On-the-Fly Macros

Hubie Chen[1] and Omer Giménez[2]

[1] Dept. of Information and Communication Technologies, UPF (Barcelona, Spain)
hubie.chen@upf.edu
[2] Dept. de Llenguatges i Sistemes Informàtics, UPC (Barcelona, Spain)
omer.gimenez@upc.edu

Abstract. We present a domain-independent algorithm for planning 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

A planning instance involves deciding if an *initial state* can be transformed into a *goal state* via the application of a sequence of *actions*. Each planning instance has a set of variables associated with it; a *state* is a mapping defined on these variables that describes the relevant features of the situation being modelled. An action describes how a state can be transformed into another. For instance, in the well-known 15 puzzle, there is a 4-by-4 grid with 15 tiles labelled with the numbers 1 through 15, and the objective is to arrange the tiles in increasing order via a sequence of moves; a move consists of sliding a tile into the unoccupied position. This puzzle can be formulated as a planning problem by letting the variables represent locations of the grid, and a state would map each location to a tile or the unoccupied position; the act of sliding a tile into the unoccupied position corresponds to the application of an action.

A *domain* is a collection of related planning instances that typically model a particular application area. For instance, the set of "sliding tile" problem instances on n-by-n grids with $n \geq 2$ over all possible initial states might be taken as a domain. An overarching research goal in planning is to develop an automated planner that can robustly solve problems in a domain-independent manner.

Macros—a term we use broadly to refer to combinations of actions—have long been studied in planning [1, 2]. Many domain-dependent applications of macros have been exhibited and studied [3–5]; 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 [6–8].

In this paper, we present a domain-independent algorithm that computes macros in a novel way. Our algorithm computes macros "on-the-fly" and does not require previously learned or inferred information, nor any prior domain knowledge. This stands in contrast to previous work on macros, since our macros are generated and applied not over a domain or even over an instance, but with respect to a "current state" s and a

H. Ono, M. Kanazawa, and R. de Queiroz (Eds.): WoLLIC 2009, LNAI 5514, pp. 155–169, 2009.
© Springer-Verlag Berlin Heidelberg 2009

(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.

We exhibit the power of our algorithm by using it to define new domain-independent tractable classes of planning that strictly extend previously defined such classes [9], 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 Π of planning and a subset S of the state set of Π, 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 actions 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 (which, under the usual propositional STRIPS formulation, is known to be PSPACE-complete [10]).

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 action, 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 action, as in the just-described algorithm, but each pair is labelled with its connecting action. 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 action 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. The second type of transformation is to take a state $s \in S$ and a label of an existing pair (s_1, s_2) with $s \neq s_1$, 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 action 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 another state s 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. 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 a concrete result to offer the reader a feel for the power of our algorithm. Consider the *Towers of Hanoi* domain. Figure 1 shows the initial state init and the goal state goal for the case $n = 5$. Although these two states only differ in the values of three variables—namely, d_n-on, p_1-clear and p_3-clear—it is well known that $(2^n - 1)$ disk movements are required to reach the goal state from the initial state. We prove that our algorithm, given the set $S = H(\mathsf{init}, 4)$, by which we denote the set of states within Hamming distance 4 of init, will discover macros that move any subtower of discs from one peg to another; in particular, it will derive a macro for moving the whole tower from the first to the third peg, thus solving the problem.The radius 4 arises from the local transformations that our algorithm needs to discover the macros, and is completely independent of the number n of discs of the Towers of Hanoi instance. Since the set $S = H(\mathsf{init}, 4)$ is of polynomial size $O(n^4)$, our algorithm finds the exponentially long solution in polynomial time! Indeed, this is only possible due to the use of macros, as in [11]: the macro solving the problem is defined in terms of other macros, which are in turn defined in terms of other macros, and so on.

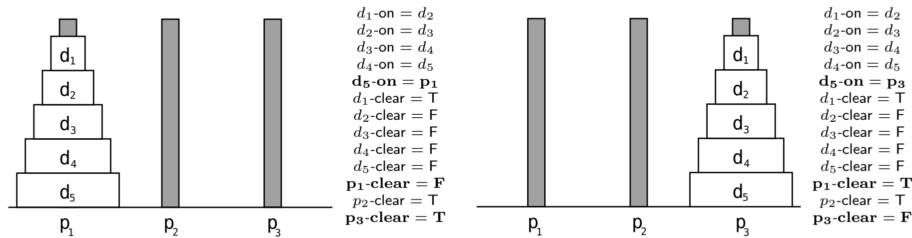

Fig. 1. Initial and goal states in the Towers of Hanoi planning problem of size $n = 5$

Our algorithm is fully domain-independent, and does not require the particular characteristics of the *Towers of Hanoi* domain to produce interesting macros. Indeed, we also obtain useful macros for the *Blocksworld-arm* domain, where a robotic arm has to move and stack blocks to reach a goal configuration. In this domain, we show that for a state s and the set $S = H(s, 4)$, our algorithm derives macros moving any subtower of blocks into the ground, preserving the subtower structure.

