# Detecting and Exploiting Decomposability in Update Graphs

**Prasad Chalasani**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
chal+@cs.cmu.edu

**Oren Etzioni***
Department of Computer Science
and Engineering, FR-35
University of Washington
Seattle, WA 98195
etzioni@cs.washington.edu

**John Mount**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jmount+@cs.cmu.edu

## Abstract

Model-learning and problem-solving algorithms can only be integrated when the output of the former fulfills the expectations of the latter. Korf's Macro-Table (KMT) algorithm requires serial or total decomposability of the operators in its world model. The problem of detecting decomposability has been open since Korf introduced the KMT algorithm in 1983. We present a polynomial-time algorithm for detecting total decomposability in update graphs, an efficiently learnable model of external environments, and an exponential-time algorithm for detecting serial decomposability. A number of additional results are presented including a transformation of update-graph models to permutation group problems that can be handled by Furst, Hopcroft and Luks's (FHL) permutation-group algorithm. The results facilitate the integration of Schapire and Rivest's update-graph-learning algorithm with the KMT and FHL algorithms.

## 1 INTRODUCTION

An autonomous agent seeking to solve problems in an external environment faces two related challenges: inferring an internal model of the environment and using the model to efficiently solve problems in the environment. Whereas previous work has addressed these challenges in isolation ([RS87], [Sch88], [Kor85], [FHL80]), this paper discusses solving these in concert.

In [RS87], Rivest and Schapire present a polynomial-time algorithm that reliably learns an internal model for a class of finite automaton environments, called permutation environments. They show that their internal model, called the *update graph*, is a natural encoding of the behavior of finite automaton environments, and could be as small as logarithmic in the size of the state graph of the automaton. However, they do not consider the possibility of using the update graph as a state representation for solving problems.

Two candidates for use in problem solving using update graphs are: Korf's [Kor85] Macro-Table (KMT) algorithm, and Furst, Hopcroft, and Luks's (FHL) [FHL80] permutation-group algorithm. However, an update graph representation may not necessarily satisfy the assumptions these algorithms make about their input representation. The KMT algorithm can only be applied to a problem representation which has serially/totally decomposable actions. The FHL algorithm only solves permutation group problems. So an autonomous agent that uses these problem-solving algorithms must be able to efficiently perform two tasks:
(b) *detect* whether the learned update graph satisfies the input requirements of the algorithms, or
(a) *Transform* the update graph into a representation which does satisfy those properties.
Our main aim in this paper is to show polynomial-time algorithms that can be used to accomplish these tasks under different conditions.

## 2 OVERVIEW

An update graph is a graph model of a finite automaton environment, whose node-values (called the *labeling*) uniquely characterize the state of a finite automaton, so an update graph can be used as a state repre-

*This paper describes research done primarily at Carnegie Mellon University's School of Computer Science. The second author was also supported by an AT&T Bell Labs Ph.D. Scholarship.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

sentation for problem-solving. A problem to be solved using an update graph can be posed as a goal-labeling. In a permutation environment, executing action $b$ has the effect of permuting the node-values according to a permutation $\pi_b$. Thus, problem-solving with the update graph of a permutation environment can be viewed abstractly as the following problem, which we call the *general permutation problem*, GPP: *Given a set of permutations* $\Pi$ *of degree* $n$*, two* $n$*-dimensional state vectors* $V_o$ *(initial vector) and* $V_g$ *(goal vector), efficiently find a short sequence of permutations from* $\Pi$ *that transform* $V_o$ *into* $V_g$*.*

Two algorithms exist that solve variants of the GPP:
(a) Furst, Hopcroft, Luks's [FHL80] (FHL) algorithm solves a restricted version of GPP in which the values of the components of $V_o$ are distinct, and
(b) Korf's Macro Table (KMT) algorithm solves problems in which actions (not necessarily permutations) are serially/totally decomposable.

Our main results are polynomial-time algorithms that analyze the structure of a GPP, in order to:
(a) *detect* reliably whether the permutations $\Pi$ are totally decomposable with respect to $V_o$, (which implies they are serially decomposable) so that the KMT algorithm can be applied,
(b) *reduce* the GPP to a new GPP of smaller size that may satisfy the requirements of the FHL or KMT algorithms, and
(c) *transform* the GPP to another GPP so that either the FHL or the KMT algorithms can be applied.

## 3 DEFINITIONS

The "world" in which an intelligent agent finds itself can have different degrees of structure. The language of group theory provides a succinct way of describing regularities in the environment.

A **group** [Mac68] is defined by a set $G$ and a binary operation $\circ$ such that (a) $G$ is closed under $\circ$, (b) the operation $\circ$ is associative, (c) every element of $G$ has a inverse, and (d) there exists an identity element $e$ in $G$. If there is a subset $X$ of $G$ such that all elements of $G$ can be obtained by repeatedly applying the $\circ$ operator to elements of $X$, then $X$ is said to generate $G$, and the elements of $X$ are called *generators* of $G$.

By a **permutation** $\pi$ of an arbitrary set $X$ we shall mean a bijection from $X$ to itself. Any given collection of permutations over the set $X$ generates a permutation group under the operation of composition of permutations, and the permutations in the group are said to have *degree* $n = |X|$. We use the notation $\langle\Pi\rangle$ to denote the group generated by the permutations $\Pi$. Note that the collection of *all* permutations of $X$ forms a group $S_X$ under composition of functions. In particular, if $X = \{1, 2, ...n\}$, we denote the group of all permutations over $X$ by $S_n$, and call $S_n$ the **symmetric group** of degree $n$. If $g$ is a permutation of degree $n$, then the result of "applying the permutation $g$ to an $n$-dimensional vector $V$" is a new vector $V' = g(V)$ where $g(i) = j$ implies $V'[j] = V[i]$, for $i, j = 1, 2, ...n$.

