Efficient Pruning of Operators in Planning Domains

Anders Jonsson

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

Abstract. Many recent successful planners use domain-independent heuristics to speed up the search for a valid plan. An orthogonal approach to accelerating search is to identify and remove redundant operators. We present a domain-independent algorithm for efficiently pruning redundant operators prior to search. The algorithm operates in the domain transition graphs of multi-valued state variables, so its complexity is polynomial in the size of the state variable domains. We prove that redundant operators can always be replaced in a valid plan with other operators. Experimental results in standard planning domains demonstrate that our algorithm can reduce the number of operators as well as speed up search.

1 Introduction

Planning is the problem of finding a sequence of operators for moving from a start state to a goal state. The search space is usually very large, so most research in planning has focused on making search faster. The most successful approach has been to devise domain-independent heuristics for guiding the search along promising paths. Another approach, which has been explored less, is to identify redundant operators and remove them prior to search. There are usually several ways to reach the goal, and under certain conditions, some of these may be immediately discarded. Reducing the number of operators means reducing the branching factor, typically making search faster.

We present a domain-independent algorithm for identifying and pruning redundant operators in planning domains. We use the SAS^+ formalism [1] to represent planning domains using multi-valued state variables. The algorithm constructs domain transition graphs of individual state variables and performs search in the graphs to identify redundant operators. We prove that redundant operators can always be replaced in a valid plan with other operators. Thus, it is safe to prune the redundant operators. Experiments in standard planning domains demonstrate the utility of our approach.

Several other researchers have exploited domain knowledge to simplify the planning problem prior to search. Nebel et al. [2] designed an algorithm for removing irrelevant facts and operators, which works well in certain planning problems but is not solution-preserving. Scholz [3] defined a concept of redundant sequences of actions, restricted to sequences of length 2, and used it as a constraint to exclude plans that contained redundant sequences. Haslum and Jonsson [4] defined redundant operators as operators that can be replaced by operator sequences, and designed an algorithm for identifying and pruning redundant operators. Scholz [5] developed a technique for pruning operators similar to ours, using information about the local effect of operators. However, the author reported that the approach did not work well in LOGISTICS and BLOCKSWORLD, two domains in which our approach achieves good results. Vidal and Geffner [6] used inference to solve simple planning problems without performing search.

2 Notation

Let $\mathbf{V} = \{v_1, \ldots, v_n\}$ be a set of state variables, and let $\mathcal{D}(v_i)$ be the finite domain of state variable $v_i \in \mathbf{V}$. Let $\mathcal{D}_\mathbf{C} = \times_{v_i \in \mathbf{C}} \mathcal{D}(v_i)$ be the joint domain of a subset $\mathbf{C} \subseteq \mathbf{V}$ of state variables. We define a *context* $\mathbf{c} \in \mathcal{D}_\mathbf{C}$ as an assignment of values to the subset $\mathbf{C}$ of state variables. Let $\mathbf{c}[v_i] \in \mathcal{D}(v_i)$ be the value that context $\mathbf{c}$ assigns to state variable $v_i \in \mathbf{C}$. We use the convention of capitalizing a context to denote its associated subset of state variables. For example, $\mathbf{C}$ is the subset of state variables associated with context $\mathbf{c}$. A *state* $\mathbf{s} \in \mathcal{D}_\mathbf{V}$ assigns a value to each state variable in $\mathbf{V}$.

We define two operations on contexts. Let $f_\mathbf{W}(\mathbf{c})$ be the projection of context $\mathbf{c}$ onto the subset $\mathbf{W} \subseteq \mathbf{V}$ of state variables. The result of $f_\mathbf{W}(\mathbf{c})$ is a context $\mathbf{x}$ such that $\mathbf{X} = \mathbf{C} \cap \mathbf{W}$ and $\mathbf{x}[v_i] = \mathbf{c}[v_i]$ for each $v_i \in \mathbf{X}$. Also, let $\mathbf{c} \oplus \mathbf{w}$ be the composition of contexts $\mathbf{c}$ and $\mathbf{w}$. The result of $\mathbf{c} \oplus \mathbf{w}$ is a context $\mathbf{x}$ such that $\mathbf{X} = \mathbf{C} \cup \mathbf{W}$, $\mathbf{x}[v_i] = \mathbf{w}[v_i]$ for each $v_i \in \mathbf{W}$ and $\mathbf{x}[v_i] = \mathbf{c}[v_i]$ for each $v_i \in \mathbf{C} - \mathbf{W}$. Note that the right operand overrides the values of the left operand.

