

On-the-fly Macros

Hubie Chen

Dept. of Information and Communication Technologies
 Universitat Pompeu Fabra
 Passeig de Circumval·lació, 8
 08003 Barcelona, Spain
 hubie.chen@upf.edu

Omer Giménez

Dept. of Llenguatges i Sistemes Informàtics
 Universitat Politècnica de Catalunya
 Jordi Girona, 1-3
 08034 Barcelona, Spain
 omer.gimenez@upc.edu

November 23, 2013

Abstract

We present a domain-independent algorithm that computes macros in a novel way. Our algorithm computes macros “on-the-fly” for a given set of states and does not require previously learned or inferred information, nor prior domain knowledge. The algorithm is used to define new domain-independent tractable classes of classical planning that are proved to include *Blocksworld-arm* and *Towers of Hanoi*.

1 Introduction

Macros have long been studied in AI planning [9, 18]. Many domain-dependent applications of macros have been exhibited and studied [15, 17, 12]; also, a number of domain-independent methods for learning, inferring, filtering, and applying macros have been the topic of research continuing up to the present [2, 7, 20].

In this paper, we present a domain-independent algorithm that computes macros in a novel way. Our algorithm computes macros “on-the-fly” for a given set of states and does not require previously learned or inferred information, nor does it need any prior domain knowledge. We exhibit the power of our algorithm by using it to define new domain-independent tractable classes of classical planning that strictly extend previously defined such classes [6], and can be proved to include *Blocksworld-arm*

and *Towers of Hanoi*. We believe that this is notable as theoretically defined, domain-independent tractable classes have generally struggled to incorporate construction-type domains such as these two. We hence give theoretically grounded evidence of the computational value of macros in planning.

Our algorithm. Consider the following reachability problem: given an instance of planning and a set S of states, compute the ordered pairs of states $(s, t) \in S \times S$ such that the second state t is reachable from the first state s . (By *reachable*, we mean that there is a sequence of operators that transforms the first state into the second.) This problem is clearly hard in general, as deciding if one state is reachable from another captures the complexity of planning itself.

A natural—albeit incomplete—algorithm for solving this reachability problem is to first compute the pairs $(s, t) \in S \times S$ such that the state t is reachable from the state s by application of a single operator, and then to compute the transitive closure of these pairs. This algorithm is well-known to run in polynomial time (in the number of states and the size of the instance) but will only discover pairs for which the reachability is evidenced by plans staying within the set of states S : the algorithm is efficient but incomplete.

The algorithm that we introduce is a strict generalization of this transitive closure algorithm for the described reachability problem. We now turn to a brief, high-level description of our algorithm. Our algorithm begins by computing the pairs connected by a single operator, as in the just-described algorithm, but each pair is labelled with its connecting operator. The algorithm then continually applies two types of transformations to the current set of pairs until a fixed point is reached. Throughout the execution of the algorithm, every pair has an associated label which is either a single operator or a macro derived by combining existing labels. The first type of transformation (which is similar to the transitive closure) is to take pairs of states having the form (s_1, s_2) , (s_2, s_3) and to add the pair (s_1, s_3) whose new label is the macro obtained by “concatenating” the labels of the pairs (s_1, s_2) and (s_2, s_3) . If the pair (s_1, s_3) is already contained in the current set, the algorithm replaces the label of (s_1, s_3) with the new label if the new label is “more general” than the old one.¹ The second type of transformation is to take a state $s \in S$ and a label of an existing pair, and to see if the label applied to s yields a state $t \in S$; if so, the pair (s, t) is introduced, and the same replacement procedure as before is invoked if the pair (s, t) is already present.

Our algorithm, as with the transitive closure, operates in polynomial time (as proved in the paper) and is incomplete. We want to emphasize that it can, in general, identify pairs that are not identified by the transitive closure algorithm. Why is this? Certainly, some state pairs (s, t) introduced by the first type of transformation have macro labels that, if executed one operator at a time, would stay within the set S , and hence are pairs that are discovered by the transitive closure algorithm. However, the second type of transformation may apply such a macro to other states to discover pairs $(s, t) \in S \times S$ that would *not* be discovered by the transitive closure: this occurs when a step-by-step execution of the macro, starting from s , would leave the set S before arriving to t .

¹ For the precise definitions of “concatenation” and “more general”, please refer to the technical sections of the paper.

Indeed, these two transformations depend on and feed off of each other: the first transformation introduces increasingly powerful macros, which in turn can be used by the second to increase the set of pairs, which in turn permits the first to derive yet more powerful macros, and so forth.

We now describe two concrete results to offer the reader a feel for the power of our algorithm. Let s be any state of a *Blocksworld-arm* instance, and let S be the set $H(s, 4)$ of states within Hamming distance 4 of s .² Let us use the term *subtower* to refer to a sequence of blocks stacked on top of one another such that the top is clear. We prove that our algorithm, given the set S , will discover macros that move any subtower of s onto the ground (preserving the subtower structure). As another result, let s be the initial state of the *Towers of Hanoi* problem, for any number of discs; and, let S be the set $H(s, 7)$ of states within Hamming distance 7 of s . We prove that our algorithm, given the set S , will discover macros that, starting from the state s , move any subtower of discs from the initial peg to either of the other pegs. In particular, our algorithm will report that the goal state is reachable from the initial state s . Note that, in the case of *Blocksworld-arm*, the constant 4 is independent of the state s , and in particular is independent of the height of subtowers; likewise, in *Towers of Hanoi*, the constant 7 is independent of the number of discs. Note also that, as can be proved, the transitive closure algorithm does not detect either of these reachability conditions, even when $S = H(s, k)$ for an arbitrarily large constant k .³ We emphasize again that our new algorithm is fully domain-independent.

Our algorithm not only returns pairs of states, but also returns, for each state pair (s, t) , a succinct representation of a plan from s to t , as in [16]. Note that our algorithm may discover pairs (s, t) for which the shortest plan from s to t is of exponential length, when measured in terms of the original operators, as in the *Towers of Hanoi* domain.