A *finite automaton environment* (FAE) $\mathcal{E}$ is characterized by a tuple $(Q, B, P, \delta, \gamma)$, where $Q$ is a finite set of states, $P$ is a finite set of predicates, $B$ is a finite set of basic actions, $\delta : Q \times B \rightarrow Q$ is the state transition function, $\gamma : Q \times P \rightarrow \Sigma$ is the sensor function that associates predicate-values with states, $\Sigma$ being the (finite) set of possible predicate values. We use the notation $qa$ to denote the (unique) final state reached after applying an action-sequence $a \in B^*$ from state $q$. Note that in a finite automaton environment, for all states $q$, and for all $b \in B$, $qb$ must be well defined, and unique.

An environment with more structure is the **Permutation Environment**(PE), [RS87], which is a finite automaton environment where the transition function of each action $b$: $\delta(., b) : Q \rightarrow Q$ is a bijection on $Q$. This definition implies that for each state of the automaton, there is exactly one incoming and one outgoing $b$-arrow, for every action $b \in B$. Thus for a PE, all the actions $B$ are *invertible.*

Rivest and Schapire [RS87] devise an efficient inference algorithm for permutation environments.

**Examples:**
An example of an FAE is the 8-puzzle environment, $\mathcal{E}_8$, whose action-set $\{L, R, U, D\}$ corresponds to the four directions (left, right, up, down) in which the blank can be moved. The predicates are $p_i$, $i = 1, 2, ...8$ whose value is the position of tile $i$. The actions of $\mathcal{E}_8$ are not very regular. For example, the action $L$ has "different kinds of effects" in different states. When the blank is in the leftmost position, this operator has no effect. (In order to model the 8-puzzle as a finite automaton, we *must* define a next state for every action, so when the action is "not applicable" in the usual sense, we say that the next-state is the current state itself.) The 8-puzzle environment $\mathcal{E}_8$ is clearly not a PE because, for example, two different states can lead to the same state through the action $L$: A state $s$ whose bottom row is [$B$ 1 2], and a state which is the same as $s$ except that its bottom row is [1 $B$ 2]. (B represents the blank position). So, it can be pointed out that while the 8-puzzle operators are "invertible", they are only invertible in the colloquial sense. They are not invertible in the corresponding finite automaton.

The Rubik's Cube environment $\mathcal{E}_{Rub1}$ with the usual operators, and predicates with values that each represent the position of a given square on the cube is a $PE$. This is because the effects of the operators are uniform: any given state is reachable from exactly one other state, through a given operator. The Rubik's Cube environment $\mathcal{E}_{Rub2}$ with the same operators, and predicates whose values each represent which square is

in a fixed position is also a $PE$ by the same reasoning. ■

We now turn to another dimension of structure and regularity in an environment.

### 3.1 SERIAL & TOTAL DECOMPOSABILITY

Consider a *state vector representation* for a FAE where each state is associated with an assignment of values to $n$ state variables. A given assignment to the $n$ variables is called a *state vector*. Note that different states need *not* have distinct state vectors. Now, suppose the $n$ variables are ordered $v_1, v_2, ...v_n$. When a basic action $b \in B$ is applied to the automaton, the variables acquire new (or possibly the same) values. Let $c_i$ denote the new value acquired by $v_i$. Then the action $b$ is said to be **serially decomposable** w.r.t. the ordering of those variables if each $c_i$ is expressible as a function only of the variables $v_j$ such that $j \leq i$. If each $c_i$ is expressible as a function of only $v_i$, then the action $b$ is said to be **totally decomposable**.

**Examples:** The variables from the environment $\mathcal{E}_{Rub1}$ are totally decomposable because the next position of a face on the cube is a function of its current position and the operator applied. However the variables from the environment $\mathcal{E}_{Rub2}$ are not totally decomposable because on the Rubik's cube the next face to occupy a position is not a function of the face currently in the position and the operator applied. It is not hard to see the variables of $\mathcal{E}_{Rub2}$ are not serially decomposable for any ordering. ■

### 3.2 UPDATE GRAPHS

In [RS87], Rivest and Schapire show a probabilistic polynomial-time algorithm to infer an internal model of a permutation environment, given the ability to apply actions to the environment and sense predicates from it. Their internal model is called the update graph. We describe briefly the structure of this model.

An update graph can be defined for any FAE $\mathcal{E} = (Q, B, P, \delta, \gamma)$. Every node of an update graph has a value in a given state, and the combination of node values of an update graph uniquely characterizes the state of the automaton. For each predicate $p_i$ there is a node $v_i$ that gives the value of the predicate in any state. Also, for every node and every action $b \in B$, there is exactly one directed edge labeled $b$ going into that node. In the case of a permutation environment(PE), there is exactly one outgoing edge labeled $b$ from any node. *When an action $b$ is applied to the automaton, the new values of the nodes are given by simply moving the node values in the direction of the $b$-edges:* the new value of a node $v_i$ is simply the current value of the unique node $v_j$ which has a $b$-edge going out from it to $v_i$. Thus, given the initial labeling (node-values) of the update graph, it is possible to determine the labeling (and hence the values of the predicates P) after any sequence of actions $a \in B^*$. In the case of a PE, the update graph is completely characterized by:
(a) an $n$-dimensional vector $V_o$ that gives the initial labeling of the graph, $n$ being the number of nodes of the update graph.
(b) a set $\Pi$ of permutations of degree $n$ that correspond to the actions.
We will henceforth denote an update graph by the pair $(\Pi, V_o)$. Thus, for a PE, the update graph provides a state vector representation which has a particularly simple behavior: the values of the state variables are rearranged when actions are performed. Our algorithms take advantage of this simple behavior.

