof. Proving by contradiction let's assume that $\mathcal{S}$ is safe to remove but there is a subset $\int \subseteq \mathcal{S}$ that isn't safe to remove. This is not possible, as whatever solution that was used with $\mathcal{S}$ removed can be used on any subsets as well. □

Proposition 2. *If $\mathcal{S}$ cannot be safely removed from $\mathcal{D}$ then any superset $\mathcal{Z} \supset \mathcal{S}$ can not safely be removed either.*

Proof. Proving by contradiction lets assume that $\mathcal{S}$ can not be safely removed but there is a superset $\mathcal{Z} \supset \mathcal{S}$ that can. If that is the case then $\mathcal{S}$ would be safe to remove too as whatever solution for $\mathcal{Z}$ could be used for $\mathcal{S}$ as well, which is a contradiction. □

To help visualize these properties, a description of a search tree when exploring for working-sets is in order1.

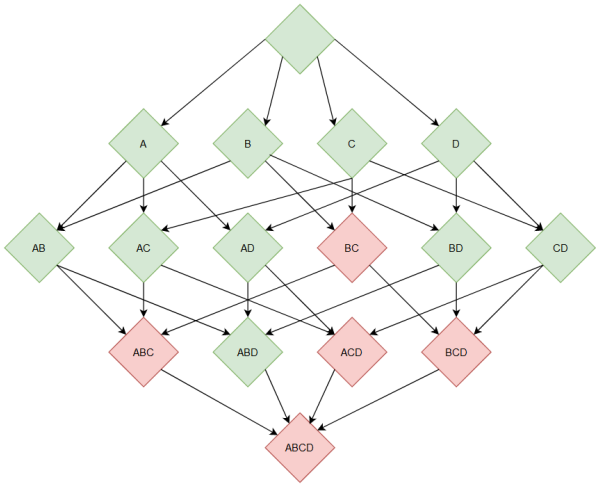

Figure 1: Figure showing when searching for a working-set in a domain with 4 actions A,B,C,D. The blank square represents removing zero actions while the bottom ABCD represents removing all actions

Confirming ABD(working-set technically being C) to be a working-set would confirm any subsets higher in the tree by proposition 1. Likewise confirming BC to **not** be a working-set would also mean that any following superset is not a working-set by proposition 2. The minimal working-sets in 1 are AC,ABD,CD. These are the working-sets that form the frontier of what working-sets are possible, as any working-set later being explored in step 2, has to be a subset of at least one of these sets.

The exploration algorithm was also used in the previous work but with the desire to include more domains and include more meta-actions when exploring working-sets, on top of meta-actions themselves being candidates to remove, it was necessary to optimize the previous project's exploration and confirmation of working-sets.

The propositions regarding working-sets let the algorithm skip using the planner sometimes, which is a significantly expensive operation. Algorithm 1 takes as input a Domain, a single problem, and all the meta-actions in the domain. *Removed_sets* is a set of sets that contains the actions you would remove to form a working-set. *Queue* is simply a queue that will contain the sets

Algorithm 1: finding_working_sets

Data: A domain D containing a set of actions A(both base and meta are included) , a single problem from P, a set containing only D's meta-actions M.
Result: A set of potential *Removed_sets*

```
1  Removed_sets = {M};
2  failed_sets = ∅;
3  Queue = PriorityQueue();
4  for Action in A do
5  |  Queue.push({Action})
6  end
7  while Queue NOT empty do
8  |  RA = Queue.pop();
9  |  if ∃set ∈ Removed_sets such that RA ⊆ set
   |   then
10 |  |  add_successors_to_queue(RA, Queue);
11 |  |  continue
12 |  else if ∃set ∈ failed_sets such that RA ⊇ set
   |   then
13 |  |  continue
14 |  plan = check_if_sovable(D(A \ RA);
15 |  if plan is not False then
16 |  |  NA = Analyse_plan(plan, A);
17 |  |  Removed_sets.add(NA);
18 |  |  add_successors_to_queue(NA, Queue);
19 |  |  Removed_sets.add(RA);
20 |  |  add_successors_to_queue(RA, Queue);
21 |  else
22 |  |  failed_sets.add(RA);
23 end
24 return Removed_sets
```

to explore. Each individual action is added as a set to the queue, to be explored. After the preliminary steps, *Queue* is popped to *RA* (Removed actions) on line 8 and the element is explored. On line 9 it is checked to see if property 1 applies. If that is the case, its successors (sets with an added action) are added to the queue to be explored, and the set itself is skipped. Otherwise on line 12 *RA* is checked to see if property 2 applies, if that is the case then it is simply skipped. Line 14 the modified problem is finally sent to the planner if the previous checks failed. The *RA* are removed from the domain and the task is run on the problem. If it succeeds, then $(A \setminus RA)$ is added

to *Removed_sets*, and its successors are added to the queue to be explored.

