PSVN: A Vector Representation for Production Systems

István T. Hernádvölgyi and **Robert C. Holte**
University of Ottawa
School of Information Technology & Engineering
Ottawa, Ontario, K1N 6N5, Canada
Email: {istvan,holte}@site.uottawa.ca

Abstract

In this paper we present a production system which acts on fixed length vectors of labels. Our goal is to automatically generate heuristics to search the state space for shortest paths between states efficiently. The heuristic values which guide search in the state space are obtained by searching for the shortest path in an *abstract* space derived from the definition of the original space. In PSVN, a state is a fixed length vector of labels and abstractions are generated by simply mapping the set of labels to another smaller set of labels (*domain abstraction*). A domain abstraction on labels induces a *state space abstraction* and this abstract space preserves important properties of the original space while usually being significantly smaller in size. It is guaranteed that the shortest path between two states in the original space is at least as long as the shortest path between their images in the abstract space. Hence, such abstractions provide admissible heuristics for search algorithms such as A* and IDA*. The mapping of states and operators can be efficiently obtained by applying the domain map on the labels. We explore important properties of state spaces defined in PSVN and abstractions generated by domain maps. Despite its simplicity, PSVN is capable to define all finitely generated permutation groups and such benchmark problems as Rubik's Cube, the sliding-tile puzzles and the Blocks World.

Introduction

It is quite common in Artificial Intelligence to represent a problem in a formalism which can be easily implemented and has properties that make problem solving computationally feasible. It is often a natural approach to model the problem with an implicitly generated graph where the vertices are the reachable states and the directed labeled edges correspond to the application of operators. Games like Chess, Checkers and Go-Moku and puzzles like the Blocks World, Rubik's Cube and the sliding tile puzzles are usually represented this way. These problems are often solved by searching the state space for a goal state. For large problems, like the ones mentioned above, it is computationally infeasible to enumerate the entire state space and search it as an explicit graph. Instead states are generated as they are encountered by applying operators. Often these problems are defined in production system notations, which provide state and operator descriptions (like STRIPS (Fikes & Nilsson 1971)). Blind search will eventually expand most of the state space. To reduce the number of states expanded by the algorithm, heuristic estimates are calculated to rank the states with respect to how close they are to the goal. It is not an easy problem in general to invent accurate heuristics (Pearl 1984; Prieditis 1993). Some search algorithms, like A* and IDA* (Korf 1985) are guaranteed to find a shortest path if the heuristic values are underestimates of the true distance from the state to the goal. Such heuristics are also called *admissible*.

In PSVN, states are represented by fixed length vectors of labels. A domain abstraction ϕ is a map from the set of labels to another smaller set of labels. ϕ is applied to the operators O and to the seed state s_0. The transitive closure of $\phi(O)$ and $\phi(s_0)$ – *the set of reachable states from* s_0 *by applying operators from* $\phi(O)$ – is another state space which preserves many important properties of the original space, but it is usually magnitudes smaller. It is also the case, which we will prove, that the shortest path connecting two states in the original space is at least as long as the shortest path connecting their images in the abstract space. Hence the length of the shortest path connecting the image of a state s to the image of the goal state in the abstract space is an admissible heuristic. The heuristic function can be stored as a table of states and the length of their shortest path to the image of the goal state. The heuristic value $h(s)$ is obtained by applying the domain map to s and retrieving the value from the table. Precomputed heuristic values stored in tables are also called *pattern databases* (Culberson & Schaeffer 1996), and have been successfully used to find shortest paths in very large state spaces (Rubik's Cube (Korf 1997), 15-Puzzle (Culberson & Schaeffer 1994)). Encouraged by these results, we designed PSVN to be able to represent these problems and at the same time provide a conve-

nient and efficient way to generate pattern databases.

State Space Representation & Generation

Formally, a state space is defined by a triple $S =< s_0, O, L >$, where s_0 is a state, O is a set of operators, and L is a finite set of labels. The state space is the transitive closure of s_0 and the operators, *i.e.*, it consists of all reachable states from s_0 by any sequence of operators. s_0 is the *seed state*. For a problem to be solvable, both goal state g and start state s must be in S and g must be reachable from s. For example, the state space of Rubik's Cube[1] is the set of all scrambled cubes reachable by applying the quarter turns. If one decides to arbitrarily rearrange the colored stickers, it is not guaranteed anymore that it can be obtained from the seed state, as not all permutations of the 54 stickers can be obtained by legal moves.

A state in PSVN is a fixed length vector of labels from L. Operators are defined by a left-hand-side (*LHS*) and a right-hand-side (*RHS*) each having the same length as a state vector. *LHS* represents a precondition and may introduce variables which are bound to labels. *RHS* defines the resulting state. A constant in *LHS* represents an exact match for the state at that position. An variable in *LHS* represents a binding of the label in the state to which *LHS* is being applied and an underscore (_) ignores the label. In *RHS*, a constant label or an identifier bound in *LHS* designates the assignment of the new label at the position and the underscore leaves the label untouched. Without loss of generality, we assume that labels are non-negative integers. For example, consider the following operator definition:

$$< A, A, 1, \_, B, C > \rightarrow < 2, \_, \_, \_, C, B >$$

The operator above applies to states whose first two labels are identical and the third label is 1. The fifth and sixth labels are bound to B and C respectively. The resulting state has the first element relabeled to 2 and the last two elements swapped. Applying this operator o to $s =< 4, 4, 1, 7, 5, 6 >$ results in $o(s) =< 2, 4, 1, 7, 6, 5 >$. Every variable of *RHS* must be bound in *LHS* and every constant label must belong to the set of declared labels.

Domain and State Space Abstractions

Our ultimate goal is to generate state spaces which provide admissible heuristics to search in the original space. First we introduce the concept of *domain abstraction* and show how it generates abstract state spaces for problems defined in PSVN.

[1] see the appendix

A domain abstraction is a map $\phi : L \rightarrow K$, where L and K are sets of labels and $|K| \leq |L|$. In other words, a domain abstraction is just a simple mapping of the labels to another smaller set of labels. K may be a subset of L.

A state space abstraction is induced by a domain abstraction by applying the map ϕ to the seed state and the operators: $S' = \phi(S) =< \phi(s_0), \phi(O), K >$. The action of ϕ on an operator is to relabel the constants.