Towards a tractability theory of domain-independent planning. Many of the benchmark domains—such as *Blocksworld-arm*, *Gripper*, and *Logistics*—can now be handled effectively and simultaneously by domain-independent planners, as borne out by empirical evidence [14]. This *empirically observed* domain-independent tractability of many common benchmark domains naturally calls for a *theoretical explanation*. By a theoretical explanation, we mean the formal definition of tractable classes of planning instances, and formal proofs that domains of interest fall into the classes. Clearly, such an explanation could bring to the fore structural properties shared by these benchmark domains.

To the best of our knowledge, research proposing tractable classes has generally had other foci, such as understanding syntactic restrictions on the operator set [5, 1, 8], studying restrictions of the causal graph, as in [3, 4, 11, 16], or empirical evaluation of simplification rules [10]. Aligned with the present aims is the work of Hoffmann [13] that gives proofs that certain benchmark domains are solvable by local search with respect to various heuristics.

² The Hamming distance between two states is defined as the number of variables at which they differ.

³ In the case of *Towers of Hanoi*, this follows immediately from the known exponential lower bound on the length of a plan transforming the initial state to the goal state. For a fixed $k \geq 1$, when given the initial state and $H(s, k)$, the transitive closure algorithm “stays within the set” $H(s, k)$, which is of polynomial $O(n^k)$ size, and will not discover pairs (v, v') which are not linked by polynomial length plans.

To demonstrate the efficacy of our algorithm, we use it to extend previously defined tractable classes. In particular, previous work [6] presented a complexity measure called *persistent Hamming width (PH width)*, and demonstrated that any set of instances having bounded PH width—PH width k for some constant k —is polynomial-time tractable. It was shown that both the *Gripper* and *Logistics* domains have bounded PH width, giving a uniform explanation for their tractability. In the present paper, we show that an extension of this measure yields a tractable class containing both the *Blocksworld-arm* and *Towers of Hanoi* domains, and we therefore obtain a single tractable class which captures all four of these domains. As mentioned, we believe that this is significant as theoretical treatments have generally had limited coverage of construction-type domains such as *Blocksworld-arm* and *Towers of Hanoi*.

We want to emphasize that our objective here is *not* to simply establish tractability of the domains under discussion: in them, plan generation is already well-known to be tractable on an individual, domain-dependent basis. Rather, our objective is to give a *uniform, domain-independent* explanation for the tractability of these domains. Neither is our goal to prove that these domains have low time complexity; again, our primary goal is to present a simple, domain-independent algorithm for which we can establish tractability of these domains with respect to the heavily-studied and mathematically robust concept of polynomial time.

Previous work on macros. Macros have long been studied in planning [9]. Early work includes [19], which developed filtering algorithms for discovered macros, and [18], which demonstrated the ability of macros to exponentially reduce the size of the search space.

Macros have been thoroughly applied in domain-specific scenarios such as puzzles and other games. To name some examples, there has been work on the sliding tile puzzle [15], Sokoban [17], and Rubik’s cube [12].

Some recent research on integrating macros into domain-independent planning systems is as follows. *Macro-FF* [2] is an extension of FF that has the ability to automatically learn and make use of macro-actions. *Marvin* [7] is a heuristic search planner that can form so-called macro-actions upon escaping from plateaus that can be reused for future escapes. Both of these planners participated in the International Planning Competition (IPC). A method for learning macros given an arbitrary planner and example problems from a domain is given in [20].

A more theoretical approach was taken by [16], who studied the use of macros in conjunction with causal graphs. This work gives tractability results, and in particular shows that domain-independent planners can cope with exponentially long plans in polynomial time, which is also a feature of the present work.

The use of macros in this paper contrasts with that of most works in that macros are generated and applied not over a domain or even over an instance, but with respect to a “current state” s and a (small) set of related states S . This ensures that the macros generated are tailored to the state set S , and no filtering due to over-generation of macros is necessary.

2 Preliminaries

An instance of the planning problem is a tuple $\Pi = (V, \text{init}, \text{goal}, A)$ whose components are described as follows.

- V is a finite set of variables, where each variable $v \in V$ has an associated finite domain $D(v)$. Note that variables are not necessarily propositional, that is, $D(v)$ may have any finite size. A *state* is a mapping s defined on the variables V such that $s(v) \in D(v)$ for all $v \in V$. A *partial state* is a mapping p defined on a subset $\text{vars}(p)$ of the variables V such that for all $v \in \text{vars}(p)$, it holds that $p(v) \in D(v)$.
- init is a state called the *initial state*.
- goal is a partial state.
- A is a set of *actions*. An action a consists of a *precondition* $\text{pre}(a)$, which is a partial state, as well as a *postcondition* $\text{post}(a)$, also a partial state. We sometimes denote an action a by $\langle \text{pre}(a); \text{post}(a) \rangle$.

Note that when s is a state or partial state, and W is a subset of the variable set V , we will use $(s \upharpoonright W)$ to denote the partial state resulting from restricting s to W . We say that a state s is a *goal state* if $(s \upharpoonright \text{vars}(\text{goal})) = \text{goal}$.

We say that an action a is *applicable* at a state s if $(s \upharpoonright \text{vars}(\text{pre}(a))) = \text{pre}(a)$. We define a *plan* to be a sequence of actions $P = a_1, \dots, a_n$. We will always speak of actions and plans relative to some planning instance $\Pi = (V, \text{init}, \text{goal}, A)$, but we want to emphasize that when speaking (for example) of an action, the action need not be an element of A ; we require only that its precondition and postcondition are partial states over Π .

Starting from a state s , we define the state resulting from s by applying a plan P , denoted by $s[P]$, inductively as follows. For the empty plan $P = \epsilon$, we define $s[\epsilon] = s$. For non-empty plans P , denoting $P = P', a$, we define $s[P', a]$ as follows.

- If a is applicable at $s[P']$, then $s[P', a]$ is the state equal to $\text{post}(a)$ on variables $v \in \text{vars}(\text{post}(a))$, and equal to $s[P']$ on variables $v \in V \setminus \text{vars}(\text{post}(a))$.
- Otherwise, $s[P', a] = s[P']$.