## 4 THE GENERAL PERMUTATION PROBLEM (GPP)

We can now state the general problem that we are concerned with in this paper. Once the agent infers an update graph for the permutation environment, it is presented with problems of the following form: "Achieve a particular set of values of the predicates $P$." This is equivalent to specifying goal values on a subset of the nodes of the update graph. We consider, the probably more natural, version of this problem where the goal specifies the values of *all* nodes of the update graph. Thus the problem is, "given an update graph $(\Pi, V_o)$, achieve a labeling $V_g$, using a sequence of permutations (actions) from $\Pi$".

If $G = \langle\Pi\rangle$ is the group generated by $\Pi$, then we use the notation $G(V_o)$ to denote the set of possible $n$-dimensional vectors that are reachable from $V_o$ by permutations of $\Pi$.

Any problem on an update graph of a PE can be viewed abstractly as the following *General Permutation Problem*(GPP) of degree $n$, denoted by $(\Pi, V_o, V_g)$:

> Given a set of permutations $\Pi$, two $n$-dimensional vectors $V_o$, and $V_g$, efficiently determine if $V_g \in G(V_o)$ and if so find a permutation $h$ in the group $G = \langle\Pi\rangle$ such that $h(V_o) = V_g$, and $h$ has a short expansion in terms of $\Pi$.

We will take "efficient" and "short" to mean that the solution time and length do not grow too fast with the dimension $n$. Unless otherwise stated we will implicitly assume that the degree of a GPP is $n$. We will often find it convenient to associate a variable (called a *state variable*) with each component of the $n$-dimensional vectors, called *state vectors*. Then the permutations can be said to permute the values of the $n$ state variables, $v_1, v_2..., v_n$. Thus, for all $i$, if $v_i'$ denotes the new value of $v_i$ after applying a permutation $g \in \Pi$, then

$v'_i = v_j$ such that $g(j) = i$.

We will be most interested in using the information from $\Pi$ and $V_o$ to derive information that allows us to quickly solve any GPP of the form $(\Pi, V_o, Y)$. This is because by the transitivity of actions this would allow us to quickly realize a solution of any GPP problem of the form $(\Pi, X, Y)$ (where $X, Y \in G(V_o)$) which is exactly what is needed to plan the actions of an automata. To avoid introducing additional terminology we will use the term GPP to refer to both a triple $(\Pi, V_o, V_g)$ and a pair $(\Pi, V_o)$. The latter style of problem will indicate that we are trying to solve $(\Pi, V_o, Y)$ for arbitrary $Y$. We will use this notion of a GPP whenever the goal vector is superfluous. Anything shown for a $(\Pi, V_o)$ GPP will be true for $(\Pi, V_o, V_g)$ problems.

### 4.1 HARDNESS OF GPP

It is natural to wonder how "hard" GPP is. Briefly, we can show (sketches of the proofs are in the appendix):
(a) Determining if $V_g \in G(V_o)$ is no harder than the Setwise Stabilizer Problem[1] (but we have not shown it to be necessarily as hard as this problem).[2]
(b) Finding $h \in G$ such that $h(V_o) = V_g$ is polynomial time equivalent (with respect to randomized Turing reductions) to the Setwise Stabilizer Problem.
And it is known that determining the length of the shortest expansion of $h$ in terms of $\Pi$ is P-space complete [FSS+89]. It must be emphasized that these are worst case results and may not correspond to the typically observed behavior. Furthermore, these complexity results are all relative to the problem size, $n$, but it may be to measure complexity in terms of the length of the shortest expansion of $h$ in terms of $\Pi$.

### 4.2 GPP AND PERMUTATION GROUP ALGORITHMS

If the values of the state variables in $V_o$ are all distinct, then all vectors in $G(V_o)$ will have distinct state variables. Thus given a pair of vectors $(V_o, V_g)$, where $V_o$ contains distinct state variables, the permutation (if any) $\pi_g$ that maps $V_o$ to $V_g$ is uniquely defined and can be found simply by examining the two vectors. To solve the problem, it then only remains to determine a short expansion (if any) of $\pi_g$ in terms of permutations from $\Pi$. We refer to this special case of GPP, where $V_o$ has distinct components, as a *Standard Permutation Problem*, SPP.

[1]The Setwise Stabilizer Problem is known to be at least as hard as Graph Isomorphism but is not known to be NP-Complete[Hof82]

[2]Determining if a partially specified labeling $\tilde{V_g}$ can be extended into a labeling $V_g$ such that $V_g \in G(V_o)$ is easily seen to be NP-complete (reduction from the Clique Problem[GJ79]).

This is a classic problem in computational group theory. In [FHL80], Furst, Hopcroft and Luks describe a polynomial-time (in $n$) algorithm (the FHL-algorithm) to construct a table for a permutation group $G \subseteq S_n$ presented in terms of its generators $\Pi$. This table can then be used to find an expansion in terms of $\Pi$ for any permutation $g \in G$. The *length* of this expansion may be exponentially long in $n$, but no results exist that guarantee that the expansion length is polynomialy related to the optimal expansion. It must be noted that certain generator sets $\Pi$ exist that generate permutations $g$ whose optimal expansions are exponentially long in $n$. Clearly, more work needs to be done in this area.

In general, in a GPP, the values of the components of $V_o$ need not be distinct, and instead of a unique permutation $\pi_g$ that maps $V_o$ to $V_g$ there may be a set $H \subseteq S_n$ of permutations that do the mapping. However, only the permutations in $G \cap H$ are *feasible*, i.e., expandable in terms of $\Pi$. The set $H$ may be exponentially large (in $n$), so a naive approach that checks each $h \in H$ for feasibility is ruled out. Thus one must first find feasible permutations $G \cap H$, before finding a feasible permutation with a short expansion. In section 6 we will show polynomial time algorithms that attempt to transform the original GPP to a smaller GPP which may turn out to be an SPP, so that the existing algorithms for SPP may be used.