For example, consider the map $\phi : L \rightarrow K$ for $L = \{1, 2, 3, 5\}$ and $K = \{4, 6\}$:

$$\phi(x) = \begin{cases} 4 & x \in \{1, 2\} \\ 6 & x \in \{3, 5\} \end{cases}$$

Let the original state space be defined as $S =< s_0, O, L >$ where:

$$O = \left\{ \begin{array}{l} o_1 =< A, A, 1, \_ > \rightarrow < 2, \_, 3, \_ >, \\ o_2 =< \_, X, \_, 2 > \rightarrow < \_, 2, \_, X >, \\ o_3 =< 5, \_, 1, B > \rightarrow < B, \_, 5, 1 >, \\ o_4 =< B, \_, 5, 1 > \rightarrow < 5, \_, 1, B > \end{array} \right\}$$

$$s_0 =< 5, 5, 1, 2 >$$

Then, $S' = \phi(S) =< \phi(s_0), \phi(O), K >$, where

$$\phi(O) = \left\{ \begin{array}{l} \phi(o_1) =< A, A, 4, \_ > \rightarrow < 4, \_, 6, \_ >, \\ \phi(o_2) =< \_, X, \_, 4 > \rightarrow < \_, 4, \_, X >, \\ \phi(o_3) =< 6, \_, 4, B > \rightarrow < B, \_, 6, 4 >, \\ \phi(o_4) =< B, \_, 6, 4 > \rightarrow < 6, \_, 4, B >, \end{array} \right\}$$

$$\phi(s_0) =< 6, 6, 4, 4 >$$

The state spaces defined by the above seed states, operators and abstraction are depicted in figure 1.

The key property of state space abstractions is that they are homomorphisms and therefore the distance between two abstract states is never more than the distance between their pre-images in the original space. Moreover this distance provides a monotonic heuristic for A*.

Definition: *State Space Homomorphism.* Let S and T be state spaces with operators O_S and O_T respectively. $\phi : S \rightarrow T$ is a *state space homomorphism* if for every state $u, v \in S$ and for every operator $o \in O_S$ $v = o(u)$ implies that $\phi(o)$ is applicable to $\phi(u)$ and $\phi(v) = \phi(o)(\phi(u))$.

Theorem 1: If ϕ is a domain abstraction then ϕ induces a state space homomorphism.

Proof: Because o applies to u the constant labels in LHS_o match the values of u at those positions. ϕ maps the matching labels of LHS_o and u to identical ones, hence $\phi(o)$ applies to $\phi(u)$. Now consider the action of o and $\phi(o)$ on u and $\phi(u)$ elementwise. Let $u[i]$, $LHS_o[i]$ and $RHS_o[i]$ represent the element at position i of state u, and the left and right hand sides of o respectively. Let c be a constant and A a variable bound in LHS_o.

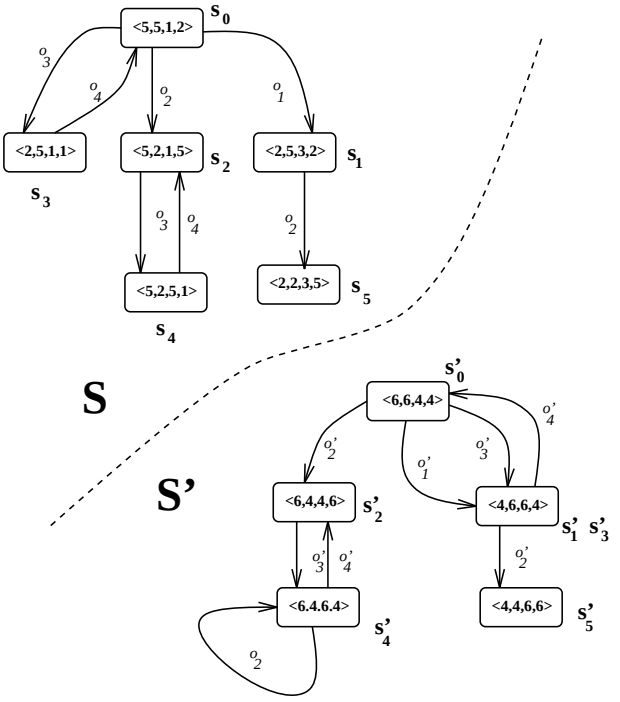

Figure 1: A state space and an abstraction

First let us derive the elements of $w = \phi(o)(\phi(u))$. If $RHS_o[i] = \_$ then $w[i] = \phi(u[i])$. If $RHS_o[i] = c$ then $w[i] = \phi(c)$. If $RHS_o[i] = A = LHS_o[j]$ is a variable bound at position j in LHS_o then $w[i] = \phi(u[j])$. The elements of $v = o(u)$ are as follows: if $RHS_o[i] = \_$ then $v[i] = u[i]$. If $RHS_o[i] = c$ then $v[i] = c$. And if $RHS_o[i] = LHS_o[j] = A$ then $v[i] = u[j]$. Hence $w = \phi(o)(\phi(u)) = \phi(v)$.

Theorem 2: If ϕ is a state space homomorphism then for any states $u_1, u_k \in S$ the length of the shortest path between states $\phi(u_1) = u'_1$ and $\phi(u_k) = u'_k$ in $\phi(S) = S'$ is less or equal to the length of the shortest path between u_1 and u_k in S.

Proof: Let $o_1 o_2 ... o_{k-1}$ be a shortest path between u_1 and u_k in S. Since $\phi : S \rightarrow S'$ is a homomorphism, $\phi(o_1 o_2 ... o_{k-1}) = \phi(o_1)\phi(o_2)...\phi(o_{k-1})$ is an applicable sequence of operators forming a path between states u'_1 and u'_k. The shortest path between u'_1 and u'_k cannot be longer than this, hence the result follows.

It is instructive to consider how there can be a shorter path than $\phi(o_1 o_2 ... o_{k-1})$. Let the states on this path be $u'_1 ... u'_i ... u'_j ... u'_k$. If $u'_i = u'_j$ then $\phi(o_i)...\phi(o_{j-1})$ can be eliminated from the path, because the sequence corresponds to a loop. As a special case if $u'_j = u'_k$ then $\phi(o_1)...\phi(o_{j-1})$ is a path between u'_1 and u'_k shorter than $\phi(o_1 o_2 ... o_{k-1})$. Alternatively suppose there is a state $v \in S$ such that v is not equal to u_i for any i but there is a path between v and u_k in S. If $v' = \phi(v) = u'_j$ then $u'_1 ... v' ... u'_k$ is also a path between u'_1 and u'_k which might be shorter than $u'_1 ... u'_j ... u'_k$. Note that this path does not correspond to a path between u_1 and u_k in S.