The plan is also analysed to see what actions weren't used. Clearly this unused set must also be safe to remove and therefore a removed-set, this addition lets algorithm 1 get sets that potentially are very far into the search tree allowing it to later skip a considerable amount of sets by the check of property 1 on line 9. The set is therefore also added to *Removed_sets* and its successors are added to the *Queue*. If the planner returns false, then *RA* is added to *failed_sets*. This analysis of the plan is the most significant change compared to the previous works version.

The algorithm is ensured to terminate as when adding more elements to the queue inside the add_successors_to_queue function there is a set containing all sets that have already been added to the queue, ensuring that no set is added twice. The search space grows exponentially with the number of starting actions 2^n. It is finite but can be excessively large. Since the runtime is dependent on *Queue* whose size ultimately depends on the searchspace this mean that the worst case runtime is $O(2^n)$

The algorithm 1 is ensured to only return working-sets (technically the inverse of the working-sets) for whatever problem was input.

Proof. Proving by contradiction let's assume that *Removed_sets* contains a set that doesn't form a working-set after being removed from *A*. This is not possible as the only place sets are added to the *Removed_sets* is specifically when that set has been tried on line 14 or when the set was concluded to not being used on line 16. □

The algorithm is also ensured to implicitly return all possible working sets.

Proof. **1.** From proposition 1 a working-set's subsets are all also a working-set.
2. Proving by contradiction let's assume that there is an actual working-set *W* that exist that is not found by the algorithm. This would mean that *W* exists beyond the frontier of working-sets found. This is not possible, as proposition 1 and 2 explains. Either the path to *W* is possible by proposition 1 and *W* would have been found along with any intermediary working-sets. Or it is impossible by proposition 2. □

After algorithm 1 returns, the found working-sets are tried on the rest of problems in *P*. The version used in the previous project would test every single working-set. However, there was no need for this as proposition 1 says, when confirming a superset it would also confirm all its possible subsets. This project's version instead only tests the distinct supersets. The found working-sets are filtered so that only supersets remains. These supersets are then all tested on 1 problem at a time. If any set fails, it is removed and instead added to *failed_sets*. When a problem has been tried on all sets, if not a single 1 failed then go to the next problem, if any of them failed instead then algorithm 1 is essentially run again, except the *failed_sets* kept, the sets that succeeded are used as *Removed_sets* and the problem used is the one that was just tried.

V. Selecting Working-sets

After the frontier of working-sets has been found, individual working-sets are selected. They are selected based on a number of metrics to later be sent to a cluster for testing. The metrics and their reason for inclusion are as follows.

- **Totaltime:** The total time it takes for a planner to find a solution. As totaltime is one of the 2 metrics that is valued in the end, it makes sense to include it when picking working-sets.
- **Searchtime**: The time strictly spend searching for a solution after any preliminary steps. If this metric proves to find the best performing working-set, then it suggests that the searching time was the biggest bottleneck in its planning speed.
- **Planlength**: The number of actions performed to reach the goal state of a problem. A well performing working-set here suggests the best performing working-set solves plans in fewer actions than other sets of actions. This is one of the qualities using composite actions is supposed to contribute to.

- **Generated states**: The number of states generated in the frontier when the planner is exploring. A working-set being best from this metric would mean, that the planner is likely overwhelmed by the number of potential states it creates, when using other sets of actions.
- **Expanded states**: The number of states the planner has explored from the frontier. This one being best could possibly mean that using other sets of actions than this one results in the heuristics getting sabotaged, as the planner has to explore more states.
- **Individual components**: Converting the found plan to a partial order plan, and then counting the number of Independent Graphs that could be constructed from the partial order.
- **Makespan**: The length of the largest individual component. This one together with individual components would mean that the domain is particularly suited for partial order planning.

For each metric listed, the working-set found that performed best is selected. In most metrics a lower number is desired, except for individual components where a high number is desired instead. Despite knowing the frontier of possible working-sets, the number of possible sets is still a too high in most domains to explore all of them. Instead an ML tool called SMAC[5] is used to select these working-sets. From the documentation of SMAC it describes it as *"SMAC is a tool for algorithm configuration to optimize the parameters of arbitrary algorithms, including hyperparameter optimization of Machine Learning algorithms. The main core consists of Bayesian Optimization in combination with an aggressive racing mechanism to efficiently decide which of two configurations performs better."* [8]. SMAC offers multiple facades which are presets to ease the use ML without the user needing to know all the complexities of Machine learning. As the issue being explored in this project is not a really a machine learning problem, the AlgorithmConfigurationFacade(AC) is chosen, which is more of a generalised way of hyperparameter optimization for algorithms. RandomFacade is also chosen to see if choice of facade is important

The SMAC tool will do numerous trials, each trial a working-set being explored. A Configuration defines for the SMAC tool which actions to include or exclude in the working-set being explored during a single trial. The facades choses which configuration to use during each trial. The working-set created from the configuration is first checked to see if it lies beyond the frontier previously defined. If that is the case, it is immediately discarded. Otherwise the working-set is tried on 3 problems which was also used in the Finding working-setsIV section. The average of every metric is saved as a container in a list for later, but the metric that is used by SMAC for optimization is strictly the average total time. When SMAC is done running, the list of containers are searched to find the best performing working-sets in each metric, whereafter a PDDL domain file is created for each picked working-set. Any metrics that share a working-set as best performing are combined. It might seem counterproductive to let SMAC optimize towards best total time, while still being interested in other features as well. This is fine as the ultimate goal of this project is to achieve better performance in time. The metrics are still considered as the increased difficulty of the test problems, might reveal that going strictly for total-time on easier problems is not suited for the domains, and focusing on some other metric is more important.