We say that a state s is *reachable* (in an instance Π) if there exists a plan P such that $s = \text{init}[P]$. We are concerned with the problem of *plan generation*: given an instance $\Pi = (V, \text{init}, \text{goal}, A)$ obtain a plan P that *solves* it, that is, a plan P such that $\text{init}[P]$ is a goal state.

Note that sometimes we will use the representation of a partial function f as the relation $\{(a, b) : f(a) = b\}$.

3 Macro Computation Algorithm

In this section, we develop our macro computation algorithm. This algorithm makes use of a number of algorithmic subroutines. In particular, we will present the two

macro-producing operations discussed in the introduction, apply and transitive. First, we define the notion of *action graph*, the data structure on which these operations work.

Definition 1 *An action graph is a directed graph G whose vertex set, denoted by $V(G)$, is a set of states, and whose edge set, denoted by $E(G)$, consists of labelled edges that are actions; we denote the label of an edge e by $l_G(e)$ (or $l(e)$ when G is clear from context). Note that for every ordered pair of vertices (s, s') , there may be at most one edge (s, s') in $E(G)$,⁴ and each edge has exactly one label.*

We now define three functions which will themselves be used as subroutines in apply and transitive.

Definition 2 *We define the algorithmic function $\text{better}(a, (s, s'), G)$ as follows. Type-wise, the function $\text{better}(a, (s, s'), G)$ requires that a is an action, G is an action graph, and s and s' are vertices in G . The pseudocode for $\text{better}(a, (s, s'), G)$ is as follows:*

```

better(a, (s, s'), G) returns boolean
{
  if((s, s') not in E(G))
    return TRUE;

  if(pre(a) strictly contained in pre(l(s, s')) AND
     post(a) contained in post(l(s, s')))
    return TRUE;

  if(pre(a) contained in pre(l(s, s')) AND
     post(a) strictly contained in post(l(s, s')))
    return TRUE;

  return FALSE;
}

```

Definition 3 *We define the algorithmic function $\text{addlabel}(G, s, s', a)$ as follows. Type-wise, the function $\text{addlabel}(G, s, s', a)$ requires that G is an action graph, s and s' are vertices in G , and a is an action. The pseudocode for $\text{addlabel}(G, s, s', a)$ is as follows:*

```

addlabel(G, s, s', a) returns G'
{
  G' := G;
  if((s, s') not in E(G))
  {
    place (s, s') in E(G');
  }
  l_{G'}(s, s') := a;
  return G';
}

```

We remark that in our pseudocode, the assignment operator $:=$ is intended to be a value copy (as opposed to a reference copy, as in some programming languages).

Definition 4 *We define the algorithmic function $\text{combine}(a, a')$ as follows. Type-wise, the function $\text{combine}(a, a')$ requires that a and a' are actions. We remark that in all cases where we use the function $\text{combine}(a, a')$, there will exist states s_1, s_2 such that a is applicable at state s_1 , $s_1[a] = s_2$, and a' is applicable at state s_2 . The pseudocode for $\text{combine}(a, a')$ is as follows:*

⁴ That is, an action graph is not a multigraph.

```

combine(a, a') returns action a''
{
  R := vars(pre(a)) setminus vars(post(a));
  s := post(a) union (pre(a) | R);
  O := vars(post(a)) setminus vars(post(a'));
  pr := pre(a) union (pre(a') - s);
  pos := post(a') union (post(a) | O);
  return <pr; pos setminus pr>;
}

```

Here, the pipe symbol $|$ should be interpreted as function restriction, and the subtraction symbol in $(\text{pre}(a') - s)$ should be interpreted as a set difference, where the partial functions $\text{pre}(a')$ and S are viewed as relations. Intuitively, the partial state s represents what we know about a state if all we are told is that the action a has just been successfully executed.

The following propositions identify key properties of the combine function.

Proposition 5 *Let a, a' be actions and let s be a state. The action $\text{combine}(a, a')$ is applicable at s if and only if a is applicable at s and a' is applicable at $s[a]$. When this occurs, $s[\text{combine}(a, a')]$ is equal to $s[a, a']$.*

Proposition 6 *The function combine is associative. That is, the action $\text{combine}(\text{combine}(a_1, a_2), a_3)$ is equal to the action $\text{combine}(a_1, \text{combine}(a_2, a_3))$, assuming that there exists a state s such that a_1 is applicable in s , a_2 is applicable in $s[a_1]$, and a_3 is applicable in $s[a_1, a_2]$.*

We may now define the promised macro-producing operations.

Definition 7 *We define two algorithmic functions $\text{apply}(G, A, a, s)$ and $\text{transitive}(G, s_1, s_2, s_3)$. Type-wise, the function $\text{apply}(G, A, a, s)$ requires that G is an action graph, A is a set of actions, a is an action, and s is a vertex of G . The pseudocode for $\text{apply}(G, A, a, s)$ is as follows:*

```

apply(G, A, a, s) returns G'
{
  G' := G;
  if( a in A OR a appears as a label in G' ) {
    if( s[a] != s AND s[a] in V(G) ) {
      if( better(a, (s, s[a]), G) ) {
        G' := addlabel(G, s, s[a], a);
      }
    }
  }
  return G';
}

```

Type-wise, the function $\text{transitive}(G, s_1, s_2, s_3)$ requires that G is an action graph, and that s_1, s_2 , and s_3 are vertices in G . The pseudocode for $\text{transitive}(G, s_1, s_2, s_3)$ is as follows.

```

transitive(G, s_1, s_2, s_3) return G'
{
  G' := G;
  if( (s_1, s_2) in E(G) and
      (s_2, s_3) in E(G) ) {
    a := l(s_1, s_2);
    a' := l(s_2, s_3);
    a'' := combine(a, a');
    if( better(a'', (s_1, s_3), G) ) {
      G' := addlabel(G, s_1, s_3, a'');
    }
  }
  return G';
}

```

Within the function `transitive`, in the case that the `addlabel` function is called and returns a graph G' that is different from the input graph G , we say that the transition (s_1, a'', s_3) (where s_1, s_3, a'' are the arguments passed to the `addlabel` function) is produced by the function.