Theorem 3: Let S be state space and S' be obtained by domain abstraction ϕ. Let heuristic function $h(u)$ for state u and goal state g ($u, g \in S$), be defined as the length of the shortest path between $\phi(u)$ and $\phi(g)$ is S'. $h(u)$ is a monotonic heuristic function.

Proof: Let $d(u, v)$ denote the length of the shortest path between u and v in S and let $d'(\phi(u), \phi(v))$ denote the length of the shortest path between $\phi(u)$ and $\phi(v)$ in $S' = \phi(S)$. Pearl (Pearl 1984) has shown that a heuristic is *monotonic* if $d(u, v) + h(u) \geq h(v)$ for all states $u, v \in S$. Consider the path obtained by concatenating the shortest path between $\phi(v)$ and $\phi(u)$ and the shortest path between $\phi(u)$ and $\phi(g)$. The length of this path is $h^+(y) = d'(\phi(u), \phi(v)) + h(u)$. But the length of the first leg $d'(\phi(u), \phi(v))$ between $\phi(v)$ and $\phi(u)$ is less or equal to $d(u, v)$ because $\phi : S \rightarrow S'$ is a homomorphism. $h^+(v)$ is the length of a path between $\phi(v)$ and $\phi(g)$, hence $d(u, v) + h(u) \geq h^+(v) \geq h(v)$.

Building the Abstract Space

Ultimately, we want to build the abstract space $\phi(S)$. We also want to calculate the distance $h(s)$ between $\phi(s)$ and $\phi(g)$ for all $\phi(s) \in \phi(S)$ to obtain heuristic values for $s \in S$. In this section we also address the issue of operator invertibility and non-surjective state space homomorphisms.

Definition: *Pattern Database.* Let $g \in S$ be a goal state and ϕ a domain abstraction. A *pattern database* is a table indexed by $\phi(s)$ ($s \in S$) containing values of the length of the shortest path between $\phi(s)$ and $\phi(g)$ in $\phi(S)$.

Because $\phi(S)$ will be entirely expanded, to construct the pattern database we can employ *breadth first* traversal from the goal state using the *inverses* of the operators. This way, we expand $\phi(S)$ and at the same time obtain $h(s)$ for all $s \in S$.

However, operators defined in PSVN are not necessarily invertible. First we examine what operator definitions give rise to invertible operators, and then we describe how to build the abstract space and obtain $h(s)$ if some of the operators are not invertible.

Invertibility

Definition: *Invertible Operator.* An operator o is invertible if all of the labels of the source state s can be *uniquely* recreated from the definition of o and the labels of $o(s)$.

Consider the following operator definitions:

$$o_1 :< A, A, \_, \_ > \rightarrow < 1, 2, \_, \_ >$$

$$o_2 :< A, A, \_, \_ > \rightarrow < \_, 2, \_, \_ >$$
$$o_3 :< A, A, \_, \_ > \rightarrow < \_, \_, A, \_ >$$
$$o_4 :< 3, \_, \_, \_ > \rightarrow < \_, \_, 2, \_ >$$

o_1 is not uniquely invertible. Since A does not occur in *RHS*, its inverse cannot have a binding in its *LHS*. In o_1, A can be bound to any label of L, hence all states resulting from o_1^{-1} could be of the form $< A, A, \_, \_ >$ where $A \in L$. o_2 is invertible, because A overwritten on the second position is implicitly bound at the first position:

$$o_2^{-1} =< A, 2, \_, \_ > \rightarrow < A, A, \_, \_ >$$

o_3 is is not invertible. In o_3^{-1} the value bound at the third position, uniquely determines the labels for the first two, but o_3^{-1} cannot know what label to assign to the third position. For example

$$< 1, 1, 2, 5 > \ -o_3 \rightarrow < 1, 1, 1, 5 >$$

The label 2 on the third position of the source state is not present in the resulting state nor does o_3 explicitly specify this label to match. Hence the source state cannot be recreated from the o_3 and the resulting state. o_4 is also not invertible, because it cannot be determined for o_4^{-1} what the label of the third position used to be before label 2 overwrote it.

Definition: *Present Label Binding.* Let o : *LHS* $\rightarrow$ *RHS*. A variable A bound in *LHS* is *present* in *RHS*, if it is used to assign a label in *RHS or* if there is a j such that $LHS[j] = A$ and $RHS[j] = \_$. The latter case is equivalent to $LHS[j] = A$ and $RHS[j] = A$.

Theorem 4: An PSVN operator is invertible if and only if there is no i such that $LHS[i] = \_$ and $RHS[i] \neq \_$ and every variable bound in *LHS* is *present* in *RHS*.

Proof: Let us consider every pair of label designations at corresponding positions. Let c, c_1, c_2 represent constant labels, A, B variable bindings, and i a position. According to our definition of invertible operator, the inverse operator must be able to recreate the original source state from its definition and the original resulting state.

- $LHS[i] = \_$ and $RHS[i] = \_$

  The label at position i does not change, hence it does not violate invertibility

- $LHS[i] = \_$ and $RHS[i] = A$
  $LHS[i] = \_$ and $RHS[i] = c$

  The label of the source state in position i is not bound and not specified to match any particular label. This label is overwritten by the label bound to A or by label c. Hence the inverse operator cannot determine what label to assign to position i to recover the overwritten label. Such a condition necessarily violates invertibility.

- $LHS[i] = A$ and $RHS[i] = \_$
  $LHS[i] = A$ and $RHS[i] = c$
  $LHS[i] = A$ and $RHS[i] = B$

  If A is *present* in *RHS* then the inverse operator can assign the label bound to it to position i, otherwise the label cannot be uniquely determined.

- $LHS[i] = c$ and $RHS[i] = \_$
  $LHS[i] = c$ and $RHS[i] = A$
  $LHS[i] = c_1$ and $RHS[i] = c_2$

  The label to assign to position i by the inverse operator is explicitly specified and hence invertibility is not violated.

Theorem 5: If ϕ is a domain abstraction and o is an invertible operator then $\phi(o)$ is also invertible.

Proof: ϕ only relabels constants in an operator definition, hence the conditions listed for the proof of theorem 4 apply the same way.

If all operators of $o \in O$ are invertible, then a breadth first traversal using $\phi(O)^{-1}$ from $\phi(g)$ will generate $\phi(S)$ and can be used to calculate the heuristic values. If one or more of the operators are not invertible we can still achieve our goal, but some extra processing is needed.

