Summary

A meta action is an action not inherent to the original planning task, but rather added to improve search times. As such, using meta actions entails three problems: generating meta actions, deciding which to add to the task, and reconstruction of a plan containing meta actions.
This project aims to improve how meta actions are generated, while utilising a naive method for choosing, and prior work for reconstruction. The generation of meta actions in prior work, and this project, is a two phase process. Where first, a set of meta action candidates are generated, which then secondly, are checked for validity. A meta action candidate is valid iff its exact effect can be reproduced by a sequence of actions from the original planning task. The method of meta action generation proposed is a combination of two novel techniques: one of meta action candidate generation and one that converts invalid candidates to valid meta actions.
The proposed method of meta action candidate generation finds a desired effect, then tries to convert it into a meta action candidate that upholds mutex groups. This process often leads to valid meta actions by itself, however, in the cases where it does not the second proposed method is used. Which, given an invalid meta action, restricts the states wherein the action can be used to those where it is reproducible. In turn, leading to a set of valid meta actions.
This whole process leads to a set of meta actions that when added to the planning task can greatly improve the search speed. In some planning task, it does so by orders of magnitude.

Focused Meta Actions

Jan Mackeprang Damgaard Hansen
Aalborg University
jmdh19@student.aau.dk

Kristian Skov Johansen
Aalborg University
kjohan19@student.aau.dk

Abstract

Lots of classical planning research is based on attempting to abstract problems to make the search process easier. This is commonly done by means of macros[1, 4, 5], however these carry is downside of being rigid and not very flexible. Hence, the concept of meta actions come in, where the general goal of a meta action is to achieve some effect, but without knowing what sequence of primitive actions can actually do it. Meta actions are very powerful, and the works of Pham and Torralba[19] showed that one can validate that a meta action can be replaced by sequences of primitive actions. While it have been shown how to validate meta actions, how to actually generate good meta actions is still an open problem. This is what this paper tackles, by making mutex valid meta action candidates and then adding preconditions based on state exploration. It is shown that this process of generating meta actions can find good valid meta actions and helps decrease search time in benchmarked domains.

1 Introduction

Classical planning is a well explored field, focusing on solving planning problems by executing sequences of actions. While the concept is simple, the actual process of searching can be very complex with larger planning problems and can explode out of proportions in space and time[2] needed to find a solution. This is a core problem of the field of classical planning, so much effort have gone into trying to reduce the complexity as much as possible. One common way to reduce said complexity, is by abstracting the planning task.

One of the earliest abstraction ideas have been to use so called Macro Actions[7]. A Macro Action is simply a sequence of actions combined into a single action. This can make so that instead of having to consider each step possible after executing an action, one can combine them to make "jumps" in the planning task. While these Macro Actions are quite simple, the primary problem with them is finding "good" sequences of actions.

A lot of research over the years have gone into this problem of finding "good" macros, such as looking at common action sequences in plans made by solving simpler versions of a planning task[3]. While adding macros to the planning task adds to the complexity, it have been shown that with "good" macros it is still worth it in terms of improving search time since they can make radical jumps in the planning task[1, 4–6].

One issue with these macros is that they rely on there existing common action sequences in a planning problem. Picture a planning problem about moving packages between locations, one package can be moved to its goal position in one move, i.e. `From -> To`, while the other needs four moves `From -> Loc -> Loc -> Loc -> To`. This means that there is no common action sequence that can get a package from the start to the goal.

This is where the concept of Meta Actions[19] come in. These are essentially actions that aim to achieve a certain property, e.g. from the example before we want the package to be in the goal location. Such a meta action does not care how to actually move the packages, but only says that you *can* get the package to another location.

While this is a powerful idea, there are some large issues with it. The first issue is making sure a meta action is **valid**, i.e. it is *always* possible to execute this meta action and there *will* be a sequence of primitive actions that can be executed to get to the same state. This is not a trivial property to uphold, however Pham and Torralba[19] used the concept of Stackelberg Planning to validate meta actions. A Stackelberg Planning task is an adversarial one, where two agents "compete" against each other. The idea is then to make one of the agents try and execute the meta action in all possible states, while the other agent tries to "reconstruct" the state by executing only primitive actions.

To test their validation method some meta actions was needed, so they generated meta action candidates by means of first finding a set of macros and then trying to reduce them down by removing parameters. While this generated some meta action candidates, a lot of them where **mutex breaking**[9] and such is invalid by default.

These issues funnels into the idea of *Focused Meta Actions* that this paper proposes. The general idea is to make mutex upholding meta action candidates with few effects, and then refine their preconditions to be more valid across multiple problems. The ultimate goal of this new meta action generation method, is to improve the search time in classical planning tasks as much as possible.

The main contribution of this paper can be summed up as:

- A method of generating mutex valid meta action candidates.
- A method of refining candidates to become valid.

2 Background

To illustrate concepts during the paper, a common example will be used called `Blocksworld`[20], where the goal is to stack blocks in the correct order. A block can only be in a stack of blocks, or on an infinitely large table. It is only possible to pick up a block from the table, or the top of a stack, and only one block can be held at a time. Given that a block is held, it can be placed on the table or the top of a stack. This example then has the actions `pickup`, `put-down`, `stack`, and `unstack`. A detailed description of all the Blocksworld components can be seen in appendix 12.

2.1 STRIPS

STRIPS[8], originally a tool for solving planning problems, has since evolved into formalism for planning problems. The original STRIPS definition is rather simple, so for the purposes of this paper an extended format is used that gives more flexibility in regards to negative preconditions. A STRIPS planning task is a tuple $\Pi = (F, O, I, G)$ where F is a finite set of fact-literals, O is a finite set of operators, $I \subseteq F$ is the initial state, and $G \subseteq F$ is the goal partial state.

A **fact-literal** $f \in F$ is a tuple $f = (n, b)$ where n is the name of the fact and b is a boolean that describes the state of the fact. As an example a fact-literal $f_1 = (ball\ b1, \top)$ tells that the fact *ball b1* is true, while $f_1 = (ball\ b1, \bot)$ would mean it is false. A fact-literal cannot be true and false at the same time. The set of true and false fact-literals define the **state space** for a given planning task. For simplification, all unmentioned fact-literals are false in the state space by default.

The start and goal are **partial states**, where a partial state is one where only a subset of F needs to exist.

Operators can be seen as ways to move around the state space F. Each operator has a set of fact-literals that needs to exist for it to be **applicable**, i.e. it can be executed, as well as a set of effects that modify the state. This will be represented by two functions, one for the **preconditions** for the operator, $pre(o)$, and one for the **effects** of the operator $\mathit{eff}(o)$. As an example one can look at the precondition and effect of the following operator o_1:

$$pre(o_1) = \langle(\text{fact1}, \top), (\text{fact2}, \bot)\rangle$$
$$\mathit{eff}(o_1) = \langle(\text{fact1}, \bot), (\text{fact2}, \top)\rangle$$

These sets tells that if in the current state space $fact1$ is true and $fact2$ is false, then this operator can execute. Executing an operator on the set of states will be defined as $F[o_1]$. If it is executed, the effect set is applied to the state space, i.e. $fact1$ is set to false while $fact2$ is set to true.

A solution to a STRIPS planning task can then be seen as a sequence of applicable operators, executed in sequence from the initial state I to the goal state G. This sequence of operators will be referred to as a **plan** for the planning task. A plan will always lie in the **reachable state space**, meaning all the states that can be reached by executing some sequence of actions.

A traditional STRIPS planning task is **grounded**. This means that the entire state space of fact-literals is already fully defined. However, this is not very useful when making new planning tasks, since making sure that everything is grounded correctly can be quite difficult, especially if done by hand. Not only that, but transferring operators from one planning task to another is not that straight forward, since another planning task might be structured with different fact-literals. Therefore a more intermediate format is needed, being called a **lifted**[18] one.

Lifted STRIPS can be seen as an extension to the grounded representation, defined as $\Pi_L = (P_L, O_L, A_L, I_L, G_L)$ where P_L is a finite set of predicate-literals, O_L is a finite set of **objects**, A_L is a finite set of actions, I_L is the initial state, and G_L is the goal partial state.

Every **predicate-literal** $p \in P_L$ is a tuple $p = (n, V_p, b)$ where n is a predicate name and $V_p = \langle pv_1, ..., pv_i\rangle$ is an arbitrary number of parameter variables and b is a boolean telling if the predicate is true or false.. i is the **arity** associated with the predicate p, i.e. how many parameters it has. A predicate p can be instantiated by substituting each parameter in p with an object from O_L, creating a fact-literal[12]. As an example, take the predicate-literal `holding`:

$$(\text{holding}, \langle ?x\rangle, \top)$$

Grounding this with the mapping of $x \mapsto$ block, gives the fact-literal:

$$(\text{holding}(block), \top)$$

An **action** $a \in A_L$ is a tuple $a = (V_a)$ where $V_a = \langle av_1, ..., av_i\rangle$ is a set of parameters for the action where i is the arity associated with a. An action also has the same functions as those for operators, being $pre(a)$ and $\mathit{eff}(a)$. The two methods returns predicate-literals instead of fact-literals, where each of the parameters in the predicates correspond to a parameter in the action. Grounding of an action a is also based on substituting each parameter in a with an object from O_L where, additionally, each predicate in $pre(a)$ and $\mathit{eff}(a)$ is grounded to fact-literals. This effectively grounds an action to an operator[12]. As an example, take the `putdown` action:

$$\begin{aligned} a_{putdown} &= (\text{putdown}, \langle ?ob\rangle) \\ pre(a_{putdown}) &= \langle(\text{holding}, \langle ?ob\rangle, \top)\rangle \\ \mathit{eff}(a_{putdown}) &= \langle \\ &\quad (\text{clear}, \langle ?ob\rangle, \top), \\ &\quad (\text{arm-empty}, \langle\rangle, \top), \\ &\quad (\text{on-table}, \langle ?ob\rangle, \top), \\ &\quad (\text{holding}, \langle ?ob\rangle, \bot)\rangle \end{aligned}$$

Grounding this action with the action parameter mapped as $pv_1 \mapsto block$, yields the operator o_2:

$$\begin{aligned} pre(o_2) &= \langle(\text{holding}(block), \top)\rangle \\ \mathit{eff}(o_2) &= \langle \\ &\quad (\text{clear}(block), \top), \\ &\quad (\text{arm-empty}, \top), \\ &\quad (\text{on-table}(block), \top), \\ &\quad (\text{holding}(block), \bot)\rangle \end{aligned}$$

As mentioned earlier, one fact-literal can only be true or false, it cannot be both at the same time. So by executing o_2 its implied that the (pred(ball), $\top$) is removed from the state space and the (pred(ball), $\bot$) is added to the state space. Just as with the operators, executing a action a on the state space P_L is defined as $P_L[a]$.