In general, we use the term *transition* to refer to a triple (s, a, s') consisting of states s, s' and an action a such that a is applicable at s and $s[a] = s'$.

Definition 8 An action graph program over a set of states S and a set of actions A is a sequence of commands $\Sigma = \sigma_1, \dots, \sigma_n$ of the form `apply`(G, A, a, s), with $s \in S$, or `transitive`(G, s_1, s_2, s_3), with $s_1, s_2, s_3 \in S$. The execution of an action graph program takes place as follows. First, G is initialized to be the action graph with S as vertices and no edges. Then, the commands of Σ are executed in order; for each i , after σ_i is executed, G is replaced with the returned value.

The following is our macro computation algorithm. As input, it takes a set of states S and a set of actions A . The running time can be bounded by $O(n|S|^3(|A| + |S|^2))$, where n denotes the number of variables.

```
compute_macros(S, A) returns G, M
{
  M := empty;
  V(G) := S;
  E(G) := empty set;

  do {
    A' := (A union l(E(G)));
    for all: a in A', s in V(G) {
      G := apply(G, A, a, s);
    }

    for all s1, s2, s3 in V(G) {
      G := transitive(G, s1, s2, s3);
      if(transitive produces a transition) {
        append "l(s1, s3) = l(s1, s2), l(s2, s3)" to M;
      }
    }
  }
  while(some change was made to G)

  return (G, M);
}
```

Understanding `compute_macros`. By a *combination* over A , we mean an action in A or an action that can be derived from actions in A by (possibly multiple) applications of the `combine` function.

Definition 9 We say that a transition (s, a, s') is *condition-minimal* with respect to a set of actions A if for any combination a' over A , if $s[a'] = s'$ then $\text{pre}(a) \subseteq \text{pre}(a')$ and $\text{post}(a) \subseteq \text{post}(a')$ (when $\text{pre}(a)$, $\text{pre}(a')$, $\text{post}(a)$, and $\text{post}(a')$ are viewed as relations).

Having defined the notion of a *condition-minimal* transition, we can now naturally define the notion of a *condition-minimal* program.

Definition 10 Relative to a planning instance Π , let S be a set of states, and let A, A' be sets of actions. An A -condition-minimal-program (for short, A -CM-program) over states S and actions A' is an action graph program over S and A such that

when executed, `apply` is only passed pairs (a, s) such that $(s, a, s[a])$ is condition-minimal with respect to A , and the transitive commands produce only transitions that are condition-minimal with respect to A .

We now define a notion of *derivable* action. This notion is defined recursively. Roughly speaking, derivable actions are actions that will provably be discovered as macros by the algorithm.

Definition 11 *Relative to a planning instance Π , let S be a set of states, and let A be a set of actions. We define the set of (S, A) -derivable actions recursively, as the smallest set satisfying: any action of a transition produced by an A -CM-program over states S and the set of actions that are (S, A) -derivable or in A , is (S, A) -derivable.*

Lemma 12 *Relative to a planning instance Π with action set A , let s be a state. Any $(H(s, k), A)$ -derivable action is discovered by a call to the function `compute_macros` with the first two arguments $H(s, k)$ and A , by which we mean that any such an action will appear as an edge label in the graph output by `compute_macros`.*

We emphasize that, in the `compute_macros` procedure, labels of edges are merely actions, which (as defined) are precondition-postcondition pairs that need not appear in the original set of actions A . When new edge labels are introduced, they are always obtained from existing labels or from A via the `combine` procedure, which permits the general applicability of edge labels.

Proof (Sketch). Let $\Sigma = \sigma_1, \dots, \sigma_n$ be an A -CM-program over $H(s, k)$ and actions that are discovered by `compute_macros`, and let H be the graph returned by `compute_macros`; we prove the result by induction.

We consider the execution of the program Σ with graph G . We prove by induction on $i \geq 1$ that after the command σ_i is executed and returns graph G_i , for every edge $(s, s') \in E(G_i)$, it holds that $(s, s') \in E(H)$ and $l_{G_i}(s, s') = l_H(s, s')$.

If σ_i is an `apply` command (with arguments s and a) that effects a change in the graph, then the input action must be in $l(E(G_i))$. The command σ_i can be successfully applied at H . Since H is a fixed point over all `apply` and transitive commands, the action a passed to `apply` or one that is better (according to the function `better`) must appear in H at $l_H(s, s[a])$. By condition-minimality of $(s, a, s[a])$, we have that $a = l_H(s, s[a])$.

If σ_i is a transitive command that produces a transition (s, a, s') , then the actions a' and a'' (from within the execution of the command), by induction hypothesis, appear in H . Since H is a fixed point over all `apply` and transitive commands, the action `combine`(a, a') or one that is better must appear in H at $l_H(s, s')$. By condition-minimality of $(s, \text{combine}(a, a'), s')$, we have that `combine`(a, a') = $l_H(s, s')$. \square

4 Examples

Blocksworld-arm. We will present results with respect to the following formulation of the Blocksworld-arm domain, which is based strongly on the propositional STRIPS formulation. We choose this formulation primarily to lighten the presentation, and

remark that it is straightforward to verify that our proofs and results apply to the propositional formulation.

Domain 13 (Blocksworld-arm domain) We use a formulation of this domain where there is an arm. Formally, in an instance $\Pi = (V, \text{init}, \text{goal}, A)$ of the Blocksworld-arm domain, there is a set of blocks B , and the variable set V is defined as $\{\text{arm}\} \cup \{b\text{-on} : b \in B\} \cup \{b\text{-clear} : b \in B\}$ where $D(\text{arm}) = \{\text{empty}\} \cup B$ and for all $b \in B$, $D(b\text{-on}) = \{\text{table}, \text{arm}\} \cup B$ and $D(b\text{-clear}) = \{\text{T}, \text{F}\}$. The $b\text{-on}$ variable tells what the block b is on top of, or whether it is being held by the arm, and the $b\text{-clear}$ variable tells whether or not the block b is clear.

There are four kinds of actions.