VI. Experiments

A. finding_working_sets runtime experiment

Setup: The tests in this subsection was performed on a Windows OS laptop, with an i5-1035G1 1.00GHz CPU and with 8 gigs of ram.

Popping elements in Algorithm 1 can be done with either Breadth-first or Depth-first. To find out which method to use along with a comparison to the previous work's method a runtime test is done across the domains. Important to note here, this test is strictly the running of finding_working_sets 1 once, without the following confirmation of

Domains	Number of Actions	Meta-actions	New - Depth	New - Breadth	Old - Depth	Old - Breadth
barman-sat11-strips	28	17	time	time	time	time
depots	11	6	104,07	**10,25**	446,26	714,34
grid	8	3	21,84	**5,35**	28,75	35,04
gripper	4	1	2,86	**2,28**	2,56	2,85
logistics00	8	2	19,40	**2,95**	35,21	44,98
miconic	10	6	19,85	**5,00**	158,32	302,24
rovers	21	12	time	time	time	time
satellite	12	7	83,91	**11,90**	1053,26	2024,04
zenotravel	9	4	26,73	**5,37**	97,82	161,68

Table I: Tabel showing the number of actions and how many of those actions are meta actions. New is the algorithm 1 while Old is the one used in the previous work. The numbers are in seconds. Time means the time limit was reached. Bold numbers show best performing configuration in domain.

working-sets. The domains were given 2 hours to complete the task of finding the frontier, i.e. the minimal working-sets.
As can be seen in table I the new version using Breadth-first search is strictly better than all the other options in the domains that finish at least. This is likely due to the addition of the plan analysis that adds the unused sets of actions. Before the quickest way to confirm big sets of actions that allowed to skip solving problems was by going depth first, furthermore the following set to get confirmed then, is likely similar to the last one, not giving a lot of new options to skip. Going breadth-first instead, a lot of differing working-sets deep in the search tree are found, quickly allowing the skipping of solving numerous smaller working-sets.

Barman with its 28 and rovers with its 21 number of actions did not finish in any of the attempts. The number of possible sets is simply too large. Barman is therefore tested with an increasing number of meta-actions added to both see the significance of the number of actions across the algorithm configurations, but also to see how many actions should be removed to let Algorithm1 finish in a reasonable amount of time.

The classic barman domain contains 12 base actions, so the first domain to be run is named barman13 signifying the inclusion of the 12 base-action and a single meta-actions. The following domain, named barman14 means the same actions as barman13 plus 1 new action and so on. Each attempt at a domain is give one hour and the planner is give max 3 mins per run.
Again, the new version going breath first performs best, but it still hits the time limit fairly quickly when increasing the number of initial actions. With this, barman 21 is selected as the set to continue with, as though while it wasn't tested, it is still assumed to be finishing in a reasonable amount of time.
Rovers is also tested to find a smaller domain and to see if Breadth first, being better, holds in more than one complex domain. This time the old algorithm is skipped as the improvements of the new algorithm has already been shown multiple times. It is given 2 hours to complete. Again breadth first performs best. and rovers18 was chosen to be carried forward.

B. Cluster test results

To test the results of using the working-sets found from section V, they are compared against the original domains without any meta-actions along, with the domain_best mentioned in section I[7].
The chosen domains to experiment with are gripper, grid, barman, logistics, zenotravel, miconic, satellite, and depots (rovers was unfortunately skipped due to an error occurring when confirming its minimal working-sets).
The SMAC tool was given 5 hours and, 20000 max number of trials, to find working-sets for each domain. Table II shows the found working-sets number of actions. They are named depending on the facade that was used to find them and the metric that selected them. Sometimes the metrics when using a facade will produce duplicates, these

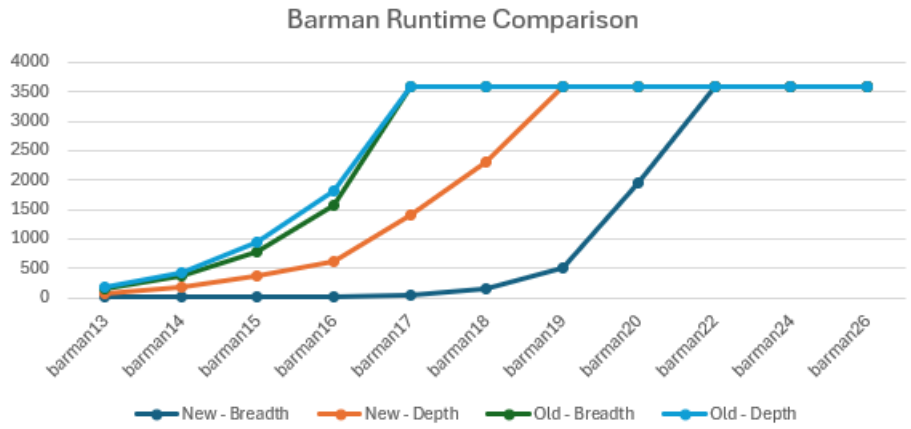

Figure 2: Figure showing the growing runtime of adding actions to barman. The numbers on the Y axis represents seconds. The number, after barman, means the number of actions.