Theorem 6: Let $S =< s_0, O, L >$, $g \in S$ and ϕ be a domain abstraction. If all operators of O are invertible then the pattern database obtained by a breadth first traversal from $\phi(g)$ using $\phi(O)^{-1}$ (*the inverses of the abstract operators*) has entries for all $s \in S$ from which g can be reached.

Proof: If g is reachable from s in S, then so is $\phi(g)$ from $\phi(s)$ in $\phi(S)$, because $\phi : S \rightarrow \phi(S)$ is a homomorphism. If o is invertible then so is $\phi(o)$ – as we proved –, hence there is a path in terms of the operators $\phi(O)^{-1}$ from $\phi(g)$ to $\phi(s)$ if there is a path in terms of the operators from O connecting s and g.

While operators of PSVN are not necessary invertible, for every operator there is a finite number of possible inverses. It is justified to use the *closed world* assumption, because L is a finite set of labels. However when more than one variables may have ambiguous bindings, the number of inverses can be potentially large. Even if one generates all inverses, the technique of generating $\phi(S)$ from $\phi(g)$ using these inverse operators, may give rise to a much larger space which embeds the real $\phi(S)$.

We present two techniques to build the abstract state if the operators are not invertible. First we generate $\phi(S)$ from $\phi(g)$ using all *possible* inverses of the operators.

Consider state space $S =< s_0, O, L >$, goal state g and abstraction $\phi : L \rightarrow K$ where

$$L = \{1, 2, 3, 4, 5\}$$

$$K = \{1, 2\}$$
$$s_0 =< 1, 2, 3, 4 >$$
$$g =< 2, 3, 3, 2 >$$
$$O = \left\{ \begin{array}{c} o_1 :< \_, 3, \_, \_ > \rightarrow < 2, \_, \_, \_ > \\ o_2 :< A, \_, B, \_ > \rightarrow < \_, B, \_, A > \end{array} \right\}$$
$$\phi(x) = \left\{ \begin{array}{ll} 1 & x \in \{1, 2, 3\} \\ 2 & x \in \{4, 5\} \end{array} \right.$$

Then, by generating all possible inverses and applying ϕ,

$$\phi(s_0) =< 1, 1, 1, 2 >$$
$$\phi(g) =< 1, 1, 1, 1 >$$
$$\phi(O) = \left\{ \begin{array}{c} \phi(o_1) :< \_, 1, \_, \_ > \rightarrow < 1, \_, \_, \_ > \\ \phi(o_2) :< A, \_, B, \_ > \rightarrow < \_, B, \_, A > \end{array} \right\}$$

$$\phi(O)^{-1} =$$
$$\left\{ \begin{array}{c} \phi(o_1)_1^{-1} :< 1, 1, \_, \_ > \rightarrow < 1, 1, \_, \_ > \\ \phi(o_1)_2^{-1} :< 1, 1, \_, \_ > \rightarrow < 2, 1, \_, \_ > \\ \phi(o_2)_1^{-1} :< A, B, B, A > \rightarrow < A, 1, B, 1 > \\ \phi(o_2)_2^{-1} :< A, B, B, A > \rightarrow < A, 1, B, 2 > \\ \phi(o_2)_3^{-1} :< A, B, B, A > \rightarrow < A, 2, B, 2 > \\ \phi(o_2)_4^{-1} :< A, B, B, A > \rightarrow < A, 2, B, 1 > \end{array} \right\}$$

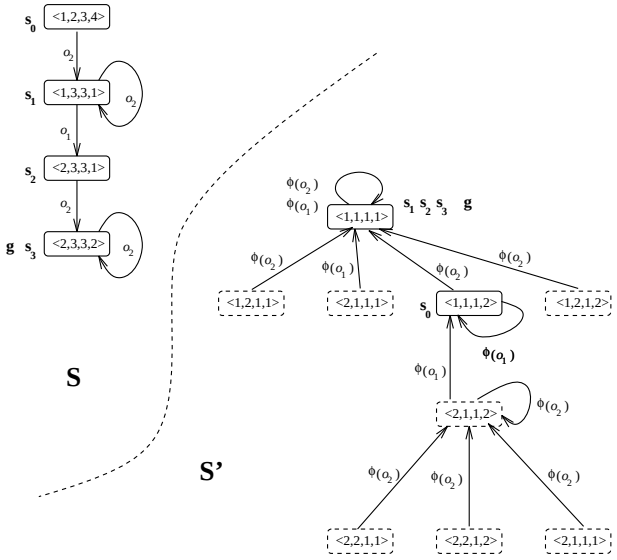

Figure 2: A state space and its abstraction expanded by generating all possible inverses of non-invertible operators

The state space S and its abstraction S' is shown on figure 2. The states in the abstract space enclosed in *dashed* boundaries do not have pre-images in S and were obtained because unique inverses of the operators could not be determined. Starting the expansion from $\phi(s_0)$ as opposed to $\phi(g)$ and using $\phi(O)$ as opposed to $\phi(O)^{-1}$ does not generate these spurious states.

Another technique to generate $\phi(S)$ and to calculate the pattern database is to search for $\phi(g)$ in $\phi(S)$ from $\phi(s)$ and building an explicit graph with backwards edges corresponding to the inverses of the operators. First one would use a depth first traversal of $\phi(S)$ from $\phi(s_0)$ and build $\phi(S)$ as an explicit graph (*adjacency list*), but if $\phi(u)$ is connected to $\phi(v)$ then the backward edge $\phi(v) \rightarrow \phi(u)$ is added (*instead of* $\phi(u) \rightarrow \phi(v)$). Then a breadth first traversal of this explicit graph from $\phi(g)$ can be used to obtain the heuristic values.

Surjectivity

Definition: *Surjective State-Space Homomorphism.* Let $\phi : S \rightarrow S'$ be a state space homomorphism. ϕ is surjective if and only if for every state $s' \in S'$ there is a state $s \in S$ such that $s' = \phi(s)$.