- $\forall b \in B$, $\text{pickup}_b = \langle b\text{-clear} = \text{T}, b\text{-on} = \text{table}, \text{arm} = \text{empty}; b\text{-clear} = \text{F}, b\text{-on} = \text{arm}, \text{arm} = b \rangle$
- $\forall b \in B$, $\text{putdown}_b = \langle \text{arm} = b; \text{arm} = \text{empty}, b\text{-clear} = \text{T}, b\text{-on} = \text{table} \rangle$
- $\forall b, c \in B$, $\text{unstack}_{b,c} = \langle b\text{-clear} = \text{T}, b\text{-on} = c, \text{arm} = \text{empty}; b\text{-clear} = \text{F}, b\text{-on} = \text{arm}, \text{arm} = b, c\text{-clear} = \text{T} \rangle$
- $\forall b, c \in B$, $\text{stack}_{b,c} = \langle \text{arm} = b, c\text{-clear} = \text{T}; \text{arm} = \text{empty}, c\text{-clear} = \text{F}, b\text{-clear} = \text{T}, b\text{-on} = c \rangle$

□

Definition 14 *Relative to an instance Π of Blocksworld-arm and a reachable state s of Π , a pile P of s is a non-empty sequence of blocks (b_1, \dots, b_k) such that $s(b_i\text{-on}) = b_{i+1}$ for all $i \in [1, k-1]$. The top of the pile P is the block $\text{top}(P) = b_1$, and the bottom of the pile is the block $\text{bottom}(P) = b_k$. The size of P is $|P| = k$.*

A sub-tower of s is a pile P such that $s(\text{top}(P)\text{-clear}) = \text{T}$; a tower is a sub-tower such that $s(\text{bottom}(P)\text{-on}) = \text{table}$.

We use the notation $P_{\geq}(b)$ (respectively, $P_{>}(b)$, $P_{\leq}(b)$, $P_{<}(b)$) to denote the sub-tower with bottom block b (respectively, the sub-tower stacked on b , and the piles supporting b , either including b or not.)

Definition 15 *Let Π be a planning instance of Blocksworld-arm. Let $P = (b_1, \dots, b_k)$ be a sequence of blocks, and b and b' two different blocks not in P . Let S be the partial state $\{b_1\text{-clear} = \text{T}, \text{arm} = \text{empty}, b_1\text{-on} = b_2, \dots, b_{k-1}\text{-on} = b_k\}$. We define several actions with S as common precondition.*

- *The action $\text{subtow-table}_{P,b} = \langle S, b_k\text{-on} = b; b_k\text{-on} = \text{table}, b\text{-clear} = \text{T} \rangle$ moves a sub-tower P from a block b to the table.*
- *The action $\text{subtow-block}_{P,b,b'} = \langle S, b_k\text{-on} = b, b'\text{-clear} = \text{T}; b_k\text{-on} = b', b\text{-clear} = \text{T}, b'\text{-clear} = \text{F} \rangle$ moves a sub-tower P from a block b onto a block b' .*
- *The action $\text{tow-block}_{P,b'} = \langle S, b_k\text{-on} = \text{table}, b'\text{-clear} = \text{T}; b_k\text{-on} = b', b'\text{-clear} = \text{F} \rangle$ moves a tower P onto a block b' .*

Theorem 16 Let Π be a planning instance of *Blocksworld-arm*, and let s be a reachable state with $s(\text{arm}) = \text{empty}$.

- If P is a sub-tower of s and $s(b_k\text{-on}) = b$, then $\text{subtow-table}_{P,b}$ is $(H(s, 4), A)$ -derivable.
- If P is a sub-tower of s , $s(b_k\text{-on}) = b$ and $s(b'\text{-clear}) = \text{T}$, then $\text{subtow-block}_{P,b,b'}$ is $(H(s, 5), A)$ -derivable.
- If P is a tower of s , $s(b_k\text{-on}) = \text{table}$ and $s(b'\text{-clear}) = \text{T}$, then $\text{tow-block}_{P,b'}$ is $(H(s, 4), A)$ -derivable.

Proof (Sketch). The proof has two parts. First, we show that the aforementioned actions are condition-minimal. Then, we describe how to obtain an A -CM-program that produces the actions inside $H(s, 5)$. We consider the case $a = \text{subtow-block}_{P,b,b'}$; the remaining actions admit similar proofs that only require Hamming distance 4.

To prove condition-minimality of action a we consider any combination $C = (a_1, \dots, a_t)$ of primitive actions from A such that $s[C] = s[a]$. We must show that the actions $\text{unstack}_{b_1,b_2}, \dots, \text{unstack}_{b_k,b}, \text{stack}_{b_k,b'}$ appear in C in the given relative order, and that no matter what are the remaining actions of C , this already implies that $\text{pre}(a) \subseteq \text{pre}(C)$ and $\text{post}(a) \subseteq \text{post}(C)$. We remark that the proof is not straightforward, since $\text{pre}(C)$ and $\text{post}(C)$ are the result of applying the combine subroutine to several actions not yet determined.

To prove that there exists an A -CM-program that produces actions subtow-table and tow-block inside $H(s, 4)$ we use a mutual induction; we omit the proof here. We then use these results for subtow-block , the proof for which we sketch here. Precisely, we now show that $\text{subtow-block}_{P,b,b'}$ is $(H(s, 5), A)$ -derivable.