Within planning papers, it is common to use a format called PDDL[17]. This is a format the represents lifted STRIPS definition with the major difference being that the planning task is split into a **domain** and **problem** part. The Domain contains the structural information of the planning task Π_L, i.e. P_L and A_l sets. The Problem on the other hand, contains the more dynamic elements O_L, I_L and G_L. This makes it so one can have the same domain but many different problems that can vary in difficulty or purpose.

2.2 Meta Actions

A Macro Action is, conceptually, a shortcut in the state space. It was first proposed by Fikes, Hart, and Nilsson[7] upon which it has been expanded since[4–6]. Each macro action is the result of combining a sequence of primitive actions into a single equivalent action. There is, however, an issue inherent in macro actions: they are inflexible.

An alternative approach that provides more flexibility is the Meta Action[19]. A meta action expresses a desired effect but leaves achieving it ambiguous. An example could be the effect "*I want this ball to be in that room*", what actions to take to achieve this is unspecified.

To generate meta actions one approach, proposed by Pham and Torralba[19], first finds a macro action which is then converted to a meta action. This conversion is through a reduction of the precondition and effect of the action. Which, by definition, is limited to the macro actions it can find, as the method relies on those to generate meta actions. An example of this can be seen in example 1.

Example 1

An example of a macro from blocksworld that could be useful is one that unstacks a block from another and puts it on the table, i.e. combine the **`unstack`** *and* **`putdown`** *actions:*

$$
\begin{aligned}
a_{macro} &= (macro, \langle ?ob, ?underob \rangle) \\
pre(a_{macro}) &= \langle \\
&\quad (on, \langle ?ob, ?underob \rangle, \top), \\
&\quad (clear, \langle ?ob \rangle, \top), \\
&\quad (arm\text{-}empty, \langle\rangle, \top) \rangle \\
eff(a_{macro}) &= \langle \\
&\quad (holding, \langle ?ob \rangle, \bot), \\
&\quad (clear, \langle ?underob \rangle, \top), \\
&\quad (on, \langle ?ob, ?underob \rangle, \bot), \\
&\quad (clear, \langle ?ob \rangle, \top), \\
&\quad (on\text{-}table, \langle ?ob \rangle, \top), \\
&\quad (arm\text{-}empty, \langle\rangle, \top) \rangle
\end{aligned}
$$

The method would then try and see what preconditions can reasonably be removed to make the macro less restrictive. One option would be to use the C_{eff} removal method and remove the ?ob parameter, yielding the meta action candidate:

$$
\begin{aligned}
a_{macro} &= (macro, \langle ?underob \rangle) \\
pre(a_{macro}) &= \langle (arm\text{-}empty, \langle\rangle, \top) \rangle \\
eff(a_{macro}) &= \langle \\
&\quad (clear, \langle ?underob \rangle, \top), \\
&\quad (arm\text{-}empty, \langle\rangle, \top) \rangle
\end{aligned}
$$

That is a candidate that practically says that any block can become clear no matter how many blocks are on top of it.

Reconstruction is the process of finding an actual plan from the execution of a meta action. Since meta actions can have any number of sequences of actions that can replace them, this can be a pretty expensive process. Pham and Torralba used a planner to reconstruct the execution of a meta action. While this works, it is also very expensive to execute, however other methods exist. The work done by this papers authors in a previous semester project was about this exact issue. There, the process of reconstruction was improved by saving a **macro-cache** of possible replacement sequences that was found during the verification of meta actions[15]. This significantly improved the reconstruction time necessary and made the reconstruction in general pretty practical.

2.3 Stackelberg Planning

Stackelberg Planning[22] is an extension to lifted STRIPS that is based on adversarial planning. The basic idea is to have two disjoint sets of actions one for a **follower** and one for a **leader**, giving us the tuple

$\Pi_s = (P_s, A_{L_s}, A_{F_s}, O_s, I_s, G_s)$. The idea is to first let the leader execute a sequence of actions, thereafter the follower will then attempt to reach the goal with the shortest possible cost.

The purpose of a Stackelberg planning is then for the leader to exhaust the search space to determine the followers response from every possible state. Each of the sequences of actions the leader takes is referred to as **leader plans** and the set of follower response action sequences are **follower plans**. The goal for the leader is to take the smallest amount of steps, to make to follower take the maximum amount of steps. This means, unlike a normal planning task, that the Stackelberg planning task does not have a single plan as a result, but rather a set of plans. This set is called the Pareto Frontier[22] that represents the set of leader costs versus best follower cost.

One utility of this Stackelberg planning task, is that it can be used to verify if a meta action is valid or invalid. This is work that have been done by Pham and Torralba[19] where they used the concept of Stackelberg planning to validate meta actions. The general idea of their method was to take a planning task and a meta action, split the actions in two to represent leader actions and follower actions. Both the leader and follower starts with the same actions with the difference being that the leader actions' predicate-literals get prefixed with a "leader" name, so that only the leader can modify leader predicate-literals. The same is done for the follower, where it can only modify follower predicate-literals. The leader will start off with executing some sequence of actions where the final action is the meta action to be verified. The follower then has to start from the state just before the meta action was executed and try to see if it can find a plan with the primitive actions that can replace the executed meta action. The general flow can be seen in figure 1 below:

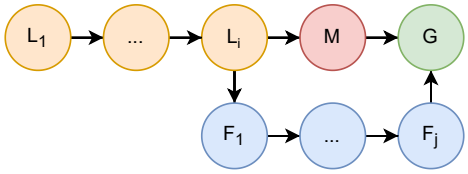


Figure 1: *General process of the Stackelberg verification. The L circles represents leader actions, the F circles follower actions, M the meta action the leader can execute and the G is an equivalent state.*

The goal of the entire planning problem is then for the follower to get to a state where all the follower predicate-literals are equivalent to the leader predicate-literals. If a sequence of follower actions can be executed that leads to an equivalence between the leader and follower states for all possible paths the leader can take, the meta action is considered valid. If there at any point was a single path where the follower could not find a sequence of actions that made the states equivalent, the meta action is considered invalid.

This process of verifying meta actions with Stackelberg planning has its limits. One major issue is that even though it is possible to verify that a meta action is valid in a set of problems, it is not possible to guarantee it will work in *all* possible problems.

2.4 Binary Decision Diagrams

As mentioned in the introduction, planning problems can explode out of proportions in space and time[2]. While reducing the complexity of the search process in an entire field in of itself, the area of reducing the space complexity in practice is something that can be managed.

This comes into play when one considers how to store a state during search of a planning task. A naive way of storing states during search would be to save an array of all possible facts that can be true or false. This, however, will get very large in bigger problems and becomes unreasonable. For that, another method of storing states is desired, one of those is that of **Reduced Order Binary Decision Diagrams** (ROBDD)[16].

A ROBDD can store any boolean expression in its structure, where the idea is to start in the root node, and follow either a true or false path down to the bottom of the tree. The path that is taken, can then represent an assignment of variables. An example of this can be seen in figure 2, that shows the possible assignments of two variables X_1 and X_2:

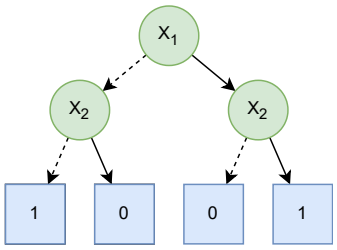


Figure 2: *ROBDD graph for two variables, X_1 and X_2, where the dotted lines represent the "true" path and the other the "false" path. This tree in effect corresponds to the boolean expression $X_1 \leftrightarrow X_2$*

Instead of saving exactly what facts are true for every state, one can save this assignment as a ROBDD instead. The general idea is to make a ROBDD where each layer correspond to some fact in the state space. This then means that instead of saving each assignment of facts in every state, one can traverse through the ROBDD in any way possible that leads to the true node to get a state instead.

This is more compact than simply storing the state as an array, however it does also come with a caveat, being that if **variable ordering**. Variable ordering is the order of which the variables are put in the ROBDD tree. In the example before, X_1 comes before X_2, so the order is rather trivial. But when it comes to large planning problems finding a good variable ordering can be very difficult[16] and sometimes the resulting ROBDD can explode in size still. However, for most cases storing a state as a ROBDD does reduce the size of a "state".