Towards a tractability theory of domain-independent planning. Many of the benchmark domains—such as *Gripper*, *Logistics* and *Blocksworld-arm*—can now be handled effectively and simultaneously by domain-independent planners, as borne out by empirical evidence [12]. 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 set of actions [10, 13, 14], studying restrictions of the causal graph, as in [15–17, 11], or empirical evaluation of simplification rules [18]. Aligned with the present aims is the work of Hoffmann [19] that gives proofs that certain benchmark domains are solvable by local search with respect to various heuristics.

To demonstrate the efficacy of our algorithm, we use it to extend previously defined tractable classes. In particular, previous work [9] 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 appendix, 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 [1]. Early work includes [20], which developed filtering algorithms for discovered macros, and [2], which demonstrated the ability of macros to exponentially reduce the size of the search space. Some recent research on integrating macros into domain-independent planning systems is as follows. *Macro-FF* [6] 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 [8].

A more theoretical approach was taken by [11], 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.

2 Preliminaries

An instance of the planning problem is a tuple $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$ whose components are described as follows. The set 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 $\mathsf{vars}(p)$ of the variables V such that for all $v \in \mathsf{vars}(p)$, it holds that $p(v) \in D(v)$. Then, init is a state called the initial state, and goal is a partial state. Finally, A is a set of *actions*. An action a consists of a *precondition* $\mathsf{pre}(a)$, which is a partial state, as well as a *postcondition* $\mathsf{post}(a)$, also a partial state. We sometimes denote an action a by $\langle \mathsf{pre}(a); \mathsf{post}(a) \rangle$.

The Hamming distance between two states s, s' is the number of differing variables. The set $H(s, k)$ is the set of all states s' within distance k of s. For a state or partial state s and a subset W of the variable set V, we use $s \upharpoonright W$ to denote the partial state resulting from restricting s to W. Sometimes we will use the representation of a partial function f as the relation $\{(a, b) : f(a) = b\}$. We say that a state s is a *goal state* if $(s \upharpoonright \mathsf{vars}(\mathsf{goal})) = \mathsf{goal}$ or, equivalently, viewing s and goal as relations, $\mathsf{goal} \subseteq s$.

An action a is *applicable* at a state s if $\mathsf{pre}(a) \subseteq s$. We define a *plan* to be a sequence of actions $P = a_1, \ldots, a_n$, with $a_i \in A$. We will always speak of actions and plans relative to some planning instance $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$, but we want to emphasize that when speaking 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, where $P = P', a$ and a applicable on $s[P']$, we define $s[P', a]$ as the state equal to $\mathsf{post}(a)$ on variables $v \in \mathsf{vars}(\mathsf{post}(a))$, and equal to $s[P']$ on variables $v \in V \setminus \mathsf{vars}(\mathsf{post}(a))$. If a not applicable on $s[P']$, then $s[P', a] = s[P']$.

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

3 Macro Computation Algorithm

In this section, we develop our macro computation algorithm. This algorithm makes use of a number of algorithmic subroutines. Before defining them, we introduce the notion of *action graph*, the data structure on which these operations work. We emphasize that whenever we refer to actions, both in the definitions and in the algorithms, we mean precondition-postcondition pairs that need not appear in the original set of actions A.

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, with the restriction that the label a of an edge (s, s') must be applicable at s and $s[a] = s'$. We denote the the label of an edge $e = (s_1, s_2)$ by $a = l_G(e)$ (or $l(e)$ when G is clear from context), and we say that the triplet (s, a, s') forms a* transition.

Note that for every ordered pair of vertices (s, s'), there may be at most one edge (s, s') in $E(G)$, and each edge has exactly one label.

We now give the pseudo-code of the two macro-producing operations discussed in the introduction, apply and transitive, whose aim is to add as many edges as possible to

the action graph G. They depend on the algorithmic functions better, addlabel and combine, which we define later. Notice 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 2. *The pseudocode of the macro-producing operations* apply(G, A, a, s) *and* transitive(G, s_1, s_2, s_3) *is as follows. Typewise, G is an action graph, A is a set of actions, a is an action and s, s_1, s_2 and s_3 are vertices in G.*

```
apply(G, A, a, s) returns G' {
        G' := G;
        if  (a∈A ∨ a appears as a label in G') ∧
            s[a]≠s ∧ s[a]∈V(G) ∧ better(a, (s,s[a]), G)  then
                G' := addlabel(G, s, s[a], a);
        return G';
}
transitive(G, s1, s2, s3) returns G' {
        G' := G;
        if  (s1,s2)∈E(G) ∧ (s2,s3)∈E(G)  then {
                a := l(s1,s2);
                a' := l(s2,s3);
                a'' := combine(a, a');
                if  better(a'', (s1,s3), G)  then G' := addlabel(G, s1, s3, a'');
        }
        return G';
}
```