When $|P| = 1$, we derive $\text{subtow-block}_{P,b,b'}$ by combining actions $a_1 = \text{unstack}_{b_1,b}$ and $a_2 = \text{stack}_{b_1,b'}$. The states $s[a_1]$ and $s[a_1, a_2]$ differ from s respectively 4 and 3 variables, so both states lie inside $H(s, 5)$. When $|P| = k$, let $P' = P_{>}(b_k)$ in state s . We use the derivable actions $a_1 = \text{subtow-table}_{P',b_k}$, $a_2 = \text{unstack}_{b_k,b}$, $a_3 = \text{stack}_{b_k,b'}$ and $a_4 = \text{tow-block}_{P',b_k}$. It is easy to check that the state $s[a_1, a_2, a_3]$ is the one that is furthest from s , differing at the 5 variables $b\text{-clear}$, $b_{k-1}\text{-on}$, $b_k\text{-clear}$, $b_k\text{-on}$ and $b'\text{-clear}$. \square

Towers of Hanoi. We study the formulation of *Towers of Hanoi* where, for every disk d , a variable stores the position (that is, the disk or the peg) the disk d is on. Formally, in an instance $\Pi = (V, \text{init}, \text{goal}, A)$ of the Towers of Hanoi domain, there is an ordered set of disks $D = \{d_1, \dots, d_k\}$ and a partially ordered set of positions $P = D \cup \{p_1, p_2, p_3\}$, where $d_i < p_j$ for every i and j . The set of variables V is defined as $\{d\text{-on} : d \in D\} \cup \{x\text{-clear} : x \in P\}$, where $D(d\text{-on}) = P$ and $D(x\text{-clear}) = \{\text{T}, \text{F}\}$.

The only actions in Towers of Hanoi are movement actions that move a disk d into a position x , provided that both d and p are clear and $d < x$.

- $\forall d \in D, \forall x, x' \in P$, if $d < x$, then define $\text{move}_{d,x',x} = \langle d\text{-clear} = \text{T}, x\text{-clear} = \text{T}, d\text{-on} = x'; x\text{-clear} = \text{F}, x'\text{-clear} = \text{T}, d\text{-on} = x \rangle$

We define this planning domain as the set of those planning instances Π such that the init and goal are certain predetermined total states. Namely, in both states init and goal it holds $d_i\text{-on} = d_{i+1}$ for all $i \in [1, \dots, k-1]$, $d_1\text{-clear} = \text{T}$, $d_i\text{-clear} = \text{F}$ for all $i \in [2, k]$ and $p_2\text{-clear} = \text{T}$. They only differ in three variables: $\text{init}(d_k\text{-on}) = p_1$, $\text{init}(p_1\text{-clear}) = \text{false}$ and $\text{init}(p_3\text{-clear}) = \text{T}$, but $\text{goal}(d_k\text{-on}) = p_3$, $\text{goal}(p_1\text{-clear}) = \text{T}$ and $\text{goal}(p_3\text{-clear}) = \text{F}$.

Definition 17 Let Π be a planning domain instance of Towers of Hanoi. Let i be an integer $i \in [1, k]$. Let $x = \text{init}(d_i\text{-on})$ and $x' \in \{p_2, p_3\}$. We define the action $\text{subtow-pos}_{i,x,x'} = \langle d_1\text{-clear} = \text{T}, d_1\text{-on} = d_2, \dots, d_{i-1}\text{-on} = d_i, d_i\text{-on} = x, x'\text{-clear} = \text{T}; d_i\text{-on} = x', x\text{-clear} = \text{T}, x'\text{-clear} = \text{F} \rangle$, that is, the action that moves the tower of depth i from x to x' .

Theorem 18 The actions $\text{subtow-pos}_{i,x,x'}$ are $(H(\text{init}, 7), A)$ -derivable.

We prove this by induction on i , the height of the subtower. To derive actions of the form $\text{subtow-pos}_{i+1,x,x'}$ from the actions of the form $\text{subtow-pos}_{i,x,x'}$, we make use of the classical recursive solution to Towers of Hanoi; an analysis shows that this recursive step stays within Hamming distance 7 of the initial state.

5 Width

In this section, we present the definition of macro persistent Hamming width and present the width results on domains. For a state s , we define $\text{wrong}(s)$ to be the variables that are not in the goal state, that is, $\text{wrong}(s) = \{v \in \text{vars}(\text{goal}) \mid s(v) \neq \text{goal}(v)\}$.

Definition 19 With respect to a planning instance $(V, \text{init}, \text{goal}, A)$, we say that a state s' is an improvement of a state s if

- for all $v \in V$, if $v \in \text{vars}(\text{goal})$ and $s(v) = \text{goal}(v)$, then $s'(v) = \text{goal}(v)$; and,
- there exists $u \in \text{vars}(\text{goal})$ such that $u \in \text{wrong}(s)$ and $s'(u) = \text{goal}(u)$.

In this case, we say that such a variable u is a variable being improved.

Definition 20 With respect to a planning instance $(V, \text{init}, \text{goal}, A)$, we say that a plan P improves a state s if $s[P]$ is a goal state, or $s[P]$ is an improvement of s .

Relative to a planning instance, we say that a state s dominates another state s' if $\{v \in V : s(v) \neq s'(v)\} \subseteq \text{vars}(\text{goal})$ and $\text{wrong}(s) \subseteq \text{wrong}(s')$; intuitively, s' may differ from s only in that it may have more variables set to their goal position. Recall that for a state s and natural number $k \geq 0$, we use $H(s, k)$ to denote the set of all states within Hamming distance k from s .

We now give the official definition of our new width notion.

Definition 21 A planning instance $(V, \text{init}, \text{goal}, A)$ has macro persistent Hamming width k (for short, MPH width k) if no plan exists, or for every reachable state s dominating the initial state init , there exists a plan over $(H(s, k), A)$ -derivable actions improving s that stays within Hamming distance k of s .

It is straightforwardly verified that if an instance has PH width k , then it has MPH width k .

We now give a polynomial-time algorithm for sets of planning instances having bounded MPH width. We establish the following theorem.

Theorem 22 Let \mathcal{C} be a set of planning instances having MPH width k . The plan generation problem for \mathcal{C} is solvable in polynomial time via the following algorithm, in time $O(n^{3k+2}d^{3k}(a + (nd)^{2k}))$. Here, n denotes the number of variables, d denotes the maximum size of a domain, and a denotes the number of actions.

```

solve_mph((V, init, goal, A), k)
{
  Q := empty plan;
  M := empty set of macros;
  s := init;

  while( s not a goal state ) {
    (G, M') := compute_macros(H(s,k), A);
    append M' to M;

    if(an improvement s' of s is reachable from s in G) {
      s := s';
    }
    else {
      print "?";
      halt;
    }
    append l(s, s') to Q;
  }
  print M;
  print Q;
}

```

Proof (Sketch). Let $\Pi \in \mathcal{C}$ be a planning instance such that there exists a plan for $\Pi = (V, \text{init}, \text{goal}, A)$. We want to show that `solve_mph` outputs a plan. During the execution of `solve_mph`, the state s can only be replaced by states that are improvements of it, and thus s always dominates the initial state init . By definition of MPH width, then, for any s encountered during execution, there exists a plan over $(H(s, k), A)$ -derivable actions improving s staying within Hamming distance k of s . By Lemma 12, all of the actions are discovered by `compute_macros`, and thus the reachability check in `solve_mph` will find an improvement.