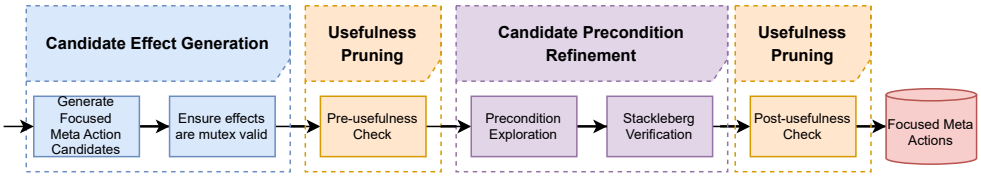


Figure 3: *Overall flow of how these Focused Meta Actions are generated. Each of the dotted boxes are sections, while the boxes inside are what is done in said section.*

3 Focused Meta Actions

The general issue this paper will tackle, is that of generating **good** meta actions. There is no clear way of defining what a "good" meta action is, however some assumptions can be made. This paper will base a good meta actions on the findings from Allen et al.[1], where they found that a macro action with the fewest possible effects tended to reduce the search time the most. Hence, this paper will attempt to make meta actions that **have the fewest possible effects**, to have the greatest impact on reducing search time.

It is also clear that for a meta action to be good, it has to be **valid** in the same way Pham and Torralba[19] found. This is important since an invalid meta action that *could* reduce search time does not generate valid plans that can be used for anything.

Definition 1 (Valid Meta Action)**.** *A meta action candidate m is a valid meta action for a planning task $\Pi_L = (P_L, O_L, A_L, I_L, G_L)$ if and only if all the reachable states s in Π_L where $pre(m) \subseteq F_L$ there exist an action sequence $A = \langle a_1, ..., a_i \rangle$ where $m \notin A$ such that $F_L[m] = F_L[a_1]...[a_i]$. A state where m can be replaced by a normal action sequence is referred to as a valid state, while one that cannot be replaced by a normal action sequence is an invalid state.*

The final requirement for a good meta action is that it should be **useful**, meaning that it should help reduce the overall search time for a planning task. This is clear since if adding a meta action to a planning task *increases* the search time, it only makes everything worse. However, two different sets of planning tasks are used for generating the meta actions and actually using them. Where the set used for generating them is significantly easier, and the difficulty of a planning task scales non-linearly, the margin of improvement might be less than noise. As such, a different metric is used which is independent of noise, namely plan length.

Definition 2 (Meta Action Usefulness)**.** *A meta action m is considered useful for a planning task $\Pi_L = (P_L, O_L, A_L, I_L, G_L)$ if adding the meta action to the planning task $A_L = m \cup A_L$ the resulting plan is shorter than without the meta action.*

Definition 3 (Focused Meta Action)**.** *A good Focused Meta Action upholds all the following properties:*

1. *must have the fewest possible effects,*
2. *must be valid and*
3. *must be useful.*

These three properties are what this paper will focus on and is what the following sections will be structured by. To make the connection between the different sections clear, one can use figure 3 as a sort of index, to understand what sections are connected.

The process starts with creating meta action candidates, which are then made to uphold mutex groups. The resulting meta action candidates are then further pruned based on usefulness. The resulting candidates are then refined into meta actions by adding precondition literals. Finally, the meta actions are pruned based on usefulness again.

4 Candidate Effect Generation

The first property that must be considered from definition 3 is the creation of meta action candidates with the fewest possible effects. The simplest way to generate these kinds of effects for a lifted planning task $\Pi_L = (P_L, O_L, A_L, I_L, G_L)$, is to take all the predicates in P_L and turn each of them into a single effect.

As an example, take the predicate $(\text{clear}, \langle ?x \rangle, \top)$, which results in the meta action candidate:

$$\begin{aligned} a_2 &= (\text{meta-clear}, \langle ?x \rangle) \\ pre(a_2) &= \langle\rangle \\ eff(a_2) &= \langle (\text{clear}, \langle ?x \rangle, \top) \rangle \end{aligned}$$

A similar one can also be made, that sets the predicate to be $\bot$ instead of $\top$.

4.1 Mutex Legal Candidates

While such a simple candidate certainly upholds the first property of definition 3, it does not necessarily uphold the second property. One large issue to why this meta action is not valid, is because it does not uphold **mutex groups**[10].

A mutex group is a set of facts where at any time one of the facts must be true, but no more than one. As an example, consider the set of fact-literals for the predicate $(\text{clear}, \langle ?x \rangle, \top)$ and $(\text{on}, \langle ?y, ?x \rangle, \top)$ for a single block. The `clear` predicate says that there is nothing on top of the block $?x$, while the `on` predicate says that block $?y$ is on top of block $?x$. It is pretty clear that for any block $?x$, it either has to have nothing on top of it, or another block on top of it, both cannot be true at the

same time. This is a mutex group for this block, that can be illustrated as a set with the mapping $x \mapsto block1, y \mapsto block2$:

$$\langle(\text{clear}(\text{block1}), \top), (\text{on}(\text{block2, block1}), \top)\rangle$$

While this is for block1 and block2, it also works the other way around, where block2 must either be free or have block1 on top:

$$\langle(\text{clear}(\text{block2}), \top), (\text{on}(\text{block1, block2}), \top)\rangle$$

These mutex groups are grounded, however one can turn them into a lifted variant. This however requires that it is *always* true that for a given $?x$ and $?y$, this group upholds. This can be a tricky thing to determine, however several tools already exists that can find such lifted mutex groups[9, 12].

The lifted mutex groups have some extra complexity, in that they also need to describe which parameters are unique in the group and which are not. Where a unique parameter is one which will be instantiated by the same object across rules. Using the example of blocks from before, one can make a lifted representation as follows:

$$\langle(\text{clear}, \langle ?x\rangle, \top), (\text{on}, \langle ?y, ?x\rangle, \top)\rangle$$

Here, the unique parameter will be $?x$, since the same parameter is used in both the `clear` and `on` predicate-literal, while the $?y$ is free since it is only used in one of the literals. The unique parameters are known as "fixed"[9] parameters, while the free parameters are known as "counted"[9] parameters. Each of the items in the lifted mutex group will be referred to as a "rule".

Definition 4 (Lifted Mutex Group)**.** *A lifted mutex group g is a set of predicate-literals, referred to as rules, $r \in g$ where at any point in the reachable state space s only a single rule in g must be upheld for one instantiation of fixed parameters and all instantiations of counted parameters.*

Definition 5 (Mutex Mentioned)**.** *A given predicate-literal p is **mentioned** in a mutex group g if the name of the predicate-literal is the same as some of the rules in the mutex group, denoted as $p(\vec{x}) \in g$*

Since it is known that these mutex groups must *always* be upheld, one can make the following assumption about meta action candidates:

Definition 6 (Mutex Upholding Candidate)**.** *A meta action candidate m_c is upholding a set of mutex groups G if for any given reachable state s where m_c is applicable, execution m_c giving the state s' is still upholding the groups in G.*

Using these definitions and knowledge of mutex groups, one can make the following theorem regarding meta action candidate mutex validity:

Theorem 1 (Mutex Validity)**.** *A meta action candidate m_c is always invalid if it is not mutex upholding.*

Proof. Assume the opposite, that a candidate m_c is valid if it is not mutex upholding for a set of mutex groups G. Then, for a given state s where m_c is applicable, executing the candidate to get the state s' will result in a state that does not uphold the set of mutex groups G. However, we know that G cannot be violated so s' must be an invalid state, hence m_c is invalid. □

This theorem can be applied to the candidate generation, where the goal is to modify a candidate's preconditions and effects so it becomes mutex upholding. While so far, groups with only two rules have been considered, one can easily have a mutex group with three or more rules in them. In the case where there are multiple rules to consider, one has to permute all possible combinations of effects that can be added. An example of this can be seen in example 2.

Example 2

As an example, take the following mutex group from blocksworld:

$$\langle(\mathit{clear}, \langle ?V0\rangle, \top), (\mathit{holding}, \langle ?V0\rangle, \top), (\mathit{on}, \langle ?C1, ?V0\rangle, \top)\rangle$$

Logically, it means that a given block has to either have no other blocks on top of it, being held in the arm or be below some other block. If there was then a meta action candidate that has a single effect, being $(\mathit{clear}, \langle ?x\rangle, \top)$. For the candidate to uphold the mutex group, either `holding` or `on` for this block has to be set to false. Here the "counted" and "fixed" variables come in to play. A "fixed" variable is one that is the same as its source rule, e.g. in this example $?x$ will be put in the place of the $?V0$ in the mutex group. The "counted" variables is one that has to be a unique one, so in this case a new one would be added. If this candidate is then to uphold the mutex group, two unique versions can be made:

$$\begin{aligned}
a_3 &= (\mathit{sample}, \langle ?x, ?C1\rangle)\\
\mathit{pre}(a_3) &= \langle(\mathit{on}, \langle ?C1, ?x\rangle, \top)\rangle\\
\mathit{eff}(a_3) &= \langle\\
&\quad (\mathit{on}, \langle ?C1, ?x\rangle, \bot),\\
&\quad (\mathit{clear}, \langle ?x\rangle, \top)\rangle
\end{aligned}$$

$$\begin{aligned}
a_4 &= (\mathit{sample}, \langle ?x\rangle)\\
\mathit{pre}(a_4) &= \langle(\mathit{holding}, \langle ?x\rangle, \top)\rangle\\
\mathit{eff}(a_4) &= \langle\\
&\quad (\mathit{holding}, \langle ?x\rangle, \bot),\\
&\quad (\mathit{clear}, \langle ?x\rangle, \top)\rangle
\end{aligned}$$

While adding effects and preconditions is simple for a single mutex group, it gets more complex when one have to consider multiple mutex groups. This is because one need to also consider what the same predicate-literal can be mentioned across several groups, which can influence what groups should be upheld for subsequent added effects.