### 4.3 GPP AND SERIAL/TOTAL DECOMPOSABILITY

Consider a variant of the GPP $(\Pi, V_o)$ where the actions need not be permutations, so that the set of bijections $\Pi$ is replaced by a finite set $\mathcal{F}$ of functions that map $n$-dimensional vectors to $n$-dimensional vectors. Suppose that the functions in $\mathcal{F}$ are, as above, serially decomposable. Such problems $(\mathcal{F}, V_o)$ are called *serially decomposable problems* (SDP). By definition, not every SDP is a GPP. We have shown, in a GPP, the permutations need not be serially decomposable, so not every GPP is an SDP.

In [Kor85], Korf shows an algorithm that constructs a *macro table* (similar to the FHL table) that can be used to solve SDPs of the form $(\mathcal{F}, V_o)$. The table-construction roughly takes the same amount of time as solving one $(\mathcal{F}, V_o, V_g)$ problem without the macro table (using conventional search techniques). This could take time exponential in $n$. However, once constructed, this table can be used to quickly generate short solutions for arbitrary $V_o$. In fact, the solutions are guaranteed to be no longer than $n$ times the optimal solution length. And such solutions can be found in time $n$ times the length of the optimal solution. Korf's Macro Table (KMT) algorithm assumes that the actions are serially decomposable w.r.t. the state variables, and does not attempt to detect whether this property holds, nor does it try to change the represen-

tation so that it does. Both these questions must be addressed when trying to apply the KMT algorithm to a GPP. We consider these questions in sections 5 and 7.

The Korf Macro Table has $m$ rows and $n$ columns, where each column corresponds to a state variable, and the rows correspond to the possible values of the variables. Thus the size of the macro table is proportional to the product of the number of variables, and the total number of possible values of the variables. This fact will be relevant in a future section.

In a GPP, the effect of any action is simply a permutation of the state vector. As we show in section 5, we can take advantage of this fact, to devise a probabilistic polynomial-time algorithm to determine if the permutations $\Pi$ are totally decomposable. If they are, since total decomposability implies serial decomposability, the KMT algorithm can be applied to the GPP. In section 6 it is demonstrated how some GPP problems can be reduced in size. In section 7 we will show that every SPP can be easily transformed to a SDP.

# 5 DETECTING SERIAL/TOTAL DECOMPOSABILITY

Given a GPP $(\Pi, V_o)$, if the permutations $\Pi$ are serially decomposable w.r.t some ordering on the state variables (i.e., vector-components), the KMT algorithm can be applied to solve problems. However, how can the agent find whether such an ordering exists, and find one if it does?

We show that this problem is reducible to determining whether an arbitrary subset of the state variables is *self-dependent*, in the sense defined below. For any permutation $g \in \langle\Pi\rangle$, and any subset $T$ of the state variables, we use the notation $g(T)$ to denote the (list of) values of the variables $T$ after applying permutation $g$ to the initial state $V_o$. Also $gh$ denotes the product of two permutations (i.e. the result of applying $g$, then $h$).

**Definition 1 (Self-Dependency)** A set of state variables $T$ is self-dependent with respect to permutation $h \in \Pi$ (and labeling $V_o$) iff the values of the variables $T$ after applying $h$ are a function only of the values of the variables $T$ before applying the action $h$. That is, there is a function $f$ such that

$$\forall g \in \langle\Pi\rangle : g(h(T)) = f(g(T))$$

If a set of variables $T$ is *not* self-dependent, then what evidence of this exists in the behavior of the environment? This is answered by the following lemma.

**Lemma 1 (Witnesses)** A set of state variables $T$ is *not* self-dependent w.r.t. permutation $h \in \Pi$ iff there are two permutations $g$ and $g'$ in $\langle\Pi\rangle$ such that

$$g(T) = g'(T) \text{ and } gh(T) \neq g'h(T)$$

We call the two permutations $g$ and $g'$ *witnesses* to the non-self dependency of $T$ w.r.t. permutation $h$.

We can show the following connection between serial decomposability and self-dependency:

**Theorem 1** *An permutation $h \in \Pi$ is serially decomposable w.r.t. a set of variables $T$, iff $T$ can be partitioned into disjoint subsets $\sigma_1, \sigma_2, ...\sigma_k$ such that each variable in $\sigma_1$ is self-dependent w.r.t. $h$, and for $i = 2, 3, ...k$, each variable $x$ in $\sigma_i$ has the property that the set of variables*

$$\sigma_1 \cup \sigma_2 ... \cup \sigma_{i-1} \cup \{x\}$$

*is self-dependent w.r.t. $h$.*

**Sketch of Proof:**
If $h$ is serially decomposable w.r.t. an ordering $< v_1, v_2, ...v_m >$, then the above conditions are satisfied by the partition $\sigma_i = \{v_i\}$. This is easy to show from the definition of serial decomposability. Conversely, if such a partition exists, then it is easy to show that $h$ is serially decomposable when the variables of $T$ are arranged in increasing order of the index $i$ of the partition $\sigma_i$ to which they belong.□

This theorem suggests the following algorithm for determining serial decomposability of an action $h \in \Pi$ w.r.t. the $n$ state variables. **self**$(\Pi, V_o, T, h)$ is a predicate that tests self-dependency of a set of variables $T$ w.r.t. action $h$. Let $X$ be the set of the $n$ state variables.

**procedure Serial** $(\Pi, V_o, h)$
$i \leftarrow 0$;
$\sigma_0 \leftarrow \phi$;
left $\leftarrow \phi$;
**repeat**
  $i \leftarrow i + 1$;
  $\sigma_i \leftarrow \{x \in (X \setminus \text{left}) |$ **self**$(\Pi, V_o, \text{left} \cup \{x\}, h)\}$;
  left $\leftarrow$ left $\cup \sigma_i$;
**until** left $= X$;
**endproc**

Central to the above algorithm is the procedure **self()**, and we show below that this could take time exponential in $|T|$. It must be noted that constructing a KMT by existing weak methods can take time exponential in the diameter of the problem space[3] so unless we dealing with a family of problem spaces where the diameter does not increase as fast as $|T|$ we see that the running time **Serial()** and **self()** is dominated by the time to actually construct a KMT once we have found a set of serial decomposable variables.