Definition 3. *The pseudocode of the functions* addlabel(G, s, s', a), better$(a,(s, s'), G)$ *and* combine(a, a') *is as follows. Typewise, G is an action graph, s and s' are vertices in G, and a and a' are actions. In the particular case of* combine(a, a') *we require that, for some state s_1, a and a' are respectively applicable in s_1 and $s_1[a]$. This requirement is enforced in the function* transitive*, which is the only place that calls* combine(a, a').

```
addlabel(G, s, s', a) returns G' {
        G' := G;
        if  (s,s')∉E(G)  then place (s,s') in E(G');
        l_{G'}(s,s') := a;
        return G';
}
better(a, (s,s'), G) returns boolean  {
         if  (s,s')∉E(G)  then return TRUE;
         a' = l(s,s')
         if  pre(a)⊆pre(a') ∧ post(a)⊆post(a') ∧ a≠a'  then return TRUE;
         else return FALSE;
}
combine(a, a') returns action a'' {
        R := vars(pre(a))\vars(post(a));
        s := post(a) ∪ (pre(a)↾R);
        O := vars(post(a))\vars(post(a'));
        pr := pre(a) ∪ (pre(a')\s);
        pos := post(a') ∪ (post(a)↾O);
        return <pr; pos\pr>;
}
```

Note that in the definition of combine 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 4. *Let a, a' be actions and let s be a state. The action* 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, $s[$*combine*$(a, a')]$ is equal to $s[a, a']$.*

This property ensures the *general applicability* of actions obtained from the combine procedure. That is, we may merge the pair of actions a, a' into a single action $a'' =$ combine(a, a'), since the sequence (a, a') and the action a'' are indistinguishable: they can be applied to the same states, and they produce the same result.

Proposition 5 (Associativity). *Assume that a_1, a_2 and a_3 are actions that are respectively applicable in states s, $s[a_1]$ and $s[a_1, a_2]$, for some s. Then, the action* combine(combine$(a_1, a_2), a_3)$ *is equal to the action* combine$(a_1,$ combine$(a_2, a_3))$.