Definition 7 (Cross Mentioned Groups)**.** *For any predicate-literal p, the set of **cross mentioned groups** is the subset of mutex groups G where $\{g \in G \mid p(\vec{x}) \in g\}$*

As an example of this issue, consider the following small example of a set of mutex groups, where the A, B, C, D, E

represents some unique predicate-literals:

$$g_1 = \langle A, B \rangle$$
$$g_2 = \langle C, B, D \rangle$$
$$g_3 = \langle C, B, E \rangle$$

Say that one wants the uphold the following meta action candidate, where the it is desired to set the predicate A to be true:

$$a = (\text{sample}, \langle\rangle)$$
$$pre(a) = \langle\rangle$$
$$\mathit{eff}(a) = \langle (A, \top) \rangle$$

It can be seen that the B predicate is mentioned in g_1, and for it to be true, it is necessary to set B to be false. This is a rather simple modification, resulting in the modified meta action candidate:

$$a = (\text{sample}, \langle\rangle)$$
$$pre(a) = \langle (B, \top) \rangle$$
$$\mathit{eff}(a) = \langle (A, \top), (B, \bot) \rangle$$

Now B is interesting, since its cross mentioned groups are equal to all the groups. This means that it is important to make sure that added effects from g_2 and g_3 does not try to also uphold those same groups.

It can clearly be seen that new effects need to know what groups its parent will uphold, when considering the example before, but where g_2 is in focus. If one chooses to set the C to be initially true and then false, resulting in the following action:

$$a = (\text{sample}, \langle\rangle)$$
$$pre(a) = \langle (B, \top), (C, \bot) \rangle$$
$$\mathit{eff}(a) = \langle (A, \top), (B, \bot), (C, \top) \rangle$$

If the effect $(C, \top)$ does not track that its parent effect $(B, \bot)$ will later uphold the group g_3 since it is cross mentioned in it, one can end up with a candidate that looks like the following:

$$a = (\text{sample}, \langle\rangle)$$
$$pre(a) = \langle (B, \top), (C, \bot), (E, \top) \rangle$$
$$\mathit{eff}(a) = \langle (A, \top), (B, \bot), (C, \top), (E, \bot) \rangle$$

However this candidate is not mutex valid, since the precondition breaks g_3. Hence, one need to *keep track of what groups an effect is already upholding based on what groups the parent effect upholds and will uphold later.*

For this, an auxiliary function is made, $IsUpholding : e \rightarrow G_e$, that operates as a map from some given effect e to the set of mutex groups $G_e \subseteq G$. This is so to keep track of the set of mutex groups that some given effect is already upholding. This function will be treated as a sort of global mutable utility function, where the mapping from e to some mutex groups can be updated.

As mentioned, finding these lifted mutex groups in the first place is an entire field in of itself, so for the purposes of this paper CPDDL[9] is used to find the groups. The method of making a meta action candidate uphold a set of lifted mutex groups can now be described, starting with algorithm 1:

Algorithm 1 $Uphold(c, g, G)$

Input: Let $c = (P, E)$ be a candidate's preconditions P and effects E
Input: Let g be the target lifted mutex group
Input: Let G be the set of all lifted mutex groups
Output: A set of candidates that upholds g.

```
R ← ∅
for e ∈ E do
    if g ∉ IsUpholding(e) ∧ e(x⃗) ∈ g then
        IsUpholding(e) ← IsUpholding(e) ∪ g
        for p ∈ g do
            R ← R ∪ Mutate(c, e, p, G)
        end for
    end if
end for
if R = ∅ then
    R ← R ∪ c
end if
return R
```

This algorithm takes a meta action candidate and a single mutex group. It then goes through each effect in the candidate, where if the effect is mentioned in the mutex group each group rule is mutated into a new candidate by the $Mutate$ method. If a given candidate has no effect mentioned in the mutex group, the candidate is simply added as the only item in the return set, meaning the $Uphold$ method will always return at least one candidate.

The method of mutating a candidate c with a rule p is to insert a true version of p into the preconditions of c and putting a false version of p into the effects. The process of this can be seen in algorithm 2:

Algorithm 2 *Mutate*(c, e, p, G)

Input: Let $c = (P, E)$ be a candidate's preconditions P and effects E
Input: Let e be the source predicate-literal
Input: Let p be the target predicate-literal rule
Input: Let G be the set of all lifted mutex groups
Output: A mutated version of c

Let e_{cpy} be a copy of p as a predicate
for Predicate argument a in p **do**
    if a is fixed **then**
        Replace argument at index a in e_{cpy} with argument at index a in e
    else
        Replace argument at index a in e_{cpy} with a new unique argument.
    end if
end for
$IsUpholding(e_{cpy}) \leftarrow \{g \in G \mid e(\vec{x}) \in g\}$
if e is true **then**
    return $(P \cup (e_{cpy}, \top), E \cup (e_{cpy}, \bot))$
else
    return $(P \cup (e_{cpy}, \bot), E \cup (e_{cpy}, \top))$
end if

One important step here is that a newly added effect, will set its $IsUpholding(e_{cpy})$ set to be that of the cross mentioned groups for the source predicate e. A final post-processing step this does, is to add any new predicate-literal parameters to the candidate meta actions parameters.

So far, only a single mutex group is upheld from this, so a further algorithm is needed to express how to make sure a set of mutex groups is upheld. This algorithm can be seen below in algorithm 3:

Algorithm 3 *UpholdAll*(c, G)

Input: Let $c = (P, E)$ be a candidate's preconditions P and effects E
Input: Let G be a set of lifted mutex group
Output: A set of candidates that upholds all $g \in G$.

$R \leftarrow \langle c \rangle$
while $|R|$ *changes* **do**
    for $g \in G$ **do**
        $R_n \leftarrow \varnothing$
        for $r \in R$ **do**
            $R_n \leftarrow R_n \cup Uphold(r, g, G)$
        end for
        $R \leftarrow R_n$
    end for
end while
return R

This goes through each of the mutex groups and generates a new set of mutated candidates that is then passed on to the next mutex group check. This is continued until all the effects of all the candidates are upholding mutex groups where they are mentioned in, resulting in the set R not changing.

Proposition 1. *All candidates returned by UpholdAll upholds all the lifted mutex groups in G.*

This entire process makes sure that the effects of a candidate upholds all the lifted mutex groups that was found from CPDDL. It is clear that a candidate given to the *UpholdAll* method will always output mutex upholding candidates, since each mutex group is iterated through against all the candidates given by each mutex group check step. A candidate will then either be unmodified, or several variations will be given, where each of them upholds each mutex group.

5 Meta Candidate Refinement

Through the prior section a set of meta action candidates is generated. However, a meta action candidate is not the final product. A meta action candidate is by definition not yet determined to be valid. As such, one method of converting a set of meta action candidates to a set of meta actions, is by filtering them based on validity, leaving only those which are valid.

Yet, the invalid candidates are not without use. The definition of a invalidity is the lack of reproducability in an applicable state, as defined in definition 1. As such, if one where to make an invalid meta candidate inapplicable in the states wherein it is not reproducible it would become valid. This process of turning an invalid meta action candidate into a meta action is called *Refinement*.

More concretely the refinement of a meta action candidate c is a mapping to a possibly empty set of meta actions M. Through this mapping, only additions are made to the precondition of c, where additions resulting in meta actions are added to M. As such, any meta action in M has the same effect as c and has the precondition of c as a subset of its precondition.

$$pre(c) \subseteq pre(m) \wedge \mathit{eff}(c) = \mathit{eff}(m) \qquad \forall m \in M$$

where: c = is a meta action candidate
m = is a meta action

The refinement of a meta action candidate c is seen in equation 1 where a set of possible refinements are generated for each planning task in P. Each refinement is a meta action candidate, and as such tested for validity in all of P.

$$\{m \mid m \in \bigcup_{\Pi \in P} Refinements(\Pi, c) \wedge IsValid(P, m)\} \qquad (1)$$

where: P = is a set of planning tasks
c = is a meta action candidate
Π = is a planning task
m = is a meta action

A meta action candidate's precondition is a set of precondition literals, as such the possible refinements is a set of sets

of precondition literals. Namely it is the cartesian product of predicates, their parameters, and the actions parameters. As an example, say that a task has the singular predicate *on* with two parameters x and y, then an action with two parameters $p1$ and $p2$ has 4 possible precondition literals: *on* $p1$ $p1$,*on* $p1$ $p2$, *on* $p2$ $p1$, and *on* $p2$ $p2$.

Let $\Pi = (P, O, A, I, G)$ be a planning task, $p \in P$ be some predicate, and c be a meta action candidate derived from Π. Then the set of literals are calculated as described in equation 2, where the product operator of the sequence is the Cartesian product.

$$l(p, c) = \prod_{p_c \in params(c)} p_c \bigtimes params(p) \tag{2}$$

where: p = is a predicate
c = is a meta action candidate

The set of precondition literals is then the union of those for each predicate, as seen in equation 3. Of note, is that the possible precondition literals is a super set of the precondition literals in the original meta action candidate.

$$L(P, c) = \bigcup_{p \in P} l(p, c) \tag{3}$$

where: P = is a set of predicates
c = is a meta action candidate

Let $\Pi = (P, O, A, I, G)$ be planning task, and c be a meta action candidate. Then let $L(P, c) = (l_1, l_2)$ be the set of literals available. Thus a generating refinements become akin to building a tree. Where the root node is the precondition of some meta action candidate c, and each possible refinement is a node, connected through branches by the addition of a precondition literal $l \in L$.