A SAS$^+$ planning problem is a tuple $\mathcal{P} = \langle \mathbf{V}, \mathbf{s}_I, \mathbf{c}_G, A \rangle$, where $\mathbf{V}$ is the set of state variables, $\mathbf{s}_I$ is an initial state, $\mathbf{c}_G$ is a goal context, and $A = \{a_1, \ldots, a_m\}$ is a set of grounded operators. Each operator $a_j \in A$ has the form $\langle \mathbf{pre}_j, \mathbf{post}_j, \mathbf{prv}_j \rangle$, where the contexts $\mathbf{pre}_j$, $\mathbf{post}_j$, and $\mathbf{prv}_j$ denote the pre-, post- and prevail-condition of a_j, respectively. For each $a_j \in A$, $\mathbf{Pre}_j = \mathbf{Post}_j$ and $\mathbf{Pre}_j \cap \mathbf{Prv}_j = \emptyset$. Operator a_j is applicable in state $\mathbf{s}$ if $f_{\mathbf{Pre}_j}(\mathbf{s}) = \mathbf{pre}_j$ and $f_{\mathbf{Prv}_j}(\mathbf{s}) = \mathbf{prv}_j$. The result of successfully applying a_j in state $\mathbf{s}$ is $\mathbf{s} \oplus \mathbf{post}_j$. The objective is to find a plan, i.e., a sequence of operators in A^*, where $*$ is the Kleene star, for moving the system from the initial state $\mathbf{s}_I$ to a state $\mathbf{s}$ such that $f_{\mathbf{C}_G}(\mathbf{s}) = \mathbf{c}_G$. In this paper, we study the class of planning problems with unary operators, i.e., for each operator $a_j \in A$, $|\mathbf{Pre}_j| = |\mathbf{Post}_j| = 1$. In this case, it is possible to form the set $A_i = \{a_j \in A \mid \mathbf{Pre}_j = \mathbf{Post}_j = \{v_i\}\}$ of operators that change the value of state variable $v_i \in \mathbf{V}$.

3 Context Subsumption

To prune operators, we are interested in determining when the prevail-condition of one operator causes the prevail-condition of another operator to hold. We formalize this idea using context subsumption:

Definition 1. *A context* $\mathbf{c}$ subsumes *a context* $\mathbf{z}$*, which we denote* $\mathbf{c} \sqsupseteq \mathbf{z}$*, if and only if* $\mathbf{C} \subseteq \mathbf{Z}$ *and* $f_\mathbf{C}(\mathbf{z}) = \mathbf{c}$*.*

If a context subsumes another, any state that satisfies the latter will also satisfy the former. In other words, if $\mathbf{c} \sqsupseteq \mathbf{z}$, for any state $\mathbf{s}$ such that $f_\mathbf{Z}(\mathbf{s}) = \mathbf{z}$ it follows that $f_\mathbf{C}(\mathbf{s}) = \mathbf{c}$.

We also introduce the idea of context paths, which are sequences of contexts.

Algorithm 1 SUBSUMES $(\mathcal{C}^{\mathcal{A}}, \mathcal{C}^{\mathcal{A}'}, i, j)$

```
1: $subsumes \leftarrow i > |\mathcal{C}^{\mathcal{A}}|$
2: for $(k \leftarrow 0; \mathbf{not}(subsumes)$ and $k \leq (|\mathcal{C}^{\mathcal{A}'}| - j) - (|\mathcal{C}^{\mathcal{A}}| - i); k \leftarrow k + 1)$
3:     if SUBSUMES($\mathbf{c}_i^{\mathcal{A}}, \mathbf{c}_{j+k}^{\mathcal{A}'}$)
4:         $subsumes \leftarrow$ SUBSUMES $(\mathcal{C}^{\mathcal{A}}, \mathcal{C}^{\mathcal{A}'}, i + 1, j + k + 1)$
5: return $subsumes$
```