Non-surjective state space homomorphisms arise quite often. When $\phi : S \rightarrow S'$ is non-surjective, the pattern database contains entries which will never be mapped to by ϕ. These entries take up space in the database but they are never used. To demonstrate non-surjective homomorphisms, consider the *2* × *2* sliding-tile puzzle depicted on figure 3. The shaded tile is *empty* so a

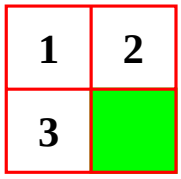

Figure 3: *2* × *2* Sliding Tile Puzzle

neighboring tile can be slid into its place. The number of possible states reachable by legal moves is 12. Label 0 represents the empty tile. The state on figure 3 is represented as $< 1, 2, 3, 0 >$: tile 1 is on position 1 (*top left*), tile 2 is on position 2 (*top right*), tile 3 is on position 3 (*bottom left*) and tile 0 (*empty*) is on position 4 (*bottom right*). The operator definitions are:

$$O = \left\{ \begin{array}{c} o_1 :< A, 0, \_, \_ > \rightarrow < 0, A, \_, \_ >, \\ o_2 :< A, \_, 0, \_ > \rightarrow < 0, \_, A, \_ >, \\ o_3 :< \_, A, \_, 0 > \rightarrow < \_, 0, \_, A >, \\ o_3 :< \_, \_, A, 0 > \rightarrow < \_, \_, A, 0 >, \\ o_1^{-1}, o_2^{-1}, o_3^{-1}, o_4^{-1} \end{array} \right\}$$

The state space $S =<< 1, 2, 3, 0 >, O, \{0, 1, 2, 3\} >$ is shown on figure 4. We consider three domain abstractions of which two give rise to non-surjective homomorphisms.

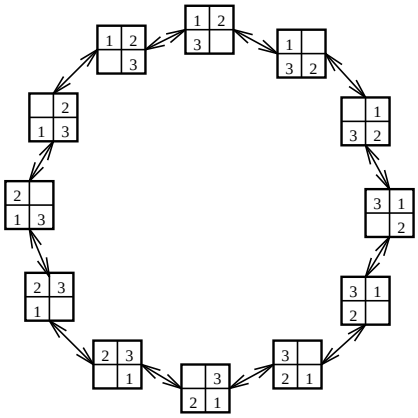

Figure 4: The *2* × *2* Sliding Tile Puzzle State Space

Let $\phi_1 : \{0,1,2,3\} \rightarrow \{2,0\}$ be defined as:

$$\phi_1(x) = \begin{cases} 0 & x = 0 \\ 2 & x \neq 0 \end{cases}$$

$\phi_1(S)$ has four states. Three states of S are mapped to each state of S'. For goal state $< 1,2,3,0 >$, the pattern database is:

$\phi(s)$	$h(s)$
$< 2,2,2,0 >$	0
$< 2,2,0,2 >$	1
$< 2,0,2,2 >$	1
$< 0,2,2,2 >$	2

This pattern database gives the exact distance as a heuristic for states $< 0,1,3,2 >$ and $< 0,2,1,3 >$ but estimates $< 0,3,2,1 >$ to be 0 moves away when it is actually 6 moves away. $\phi_1(S)$ is depicted on figure

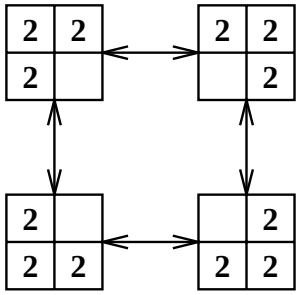

Figure 5: $\phi_1(S)$

5. This homomorphism is surjective because there is a pre-image in S for each state of S'.

Let us now consider another domain abstraction $\phi_2 : \{0,1,2,3\} \rightarrow \{1,2,0\}$ defined as:

$$\phi_2(x) = \begin{cases} 0 & x = 3 \\ x & x \neq 3 \end{cases}$$

Observe that this abstraction has two 0's (*hence two empty tiles*) and therefore it increases the branching factor. Moreover ϕ_2 is a non-surjective homomorphism, because there are states in $\phi_2(S)$ which have no pre-image in S. These states of $\phi_2(S)$ have *dashed line* boundaries in figure 6.

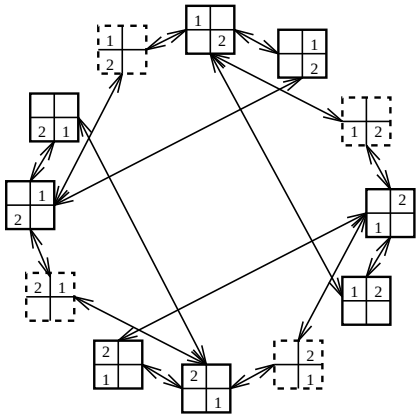

Figure 6: $\phi_2(S)$

Another illustrative example of a non-surjective homomorphism is obtained by applying domain abstraction on a different representation of this puzzle. In the previous state representation the label on position i in the vector actually corresponds to a tile in the puzzle. We can also describe states in vector form such that the indices represent the tiles and the labels represent the positions the tiles currently occupy. Examples of this *dual* representation are shown on figure 7. Vec-

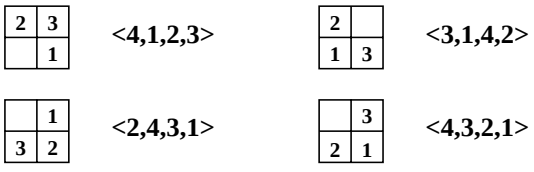

Figure 7: Dual Representation of the *2* × *2* Sliding Tile Puzzle

tor $< 4,1,2,3 >$ represents the state where tile 1 is on position 4 (*bottom right*), tile 2 is on position 1 (*top left*), tile 3 is on position 2 (*top right*) and tile 0 (*the*

empty tile) is on position 3 (*bottom left*). The operators describing the puzzle's moves are as follows:

$$O = \left\{ \begin{array}{c} o_1 =< 2,\_,\_,1 > \rightarrow < 1,\_,\_,2 >, \\ o_2 =< 3,\_,\_,1 > \rightarrow < 1,\_,\_,3 >, \\ o_3 =< \_,2,\_,1 > \rightarrow < \_,1,\_,2 >, \\ o_4 =< \_,3,\_,1 > \rightarrow < \_,1,\_,3 >, \\ o_5 =< \_,\_,2,1 > \rightarrow < \_,\_,1,2 >, \\ o_6 =< \_,\_,3,1 > \rightarrow < \_,\_,1,3 >, \\ o_7 =< 4,\_,\_,2 > \rightarrow < 2,\_,\_,4 >, \\ o_8 =< \_,4,\_,2 > \rightarrow < \_,2,\_,4 >, \\ o_9 =< \_,\_,4,2 > \rightarrow < \_,\_,2,4 >, \\ o_{10} =< 4,\_,\_,3 > \rightarrow < 3,\_,\_,4 >, \\ o_{11} =< \_,4,\_,3 > \rightarrow < \_,3,\_,4 >, \\ o_{12} =< \_,\_,4,3 > \rightarrow < \_,\_,3,4 >, \\ o_1^{-1}, o_2^{-1}, o_3^{-1}, o_4^{-1}, o_5^{-1}, o_6^{-1}, \\ o_7^{-1}, o_8^{-1}, o_9^{-1}, o_{10}^{-1}, o_{11}^{-1}, o_{12}^{-1} \end{array} \right\}$$