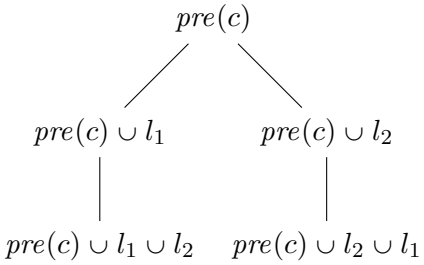


Figure 4

The number of nodes in such a tree given a set of literals L is the size of the power set of L, or $2^{|L|}$. To reduce this, a number of methods exist.

Proposition 2. *Let a be some action where $pre(a) = eff(a)$. Applying a on a state s results in the same state s.*

An action as outlined in proposition 2 is redundant by the fact that it can be omitted from any plan without changing the applicability of the plan, nor the resulting state.

Proposition 3. *Let a be some action and $(p, \top) \in pre(a)$ be some precondition literal, then if $(p, \bot) \in pre(a)$ the action is always inapplicable.*

A state by definition cannot contain both a fact and its negation. As such, an action requiring such is never applicable.

Proposition 4. *The addition of a precondition literal to an action's precondition can never increase the number of states wherein it is applicable.*

An action is inapplicable in a state iff there exists no instantiation of said action that is applicable in said state. The addition of a precondition literal to the actions precondition, does not change the fact the state still requires the prior precondition literals.

Proposition 5. *Let $\Pi = (P, O, A, I, G)$ be a lifted STRIPS planning task and $a_1 \in A$ some action in it. If a_1 is applicable in no state, then any action $a_2 \in A$ where $pre(a_1) \subseteq pre(a_2)$ is also applicable in no state.*

By proposition 4 the set of states where in a_2 is not applicable contains the set of states wherein a_1 is not applicable. And since the set of states where a_1 is not applicable is all states, b_2 is then also not applicable in all states.

In figure 4 the node $pre(a) \cup l_1$ denotes the addition of some predicate literal l_1 to the precondition of a. This addition changes the states wherein a is applicable, and by proposition 4 it reduces the applicability. As such, a child node always has the same applicability or lower than its parent. Which also means that any node applicable in zero valid states can be pruned, as by proposition 5. As an example say that adding l_1 to the precondition of a leads it to be applicable in no valid state, the tree then changes to what can be seen in figure 5 below:

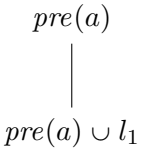


Figure 5

However, not all nodes in the tree is a refinement candidate. Namely, any refinement applicable in an invalid state is not a valid refinement. It is only those refinements that are applicable in at least one valid state, and applicable in no invalid state.

The generation of the tree can be seen in algorithm 4, which is a recursive algorithm. Its root can be seen in equation 4. Let Π be a planning task, and c a meta action candidate. Then let SS_R be the reachable state space of Π, $V \subseteq SS_R$ the set of states where c is valid by instantiation, $I \subseteq SS_R$ the set of states c is invalid by instantiation, and L the set of possible precondition literals.

$$Refinements(\Pi, c) = Refine(\Pi, V(\Pi, SS_R(\Pi), c), I(\Pi, SS_R(\Pi), c), L(\Pi, c), pre(c)) \quad (4)$$

where: Π = is a planning task
c = is a meta action candidate

Each instance of the refine algorithm 4 serves as a node in the tree. Where line 2 finds the set of valid state instantiation pairs applicable for the refinement candidate. In the case where this set is empty, the candidate and all of its descendents are ignored by returning the empty set in line 4. Then in line 6 the set of applicable invalid state instantiation pairs is found, and in the case of it being the empty set it is added to the set of final refinements in line 8. Then line 10 iterates over all the children of the node, which is those literals not already in L.

Algorithm 4 Refine(Π, valid, invalid, literals, L)

Input: Π is a lifted planning task
Input: *valid* is a set of instantiation and state set pairs
Input: *invalid* is a set of instantiation and state set pairs
Input: *literals* is a set of precondition literals
Input: L is a set of precondition literals
Output: A set of precondition literals
1: $valid \leftarrow Reduce(\Pi, L, valid)$
2: **if** $valid = \varnothing$ **then**
3: **return** $\varnothing$
4: **end if**
5: $L_o \leftarrow \varnothing$
6: $invalid \leftarrow Reduce(\Pi, L, invalid)$
7: **if** $invalid = \varnothing$ **then**
8: $L_o \leftarrow L_o \cup L$
9: **end if**
10: **for** $l \in literals \backslash L$ **do**
11: $L_o \leftarrow L_o \cup$ Refine(Π, *valid*, *invalid*, literals, $L \cup l$)
12: **end for**
13: **return** L_o

The algorithm 4 serves as the base algorithm, however, it is also quite naïve in scope. Namely, it can visit the same node multiple times. An example can be seen in figure 4, which contains both $pre(c) \cup l_1 \cup l_2$ and $pre(c) \cup l_2 \cup l_1$. As the precondition is a set, these are equivalent. As such, it would be prudent to check whether a node has already been visited.

The reduce algorithm 5 referenced in the tree generation algorithm 4 restricts a set of states to those that match the given literals. As such, the resulting set of states is always a subset of the states given as input. This set is built by the union of those states applicable for each instantiation of the literals.

Algorithm 5 Reduce(Π, I, literals, comb)

Input: Π is a lifted planning task
Input: L is a set of precondition literals
Input: I is a set of arg list and state set pairs
Output: A set of states
1: $SS \leftarrow \varnothing$
2: **for** $(args, states) \in I$ **do**
3: $SS_I \leftarrow states$
4: **for** $literal \in L$ **do**
5: $partial\_state \leftarrow Map(literal, args)$
6: $SS_I \leftarrow SS_I \cap partial\_state$
7: **end for**
8: $SS \leftarrow (args, SS \cup SS_I)$
9: **end for**
10: **return** SS

Line 2 iterates the instantiation and state pairs in I. For each pair of *args* and *states*, *states* are those applicable to *args*. As an example, say that Reduce is called for valid states, then *states* would be those states wherein the given instantiation *args* is valid. Then the lines 3 to 7 finds the subset of *states* which contains the facts created by instantiating the precondition literals in L with *args*. The resulting states are then added to the final states in line 8.

6 Usefulness Pruning

The final property of definition 3 is that of usefulness. The idea of this is to only generate meta actions that will actually help with the final search time. This is done in two separate steps, one between the effect generation and precondition refinement, and one after the precondition refinement. The reason these steps are needed, is that its technically possible to generate an effect or refine a precondition to make a candidate valid, however the resulting valid meta action is simply not useful in any way.

A common example of a useless, yet valid, meta action is one where a simple candidate gets refined up to the point where it is directly the same as a primitive action, but with some additional restricting preconditions. While the meta action is technically valid, it will clearly not do anything better than the primitive action it is similar too.

Both the usefulness pruning methods employed are based on the same principle, that being *reduction in plan lengths*. This works by taking a set of planning tasks, adding a meta action candidate to the task and then solving the task. If the plan contained the meta action and it reduces the overall plan length, it is regarded as being useful.

This is what the first usefulness pruning step does, however the second one is a bit more strict. Here, only the two meta actions that resulted in the shortest plans are picked, meaning the final usefulness check will at max return 2 meta actions. If, after refinement, the meta actions that are left are no longer used or ends up with longer plans, they are discarded at this step too.

These usefulness checks are mainly there to make the entire process of finding focused meta actions more practical, since a lot of time can be wasted in attempting to refine meta actions that has effects that is completely undesireable for the actual planning task.

7 Experimental Setup

The Focused Meta Actions will be evaluated by means of a couple of experiments. However, some setup is needed to begin with. For this, a set of benchmarks have been used, one for validating meta action candidates and one to test the meta actions on. These will be referred as "learning" and "testing" for simplicity. The learning benchmark set is made by Torralba[23] and the testing benchmark is the official Autoscale benchmark set[20]. Some of the domains in the testing set is missing in the training set, however there are still 16 domains in total to train and test on. These domains are barman, blocksworld, childsnack, depots, driverlog, floortile, grid, gripper, hiking, logistics, miconic, parking, rovers, satellite, scanalyzer and woodworking.

It should be noted, that this implementation of Focused Meta Actions does not support numerical expressions in

PDDL[11], so they have all been removed from the benchmark set. This is simply to limit the scope of what PDDL requirements that needed to be implemented.

The Autoscale Learning benchmark set generates a very large quantity of training problems, sometimes in the range of thousands, however only a few need to be selected for the purposes of this paper. To select problems, the *operator count* is used, since it can represent the difficulty of a problem. Difficulty is important here, since too large training problems will make the Stackelberg Verification time out too easily. The Autoscale Learning benchmark set contains log files for the reported operator count for each problem. For the training problems, five where selected that all had an operator count below the 10th percentile of the maximum operator count for all the problems in the domain. For usefulness check, another five problems where selected each being withing the 10th to the 50th percentile, since some slightly more difficult problems are needed[1].

The testing set all consists of 30 problems for each of the domains.

The training was run on a cluster with AMD Opteron 6376 processors and with 4GB memory limit. The testing was run on the same cluster computer, but following normal IPC limits[14] of a 30 minute time limit and also 4GB memory limit. For the testing, the solver Fast Downward[13] was used with the LAMA-First[21] configuration.

The entire training process, i.e. generation, validation and refinement, is all done fully automatically[2].

[1]Benchmarks can be found at `https://github.com/kris701/FocusedMetaActionsData`

[2]Implementation can be found at `https://github.com/jamadaha/P10`

8 Results

The first results to consider, is that of generating and refining meta actions to begin with. As mentioned, the training part is run on all 16 domains, each with 5 problems to train on. All the domains found different numbers of meta action candidates, as can be seen in table 1.