By the witness lemma above, to check if a set of variables $T$ is *not* self-dependent, it is sufficient to generate the two witness permutations $g$ and $g'$ in $G = \langle\Pi\rangle$,

[3] the minimum number of operators needed to transform any state vector in the problem space to any other

from the generators $\Pi$. Suppose that the agent generates permutations of $G$ by applying a random sequence of actions from $\Pi$. Then how likely is it that two such witness permutations will be generated, when $T$ is *not* self-dependent?

To answer this, we use arguments similar to those given in [RS87]. Note that when action (permutation) $b$ is applied to a state vector, the new value of a variable $v_i$ is the current value of the variable $v_j$ where $j = b^{-1}(i)$. As a mnemonic, we use $b^{-1}(v_i)$ to denote the variable $v_j$. Similarly, $b^{-1}(T)$ denotes the *set* of variables whose current values become the future values of the variables $T$ after action $b$. Then the witness property above can be re-stated as: If $T$ is not self dependent w.r.t. action $b$, then there are two permutations $g, g' \in G$ such that $g(T) = g'(T)$, and $g(b^{-1}(T)) \neq g'(b^{-1}(T))$.

Now consider the permutations of $G = \langle\Pi\rangle$ that map the variables of $T$ and $b^{-1}(T)$ to themselves, i.e., the subgroup $H_T$ that *stabilizes* $T \cup b^{-1}(T)$. Consider the graph $\mathcal{H}_T$ of left-cosets $gH_T$ of $H_T$ in $G$. Then, if $T$ is not self-dependent, there are two left cosets $gH_T$ and $g'H_T$ such that the above witness property is satisfied for $g, g'$. Further, applying a random sequence of actions is equivalent to taking a random walk on the coset graph $\mathcal{H}_T$, starting from $H_T$. It can be shown that to ensure that two particular nodes are visited with probability at least $1/2$, it is sufficient to take random steps of length $O(me)$ where $m$ is the number of nodes in $\mathcal{H}_T$, and $e$ is the number of edges of $\mathcal{H}_T$. So, to detect self dependency of a set of variables $T$ with a probability of error at most $\epsilon$, the agent need only take a random walk on $\mathcal{H}_T$ whose length is polynomial in $m$,$e$, and $1/\epsilon$. After each action, it must check if the witness property holds, and this can done in time polynomial in $n$, the size of the state vectors.

How large can the coset graph $\mathcal{H}_T$ be? The number of nodes $m \leq \binom{n}{|T|}$, since that is the number of different ways of mapping a given set of $|T|$ nodes to $|T|$ other nodes. Also, $e \leq m|\Pi|$. Thus the number of nodes of the coset graph is $O(n^{|T|})$, exponential in $|T|$. Since $T$ could be $O(n)$ in general, this means the serial decomposability detection algorithm above can take time exponential in $n$.

However, if the algorithm always checks for the self-dependency of a *constant-sized* set $T$, then the algorithm runs in time polynomial in $n$. We can show easily that a set of variables $T$ is totally decomposable iff every singleton subset of $T$ is self-dependent. Thus a procedure **Total()** can be constructed which simply tests for the self-dependency of each state variable. If an environment passes this test, the agent can apply the KMT algorithm to solve problems.

We have thus shown:

**Theorem 2** *Given a GPP, $(\Pi, V_o)$, of degree $n$, and a reliability $\epsilon$, the procedure* **Total()** *runs in time polynomial in ($|\Pi|$,$n$, $1/\epsilon$) and detects, with error probability $< \epsilon$, whether the permutations $\Pi$ are totally decomposable.*

It is not hard to see that simple adaptations of **Serial** and **Total** can detect decomposability (serial/total) w.r.t small *clusters* of variables rather than individual variables.

# 6 REDUCING TO A SMALLER PROBLEM

Because **self()** can run in time exponential in $|T|$ it is important to reduce the size of $|T|$ whenever possible. We now develop a polynomial-time algorithm that attempts to transform a GPP $P = (\Pi, V_o)$ to an equivalent GPP $P' = (\Pi', V_o')$ of smaller degree. The basic idea is to detect whether the permutations $\Pi$ permute *blocks* of variables in a certain structured way: if so, then the GPP can be re-cast in terms of block-level permutations, which necessarily have a degree smaller than $n$.

Two GPPs $P, P'$ are equivalent if:
(a) there is a one-one correspondence $\phi : \Pi \rightarrow \Pi'$,
(b) there is a one-one correspondence $\gamma : G(V_o) \rightarrow G'(V_o')$, where $G = \langle\Pi\rangle, G' = \langle\Pi\rangle$, and
(c) For any $V, W \in G(V_o)$ and $\pi \in \Pi$, $\pi(V) = W \Rightarrow \phi(\pi)(\gamma(V)) = \phi(W)$.

Our idea is an extension of the notion of a *closed partition* [Koh78]. Consider a partition $\tau$, and a total order $\prec$ on the index-set $I_n = \{1, 2, ...n\}$. We write $\tau(i, j)$ when $i, j$ belong to the same block of $\tau$. Then, w.r.t. a GPP $(\Pi, V_o)$, we say that $(\tau, \prec)$ is a closed, faithful (CF) partition, if

$$\forall i, j : \forall g \in \Pi : (\tau(i, j) \ \wedge \ i \prec j) \ \Rightarrow \ \begin{array}{l} \tau(g(i), g(j)) \wedge \\ g(i) \prec g(j) \end{array}$$

The ordering $\prec$ is called a *faithful ordering*. Note that the blocks of a CF partition must all have the same size. Thus, if the GPP has a CF-partition $(\tau, \prec)$, the effect of any permutation in $\Pi$ is to map $\prec$-ordered blocks of $\tau$ to $\prec$-ordered blocks of $\tau$. Further, for any vector in $G(V_o)$ we can then associate a value-pattern with each block, which is obtained by laying out the elements of each block in the order $\prec$.