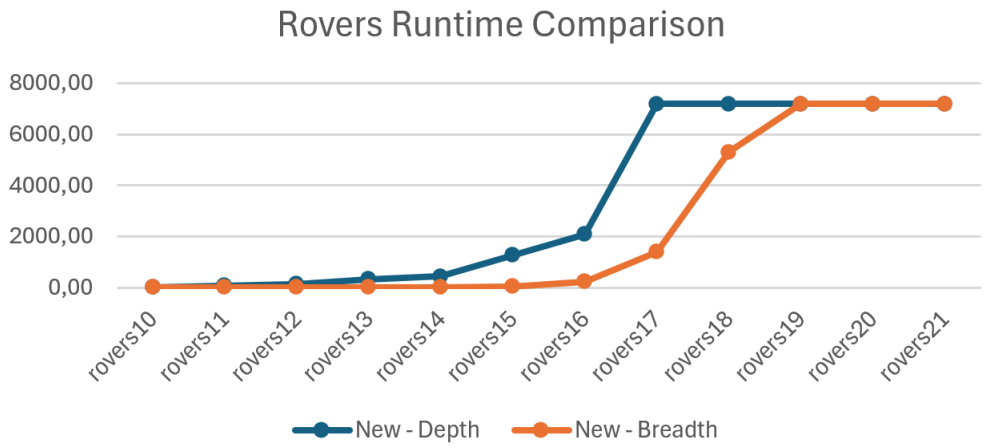

Figure 3: Figure showing the growing runtime of adding actions to Rovers. The numbers on the Y axis represents seconds. The number after rovers, means the number of actions.

are simply combined, and named appropriately (duplicates when comparing between facades still exist). These working-sets are the final sets that are sent to a cluster to evaluate their performance.

Evaluation Metrics: The metrics the results are measured on are **coverage:** The number of solved problems, and **totaltime:** the time it takes to search of a solution and any preprocessing.
Setup: The tests are performed on DEIS-MCC Naples cluster [9]. The planning engine used is the Fast downward planner[10] and for the search heuristics the state of the art lama-first[1] is used. The problems used are 30 "agile" problems created from “Auto- matic instance generation for classical planning,” [11]. The tasks were given a time limit of 10 minutes and 4 gigabytes of ram.

Results: Table III shows the coverage of all the working-sets, Domain_best, and Original. Across all the domains at least one working-set performed better or at least as good as either Domain_best or Original. Particularly Grid and Zenotravel achieved found a working-set with

Barman		Depots		Grid		Gripper		Logistics		Miconic		Satellite		Zenotravel	
Domain_best	(2)14	Domain_best	(2)7	Domain_best	(1)6	Domain_best	(1)4	Domain_best	(2)8	Domain_best	(2)6	Domain_best	(2)7	Domain_best	(2)7
Original	(0)12	Original	(0)5	Original	(0)5	Original	(0)3	Original	(0)6	Original	(0)4	Original	(0)5	Original	(0)5
AC:c	(9)21	AC:c	(4)6	AC:c	(3)8	AC:gs-ms	(0)3	AC:ex	(2)5	AC:c	(3)4	AC:c	(7)12	AC:c	(3)5
AC:ex	(4)12	AC:ex	(4)7	AC:gs	(1)5	AC:tt-st-pl-ex-c	(1)4	AC:gs	(2)4	AC:pl-ms	(6)10	AC:ex	(4)8	AC:ex	(2)7
AC:gs	(4)13	AC:gs	(3)4	AC:pl-ex-ms	(3)4			AC:pl-ms-c	(2)8	AC:st-ex-gs	(3)3	AC:pl-ms	(2)7	AC:gs	(2)2
AC:ms	(4)13	AC:ms	(4)6	AC:st	(3)6			AC:st	(2)5	AC:tt	(1)3	AC:st	(3)6	AC:ms	(4)6
AC:pl	(6)16	AC:pl	(4)5	AC:tt	(2)4			AC:tt	(2)2			AC:tt-gs	(2)4	AC:pl	(4)9
AC:st	(3)11	AC:st	(4)5											AC:st	(2)4
AC:tt	(2)10	AC:tt	(3)5											AC:tt	(1)4
RA:c	(9)21	RA:c	(3)6	RA:c	(3)8	RA:gs	(0)3	RA:ex-gs	(2)4	RA:c	(2)3	RA:c	(7)12	RA:c	(3)5
RA:ex	(6)15	RA:ex	(4)7	RA:gs	(1)5	RA:st-pl-ex-ms-c	(1)4	RA:pl-ms-c	(2)8	RA:ex-gs	(2)3	RA:ex	(3)6	RA:ex	(2)7
RA:pl-gs	(8)18	RA:gs	(2)4	RA:st-pl-ex-ms	(3)4	RA:tt	(1)2	RA:st	(2)6	RA:pl-ms	(6)10	RA:gs	(2)4	RA:gs	(2)4
RA:st-ms	(7)16	RA:ms	(3)6	RA:tt	(2)4			RA:tt	(2)6	RA:tt-st	(2)3	RA:pl-ms	(3)8	RA:ms	(3)6
RA:tt	(6)14	RA:pl	(3)4									RA:tt-st	(3)5	RA:pl	(4)9
		RA:tt-st	(2)2											RA:st	(2)2
														RA:tt	(1)5