Domain	G	C	C_{pre}	M_{valid}	M_{post}
Barman	9	15	8	15	2
Blocksworld	3	19	11	41	2
Childsnack	1	15	3	0	0
Depots	6	91	60	4	2
Driverlog	4	6	6	23	4
Floortile	3	7	4	1	1
Grid	4	6	5	48	2
Gripper	3	4	3	48	2
Hiking	5	6	6	9	2
Logistics	1	2	2	67	2
Miconic	1	5	2	2	2
Parking	3	6	3	1	0
Rovers	4	21	7	357	2
Satellite	1	9	3	1	1
Scanalyzer	2	3	1	1	1
Woodworking	7	11	0	0	0

Table 1: *Usefulness pruning information. G is the amount of mutex groups for each domain. C is the initial candidate meta actions. C_{pre} is the candidates after the pre-usefulness check. M_{valid} is the valid refined meta actions. M_{post} is the valid meta actions after the post-usefulness check.*

As can be seen, it is possible to find focused meta actions in most of the domains. The time it takes to find all these valid meta actions, depends a lot on how many there is to check through and how difficult the training problems are. During training, some timeouts was given for different parts, such as timeouts for the pre-usefulness checks, timeouts for the state space search, etc.

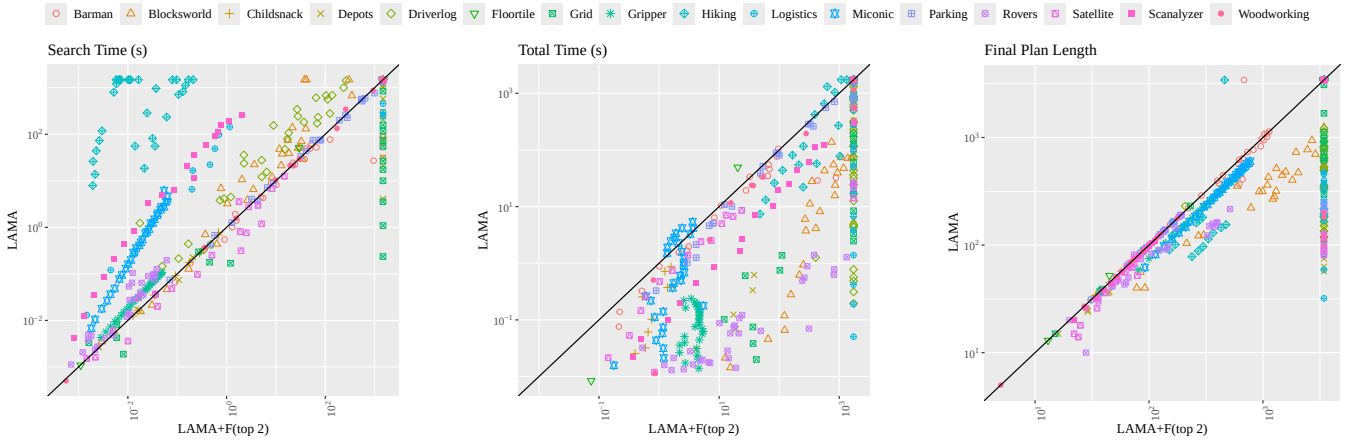
An overview of the timeouts given can be seen in the list below:

- **Pre-usefulness check**: 5m
- **State Space Exploration**: 10m
- **Meta Action Candidate Validation**: 5m
- **Candidate Refinement**: 60m
- **Post-usefulness check**: 5m

A lot of the time, these time limits are reached as the refinement process gets to the more difficult problems. It should also be noted, that for the testing problems all the meta actions was verified to be correct, this however does not mean that they are correct in the testing problems which will be discussed later.

Now that the meta actions have been made, the real question from this is if these meta actions are helpful in regards to search time. For this, a scatter plot of the directly stated search time from Fast Downward can be seen in figure 6a.

In the figure, it can be seen that the meta actions added to the domain helps the search time in almost all of the domains.



(a) *Stated search time from the Fast Downward planner running the lama-first configuration with and without meta actions. Do note this is without reconstruction, this is just the raw search time.*

(b) *Total Time to solve problems in the domains. The total time is that of translation + search + reconstruction.*

(c) *Final plan lengths with reconstruction. The LAMA side is simply the plan it found while the one with Focused Meta Actions are the final plan length after reconstruction.*

Figure 6: *Results for comparing the Focused Meta Actions against LAMA.*

One large exception is the *Grid* domain, a larger reason for this is that grid is a domain that can really stress test the Fast Downward translator. More details on specifically this domain will be discussed in section 9. However, other domains such as *Scanalyzer*, *Hiking* and *Miconic* seem to benefit a lot from these added meta actions.

While it would seem that the meta actions found help the search a lot, if one however looks at the total coverage of the training domains, they do not seem better, as can be seen in table 2:

Domain	LAMA+F(top 2)	LAMA
Barman	20	24
Blocksworld	30	27
Childsnack	9	9
Depots	5	17
Driverlog	30	30
Floortile	2	2
Grid	7	18
Gripper	30	30
Hiking	30	19
Logistics	11	14
Miconic	30	30
Parking	22	22
Rovers	30	30
Satellite	18	18
Scanalyzer	19	19
Woodworking	10	10
Total	303	319

Table 2: *Coverage of how many problems each method was able to solve within the time limit. Each domain has 30 problems in total. Do note, this is without reconstruction but purely based on being able to find a plan.*

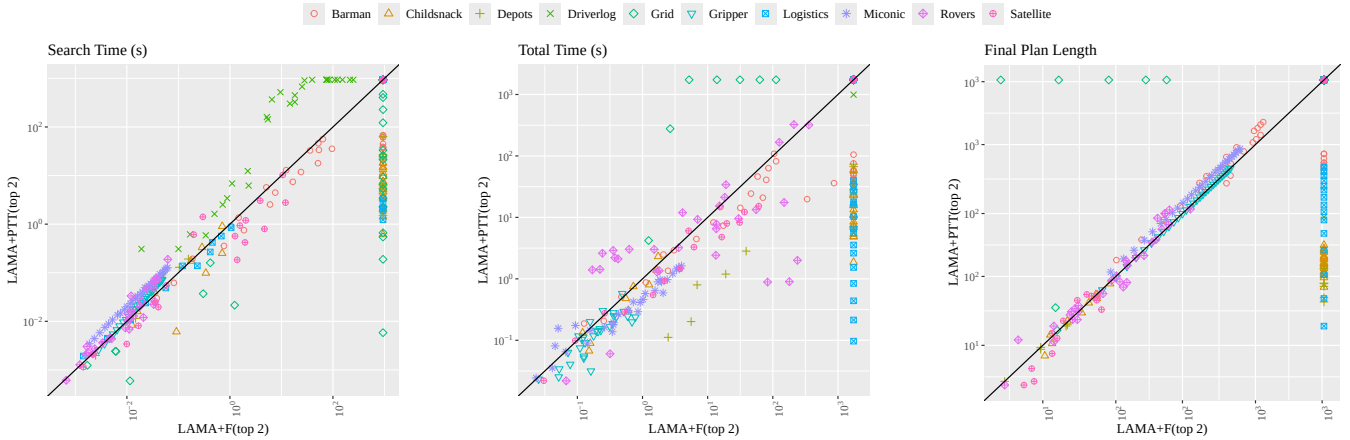
From this table, it is clear that it is a bit hit and miss with the domains. A few domains benefit from the meta actions, such as *Hiking* and *Blocksworld*, however quite a few also suffers a lot from the added meta actions, such as *Depots* and *Grid*.

Much of this can be contributed to the fact that the higher problems in the Autoscale Benchmark set, is simply very difficult, and even LAMA struggles with getting full coverage. It should also be noted, that on a lot of the difficult problems ends up running out of memory during translation.

While figure 6a is directly the stated search time of Fast Downward, it is also interesting to take reconstruction into account. The reconstruction method used is that of this authors previous work, MARMA[15], that is able to reconstruct meta actions by macros learned through the initial validation of the meta actions. Another thing this MARMA tool is able to do, is detect if a meta action is invalid in a plan. This is done by first checking if there is a macro that can replace the execution of a meta action, if not then a regular planner is used to reconstruct. If that planner cannot reconstruct the meta action either, it is regarded as an invalid meta action. One can then make a new scatter plot with the total time for the entire planning process, as can be seen in figure 6b.

It is quite clear, that the total time of using these meta actions are not great. However, one have to consider the fact that most of the time is spent on reconstruction and not the search. This indicates that the meta actions themselves are helping quite a lot in general, but the reconstruction technique is not good enough to help the total planning time.

One domain stands out in the total time results, being the *Logistics* domain, where every single problem was unsolved for the Focused Meta Actions. The reason for this is that the MARMA macro generation process used for reconstruction, generated way too many macros. The MARMA tool simply runs out of time, not in regards to search, but in regards to



(a) *Stated search time between the Focused Meta Actions and the PTT meta actions. Do note this is without reconstruction, this is just the raw search time.*

(b) *Total Time to solve problems in the domains. The total time is that of translation + search + reconstruction.*

(c) *Final Plan lengths between the two methods.*

Figure 7: *Results for comparing the Focused Meta Actions against the previous work of Pham and Torralba, denoted as PTT.*

looking through all the macros to find one that fits the usage of the meta action.

One other aspect that the MARMA tool can do, is to detect if the usage of a meta action is invalid. The MARMA tool showed that the Focused Meta Actions generated invalid meta actions in two domains, being *Grid* and *Rovers*. These two domains will not be removed in the following results, since the MARMA tool can technically fix the use of an invalid meta action, by reverting back to solving without that meta action.