We now perform a running time analysis of the algorithm. Let v denote the number of vertices in the graphs in `compute_macros`, that is, $|H(s, k)|$. We have $v \leq \binom{n}{k}d^k \in O((nd)^k)$. Let e be the maximum number of edges; we have $e = \binom{v}{2} \in O((nd)^{2k})$. The do-while loop in `compute_macros` will execute at most $2n \cdot e \in O(ne)$ times, since once an edge is introduced, its label may change at most $2n$ times, by definition of *better*. Each time this loop iterates, it uses no more than $(a + e)v + v^3$ time: *apply* can be called on no more than $(a + e)v$ inputs, and *transitive* can be called on no more than v^3 inputs. The while loop in `solve_mph` loops at most n times, and each time, by the previous discussion, it requires $ne((a + e)v + v^3)$ time for the call to `compute_macros`, and $(v + e)$ time for the reachability check. The total time is thus $O(n(ne((a + e)v + v^3) + (v + e)))$ which is $O(n^2e((a + e)v + v^3))$ which is $O(n^2e(a + e)v)$ which is $O(n^{3k+2}d^{3k}(a + (nd)^{2k}))$. \square

Blocksworld.

Theorem 23 *All instances of the Blocksworld-arm domain have MPH-width 10.*

According to Theorem 16, at any state s we may consider our set of applicable actions enriched by this new macro-actions. We now show how can these new actions be used to improve any reachable state s . The proof is conceptually simple: improve s just by moving around a few piles of blocks. For instance, if $s(b\text{-on}) = b'$ but $\text{goal}(b\text{-on}) = b''$, apply actions $\text{subtow-table}_{P_{>(b''),b''}}$, $\text{subtow-block}_{P_{\geq(b),b',b''}}$. However, we must not forget that variables that were already in the goal state in s must remain so after the improvement. For instance, if b was on top of b' in s , then unstacking b from b' will make $b'\text{-clear}$ change from F to T. We may try to solve this by placing anything whatever on top of b' , but then this movement may affect some other variable which was already in the goal state, and so forth.

The following lemma is a case-by-case analysis of the solution to the difficulty we have described.

Lemma 24 *Let Π be an instance of the Blocksworld-arm domain, and let s be a reachable state of Π such that $s(\text{arm}) = \text{empty}$. If a block b is such that $s(b\text{-clear}) = \text{T}$ but $\text{goal}(b\text{-clear}) = \text{F}$, then there is a plan using $(H(s, 6), A)$ -derivable actions that improves the variable $b\text{-clear}$ in s .*

Proof (Sketch). Clearly, $b = \text{top}(P_1)$ for some tower P_1 of s . Let P_2, \dots, P_t be the remaining $t - 1$ towers of s , and let t' be the number of towers of goal.

The proof proceeds by cases. If there is i such that $\text{goal}(\text{bottom}(P_i)\text{-on}) \neq \text{table}$, we say we are in Case 1. Otherwise, it holds that $t \leq t'$. In particular, there are t' blocks b' such that $\text{goal}(b'\text{-clear}) = \text{T}$ (block b not one of them), and t blocks $b' \neq b$ such that $s(b'\text{-clear}) = \text{T}$ (block b being one of them). It follows that it exists a block b' such that $\text{goal}(b'\text{-clear}) = \text{T}$ but $s(b'\text{-clear}) = \text{F}$. We say we are in Case 2 if the block b' belongs to the tower P_1 , and in Case 3 if not. Throughout this proof we say that a block b' is badly placed if $s(b'\text{-on}) \neq \text{goal}(b'\text{-on})$.

Case 1. The tower P_i is wrongly placed in the table, so we are allowed to change the value of $\text{bottom}(P_i)\text{-on}$ without worry.

- (a) If $i \neq 1$, then use $\text{tow-block}_{P_i,b}$ to stack the tower P_i on top of b .
- (b) If $i = 1$ and a tower P_j with $j > 1$ has a badly placed block b' , then a possible solution is to insert P_1 below b' . That is, move the sub-tower $P_{\geq(b')}$ on top of P_1 , and then move the new resulting tower on top of the place where b' was in state s , that is, on top of $s(b'\text{-on})$.
- (c) If $i = 1$ and no tower P_j with $j > 1$ has badly placed blocks., then consider the pile P'_i in state goal that b belongs to, and let $b' = \top(P'_i)$. If block b' is in P_j for $j > 1$ in state s , then P_j would have some badly placed block, since b' and b , sharing pile P'_i in the goal state, would be in different piles in state s . So b' is in P_1 , $\text{goal}(b'\text{-clear}) = \text{T}$ but $s(b'\text{-clear}) = \text{F}$, since b is the top of P_1 . It follows that the block on top of b' in pile P_1 is badly placed. To improve $b\text{-clear}$ use actions $\text{subtow-table}_{P_{>(b'),b'}}$ and $\text{tow-block}_{P_{\leq(b'),b}}$, that is, break the tower over block b' and swap the two parts.

Note that an action like $\text{tow-block}_{P_{\leq}(b'),b}$ is not derivable from s since the pile $P_{\leq}(b')$ is not a subtower of s , but it is derivable from $s' = s[\text{subtow-table}_{P_{>}(b'),b'}]$, a state within distance 2 from s . This fact may increase the width required to discover the derivable actions. In our case, a careful examination reveals that Situation (b) requires width 5 and Situation (c) requires width 4.