Now let us consider domain abstraction ϕ_3 defined as

$$\phi_3(x) = \left\{ \begin{array}{ll} 2 & x = 4 \\ x & x \neq 4 \end{array} \right.$$

ϕ_3 renders *positions* 2 and 4 indistinguishable. $\phi_3(S)$ has 12 states (*figure 8*); 3 of them do not have a pre-

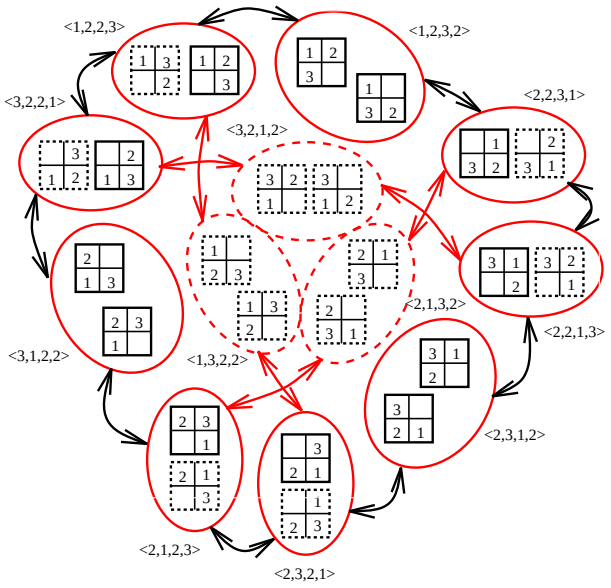

Figure 8: $\phi_3(S)$

image in S. ϕ_3 abstracts the positions the tiles occupy rather than the tiles themselves. The actual puzzle states of S enclosed in the ellipses on figure 8 correspond to one state of $\phi_3(S)$. The states enclosed in dashed boundaries do not have pre-images in S. It is interesting to see how the branching factor increases: if the empty tile is on position 1 or 3, it can be swapped with the tile across the diagonal – an otherwise illegal move. In this case, either tile 2 or tile 4 or both are on position 2, which represents positions 2 and 4 in the original space.

$\phi(S)$ as a Heuristic

Naturally, we are interested in knowing how such a pattern database is expected to perform. (Korf 1997) used pattern databases to find optimal solutions to random instances of the Rubik's Cube for the first time[2]. His abstraction rendered all edge pieces to the same identical label, while the stickers on the corner kept their identities. This mapping actually corresponds to the $2 \times 2 \times 2$ mini cube which only has 8 corner cubies. This puzzle is also sold commercially. (Culberson & Schaeffer 1996) have used pattern databases together with the Manhattan distance heuristic to search the 15 sliding-tile puzzle. They also took advantage of geometrical symmetries offered by the puzzle. Considering the size of the state spaces – $\approx 10^{19}$ and $\approx 10^{13}$ –, pattern databases promise to find optimal paths in very large spaces using heuristic search. A large scale study of the relationship between the expected number of states expanded and the size of the pattern database was conducted by (Holte & Hernádvölgyi 1999). A linear relationship was first conjectured by (Korf 1997) and he later refined it based on (Korf & Reid 1998).

Let t be the expected number of states expanded in the worst case, and b and d be the effective branching factor and the depth of the optimal solution respectively. $P(x)$ is the probability that state $s \in S$ has a heuristic value $h(s) \leq x$. Then

$$t(b, d, P) = \sum_{i=1}^{d+1} b^i P(d - i + 1)$$

Korf suggested that the above formula can be reduced to

$$t \approx n \frac{\log_b m}{m}$$

where m is the size of the pattern database. Our study (Holte & Hernádvölgyi 1999) has confirmed that a simple relationship of the form

$$t \cdot m^{\alpha} = c$$

holds for some state space and solution specific constants α and c. This is a very important property because the performance of the pattern database with respect to the number of states expanded can be predicted

[2]Mike Reid and Herbert Kociemba also have implementations of optimal solvers, but they take advantage of the fact that Rubik's Cube is a permutation group and other special properties of the puzzle. Contact `cube-lovers@ai.mit.edu` for more information. Korf's program uses pure search (IDA*) with a heuristic defined by a pattern database.

from the amount of memory it occupies. Our randomly generated pattern databases for the 8 sliding-tile puzzle suggest that pattern databases of size 5040 – which is equivalent to 2.7% of the size of the original space – outperform the Manhattan distance heuristic. We also found that using more than one pattern databases together outperform a single pattern database with the same memory requirements. The number of states expanded using three pattern databases is approximately half of the number of states expanded by a single pattern database with size of the combined capacities of the three. Having more than three databases does not increase performance further in our experiments.

We also generated a large number of domain maps that all gave rise to abstract spaces of the same size, and we found (Holte & Hernádvölgyi 1999) that some pattern databases are clearly superior to others of the same size due to larger heuristic values on average and/or a favorable distribution of heuristic values in the original space. We are currently investigating how to predict from the domain map the performance of the pattern database it generates without calculating the distribution of heuristic values in the original space.

Concluding Remarks

In this paper we presented a production system, PSVN, which acts on fixed length vectors of labels. We defined domain maps of labels which induce state space homomorphisms and hence provide admissible heuristics – which can be stored in pattern databases – for search algorithms such as A* and IDA*. We also explored properties of state spaces defined in PSVN and abstractions generated by domain maps. We established necessary and sufficient conditions for PSVN operators to be invertible and described how to build the abstract space and calculate the pattern database which provides the heuristic estimates for search. Some abstractions give rise to *non-surjective* homomorphisms, *i.e.* the abstract state space will have states which do not have pre-images in the original space. If possible, such homomorphisms should be avoided because the corresponding pattern database will contain entries which are never used.

Pattern databases proved to be useful to guide search in very large state spaces (Korf 1997; Culberson & Schaeffer 1996). Such pattern databases can automatically be obtained for spaces defined in PSVN by domain abstraction. A large scale experimental study (Holte & Hernádvölgyi 1999) suggests that the performance of a pattern database can be predicted from the amount of memory it requires. Encouraged by these early results, we continue to investigate properties of state spaces defined in PSVN and pattern databases obtained by abstractions.

Acknowledgment

The work presented in this paper is partially supported by an NSERC[3] postgraduate scholarship and an NSERC operating grant.

References

Culberson, J. C., and Schaeffer, J. 1994. Efficiently searching the 15-puzzle. Technical report, Department of Computer Science, University of Alberta.