It is also interesting to look at the differences in plan lengths between using meta action and not. This comparison can be seen in figure 6c.

One can see that adding meta actions, seem to result in a longer plans. This is to be expected, since configurations such as lama-first have no way of knowing the actual heuristic value of executing a meta action. So a meta action that seems to be cheap to execute, could result in many actions having to be executed to reconstruct it.

8.1 Comparison to Previous Work

While the results looks promising in of itself, it is still important to compare against previous work. For this, the Focused Meta Actions that was found will be compared against the previous work of Pham and Torralba.

As mentioned earlier, they made meta actions by reducing preconditions and effects of macros, creating a meta action that could potentially be less restrictive than the original macro. In the following figures, this previous work meta actions will be referred to as "LAMA+PTT(top 2)", since it is the LAMA-first[21] configuration of Fast Downward[13], and they referred their meta actions as "PTT" in their paper. Do note, this is also the "top 2" meta actions stated in their paper, not top 2 in the sense of plan lengths, as it is with the post usefulness check for the Focused Meta Actions.

A total of 10 of the 16 domains are compatible with the results found in their work. From these 10 domains, both Pham and Torralba and this paper found meta actions in all the domains, however just as the Focused Meta Actions the previous work also has invalid meta actions in *Grid* and *Rovers*. Just as before, those two domains stay.

To compare the two methods, one can start with looking at the search time in figure 7a. This represents the stated search time from Fast Downward.

It can be seen that for the most part, these two methods are quite comparable. One large exception is that of *Childsnack*, where the PTT meta actions manages to find a really good meta action that helps the search time a lot. While the *Grid* results for the PTT meta actions also look a lot better, one have to consider the fact that they are invalid meta actions. On the other hand, the Focused Meta Actions seem to have helped the *Driverlog* domain more than the PTT meta actions could.

Just as when comparing to LAMA, one can also compare the Focused Meta Actions with the PTT meta actions in regards to total time as can be seen in figure 7b-

It can also be seen that the *Depots* and *Barman* domains are better with the PTT meta action in total time. One speculation that can be made is that the PTT meta actions are easier to reconstruct than the Focused Meta Actions, since the PTT ones are made out of macros to begin with.

While earlier plan lengths versus LAMA was looked at, it is also interesting to see the differences in plan lengths between the Focused Meta Actions and those of the meta actions from Pham and Torralba. This can be seen in figure 7c.

From the figure, it can be seen that, for the problems that both meta actions could help solve, the plan lengths are pretty much identical. This is not too surprising, since in a some of the domains, the Focused Meta Actions actually manages to find the exact same meta actions as the PTT ones. One exam-

ple is the *Miconic* domain, where both methods find a meta action that directly just sets the predicate *served* to be true for some passenger. However, for most of the domains very different meta actions are found between the two methods.

It is interesting to look at a few examples of the differences between the Focused Meta Actions and the PTT meta actions. One of the domains where the PTT meta actions does well and there was found no Focused meta action, is the *Childsnack* domain.

The reason why the PTT meta actions are good here, is that they have a meta action that sets a child to be *served* and its tray to be empty. This action makes a lot of sense, since for a child to be served the tray they have been served from must also be emptied. However, CPDDL does not report this as a mutex group, instead it only reports that a tray can only be at one place at a time. This limits the amount of effects that the Focused Meta Actions can make, hence why no valid ones are found.

9 Discussion

This section will go into some further details on some of the meta actions and refinements found for the different domains. A few interesting domains are selected, either because they work well with the Focused Meta Actions method or because they work badly.

9.1 Blocksworld

Blocksworld is one of the domains that create an interesting refinement for a meta action. The meta action in question is one that is constructed from upholding mutex groups pertaining to the `on` predicate. One of the mutex valid candidates for this, is the following:

$$
\begin{aligned}
a &= (\text{meta\_on\_3}, \langle ?x, ?y \rangle) \\
pre(a) &= \langle \\
&\quad (\text{clear}, \langle ?y \rangle, \top), \\
&\quad (\text{on-table}, \langle ?x \rangle, \top) \rangle \\
eff(a) &= \langle \\
&\quad (\text{on}, \langle ?x, ?y \rangle, \top), \\
&\quad (\text{clear}, \langle ?y \rangle, \bot), \\
&\quad (\text{on-table}, \langle ?x \rangle, \bot) \rangle
\end{aligned}
$$

This is in principle a meta action that says that if block ?x is on the table and there is nothing on block ?y, then block ?x can be put on top of block ?y. This meta action is not valid in of itself, since the precondition does not say that the block ?y cannot be on block ?x, however a refinement is found that changes the entire function of this meta action. This refinement can be seen below, with the additions marked in bold:

$$
\begin{aligned}
a &= (\text{meta\_on\_3}, \langle ?x, ?y \rangle) \\
pre(a) &= \langle \\
&\quad (\text{clear}, \langle ?y \rangle, \top), \\
&\quad (\text{on-table}, \langle ?x \rangle, \top), \\
&\quad (\textbf{on-table}, \langle ?y \rangle, \top), \\
&\quad (\textbf{clear}, \langle ?x \rangle, \bot) \rangle \\
eff(a) &= \langle \\
&\quad (\text{on}, \langle ?x, ?y \rangle, \top), \\
&\quad (\text{clear}, \langle ?y \rangle, \bot), \\
&\quad (\text{on-table}, \langle ?x \rangle, \bot) \rangle
\end{aligned}
$$

These two added preconditions make so that both blocks have to be on the table, however there can be any amount of blocks on top of block ?x. The interesting part of this is that it essentially makes so that one can move an entire stack of blocks in a single action in the manner that can be seen in figure 8:

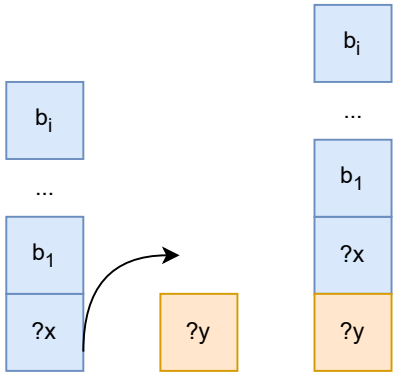


Figure 8: *What the refined meta action does. The block ?x has i amount of other blocks on top of it.*

Normally, one would have to entirely unstack all the blocks on top of ?x before moving it, meaning a large amount of actions needed. This refinement can sometimes help the search time a lot, but it very much depends on the configuration of the blocks in the initial state.

9.2 Grid

While the meta action for blocksworld is an example of a good successful meta action refinement, it is not necessarily the case for all domains. One such domain that does not work well is the Grid domain. This is a domain about moving "shapes" around in a grid to unlock further locations in the grid to get more shapes and so on.

A focused meta action that gets made for this domain is

the following:

$$
\begin{aligned}
a &= (\text{meta\_at-robot\_0}, \langle ?x, ?y \rangle) \\
pre(a) &= \langle \\
&\quad (\text{at-robot}, \langle ?x \rangle, \top), \\
&\quad (\text{open}, \langle ?y \rangle, \top) \rangle \\
eff(a) &= \langle \\
&\quad (\text{at-robot}, \langle ?y \rangle, \top), \\
&\quad (\text{at-robot}, \langle ?x \rangle, \bot) \rangle
\end{aligned}
$$

This is essentially a meta action that says one can "teleport" to any open grid cell in the problem. While this sounds like it could be rather useful, instead of having to jump from grid cell to grid cell one by one, it is in fact very bad.

This can also be seen in the result section, where grid is the domain that the focused meta actions perform the worst in. This is because of the amount of operators that gets made from this meta action. With the normal move action, one can only move from one grid cell to the adjacent one. However with this meta action, one can move from any grid cell to any other grid cell. This results in a huge amount of operators that can be made from this, an example of which can be seen in figure 9

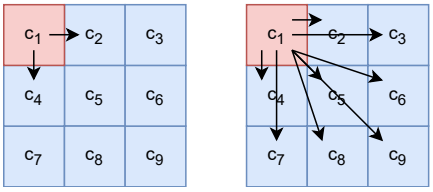


Figure 9: *A small grid example. Assume you start in c_1. With the normal move action, its possible to move to c_2 and c_4 resulting in 2 operators. However with the teleport meta action, it is now possible to move anywhere on the grid field, resulting in 8 operators instead.*

In most of the grid problems, adding this meta action simply makes the Fast Downward translator run out of memory, since there are so many operators to create. An example is the largest grid problem, that contains 40 different grid clusters with 140 cells each, giving a total of 5600 grid cells.

10 Conclusion and Future Works

It have been shown that the Focused Meta Actions can significantly improve the search time for most planning domains with a few exceptions. However, while the search time was great, the time it took to reconstruct the execution of the meta actions where not so good, resulting in higher total time.

When comparing to the previous work of Pham and Torralba, it was also shown that the Focused Meta Actions was sometimes better than their meta actions and sometimes not.

That said, the Focused Meta Actions method does show promise, in being a fully automated method of generating and validating meta actions that can work in most domains. However, there are several points that could be improved. One such point is that of the usefulness checks.

The usefulness checks heavily relies on there being large differences in plan lengths between using meta actions and not using meta actions. However, in a lot of the domains, the plans for the usefulness problems was very short, and a lot of over filtering happened as a result. One could imagine that doing usefulness checks based on search time could be better, however it requires a very well balanced benchmark dataset to work with. Things such as making sure that all the usefulness problems are solvable within a relatively short amount of time can be quite difficult, especially when using Autoscale benchmarks.