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 := {};
        V(G) := S;
        E(G) := ∅;
        do {
                A' := (A ∪ labels(E(G)));
                for all:  a∈A', s∈V(G)  {
                        G := apply(G, A, a, s);
                }
                for all:  s1,s2,s3∈V(G)  {
                        G := transitive(G, s1, s2, s3);
                        if  transitive produces a transition  then
                                M[l(s1,s3)] := l(s1,s2), l(s2,s3);
                }
        } while (some change was made to G);
        return (G, M);
}
```

The resulting action graph G contains the reachability information found by the algorithm. The mapping M contains the macros that have been used, that is, the description of how to decompose the actions appearing in G into simpler actions, up to those of A.

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. Clearly, all actions derived by the compute_macros algorithm are combinations of the original set of actions A. Although the actual combinations derived by the algorithm may depend on details such as the order in which the `for all` loops are executed, under certain assumptions it is possible to prove that some actions, which we call *derivable*, will be discovered by any run of the algorithm.

Definition 6. *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* pre$(a) \subseteq$ pre(a') *and* post$(a) \subseteq$ post(a'). *In other words, either $a = a'$ or $better(a, a')$ holds true.*

Having defined the notion of a *condition-minimal* transition, we turn our attention to those that can be found by the compute_macros algorithm.

Definition 7. *An* action graph program *over a set of states S and a set of actions A is a sequence of commands $\Sigma = \sigma_1, \ldots, \sigma_n$ 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.*

Definition 8. *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.*

Definition 9. *The set of (S, A)-derivable actions is 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 10. *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.

4 Results on compute_macros

We will present results with respect to formulations of the Blocksworld-arm and the Towers of Hanoi domains, which are based strongly on their propositional STRIPS formulations. We choose these formulations primarily to lighten the presentation, and remark that it is straightforward to verify that our proofs and results apply to the propositional formulations.

Domain 1. (Blocksworld-arm domain) We use a formulation of this domain where there is an arm. Formally, in an instance $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$ of the Blocksworld-arm domain, there is a set of blocks B, and the variable set V is defined as $\{\mathsf{arm}\} \cup \{b\text{-}\mathsf{on} : b \in B\} \cup \{b\text{-}\mathsf{clear} : b \in B\}$ where $D(\mathsf{arm}) = \{\mathsf{empty}\} \cup B$ and for all $b \in B$, $D(b\text{-}\mathsf{on}) = \{\mathsf{table}, \mathsf{arm}\} \cup B$ and $D(b\text{-}\mathsf{clear}) = \{\mathsf{T}, \mathsf{F}\}$. The b-on variable tells what the block b is on top of, or whether it is being held by the arm, and the b-clear variable tells whether or not the block b is clear.

There are four kinds of actions.

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

We say that a state s is *consistent* if it satisfies the following restrictions.

- $\forall b' \in B, s(b'\text{-}\mathsf{clear}) = \mathsf{T} \Rightarrow |\{b \in B | s(b\text{-}\mathsf{on}) = b'\}| = 0$.
- $\forall b' \in B, s(b'\text{-}\mathsf{clear}) = \mathsf{F}, s(\mathsf{arm}) \neq b' \Rightarrow |\{b \in B | s(b\text{-}\mathsf{on}) = b'\}| = 1$.
- $\forall b \in B, s(\mathsf{arm}) = b \Leftrightarrow s(b\text{-}\mathsf{on}) = \mathsf{arm}, s(b\text{-}\mathsf{clear}) = \mathsf{F}$.
- $\forall b \in B$, $\exists b_1, \ldots, b_k \in B$ such that $b\text{-}\mathsf{on} = b_k, b_k\text{-}\mathsf{on} = b_{k-1}, \ldots, b_1\text{-}\mathsf{on} = \mathsf{table}$.