Table II: Table showing the number of actions in the working-sets. The number tells the total amount of action with the number inside the parenthesis telling how many of those are meta-actions. AC means the set was explored by using AlgorithmConfigurationFacade and RA means the RandomFacade was used. The Metric shorthands are as follows. tt = totaltime, st = searchtime, c = individual components,ex = expanded states, gs = generated states, ms = makespan, pl = planlength.

Barman		Depots		Grid		Gripper		Logistics		Miconic		Satellite		Zenotravel	
Domain_best	**30**	Domain_best	16	Domain_best	20	Domain_best	**30**	Domain_best	17	Domain_best	**30**	Domain_best	16	Domain_best	9
Original	24	Original	13	Original	18	Original	**30**	Original	14	Original	**30**	Original	18	Original	15
AC:c	**30**	AC:c	5	AC:c	7	AC:gs-ms	**30**	AC:ex	17	AC:c	**30**	AC:c	10	AC:c	5
AC:ex	19	AC:ex	16	AC:gs	15	AC:tt-st-pl-ex-c	**30**	AC:gs	**18**	AC:pl-ms	**30**	AC:ex	18	AC:ex	11
AC:gs	**30**	AC:gs	**19**	AC:pl-ex-ms	7			AC:pl-ms-c	17	AC:st-ex-gs	**30**	AC:pl-ms	18	AC:gs	11
AC:ms	19	AC:ms	8	AC:st	7			AC:st	**18**	AC:tt	**30**	AC:st	18	AC:ms	8
AC:pl	**30**	AC:pl	16	AC:tt	**30**			AC:tt	**18**			AC:tt-gs	**19**	AC:pl	4
AC:st	19	AC:st	**19**											AC:st	13
AC:tt	19	AC:tt	**19**											AC:tt	19
RA:c	**30**	RA:c	14	RA:c	7	RA:gs	**30**	RA:ex-gs	**18**	RA:c	**30**	RA:c	10	RA:c	5
RA:ex	19	RA:ex	16	RA:gs	15	RA:st-pl-ex-ms-c	**30**	RA:pl-ms-c	17	RA:ex-gs	**30**	RA:ex	16	RA:ex	11
RA:pl-gs	19	RA:gs	**19**	RA:st-pl-ex-ms	7	RA:tt	**30**	RA:st	17	RA:pl-ms	**30**	RA:gs	**19**	RA:gs	12
RA:st-ms	19	RA:ms	5	RA:tt	**30**			RA:tt	17	RA:tt-st	**30**	RA:pl-ms	16	RA:ms	10
RA:tt	19	RA:pl	5									RA:tt-st	18	RA:pl	4
		RA:tt-st	**19**											RA:st	11
														RA:tt	**30**

Table III: Table showing the coverage of the selected working-sets being tried on 30 problems. Bold numbers highlights the best performing set for each domain. AC means the set was explored by using AlgorithmConfigurationFacade and RA means the RandomFacade was used. The Metric shorthands are as follows. tt = totaltime, st = searchtime, c = individual components,ex = expanded states, gs = generated states, ms = makespan, pl = planlength.

great improvement in coverage. Looking deeper into the actual working-sets Grid AC:tt and RA:tt turned out to be duplicates and in Logistics AC:gs and RA:ex-gs aswell.

The plots show the time it takes for the working-sets and Domain_best/Original to complete each problem. To reduce clutter, only Domain_best, Original, and working-sets that performed better or in otherwise interesting ways, were given their own legend, while the rest of the working-sets are grouped together in blue dots. The duplicate working-sets in Grid and Logistics are both plotted but shares the same shape and colour. The dotted line is drawn at the time of the longest solved problem, any problem that didn't get solved is drawn beyond this line.