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)* 402–416.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Holte, R. C., and Hernádvölgyi, I. T. 1999. A space-time tradeoff for memory-based heuristics. *To Appear in the Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99).*

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* 530–535.

Korf, R. E., and Reid, M. 1998. Complexity analysis of admissible heuristic search. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)* 305–310.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)* 700–705.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison & Wesley.

Prieditis, A. E. 1993. Machine discovery of effective admissible heuristics. *Machine Learning* 12:117–141.

Slaney, J., and Thiébaux, S. 1994. Adventures in blocks world. Technical report, Research School of Information Sciences and Engineering and Centre for Information Science Research, Australian National University. TR-ARP-7-94.

Valtora, M. 1984. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences* 47–59.

[3] *Natural Sciences and Engineering Research Council of Canada*

Appendix

Examples of Problem Spaces

$n \times m$ Sliding Tile Puzzle

The $n \times m$ sliding tile puzzle is played on an $n \times m$ grid of tiles of which one is missing. A neighboring tile can be slid into this missing one. The objective of the game is to bring the scrambled grid of tiles into a particular goal state. Figure 9 shows the *3* × *3* sliding tile puzzle,

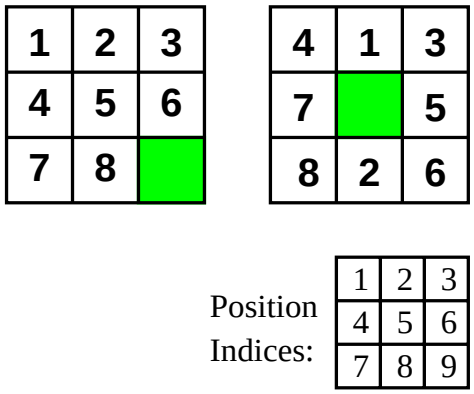

Figure 9: *3* × *3* Sliding Tile Puzzle: the goal state and a scrambled state.

with a goal and a scrambled state and the indices for the operator definitions. A state is represented as a vector of 9 labels. The labels correspond to the tiles, and the indices correspond to the positions shown in figure 9. The operators for the *3* × *3* sliding tile puzzle are defined as follows:

$$O = \left\{ \begin{array}{c} o_1, o_2, o_3, o_4, o_5, o_6, \\ o_7, o_8, o_9, o_{10}, o_{11}, o_{12}, \\ o_1^{-1}, o_2^{-1}, o_3^{-1}, o_4^{-1}, o_5^{-1}, o_6^{-1}, \\ o_7^{-1}, o_8^{-1}, o_9^{-1}, o_{10}^{-1}, o_{11}^{-1}, o_{12}^{-1} \end{array} \right\}$$

where

$$\begin{array}{l} o_1 :< 0, X, \_, \_, \_, \_, \_, \_, \_ > \rightarrow < X, 0, \_, \_, \_, \_, \_, \_, \_ > \\ o_2 :< \_, 0, X, \_, \_, \_, \_, \_, \_ > \rightarrow < \_, X, 0, \_, \_, \_, \_, \_, \_ > \\ o_3 :< \_, \_, \_, 0, X, \_, \_, \_, \_ > \rightarrow < \_, \_, \_, X, 0, \_, \_, \_, \_ > \\ o_4 :< \_, \_, \_, \_, 0, X, \_, \_, \_ > \rightarrow < \_, \_, \_, \_, X, 0, \_, \_, \_ > \\ o_5 :< \_, \_, \_, \_, \_, \_, 0, X, \_ > \rightarrow < \_, \_, \_, \_, \_, \_, X, 0, \_ > \\ o_6 :< \_, \_, \_, \_, \_, \_, \_, 0, X > \rightarrow < \_, \_, \_, \_, \_, \_, \_, X, 0 > \\ o_7 :< 0, \_, \_, X, \_, \_, \_, \_, \_ > \rightarrow < X, \_, \_, 0, \_, \_, \_, \_, \_ > \\ o_8 :< \_, 0, \_, \_, X, \_, \_, \_, \_ > \rightarrow < \_, X, \_, \_, 0, \_, \_, \_, \_ > \\ o_9 :< \_, \_, 0, \_, \_, X, \_, \_, \_ > \rightarrow < \_, \_, X, \_, \_, 0, \_, \_, \_ > \\ o_{10} :< \_, \_, \_, 0, \_, \_, X, \_, \_ > \rightarrow < \_, \_, \_, X, \_, \_, 0, \_, \_ > \\ o_{11} :< \_, \_, \_, \_, 0, \_, \_, X, \_ > \rightarrow < \_, \_, \_, \_, X, \_, \_, 0, \_ > \\ o_{12} :< \_, \_, \_, \_, \_, 0, \_, \_, X > \rightarrow < \_, \_, \_, \_, \_, X, \_, \_, 0 > \end{array}$$

Rubik's Cube

Rubik's Cube is one of the most famous combinatorial puzzles. The *3* × *3* × *3* cube (figure 10) consists of

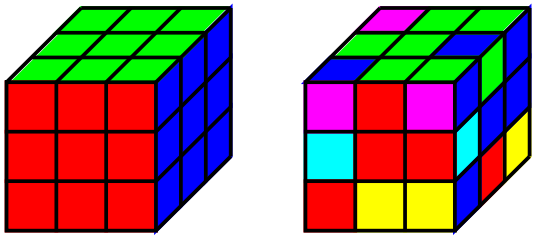

Figure 10: Rubik's Cube

8 corner cubies, 12 edge cubies and 6 middle cubies. The visible faces of these cubies are covered with colored stickers. The goal state is arranged such that all stickers on the same side have identical colors. Each face can be turned clockwise or counter clockwise 90 degrees (*quarter turns*). The state space consists of $43252003274489856000 = 4.3252 \cdot 10^{19}$ different states. $n \times n \times n$ versions for $n = 2, 3, 4, 5, 6$ are also available commercially as well as Megaminx, which is a dodecahedron rather than a cube. The middle pieces do not move with respect to each other. A state of Rubik's Cube can be represented as a vector of 48 labels, one for each of the 48 stickers[4]. The positions correspond to those on figure 11. O consists of 12 quarter turn