Suppose the blocks are numbered $1, 2, ...m$, and the value patterns are numbered $1, 2, ..., k$. Since the partition is faithful, the value patterns in any vector in $G(V_o)$ will be the same, except that their positions will have changed. So for each $g \in \Pi$, we can define a new block-level permutation $g'$. And for any vector $V \in G(V_o)$, we can define a corresponding $m$-dimensional vector $V'$ whose $i$th component gives the value-pattern number of the $i$th block of $\tau$. Thus we

can reduce the GPP $(\Pi, V_o)$ of degree $n$ to an equivalent, smaller GPP $(\Pi', V_o')$ of degree $m$. We refer to this reduction as a *CF-reduction*.

How do we find a CF-partition? We can do this with a slightly modified version of Kohavi's [Koh78] algorithm, which finds *all* closed partitions of a state graph of an automaton in time polynomial in the size of the state graph. By adding a check to determine faithfulness, this algorithm can be used to find all CF partitions in time polynomial in $n$, the degree of the GPP. We can choose which CF-partition to use, so that the resulting new GPP has useful properties. For instance, if there is a CF-partition with distinct value-patterns in each block, then we should prefer to use that partition over one which does not have this property. Then the new GPP will be an SPP/SDP, so that both the Korf Macro Table algorithm and the FHL algorithm can be applied.

**Example:**
These ideas can be illustrated by the Top-Spin puzzle environment, $\mathcal{E}_{top}$, with the "rotate" ($r$) and "swap" ($s$) actions, and binary predicates $p_{ij} = 1$ if position $i$ has disc $j$, and $p_{ij} = 0$ otherwise. The correctly inferred update graph for this environment should consist of 20 disjoint, isomorphic subgraphs $U_j, j = 1, 2, ...20,$ , where subgraph $U_j$ contains 20 nodes for the 20 predicates $p_{ij}$ associated with disc $j$. If we number the nodes in $U_j$ as $1, 2, ...20$, then the $r$ action defines a cyclic permutation (1 2 ...20), and the $s$ action defines two cyclic permutations (1 4) and (2 3). In any state, exactly one node $p_{ij}$ in each $U_j$ will have value 1, and all others have value 0. Since each disc is in a different position, the $i$ for which $p_{ij}$ is 1 will be different in each subgraph $U_j$.

A closed, faithful partition of this update graph is one in which each block $i$ contains the nodes $p_{ij}, j = 1, 2, ...20$, that is *each block corresponds to a position* . The faithful ordering on each block $i$ simply orders the $p_{ij}$ in ascending order of $j$. Then the new (meta) update graph $U'$ will have 20 (meta) nodes, and each node will have a *distinct* value, since the blocks have distinct value patterns. The $s$ and $r$ actions define cycles which are similar to the original cycles. Thus we can reduce an update graph having 400 nodes with indistinct values to an equivalent update graph with 20 distinct-valued nodes. Thus in addition to reducing the size of the update graph, we have managed to obtain a better-behaved update graph: Any problem on the new update graph is now an SPP, and, as seen before, both the FHL and KMT algorithms apply. Also, the new update graph in a sense describes the environment more "naturally": The value of the (meta-node) $i$ represents the number of the disc in position $i$, and the $r$ and $s$ arrows represent how the discs move among the positions when the corresponding actions are performed. ■

The main result of this section can be summarized as:

**Theorem 3** *If the index-set $\{1, 2, ...n\}$ has a closed, faithful (CF) partition $\tau$ relative to the GPP $(\Pi, V_o)$ of degree $n$, then this GPP is equivalent to a smaller GPP of degree equal to the number of blocks of $\tau$. Furthermore, all CF-partitions can be found in polynomial time in $n$.*

# 7 TRANSFORMING TO A TOTALLY DECOMPOSABLE PROBLEM

We show in this section that it is always possible, at the expense of problem size, to change the representation of the state vectors in a GPP so that the actions (although no longer permutations of the state vectors) are totally decomposable.

For the GPP of degree $n$, $(\Pi, V_o)$, Consider the following *inverted* representation $X_V$ of an $n-$dimensional vector $V$: $X_V =< x_1 x_2 ... x_k >$ where $x_i$ is the *set of positions (node indices) that have value $t_i$*. The number of occurrences of the $k$ different values must remain the same after any action; the values simply get rearranged in the vector. We can show easily that

**Lemma 2** *In the inverted representation $X_V$, all actions are totally decomposable.*

**Proof:**

> Whenever an action $b$ is applied to the update graph, the values $t_i$ move according to the permutation $\pi_b$. Thus the indices which acquire value $t_i$ after this action depend only on which indices currently have the value $t_i$. Thus the new value of $x_i$ depends only on its current value.

□

Since total decomposability implies serial decomposability, the KMT approach could conceivably be used with this representation, although the macro table can in general be very large: the number of rows in the table is equal to the size of the set of possible values of the variables, and this set could be large. For example, the number of possible values of $x_i$ is $\binom{n}{n_i}$, where $n = |C|$, the number of nodes in the update graph. However, if the $n_i$ are all bounded by a constant $c$, this number would be polynomial in $n$, which means the macro table will be of size polynomial in $n$, and so problems can still be solved in time polynomial in the length of the optimal solution.

Clearly, when all the $n_i = 1$, we have a standard permutation problem, SPP, so we have:

**Lemma 3** *Every SPP can be transformed to a SDP*

*by inverting the state vector representation.*

Thus, for an SPP, *both* the FHL algorithm and the KMT algorithm are applicable.