Barman 4 is a domain where Domain_best has max coverage, and of the working-sets found with max coverage, none of them performed significantly better. Of the domains explored, it is the one with the most amount of actions, resulting in a huge number of options to create potential working-sets. SMAC should likely have been give more time to search for working-sets. Also 7 meta-actions were removed before barman was explored so it is a possibility that very useful meta-actions were cut. Barman is also the only Domain where the metric totaltime was not performing well. Another point to bring up is that in all the working-sets that didn't manage to get max coverage, it is consistent which problems weren't solved, problem 20-30. This despite the problems looking to be relatively easier than the previous problem if we look at working-sets that

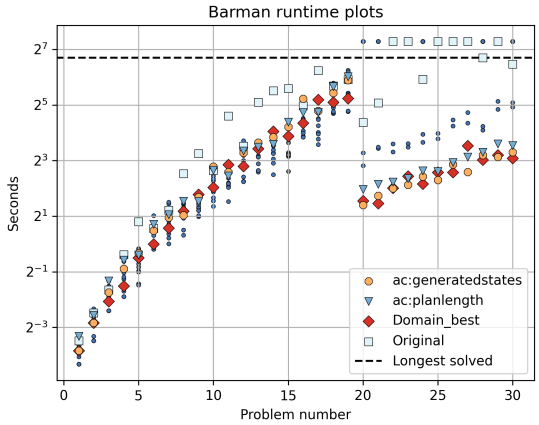

Figure 4: Barman plot

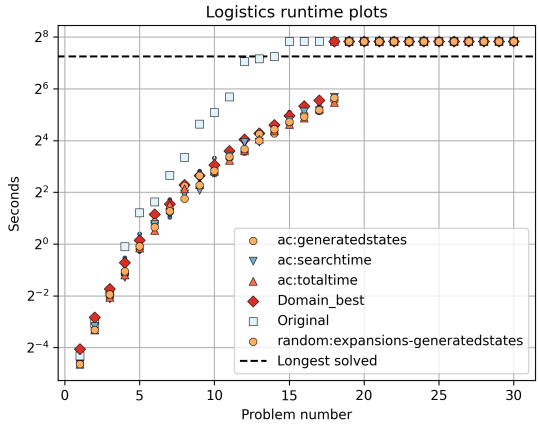

Figure 7: Logistics plot

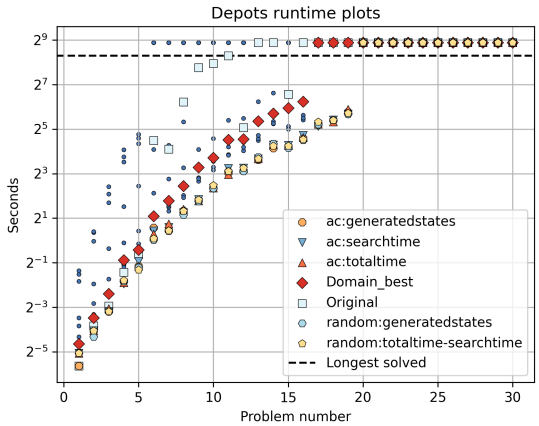

Figure 5: Depots plot

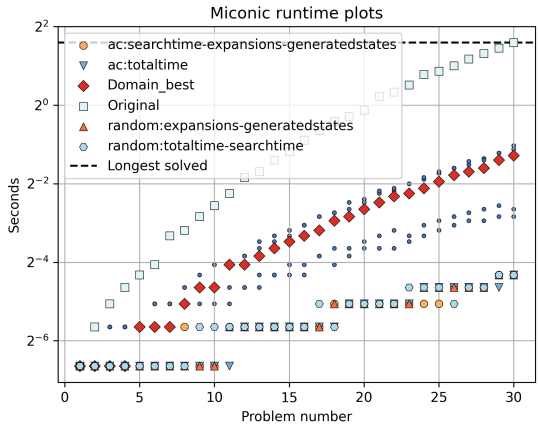

Figure 8: Miconic plot

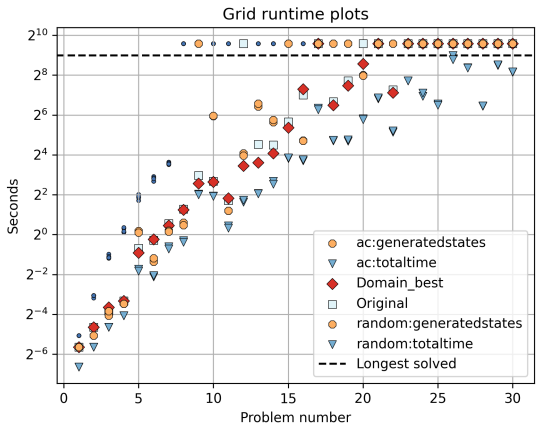

Figure 6: Grid plot

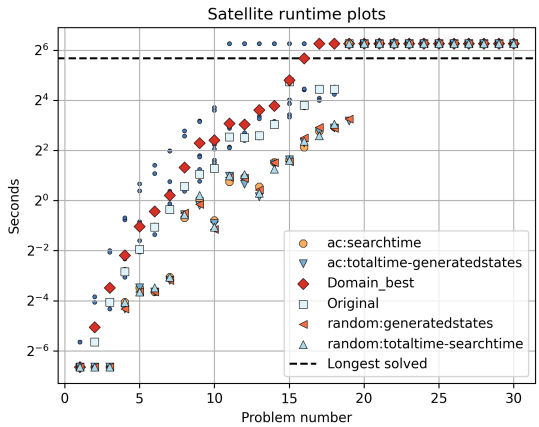

Figure 9: Satellite plot

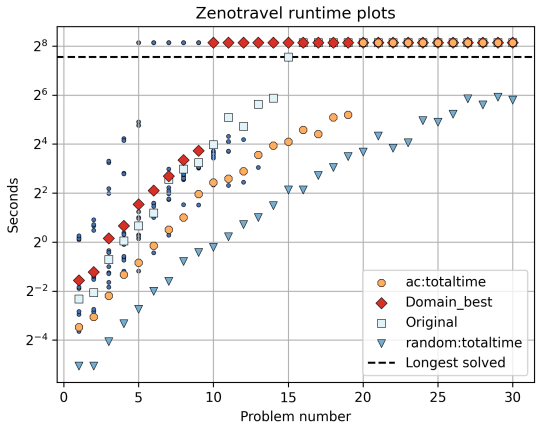

Figure 10: Zenotravel plot

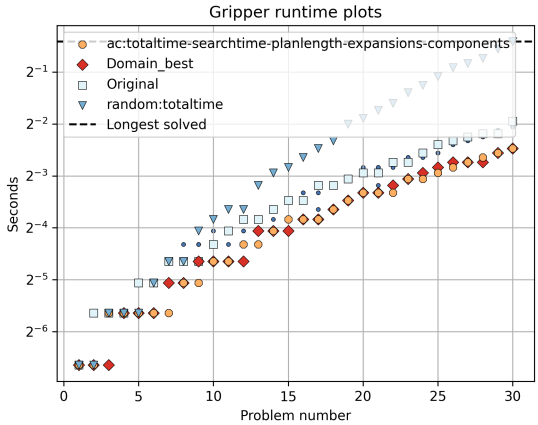

Figure 11: Gripper plot

actually solved them. Looking closer into those runs reveals that all the failures in barman is due to timeout instead of being proven unsolvable, so it is hard to say if the failure is due to the problems used during exploration not being representative enough of the infinite class of problems, or due to simply missing some action which makes the affected working-sets very inefficient while still being a safe working-set.

Depots 5 saw an improvement in coverage from 5 working-sets. Curiously, all of them perform about the same, even RA:tt-st with only 2 actions.

In **Grid** 6 the totaltime metric resulted in the same working-set being found. It performed well against Domain_best, managing to complete all the tasks. Generatedstates also resulted in duplicates. It was the second best but below even original in performance. The rest of the metrics did terribly in this domain.

In **Logistics** 7 the working-sets resulted in a single extra problem being solved, and all the highlighted working-sets had the same relative performance.

Miconic 8 had the same coverage across the board. The horizontal lines of timings are due to the smallest decimal being a hundredth of a second. The performance of all the working-sets were either better or close to as good as the Domain_best.

In **Satellite** 9 the best performing test set was Original not Domain_best. Random:gs and random ac:tt-gs managed to solve one more problem than original, and achieved better speed as well on a number of the problems