Case 2. Note that if Case 1 does not apply then $t \leq t'$. Let b' be the highest block in P_1 such that $s(b'\text{-clear}) = \text{F}$ but $\text{goal}(b'\text{-clear}) = \text{T}$.

- (a) If $t > 1$ and a tower P_j with $j > 1$ has a badly placed block b'' , then we insert the pile $P_{>}(b')$ below b'' , analogously to Situation (b) in Case 1. This procedure improves variables $b\text{-clear}$ and $b'\text{-clear}$ at the same time, but it needs width 6.
- (b) If there is a second block b'' in P_1 such that $\text{goal}(b''\text{-clear}) = \text{T}$, then swap the sub-tower $P_{>}(b')$ with the pile between b' and b'' , the block b'' not including. The procedure is similar to Situation (c) in Case 1, but it requires width 5.
- (c) If there is no second block b'' in P_1 but all the towers P_j with $j > 1$ have no badly placed blocks, it follows that either $t = 1$ or all towers P_j with $j > 1$ are exactly as in the goal state. Observe that, in this situation, the blocks of P_1 form a tower in s and in goal, but the order of the blocks in the two towers must differ: the pile $P' = P_{\leq}(b')$, which is such that $\text{goal}(\text{top}(P')\text{-clear}) = \text{T}$ and $\text{goal}(\text{bottom}(P')\text{-on}) = \text{table}$, cannot be a pile in goal. Hence there is a badly placed block below b' . This situation is analogous to Situation (b) in Case 2, and it also requires width 5.

Case 3. There is a block b' such that $s(b'\text{-clear}) = \text{F}$ but $\text{goal}(b'\text{-clear}) = \text{T}$, and the block is in some tower P_i other than P_1 . We just stack the sub-tower $P_{>}(b')$ on top of b . \square

Proof (Sketch). (of Theorem 23) Let Π be an instance of the Blocksworld-arm domain, and let s be a reachable state of Π that is not a goal state. We present the case where $s(\text{arm}) = \text{goal}(\text{arm}) = \text{empty}$.

Improving $b\text{-on}$.

- $s(b\text{-on}) = \text{table}$, $\text{goal}(b\text{-on}) = b'$. If $s(b'\text{-clear}) = \text{F}$, then move the sub-tower $P_{>}(b')$ onto the table. (This changes the variable $b''\text{-on}$, where b'' is the block on top of b' in s , which was not in the goal state in s .) Now the block b' is clear, so we stack the tower b is the bottom of onto b' .
- $s(b\text{-on}) = b''$, $\text{goal}(b\text{-on}) = b'$. If $s(b'\text{-clear}) = \text{F}$ then we can swap piles $P_{>}(b'')$ and $P_{>}(b')$. Otherwise, we stack $P_{>}(b'')$ on top of b' , but then $b''\text{-clear}$ becomes true. This is a problem if $\text{goal}(b''\text{-clear}) = \text{F}$, so we may need to apply Lemma 24 at the current state. Again, a careful examination shows that we may need width 8.
- $s(b\text{-on}) = b''$, $\text{goal}(b\text{-on}) = \text{table}$. Move $P_{>}(b)$ onto the table. As in the previous case apply Lemma 24 to the current state if $\text{goal}(b''\text{-clear}) = \text{F}$. In this case we may need width 7.

Improving b -clear.

- $s(b\text{-clear}) = \text{F}$, $\text{goal}(b\text{-clear}) = \text{T}$. Move the pile $P_{>}(b)$ onto the table, so width 4 is enough.
- $s(b\text{-clear}) = \text{T}$, $\text{goal}(b\text{-clear}) = \text{F}$. Just apply Lemma 24, which requires width 6.

Under the assumption that $s(\text{arm}) = \text{goal}(\text{arm}) = \text{empty}$, there is nothing else to show, since we have explained how to improve any variable. The width number 10 comes from the analysis of the other cases. \square

Towers of Hanoi.

Theorem 25 *All instances of the Towers of Hanoi domain have MPH-width 7.*

Each instance can be solved by a single application of the action $\text{subtow-pos}_{k,p_1,p_3}$.

References

- [1] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.
- [2] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-FF: Improving ai planning with automatically learned macro-operators. *JAIR*, 24:581–621, 2005.
- [3] R. Brafman and C. Domshlak. Structure and complexity of planning with unary operators. *JAIR*, 18:315–349, 2003.
- [4] Ronen Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *AAAI 2006*, 2006.
- [5] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.
- [6] Hubie Chen and Omer Gimenez. Act local, think global: Width notions for tractable planning. 2007. ICAPS 2007.
- [7] A. Coles and A. Smith. Marvin: A heuristic search planner with online macro-action learning. *JAIR*, 28:119–156, 2007.
- [8] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76:625–655, 1995.
- [9] R. E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971.
- [10] Patrik Haslum. Reducing accidental complexity in planning problems. In *Proc. 20th International Joint Conference on Artificial Intelligence*, 2007.
- [11] Malte Helmert. The fast downward planning system. *JAIR*, 26:191–246, 2006.
- [12] I. Hernádvölgyi. Searching for macro-operators with automatically generated heuristics. In *14th Canadian Conference on AI*, pages 194–203, 2001.
- [13] J. Hoffmann. *Utilizing Problem Structure in Planning: A Local Search Approach*, volume 2854 of *LNAI*. Springer-Verlag, 2003.
- [14] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.
- [15] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–317, 1989.
- [16] Anders Jonsson. The role of macros in tractable planning over causal graphs. In *Proc. 20th International Joint Conference on Artificial Intelligence*, pages 1936–1941, 2007.
- [17] A. Junghanns and J. Schaeffer. Sokoban: enhancing single-agent search using domain knowledge. *Artificial Intelligence*, 129:219–251, 2001.
- [18] Richard E. Korf. *Learning to solve problems by solving for macro-operators*. Research notes in artificial intelligence. Pitman, 1985.
- [19] S. Minton. Selectively generalizing plans for problem-solving. In *IJCAI-85*, pages 596–599, 1985.
- [20] M. A. H. Newton, J. Levine, M. Fox, and D. Long. Learning macro-actions for arbitrary planners and domains. In *ICAPS-07*, 2007.