**Example:**
The ideas of this section can be illustrated by the example of the Hungarian Rings puzzle: This is a pair of interlocking rings of beads. The two rings have two beads in common where they intersect, and the beads in each ring can be rotated on that ring. The beads are colored with $k$ different colors, and there are $m$ of each color. The goal is to achieve a given color-configuration. Consider the environment $\mathcal{E}_{hun}$ corresponding to this puzzle, with predicates $p_i$ giving the color of the bead in the position numbered $i$. This environment is a permutation environment, and its update graph will have $km$ nodes, and the values of the nodes will reflect the colors of the beads in the puzzle. There are thus $m$ nodes of each color. Now if we take the inverse representation, there are $k$ variables $x_1, x_2 ... x_k$, corresponding to the $k$ colors. The value of each variable $x_i$ is the set of positions containing beads of color $i$. Under this representation, all the actions are totally decomposable, so the KMT algorithm can be applied.

The number of possible values of the variables is $\binom{mk}{m}$. So if we consider the class of Hungarian Ring puzzles with constant $m$ and variable $k$, then the size of the Korf macro table is polynomial in $k$. ■

# 8 A RECIPE FOR SOLVING PROBLEMS WITH AN UPDATE GRAPH

There are many ways in which the different ideas presented in this paper can be combined into a strategy for attacking GPPs. We present here one possible recipe for solving a GPP $(\Pi, V_o)$.

**Procedure GPP-Solve**$(\Pi, V_o)$
1. $P \leftarrow (\Pi, V_o)$;
2. **If** $V_o$ has distinct values **then**
   $P$ is an SPP, so
   **apply FHL** to $P$, or
   **apply KMT** to the inverted representation of $P$.
   **return** "success"
   **endif**
3. **Find** all CF-reductions $P'$ of $P$
   **If** there is a CF-reduction $P' = (\Pi', V_o')$ **then**
   such that $V_o'$ has distinct values,
   $P \leftarrow P'$
   **goto** 2.
   **endif**
4. **Find** the smallest CF-reduction $P' = (\Pi', V_o')$ of $P$.
   **If** degree$(P')$ < degree$(P)$ **then**
   $P \leftarrow P'$
   **goto** 2.
   **endif**
5. **If** $P$ is totally decomposable **then**
   **apply KMT** to $P$.
   **return** "success"
   **endif**
6. (All simplifications and reductions fail)
   will have to use search methods to
   solve instances of $(\Pi, V_o, V_g)$
   individually, couldn't solve $(\Pi, V_o)$
   **return** "failure"

**endproc**

# 9 CONCLUSION

The fundamental problem in integrating a model-learning algorithm with a problem-solving algorithm is that the problem-solving algorithm may make assumptions about the state representation that are not always satisfied by the learned internal model. In this paper, we considered the problem of coupling Rivest and Schapire's model-learning algorithm with two problem-solving algorithms: Korf's Macro Table (KMT) algorithm and the permutation group algorithm of Furst, Hopcroft, and Luks(FHL). If the inferred representation (an update graph) is totally decomposable, the KMT algorithm can be applied. We presented a polynomial-time algorithm that detects whether the learned update graph satisfies this property. We also presented a representation-change that transforms an update graph problem into a totally decomposable problem. This same transformation can sometimes change the problem into a permutation group problem that can be handled by the FHL algorithm. Finally, we presented a polynomial-time algorithm that finds possible reductions of the update graph to a smaller, equivalent update graph. In addition to being smaller (and therefore more tractable) the reduced update graph may satisfy the requirements of the above two algorithms.

Though we have shown how to recognize when a set of variables in an update graph are serial decomposable reliably detecting serial decomposability in time polynomial in the size of the update graph is still an open problem. Since there exist problems whose optimal solutions are exponentially long in the size of the update graph, it is more practical, from an AI perspective, to measure efficiency relative to the length of the optimal solution. Developing practical algorithms which are provably efficient in this sense is an open research area. Also, methods need to be developed for analyzing the computational complexity of problem-solving algorithms relative to this measure.

## 10 ACKNOWLEDGEMENTS

We would like to thank Merrick Furst, Ravi Kannan, Craig Knoblock, Gary Miller, Tom Mitchell and Ron Rivest for their helpful discussions.

## A HARDNESS PROOFS

In this section we consider the GPP $(\Pi, V_o, V_g)$ and $G = \langle \Pi \rangle$. From [Hof82] we know that the Setwise Stabilizer Problem and the Coset Intersection Emptiness Problem problems for groups are polynomial time equivalent. So we will use them interchangeable. Some of these theorems should be known, and we are presenting them here only for completeness.

**Theorem 4** *Determining if $V_g \in G(V_o)$ can be solved by solve a Coset Intersection Emptiness Problem.*

**Proof:**

Let $\pi$ be any permutation in $S_n$ such that $\pi(V_o) = V_g$ (it is trivial to determine if such a permutation exits and if so find it, if no such permutation exists then $V_g \notin G(V_o)$). Let $H$ be the group of all permutations in $S_n$ that fix $V_o$ (it is easy to find generators for $H$). Then $V_g \in G(V_o)$ iff $H\pi \cap G \neq \emptyset$.