Another problem is that of the training problems. It was shown that in *Grid* and *Rovers* there where invalid meta actions. This is the result of the training problems not being representative enough of the domain, e.g. the *Grid* meta actions fails if there are two independent grids which usually only occurs in larger problems. Some automated method of finding *representable* problems of a domain could be useful to solve this issue.

To sum this paper up, a new novel method of generating, validating and refining meta actions have been made, that shows promise in regards to reducing search time. It was also shown that this method can be used on a large set of domains, and because of its automated nature, can easily be adapted to new domains.

References

[1] Cameron Allen et al. "Efficient Black-Box Planning Using Macro-Actions with Focused Effects". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. Ed. by Zhi-Hua Zhou. ijcai.org, 2021, pp. 4024–4031. DOI: 10.24963/IJCAI.2021/554. URL: https://doi.org/10.24963/ijcai.2021/554.

[2] Tom Bylander. "The Computational Complexity of Propositional STRIPS Planning". In: *Artif. Intell.* 69.1-2 (1994), pp. 165–204. DOI: 10.1016/0004-3702(94)90081-7. URL: https://doi.org/10.1016/0004-3702(94)90081-7.

[3] Lukás Chrpa. "Generation of macro-operators via investigation of action dependencies in plans". In: *Knowl. Eng. Rev.* 25.3 (2010), pp. 281–297. DOI: 10.1017/S0269888910000159. URL: https://doi.org/10.1017/S0269888910000159.

[4] Lukás Chrpa and Fazlul Hasan Siddiqui. "Exploiting Block Deordering for Improving Planners Efficiency". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 1537–1543. URL: http://ijcai.org/Abstract/15/220.

[5] Lukás Chrpa and Mauro Vallati. "Improving Domain-Independent Planning via Critical Section Macro-Operators". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 7546–7553. DOI: 10.1609/AAAI.V33I01.33017546. URL: https://doi.org/10.1609/aaai.v33i01.33017546.

[6] Lukás Chrpa, Mauro Vallati, and Thomas Leo McCluskey. "MUM: A Technique for Maximising the Utility of Macro-operators by Constrained Generation and Use". In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. Ed. by Steve A. Chien et al. AAAI, 2014. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7905.

[7] Richard Fikes, Peter E. Hart, and Nils J. Nilsson. "Learning and Executing Generalized Robot Plans". In: *Artif. Intell.* 3.1-3 (1972), pp. 251–288. DOI: 10.1016/0004-3702(72)90051-3. URL: https://doi.org/10.1016/0004-3702(72)90051-3.

[8] Richard Fikes and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". In: *Artif. Intell.* 2.3/4 (1971), pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5. URL: https://doi.org/10.1016/0004-3702(71)90010-5.

[9] Daniel Fiser. "Operator Pruning Using Lifted Mutex Groups via Compilation on Lifted Level". In: *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*. Ed. by Sven Koenig, Roni Stern, and Mauro Vallati. AAAI Press, 2023, pp. 118–127. DOI: 10.1609/ICAPS.V33I1.27186. URL: https://doi.org/10.1609/icaps.v33i1.27186.

[10] Daniel Fiser and Antonín Komenda. "Fact-Alternating Mutex Groups for Classical Planning". In: *J. Artif. Intell. Res.* 61 (2018), pp. 475–521. DOI: 10.1613/JAIR.5321. URL: https://doi.org/10.1613/jair.5321.

[11] Maria Fox and Derek Long. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains". In: *J. Artif. Intell. Res.* 20 (2003), pp. 61–124. DOI: 10.1613/JAIR.1129. URL: https://doi.org/10.1613/jair.1129.

[12] Malte Helmert. "Concise finite-domain representations for PDDL planning tasks". In: *Artif. Intell.* 173.5-6 (2009), pp. 503–535. DOI: 10.1016/J.ARTINT.2008.10.013. URL: https://doi.org/10.1016/j.artint.2008.10.013.

[13] Malte Helmert. "The Fast Downward Planning System". In: *J. Artif. Intell. Res.* 26 (2006), pp. 191–246. DOI: 10.1613/JAIR.1705. URL: https://doi.org/10.1613/jair.1705.

[14] *International Planning Competition*. URL: https://ipc2023-learning.github.io/.

[15] Kristian Skov Johansen and Jan M. D. Hansen. "MARMA: Meta Action Reconstruction using Macro Actions". In: (2024). URL: https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/a7me0f/alma9921651321705762.

[16] Sheldon B. Akers Jr. "Binary Decision Diagrams". In: *IEEE Trans. Computers* 27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141. URL: https://doi.org/10.1109/TC.1978.1675141.

[17] Drew McDermott et al. *PDDL – The Planning Domain Definition Language – Version 1.2*. Tech. rep. CVC TR-98-003/DCS TR-1165. Yale University: Yale Center for Computational Vision and Control, 1998.

[18] John H. Munson. "Robot Planning, Execution, and Monitoring in an Uncertain Environment". In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*. Ed. by D. C. Cooper. William Kaufmann, 1971, pp. 338–349. URL: http://ijcai.org/Proceedings/71/Papers/028.pdf.

[19] Florian Pham and Álvaro 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. URL: https://doi.org/10.24963/ijcai.2023/602.

[20] AI-Planning. *Autoscale Agile Benchmarks*. 2024. URL: https://github.com/AI-Planning/autoscale-benchmarks/tree/main/21.11-agile-strips.

[21] Silvia Richter and Matthias Westphal. "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks". In: *J. Artif. Intell. Res.* 39 (2010), pp. 127–177. DOI: 10.1613/JAIR.2972. URL: https://doi.org/10.1613/jair.2972.

[22] Patrick Speicher et al. "Stackelberg Planning: Towards Effective Leader-Follower State Space Search". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 6286–6293. DOI: 10.1609/AAAI.V32I1.12090. URL: https://doi.org/10.1609/aaai.v32i1.12090.

[23] Álvaro Torralba. *Autoscale Learning Benchmarks*. 2024. URL: https://github.com/alvaro-torralba/autoscale-learning.

11 Appendix

12 Blocksworld Domain

The blocksworld domain contains 5 predicate-literals P_L:

- (clear, $\langle ?x \rangle$): Says that there are no other block on top of block x
- (on-table, $\langle ?x \rangle$): Says that the block x is on the table (i.e. its on the bottom)
- (arm-empty, $\langle\rangle$): Says if the arm is empty or not
- (holding, $\langle ?x \rangle$): Says what block the arm is holding
- (on, $\langle ?x, ?y \rangle$): Says that the block x is on top of y

As well as the four actions A_L:

$$
\begin{aligned}
a_{pickup} &= (\text{pickup}, \langle ?ob \rangle) \\
pre(a_{pickup}) &= \langle \\
&\quad (\text{clear}, \langle ?ob \rangle, \top) \\
&\quad (\text{on-table}, \langle ?ob \rangle, \top), \\
&\quad (\text{arm-empty}, \langle\rangle, \top) \rangle \\
\mathit{eff}(a_{pickup}) &= \langle \\
&\quad (\text{holding}, \langle ?ob \rangle, \top), \\
&\quad (\text{clear}, \langle ?ob \rangle, \bot), \\
&\quad (\text{on-table}, \langle ?ob \rangle, \bot), \\
&\quad (\text{arm-empty}, \langle\rangle, \bot) \rangle
\end{aligned}
$$

$$
\begin{aligned}
a_{putdown} &= (\text{putdown}, \langle ?ob \rangle) \\
pre(a_{putdown}) &= \langle (\text{holding}, \langle ?ob \rangle, \top) \rangle \\
\mathit{eff}(a_{putdown}) &= \langle \\
&\quad (\text{clear}, \langle ?ob \rangle, \top), \\
&\quad (\text{arm-empty}, \langle\rangle, \top), \\
&\quad (\text{on-table}, \langle ?ob \rangle, \top), \\
&\quad (\text{holding}, \langle ?ob \rangle, \bot) \rangle
\end{aligned}
$$

$$
\begin{aligned}
a_{stack} &= (\text{stack}, \langle ?ob, ?underob \rangle) \\
pre(a_{stack}) &= \langle \\
&\quad (\text{clear}, \langle ?underob \rangle, \top), \\
&\quad (\text{holding}, \langle ?ob \rangle, \top) \rangle \\
\mathit{eff}(a_{stack}) &= \langle \\
&\quad (\text{arm-empty}, \langle\rangle, \top), \\
&\quad (\text{clear}, \langle ?ob \rangle, \top), \\
&\quad (\text{on}, \langle ?ob, ?underob \rangle, \top), \\
&\quad (\text{clear}, \langle ?underob \rangle, \bot), \\
&\quad (\text{holding}, \langle ?ob \rangle, \bot) \rangle
\end{aligned}
$$

$$
\begin{aligned}
a_{unstack} &= (\text{unstack}, \langle ?ob, ?underob \rangle) \\
pre(a_{unstack}) &= \langle \\
&\quad (\text{on}, \langle ?ob, ?underob \rangle, \top), \\
&\quad (\text{clear}, \langle ?ob \rangle, \top), \\
&\quad (\text{arm-empty}, \langle\rangle, \top) \rangle \\
\mathit{eff}(a_{unstack}) &= \langle \\
&\quad (\text{holding}, \langle ?ob \rangle, \top), \\
&\quad (\text{clear}, \langle ?underob \rangle, \top), \\
&\quad (\text{on}, \langle ?ob, ?underob \rangle, \bot), \\
&\quad (\text{clear}, \langle ?ob \rangle, \bot), \\
&\quad (\text{arm-empty}, \langle\rangle, \bot) \rangle
\end{aligned}
$$

The initial state and goals for the domain is some configuration where blocks are stacked on top of each other, where the goal is to shift the blocks around to get a goal configuration.