The planning domain Blocksworld-arm is the set of planning instances Π where init and goal are consistent, goal is total and $\mathsf{goal}(\mathsf{arm}) = \mathsf{empty}$. In any Blocksworld-arm planning instance, a state s is reachable if and only if s is consistent. In particular, all planning instances with consistent goal state are solvable. □

Definition 11. *A* pile P *of a state* s *is a non-empty sequence of blocks* $(b_1, \ldots, b_k)$ *such that* $s(b_i\text{-}\mathsf{on}) = b_{i+1}$ *for all* $i \in [1, k-1]$*. The* top *of the pile* P *is the block* $\mathsf{top}(P) = b_1$*, and the* bottom *of the pile is the block* $\mathsf{bottom}(P) = b_k$*. The* size *of* P *is* $|P| = k$*. A* sub-tower *of* s *is a pile* P *such that* $s(\mathsf{top}(P)\text{-}\mathsf{clear}) = \mathsf{T}$*; a tower is a sub-tower such that* $s(\mathsf{bottom}(P)\text{-}\mathsf{on}) = \mathsf{table}$*. The notation* $P_{\geq}(b)$ *(respectively,* $P_{>}(b)$*,* $P_{\leq}(b)$*,* $P_{<}(b)$*) denotes 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 12. *Let* $P = (b_1, \ldots, b_k)$ *be a pile,* b *and* b' *be two different blocks not in* P*, and* q *be the partial state* $\{b_1\text{-}\mathsf{clear} = \mathsf{T}, \mathsf{arm} = \mathsf{empty}, b_1\text{-}\mathsf{on} = b_2, \ldots, b_{k-1}\text{-}\mathsf{on} = b_k\}$*. We define actions with* q *as common precondition. The action* $\mathsf{subtow\text{-}table}_{P,b} = \langle q, b_k\text{-}\mathsf{on} = b; b_k\text{-}\mathsf{on} = \mathsf{table}, b\text{-}\mathsf{clear} = \mathsf{T}\rangle$ *moves a sub-tower* P *from* b *to the table. The action* $\mathsf{subtow\text{-}block}_{P,b,b'} = \langle q, b_k\text{-}\mathsf{on} = b, b'\text{-}\mathsf{clear} = \mathsf{T}; b_k\text{-}\mathsf{on} = b', b\text{-}\mathsf{clear} = \mathsf{T}, b'\text{-}\mathsf{clear} = \mathsf{F}\rangle$ *moves a sub-tower* P *from a* b *onto* b'*. The action* $\mathsf{tow\text{-}block}_{P,b'} = \langle q, b_k\text{-}\mathsf{on} = \mathsf{table}, b'\text{-}\mathsf{clear} = \mathsf{T}; b_k\text{-}\mathsf{on} = b', b'\text{-}\mathsf{clear} = \mathsf{F}\rangle$ *moves a tower* P *onto* b'*.*

Theorem 13. *Let* s *be a reachable state with* $s(\mathsf{arm}) = \mathsf{empty}$*. If* P *is a sub-tower of* s *and* $s(b_k\text{-}\mathsf{on}) = b$*, then* $\mathsf{subtow\text{-}table}_{P,b}$ *is* $(H(s,4), A)$*-derivable. If* P *is a sub-tower of* s*,* $s(b_k\text{-}\mathsf{on}) = b$ *and* $s(b'\text{-}\mathsf{clear}) = \mathsf{T}$*, then* $\mathsf{subtow\text{-}block}_{P,b,b'}$ *is* $(H(s,5), A)$*-derivable. If* P *is a tower of* s*,* $s(b_k\text{-}\mathsf{on}) = \mathsf{table}$ *and* $s(b'\text{-}\mathsf{clear}) = \mathsf{T}$*, then* $\mathsf{tow\text{-}block}_{P,b'}$ *is* $(H(s,4), A)$*-derivable.*

The previous theorem states that our macro computing algorithm will *always* discover these interesting actions when applied to the state s and the set $S = H(s,5)$. Note that the polynomial bound on the running time of the algorithm does not depend on the height of the subtower being moved.

Domain 2. (Towers of Hanoi domain) 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 $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$ of the Towers of Hanoi domain, there is a set of disks $D = \{d_1, \ldots, d_n\}$ and a set of positions $P = D \cup \{p_1, p_2, p_3\}$. We consider the partial order $<$ in the set of positions defined by $d_i < d_j$ for every $i < j$, and $d_i < p_j$ for every i and j. The set of variables V is defined as $\{d\text{-}\mathsf{on} : d \in D\} \cup \{x\text{-}\mathsf{clear} : x \in P\}$, where $D(d\text{-}\mathsf{on}) = P$ and $D(x\text{-}\mathsf{clear}) = \{\mathsf{T}, \mathsf{F}\}$.