□

**Lemma 4** *Given $\pi \in S_n$ and $G, H \subset S_n$ such that $H\pi \cap G \neq \emptyset$ finding $h \in H\pi \cap G$ is poly time equivalent to the Coset Intersection Emptiness Problem.*

**Proof:**

This can be done by noticing that the FHL table for G (which can be constructed from generators in polynomial time) yields (by deleting initial rows and columns) a subgroup tower $I = G^{(n)} \subset G^{(n-1)} \subset \cdots \subset G^{(0)} = G$ such that the index of $G^{(i+1)}$ in $G^{(i)}$ is no greater than $n$. Let $k$ be the largest index such that $H\pi \cap G^{(k)} \neq \emptyset$. WLOG we can assume $k = n$ (else we can replace $G$ by $G^{(k)}$). Let $G' = G^{(n-1)}$ and $\tau_1, \tau_2 \cdots \tau_k$ ($k \leq n$) be a complete right traversal of $G$ by $G'$. Then since $G = \cup_{i=1}^{k} \tau_i G'$ we must have $H\pi \cap G'\tau_j \neq \emptyset$ for some $j$ such that $1 \leq j \leq k$. We can check these $k$ possibilities by checking Coset Intersection Emptiness on $H(\pi\tau_i^{-1}) \cap G'$ for $i = 1 \cdots k$. So we (recursively) solve the smaller problem by finding $h' \in H(\pi\tau_j^{-1}) \cap G'$ and $h = h'\tau_j$ is the answer.

□

**Theorem 5** *Finding $h \in G$ such that $h(V_o) = V_g$ is polynomial time equivalent (with respect to randomized Turing reductions) to the Setwise Stabilizer Problem.*

**Proof:**

From the previous proof it is easy to see that to find $h \in G$ such that $h(V_o) = V_g$ all we have to do is find $h \in H\pi \cap G$ (when $H\pi \cap G \neq \emptyset$) which, by the previous lemma, is poly time equivalent to Coset Intersection Emptiness (and hence Setwise Stabilizer).

For the other direction let $G$ be an arbitrary subgroup of $S_n$, $\Pi$ a set of generators for $G$ and $P \subset \{1, 2, \cdots n\}$ arbitrary. We will use the GPP problem to calculate the Setwise Stabilizer of $P$ in $G$. Let $V_o$ be the length $n$ vector $(V_o)_i = 1$ if $i \in P$ else $(V_o)_i = 0$. Assume we can solve GPP instances $(\Pi, V_o, V_g)$. We can calculate the setwise stabilizer $S$ of $P$ in polynomial time with exponentially small chance of error by running the following procedure:

1. $S = \{\mathsf{identity}\}$
2. Repeat $n^2$ times:
   Let $\pi$ be a random permutation from $G$.
   Let $h$ be a solution of $(\Pi, V_o, \pi(V_o))$.
   Let $S = \langle S, \pi h^{-1} \rangle$.
   **endrep**

The procedure works because during each iteration $S$ is a subgroup of $stab(P, G)$ therefore if $S \neq stab(P, G)$ we have $|S| \leq |stab(P, G)|/2$. The GPP solution procedure has no way of knowing $\pi$ so since $\pi$ is randomly selected from $G$ we see that $\pi h^{-1}$ will be randomly selected from $stab(P, G)$, so if $S \neq stab(P, G)$ we know $\pi h^{-1} \notin S$ with probability $\geq 1/2$. and in addition we will have $|\langle S, \pi h^{-1} \rangle| \geq 2|S|$. Therefore we need only about $\log_2 n!$ successes to compute $stab(P, G)$. $\log n!$ is asymptotically equal to $n \log n$ and is therefore dominated by $n^2$, so it is a simple matter of statistics (Binomial trials) to show our chance of failure is exponentially small.

□

**Theorem 6** *Deciding if there is an extension of a partial labeling $\tilde{V_g}$ into a labeling $V_g$ such that $V_g \in G(V_o)$ is NP-complete.*

**Proof:**

Since a labeling $V_g$ and a permutation $h$ can be presented succinctly and we can check in poly time if $h \in G$ and $h(V_o) = h(V_g)$ it is clear that this problem is in NP.

To prove it is NP-complete we demonstrate that this problem can encode the Clique problem (determining if a graph has a clique of size $k$, which is known to be NP-complete). Let $K$ be an arbitrary graph of $n$ nodes encoded as a length $n^2$ vector $V_o$ such that $(V_o)_{n*(i-1)+j} = 1$ if there is an edge from node $i$ to node $j$ and $(V_o)_{n*(i-1)+j} = 0$ otherwise. Let $G$ be $S_n$ acting on $V_o$ such that for $\pi \in S_n$ and a length $n^2$ vector $v$ we have: $(\pi(v))_{n*(i-1)+j} = v_{n*(\pi(i)-1)+\pi(j)}$. Let $\tilde{V_g}$ be the length $n^2$ vector with 1's in the first $k^2$ entries and the rest undetermined. Then there exists $V_g$ such that $V_g$ agrees with $\tilde{V_g}$ in all the specified positions and $V_g \in G(V_o)$ if and only if $K$ has a $k$-clique.

□

## References


[FHL80] M.L. Furst, J.E. Hopcroft, and E. Luks. "Polynomial-time algorithms for permutation groups" In *Proc. 21st IEEE Foundations of Computer Science*, pages 36–41, 1980.

[FSS+89] A. Fiat, M. Shahar, A. Shamir, I. Shimshoni, and G. Tardos. "Planning and learning in permutation groups" In *Proc. 30th IEEE Foundations of Computer Science*, pages 274–279, 1989.

[GJ79] M. Garey, and D. Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness* W.H. Freeman and Company, 1979.

[Hof82] Christoph M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism* Springer-Verlag, 1982.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory.* Mc Graw-Hill, 1978.

[Kor85] R.E. Korf. "Macro-operators: A weak method for learning" *Artificial Intelligence*, 26:35–77, 1985.

[Mac68] I.D. Macdonald. *The Theory of Groups.* Oxford Univeristy Press, 1968.

[RS87] R. L. Rivest and R. E. Schapire. "Diversity-based inference of finite automata" In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78–87, Oct 1987.

[Sch88] R.E. Schapire. *Diversity-based inference of finite automata* Master's thesis, Massachusetts Institue of Technology, May 1988.