Of all the domains explored **Zenotravel** 10 was the one with the most success. A working-set was found that solved all of the problems. Interestingly it was the random facade that found this set, not ac facade.

Gripper 11 is not very interesting as due to the low amount of options, given its initial number action, the best working-set found was just the same as Domain_best.

Broadly speaking about the metrics value, in most of the domains, totaltime usually resulted in the best performance. Followed by generated-states. Components and makespan looks to be irrelevant in most domains and in a lot of instances components will produce the set of actions containing all the base- and meta-actions. However, the tests are likely biased against them. The way these two metric are counted is essentially by converting an already made sequential-plan to a partial-order-plan, which likely puts these two at a disadvantage.

Looking at the graphs, in general the working-sets form a curve with the time for solution found to be increasing with the difficulty of the problems. But most of them will suddenly stop and jump to "not solved". This gave a suspicion that most

Domain	Memout	Timeout
Barman Working sets	0	88
Barman Original	0	6
Barman Domain_Best	0	0
Depots Working sets	210	0
Depots Original	7	10
Depots Domain_best	14	0
Grid Working-sets	125	20
Grid Original	5	7
Grid Domain_best	5	5
Logistics Working-sets	113	0
Logistics Original	0	16
Logistics Domain_best	13	0
Satellite Working-sets	138	0
Satellite Original	14	0
Satellite Domain_best	12	0
Zenotravel Working-sets	266	0
Zenotravel Original	21	0
Zenotravel Domain_Best	15	0

Table IV: Table showing the reason for the failure of finding plans. The working-sets were grouped together for readability.

of these problems were not being solved due to hitting the memory limit. To confirm this suspicion, the failed reasons were aggregated.

As can be seen in table IV, the primary reason for the failure was due to memory limits. Only working-sets in Grid had a mix of memout and timeout, otherwise the working-sets failed due to strictly memout, except in Barmans case which failed strictly due to timeout. This points to that in some of the domains, the gain in coverage, from removing actions and using meta-action is more due to reducing the memory usage rather than it providing a faster solver. Logistics is an interesting counterpoint to this being a general statement however. In logistics the original domain failures were due to timeout not memout. Even the AC:tt working-set with only 2 actions, failed due to memout rather than time. This would mean here that the inclusion of meta-actions in Logistics is a significant drain on memory.