The only actions in Towers of Hanoi are movement actions $\mathsf{move}_{d,x,x'}$ that move a disk d into a position x from a position x', provided that both d and x are clear, and $d < x$. That is, $\mathsf{move}_{d,x',x} = \langle d\text{-}\mathsf{clear} = \mathsf{T}, x\text{-}\mathsf{clear} = \mathsf{T}, d\text{-}\mathsf{on} = x'; x\text{-}\mathsf{clear} = \mathsf{F}, x'\text{-}\mathsf{clear} = \mathsf{T}, d\text{-}\mathsf{on} = x\rangle$. The planning domain is the set of planning instances Π such that the init and goal are certain predetermined total states. Namely, in both states init and goal it holds $d_i\text{-}\mathsf{on} = d_{i+1}$ for all $i \in [1, \ldots, n-1]$, $d_1\text{-}\mathsf{clear} = \mathsf{T}$, $d_i\text{-}\mathsf{clear} = \mathsf{F}$ for all $i \in [2, n]$ and $p_2\text{-}\mathsf{clear} = \mathsf{T}$. They only differ in three variables: $\mathsf{init}(d_n\text{-}\mathsf{on}) = p_1$, $\mathsf{init}(p_1\text{-}\mathsf{clear}) = \mathsf{F}$ and $\mathsf{init}(p_3\text{-}\mathsf{clear}) = \mathsf{T}$, but $\mathsf{goal}(d_n\text{-}\mathsf{on}) = p_3$, $\mathsf{goal}(p_1\text{-}\mathsf{clear}) = \mathsf{T}$ and $\mathsf{goal}(p_3\text{-}\mathsf{clear}) = \mathsf{F}$. □

Definition 14. *Let Π be a planning domain instance of Towers of Hanoi. Let i be an integer $i \in [1, k]$. Let $x = \mathsf{init}(d_i\text{-on})$ and $x' \in \{p_2, p_3\}$. We define the action* $\mathsf{subtow\text{-}pos}_{i,x,x',x''} = \langle d_1\text{-clear} = \mathsf{T}, d_1\text{-on} = d_2, \ldots, d_{i-1}\text{-on} = d_i, d_i\text{-on} = x, x'\text{-clear} = \mathsf{T}, x''\text{-clear} = \mathsf{T}; d_i\text{-on} = x'', x\text{-clear} = \mathsf{T}, x''\text{-clear} = \mathsf{F}\rangle$*, that is, the action that moves the tower of height i from x to x'' using the clear position x' as an intermediate position.*

Theorem 15. *Any action* $\mathsf{subtow\text{-}pos}_{i,x,x',x''}$ *is $(H(\mathsf{init}, 4), A)$-derivable.*

References

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

A 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 $\mathsf{wrong}(s)$ to be the variables that are not in the goal state, that is, $\mathsf{wrong}(s) = \{v \in \mathsf{vars}(\mathsf{goal}) \mid s(v) \neq \mathsf{goal}(v)\}$.

Definition 16. *A state s' is an* improvement *of a state s if: for all $v \in V$, if $v \in \mathsf{vars}(\mathsf{goal})$ and $s(v) = \mathsf{goal}(v)$, then $s'(v) = \mathsf{goal}(v)$; and, there exists $u \in \mathsf{vars}(\mathsf{goal})$ such that $u \in \mathsf{wrong}(s)$ and $s'(u) = \mathsf{goal}(u)$. In this case, we say that such a variable u is a variable being improved. We say that a plan P* improves *the state s if $s[P]$ is a goal state, or $s[P]$ is an improvement of s.*

We remark that in the previous definition we permit P to be the empty plan ϵ; in particular, we have that the empty plan improves any goal state.

Definition 17. *(from [9]) A planning instance $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$ has* persistent Hamming width *k (for short,* PH width *k) if no plan exists solving Π, or for every reachable state s, there exists a plan (over A) improving s that stays within Hamming distance k of s.*

In this definition, when we say that a plan *stays within Hamming distance k* of a state s, we mean that when the plan is executed in s, all intermediate states encountered (as well as the final state) are within Hamming distance k of s.

Relative to a planning instance, we say that a state s dominates another state s' if $\{v \in V : s(v) \neq s'(v)\} \subseteq \mathsf{vars}(\mathsf{goal})$ and $\mathsf{wrong}(s) \subseteq \mathsf{wrong}(s')$; intuitively, s may differ from s' only in that it may have more variables set to their goal position.

Definition 18. *A planning instance $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$ has* macro persistent Hamming width *k (for short,* MPH width *k) if no plan solving Π 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 straightforward to verify 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 19. *Let $\mathcal{C}$ be a set of planning instances having MPH width k. The plan generation problem for $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$ belonging to $\mathcal{C}$ 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 $|V|$ and d denotes the maximum size of a domain.*