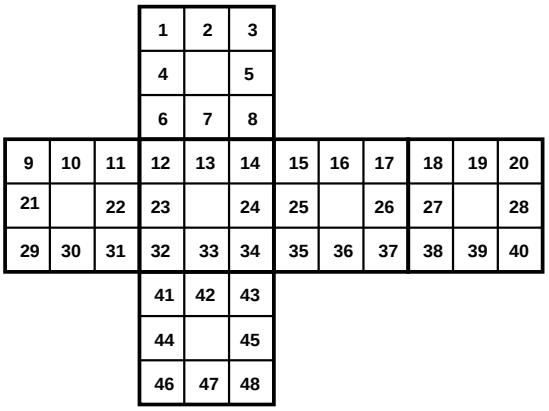

Figure 11: Indices of Rubik's Cube Stickers

[4] because the middle cubies do not move, they do not have to be represented in the states.

operations[5].

$$O = \left\{ \begin{array}{c} o_1, o_2, o_3, o_4, o_5, o_6, \\ o_1^{-1}, o_2^{-1}, o_3^{-1}, o_4^{-1}, o_5^{-1}, o_6^{-1} \end{array} \right\}$$

We show the definition of one of the (*top face*) quarter turn operators.

$$\begin{array}{cccccccccccc} & & < & A, & B, & C, & & & & & & \\ & & & D, & & E, & & & & & & \\ & & & F, & G, & H, & & & & & & \\ I, & J, & K, & L, & M, & N, & O, & P, & Q, & R, & S, & T, \\ -, & & -, & -, & & -, & -, & & -, & -, & & -, \\ -, & -, & -, & -, & -, & -, & -, & -, & -, & -, & -, & -, \\ & & & -, & -, & -, & & & & & & \\ & & & -, & & -, & & & & & & \\ & & & -, & -, & - & > & & & & & \end{array}$$

$$\rightarrow$$

$$\begin{array}{cccccccccccc} & & < & C, & E, & H, & & & & & & \\ & & & B, & & G, & & & & & & \\ & & & A, & D, & F, & & & & & & \\ R, & S, & T, & I, & J, & K, & L, & M, & N, & O, & P, & Q, \\ -, & & -, & -, & & -, & -, & & -, & -, & & -, \\ -, & -, & -, & -, & -, & -, & -, & -, & -, & -, & -, & -, \\ & & & -, & -, & -, & & & & & & \\ & & & -, & & -, & & & & & & \\ & & & -, & -, & - & > & & & & & \end{array}$$

Blocks World

The Blocks World is a famous benchmark problem for planning. An in-depth discussion of the problem and the cardinality of the state space can be found in (Slaney & Thiébaux 1994). There are n blocks which can be stacked on each other (figure 12). There is also a robot arm which can lift up a block and place it on top of another block or on the table.

Representing the Blocks World in a vector notation is not obvious because the relative order of the stacks on the table is irrelevant. We discuss two representations of Blocks World which overcomes this difficulty. The first form requires an exponential number of rules and gives rise to non-surjective homomorphisms which often have more spurious states in the abstract state than states in the original space. The second representation uses a polynomial number of rules but it also gives rise to non-surjective homomorphisms. However abstractions of this latter representation have a manageable number of states without pre-images and the size of the abstract space (including the spurious states) for coarse grained abstractions is usually smaller than the original space.

[5] sometimes a 180 degree turn is also counted as a single operation

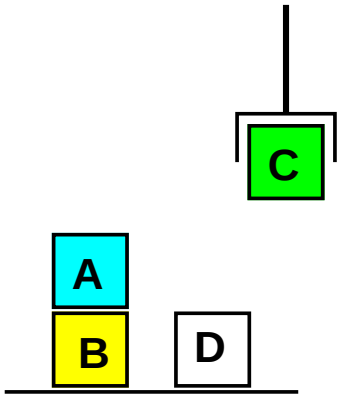

Figure 12: The 4 Blocks World

We represent each state in the n Blocks World with a vector of length $n+1$ and labels $L = \{0, 1, 2, ..., n\}$. The first n positions represent the blocks and the last position represents the robot arm. Having the label $k \neq 0$ on position $i \leq n$ means that block k is on top of block i and label k on position $n+1$ it means that block k is in the robot arm. Label 0 on position $i \leq n$ expresses that no block is on top of block i. 0 on position $n+1$ represents that the robot arm is empty. The configuration of blocks on figure 12 hence is written as $< 0, 1, 0, 0, 3 >$ for label assignments $A = 1$, $B = 2$, $C = 3$ and $D = 4$.

A rule that picks up block i from block j is defined as:

$$\begin{array}{ccccccccccc} < & ... & ,0, & ... & ,i, & , & ... & , & 0 & > \\ & & i & & j & & & , & n+1 & & \end{array}$$

$$\rightarrow$$

$$< \quad ... \quad ,\_, \quad ... \quad ,0, \quad ... \quad , \quad i \quad >$$

The inverse of the above operator would place block i on top of block j. To lift a block i from the table, the left hand side of the operator must make sure that that block i is *not* on top of any of the other blocks – which requires an exponential number of rules. Another solution is to extend PSVN to include negation.

The second representation uses $2n+1$ length vectors to describe a state and labels $L = \{0, 1, 2, ..., n+2\}$. Label $k > 1$ on position $i \leq n$ means that block $k-1$ is on top of block i. Label k on position $n+1$ means that block $k-1$ is in the robot arm. Label 0 – like in the previous notation – represents empty. The positions $n+2, ..., 2n+1$ can have labels either 0 or 1. Label 1 on position $i > n+1$ represents that block $i-n$ is on the table and label 0 on these positions means that block $i-n$ is not on the table. The configuration of blocks

on figure 12 is represented as $< 0, 2, 0, 0, 4, 0, 1, 0, 1 >$ for label associations $A = 1$, $B = 2$, $C = 3$ and $D = 4$.

A rule that picks up block i from block j is defined as:

$$\underset{i}{< .. \ ,0,} \quad \underset{j}{.. \ ,i+1,} \quad \underset{n+1}{.. \ ,0,} \quad \underset{n+i+1}{.. \ ,0,} \quad .. >$$

$$\rightarrow$$

$$< ... \ , \_, \ ... \ ,0, \ ... \ ,i+1, \ ... \ , \_, \ ... >$$

The inverse of the operator defined above puts block i from the robot arm onto top of block j.

The operator which picks up block i from the table is defined as

$$\underset{i}{< .. \ ,0,} \quad \underset{n+1}{.. \ ,0,} \quad \underset{n+i+1}{.. \ ,1,} \quad .. >$$

$$\rightarrow$$

$$< ... \ , \_, \ ... \ ,i+1, \ ... \ ,0, \ ... >$$

and its inverse places block i from the robot arm onto the table.

This work is dedicated to the memories of Rob's late father and István's late mother.