Finally, in regard to the facades chosen for SMAC, random facade was chosen to see how much the choice of facade mattered, and the results show that usually AC would have the working-sets with the best result, except for in Zenotravel were random found a working-set vastly supirior to any AC found. Given more time maybe AC would have found this working-set aswell, or maybe AC had simply discarded trying it as an option due to similar configurations performing poorly.

VII. Conclusion

In general, the tests show that using totaltime and generatedstate is more relevant than the other metrics. There was a decrease in total searchtime when removing actions across most domains, but the biggest reason for an increase to coverage during the experiments was because of less memory being used. Of course, there is a very important point that has been ignored for most of the thesis. The reconstruction of meta-actions. While including the reconstruction phase of meta-actions would increase the searchtime, and make the results look less good at timing, the coverage very likely wouldn't change very much at all, as most working-set failures are due to memorylimits not timelimits. In the end the results do show that removing actions made redundant by meta-actions is a viable way to improve both coverage and solving speed across a number of domains.

VIII. Future Work

Exploring more domains especially more complex domains, would always be relevant, to see how consistent the results of better coverage and speed is.

Another point is that as the failures show that memory was the limiting factor, using more memory is of course always an option to see how the runtime grows with speed, but more importantly, metrics that indicates memory usage should have been included when choosing working-sets, like forexample translator-operators, which tells the amount of operators the planner created during translation. Most memory failures happened during the translation.

Some or most of the code of algorithm 1 could be rewritten to c or c++ rather than use python. As looking at the stats of for example, running

algorithm 1 with barman24 it resulted in 89000 iterations of the main for loop. 5500 of those being sent to the planner, while the rest of those iteration were spent with simple set-operations but on a slow laptop. Perhaps it is not unfeasible to find the frontier of barman28 using c++ and a better processor.

Finally rather than using an ML tool like SMAC as a blackbox, with limited knowledge of how it optimizes. It would likely be suitable to go deeper into how it actually explores for working-sets, and tweak it as necessary. For example in barman21 with its 2^{21} number of possible working sets, most of the explored working-sets were discarded due to being beyond the frontier. Last point to the selections of workings-sets, is the noise that exists when solving plans. Running the same problem on the same domain twice, usually result in close but not the exact same time. This is a problem when the selection between working-sets comes down to less than a second in some domains. Using the average of 3 problems was already a way to mitigate this problem, but a more in depth approach is likely required.

IX. Acknowledgement

I want to thank my supervisor Alvaro Torralba as without his guidance this thesis would not have been possible.

Bibliography

[1] S. Richter and M. Westphal, "The LAMA planner: Guiding cost-based anytime planning with landmarks," *J. Artif. Intell. Res.*, vol. 39, pp. 127–177, 2010. DOI: 10.1613/JAIR.2972. [Online]. Available: https://doi.org/10.1613/jair.2972.

[2] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer, "Macro-ff: Improving AI planning with automatically learned macro-operators," *J. Artif. Intell. Res.*, vol. 24, pp. 581–621, 2005. DOI: 10.1613/JAIR.1696. [Online]. Available: https://doi.org/10.1613/jair.1696.

[3] R. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," in *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, D. C. Cooper, Ed., William Kaufmann, 1971, pp. 608–620. [Online]. Available: http://ijcai.org/Proceedings/71/Papers/055.pdf.

[4] M. Vallati, F. Hutter, L. Chrpa, and T. L. McCluskey, "On the effective configuration of planning domain models," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, Q. Yang and M. J. Wooldridge, Eds., AAAI Press, 2015, pp. 1704–1711. [Online]. Available: http://ijcai.org/Abstract/15/243.

[5] M. Lindauer, K. Eggensperger, M. Feurer, *et al.*, "Smac3: A versatile bayesian optimization package for hyperparameter optimization," *Journal of Machine Learning Research*, vol. 23, no. 54, pp. 1–9, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-0888.html.

[6] D. V. McDermott, "The 1998 AI planning systems competition," *AI Mag.*, vol. 21, no. 2, pp. 35–55, 2000. DOI: 10.1609/AIMAG.V21I2.1506. [Online]. Available: https://doi.org/10.1609/aimag.v21i2.1506.

[7] F. Pham and Á. Torralba, "Can I really do that? verification of meta-operators via stackelberg planning," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, ijcai.org, 2023, pp. 5420–5428. DOI: 10.24963/IJCAI.2023/602. [Online]. Available: https://doi.org/10.24963/ijcai.2023/602.

[8] (), [Online]. Available: https://automl.github.io/SMAC3/main/.

[9] AAU. "Deis-mcc, a model checking cluster." (), [Online]. Available: https://vbn.aau.dk/en/equipments/deis-mcc-a-model-checking-cluster.

[10] "Fast downward planner." (), [Online]. Available: https://www.fast-downward.org/.

[11] Á. Torralba, J. Seipp, and S. Sievers, "Automatic instance generation for classical planning," in *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, R. P. Goldman, S. Biundo, and M. Katz, Eds., AAAI Press, 2021, pp. 376–384.