```
solve_mph((V, init, goal, A), k) {
        Q := empty plan;
        M := empty set of macros;
        s := init;
        while  s not a goal state  do {
                (G, M') := compute_macros(H(s,k), A);
                append M' to M;
                if  an improvement s' of s is reachable from s in G  then s := s';
                else  {print "?"; halt;}
                append l(s, s') to Q;
        }
        print M, Q;
}
```

We remark that solve_mph can really be viewed as an extension of an algorithm for persistent Hamming (PH) width; one essentially obtains an algorithm for PH width from solve_mph by replacing the call to compute_macros with a command that simply sets G to be the directed graph with vertex set $H(s,k)$ and an edge (s_1,s_2) present if there is an action a in A such that $s_1[a]=s_2$.

Theorem 20. *All instances of the Blocksworld-arm domain have MPH-width* 8*.*

According to Theorem 13, at any state s we may consider our set of applicable actions enriched by these new macro-actions. We now show how we can use them 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 $\mathsf{goal}(b\text{-on}) = b''$, then apply action $\mathsf{subtow\text{-}table}_{P_>(b''),b''}$ (if $s(b''\text{-clear}) = \mathsf{F}$), and $\mathsf{subtow\text{-}block}_{P_\geq(b),b',b''}$. 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'-clear change from F to T. We may try to solve this by placing something else 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, which allows us to prove Theorem 20.

Lemma 21. *Let Π be an instance of the Blocksworld-arm domain, and let s be a reachable state of Π such that $s(\mathsf{arm}) = \mathsf{empty}$. If a block b is such that $s(b\text{-clear}) = \mathsf{T}$ but $\mathsf{goal}(b\text{-clear}) = \mathsf{F}$, then there is a $(H(s,6),A)$-derivable action that improves the variable b-clear in s.*

With respect to Towers of Hanoi, it is clear that any instance can be solved by a single application of the $H(\mathsf{init},4)$-derivable action $\mathsf{subtow\text{-}pos}_{n,p_1,p_2,p_3}$. This is enough to ensure MPH-width 4, since no state dominates the initial state init other than goal.

Theorem 22. *All instances of the Towers of Hanoi domain have MPH-width* 4*.*

B Proofs

Proof of Lemma 10. Let $\Sigma = \sigma_1,\dots,\sigma_n$ be an A-CM-program over $H(s,k)$ obtained by compute_macros, and let G be the resulting action graph. 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 $(s,s') \in E(G)$ and $l_{G_i}(s,s') = l_G(s,s')$.

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 G. Since G 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 G at $l_G(s,s[a])$. By condition-minimality of $(s,a,s[a])$, we have that $a = l_G(s,s[a])$.

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 G. Since G is a fixed point over all apply and transitive commands, the action $\mathsf{combine}(a,a')$ or one that is better must appear in G at $l_G(s,s')$. By condition-minimality of $(s,\mathsf{combine}(a,a'),s')$, we have that $\mathsf{combine}(a,a') = l_G(s,s')$. □

Proof of Theorem 13. 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 $a = \mathsf{subtow\text{-}block}_{P,b,b'}$; the remaining actions admit similar proofs that only require Hamming distance 4.

To prove condition-minimality of a, consider a combination $C = (a_1, \ldots, a_t)$ of primitive actions from A such that $s[C] = s[a]$. We must show that the actions $\mathsf{unstack}_{b_1,b_2}, \ldots, \mathsf{unstack}_{b_k,b}, \mathsf{stack}_{b_k,b'}$ appear in C in the given relative order, and that no matter what are the remaining actions of C, this already implies that $\mathsf{pre}(a) \subseteq \mathsf{pre}(C)$ and $\mathsf{post}(a) \subseteq \mathsf{post}(C)$. We remark that the proof is not straightforward, since $\mathsf{pre}(C)$ and $\mathsf{post}(C)$ 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 $\mathsf{subtow\text{-}block}_{P,b,b'}$ is $(H(s, 5), A)$-derivable. When $|P| = 1$, we derive $\mathsf{subtow\text{-}block}_{P,b,b'}$ by combining actions $a_1 = \mathsf{unstack}_{b_1,b}$ and $a_2 = \mathsf{stack}_{b_1,b'}$. The states $s[a_1]$ and $s[a_1, a_2]$ differ from s respectively on 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 $a_1 = \mathsf{subtow\text{-}table}_{P',b_k}$, $a_2 = \mathsf{unstack}_{b_k,b}$, $a_3 = \mathsf{stack}_{b_k,b'}$ and $a_4 = \mathsf{tow\text{-}block}_{P',b_k}$. 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-clear, b_{k-1}-on, b_k-clear, b_k-on and b'-clear. □

Proof of Theorem 15. We prove this by induction on i, the height of the subtower. The proof makes use of the classical recursive solution to Towers of Hanoi: the action $\mathsf{subtow\text{-}pos}_{i,x,x',x''}$ is the combination of actions $\mathsf{subtow\text{-}pos}_{i-1,x,x'',x'}$, $\mathsf{move}_{d_i,x,x''}$ and $\mathsf{subtow\text{-}pos}_{i-1,x',x,d_i}$.

We show that these actions are derivable from init within Hamming distance 4. This is trivial to show for the case $i = 1$, which moves a single disk; by induction, we can assume that any action of the form $\mathsf{subtow\text{-}pos}_{i-1,w,w',w''}$ is also $H(\mathsf{init}, 4)$-derivable. To prove that $\mathsf{subtow\text{-}pos}_{i,x,x',x''}$ is $H(\mathsf{init}, 4)$-derivable, we consider a state s satisfying the pre-conditions of $\mathsf{subtow\text{-}pos}_{i,x,x',x''}$ as close as possible to the initial state init. Notice that this state s is not required to be reachable (it is not even required to be consistent!) since the compute_macros algorithm takes all states within the appropiate Hamming distance into consideration. If none of the positions x, x' and x'' is a peg, let the state s be the non-consistent state obtained from init by setting d_i-on $= x$, x'-clear $= \mathsf{T}$, x''-clear $= \mathsf{T}$. We just need to check that the sequence of $H(\mathsf{init}, 4)$-derivable actions $\mathsf{subtow\text{-}pos}_{i-1,x,x'',x'}$, $\mathsf{move}_{d_i,x,x''}$ and $\mathsf{subtow\text{-}pos}_{i-1,x',x,d_i}$ is applicable to the state s, and that the number of differing variables after the application of any of these three actions never exceeds 4. A similar argument works for the case where some of the positions x, x' or x'' is a peg. □

Proof of Theorem 19. Let $\Pi \in \mathcal{C}$ be a planning instance such that there exists a plan for $\Pi = (V, \mathsf{init}, \mathsf{goal}, A)$. 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 10, 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 $O(n^{3k+2}d^{3k}(a+(nd)^{2k}))$. □

Proof of Lemma 21. Clearly, $b = \mathsf{top}(P_1)$ for some tower P_1 of s. Let $P_2, \ldots, P_t$ 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 $\mathsf{goal}(\mathsf{bottom}(P_i)\text{-on}) \neq \mathsf{table}$, we say we are in Case 1. Otherwise, it holds that $t \leq t'$. In particular, there are t' blocks b' such that $\mathsf{goal}(b'\text{-clear}) = \mathsf{T}$ (block b not one of them), and t blocks $b' \neq b$ such that $s(b'\text{-clear}) = \mathsf{T}$ (block b being one of them). It follows that it exists a block b' such that $\mathsf{goal}(b'\text{-clear}) = \mathsf{T}$ but $s(b'\text{-clear}) = \mathsf{F}$. 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 $s(b'\text{-on}) \neq \mathsf{goal}(b'\text{-on})$.

Case 1. The tower P_i is wrongly placed in the table, so we are allowed to change the value of $\mathsf{bottom}(P_i)$-on without worry. If $i \neq 1$, then use $\mathsf{tow\text{-}block}_{P_i,b}$ to stack the tower P_i on top of 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 $s(b'\text{-on})$.

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 $b' = \mathsf{top}(()P_i')$. 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, $\mathsf{goal}(b'\text{-clear}) = \mathsf{T}$ but $s(b'\text{-clear}) = \mathsf{F}$, 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-clear use actions $\mathsf{subtow\text{-}table}_{P_>(b'),b'}$ and $\mathsf{tow\text{-}block}_{P_{\leq}(b'),b}$, that is, break the tower over block b' and swap the two parts.

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

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 $s(b'\text{-clear}) = \mathsf{F}$ but $\mathsf{goal}(b'\text{-clear}) = \mathsf{T}$. 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''. This procedure improves variables b-clear and b'-clear at the same time, but it needs width 6. If there is

a second block b'' in P_1 such that $\mathsf{goal}(b''\text{-}\mathsf{clear}) = \mathsf{T}$, then swap the sub-tower $P_>(b')$ with the pile between b' and b'', the block b'' not including. This procedure requires width 5.

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 $\mathsf{goal}(\mathsf{top}(P')\text{-}\mathsf{clear}) = \mathsf{T}$ and $\mathsf{goal}(\mathsf{bottom}(P')\text{-}\mathsf{on}) = \mathsf{table}$, cannot be a pile in goal. Hence there is a badly placed block below b'. This situation requires width 5.

Case 3. There is a block b' such that $s(b'\text{-}\mathsf{clear}) = \mathsf{F}$ but $\mathsf{goal}(b'\text{-}\mathsf{clear}) = \mathsf{T}$, 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. □

Proof of Theorem 20. 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 show how to improve one of the variables of s within Hamming distance 8. We first show how to improve the variable arm; for the remaining cases, we safely assume that $s(\mathsf{arm}) = \mathsf{goal}(\mathsf{arm}) = \mathsf{empty}$.

Improving arm. We assume $s(\mathsf{arm}) \neq \mathsf{goal}(\mathsf{arm}) = \mathsf{empty}$. We use the action $\mathsf{putdown}_b$ to place the block $b = s(\mathsf{arm})$ on the table. This improves the variable arm and may also improve b-on. Note, however, that $s(b\text{-}\mathsf{clear}) = \mathsf{F}$ (according to the given formulation, the block is not clear when the arm holds it); if $\mathsf{goal}(b\text{-}\mathsf{clear}) = \mathsf{F}$, then $\mathsf{putdown}_b$ is affecting a variable which had already the right value. In that case, we use Lemma 21 to improve b-clear. The action $\mathsf{putdown}_b$ changes the value of 3 variables, including b-clear; the action of Lemma 21 changes the value of at most 6 variables, including the common variable b-clear. Hence the combination of both is $H(s, 8)$-derivable.

Improving b**-**on. Assume $s(b\text{-}\mathsf{on}) = \mathsf{table}, \mathsf{goal}(b\text{-}\mathsf{on}) = b'$. If $s(b'\text{-}\mathsf{clear}) = \mathsf{F}$, then move the sub-tower $P_>(b')$ onto the table. (This changes the variable b''-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 $P_\geq(b)$ onto b'.

Now, assume $s(b\text{-}\mathsf{on}) = b'', \mathsf{goal}(b\text{-}\mathsf{on}) = b'$. If $s(b'\text{-}\mathsf{clear}) = \mathsf{F}$ then we can swap piles $P_>(b'')$ and $P_>(b')$, which only changes the value of two variables, neither of which was in the goal state. Otherwise, we stack $P_>(b'')$ on top of b', but then b''-clear becomes true. This is a problem if $\mathsf{goal}(b''\text{-}\mathsf{clear}) = \mathsf{F}$, so we may need to apply Lemma 21 at the current state. As in the previous case, the combination of both actions is $H(s, 8)$-derivable.

Finally, we address the case $s(b\text{-}\mathsf{on}) = b'', \mathsf{goal}(b\text{-}\mathsf{on}) = \mathsf{table}$. Move $P_\geq(b)$ onto the table. As in the previous case, we can apply Lemma 21 to the current state if $\mathsf{goal}(b''\text{-}\mathsf{clear}) = \mathsf{F}$. The combination is $H(s, 8)$-derivable.

Improving b**-**clear. Assume $s(b\text{-}\mathsf{clear}) = \mathsf{F}, \mathsf{goal}(b\text{-}\mathsf{clear}) = \mathsf{T}$. This implies that the variable b'-on, where b' is the block such that $s(b'\text{-}\mathsf{on}) = b$, is not in the goal state. Thus we can safely move the pile $P_>(b)$ onto the table to improve b-clear, which only takes width 4. To solve the case $s(b\text{-}\mathsf{clear}) = \mathsf{T}, \mathsf{goal}(b\text{-}\mathsf{clear}) = \mathsf{F}$ we just need to apply Lemma 21, which requires width 6. □