Definition 2. *A context path $\mathcal{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_k\}$ is a sequence of $|\mathcal{C}| = k$ contexts $\mathbf{c}_i$ such that for each $i \in [2, \ldots, k]$, $\mathbf{c}_{i-1} \not\sqsupseteq \mathbf{c}_i$ and $\mathbf{c}_i \not\sqsupseteq \mathbf{c}_{i-1}$.*

In other words, no two neighboring contexts in a context path subsume each other. We extend the idea of context subsumption to context paths:

Definition 3. *A context path $\mathcal{C}$ subsumes a context path $\mathcal{Z}$, which we denote $\mathcal{C} \sqsupseteq \mathcal{Z}$, if and only if $|\mathcal{C}| \leq |\mathcal{Z}|$ and there exist $j_1, \ldots, j_k$, $k = |\mathcal{C}|$, such that for each $i \in [2, \ldots, k]$, $j_{i-1} < j_i$, and such that for each $i \in [1, \ldots, k]$, $\mathbf{c}_i \sqsupseteq \mathbf{z}_{j_i}$.*

In other words, there exists a subsequence of $|\mathcal{C}|$ contexts in $\mathcal{Z}$, preserving their internal order, such that each of them is subsumed by the corresponding context in $\mathcal{C}$. Algorithm 1 describes a subroutine for determining whether a context path $\mathcal{C}^{\mathcal{A}}$ subsumes a context path $\mathcal{C}^{\mathcal{A}'}$. When calling the algorithm, the indices i and j should be initialized to 1.

Let $\mathcal{A} = \{a_1, \ldots a_l\}$ be a sequence of operators in A_i^*. Given $\mathcal{A}$, $\{\mathbf{prv}_1, \ldots, \mathbf{prv}_l\}$ is the sequence of prevail-conditions of the operators in $\mathcal{A}$. Since neighboring prevail-conditions may subsume each other, this sequence may not be a context path. However, it is easy to convert it into a context path by merging neighboring prevail-conditions that subsume each other using the $\oplus$ operator. Let $\mathcal{C}^{\mathcal{A}}$ denote the context path of prevail-conditions derived from the operator sequence $\mathcal{A}$. Each context $\mathbf{c}_i^{\mathcal{A}}$ in $\mathcal{C}^{\mathcal{A}}$ corresponds to one or several operators in $\mathcal{A}$ that are applicable in states that satisfy $\mathbf{c}_i^{\mathcal{A}}$.

Theorem 1. *Let $\mathcal{A} = \{a_1, \ldots a_l\}$ and $\mathcal{A}' = \{a'_1, \ldots a'_m\}$ be two valid sequences of operators in A_i^* such that $\mathbf{pre}_1 = \mathbf{pre}'_1$ and $\mathbf{post}_l = \mathbf{post}'_m$. Assume that $\mathcal{A}'$ is part of a valid plan and that for each $j \in [1, \ldots, m-1]$, operator a'_j is not in the support of the prevail-condition of any operator in the plan. Then if $\mathcal{C}^{\mathcal{A}} \sqsupseteq \mathcal{C}^{\mathcal{A}'}$ it is possible to substitute $\mathcal{A}$ for $\mathcal{A}'$ in the plan without invalidating the plan.*

Proof. From the definition of $\mathcal{C}^{\mathcal{A}} \sqsupseteq \mathcal{C}^{\mathcal{A}'}$ it follows that there exists $j_1, \ldots, j_k$, $k = |\mathcal{C}^{\mathcal{A}}|$, such that for each $i \in [1, \ldots, k]$, $\mathbf{c}_i^{\mathcal{A}} \sqsupseteq \mathbf{c}_{j_i}^{\mathcal{A}'}$. For each $\mathbf{c}_{j_i}^{\mathcal{A}'}$, there is some operator $a'_j \in \mathcal{A}'$ that is only applicable in context $\mathbf{c}_{j_i}^{\mathcal{A}'}$. Therefore, when a'_j is applied in the plan, $\mathbf{c}_{j_i}^{\mathcal{A}'}$ must hold. But since $\mathbf{c}_i^{\mathcal{A}} \sqsupseteq \mathbf{c}_{j_i}^{\mathcal{A}'}$, this means that $\mathbf{c}_i^{\mathcal{A}}$ also holds, which makes all operators corresponding to $\mathbf{c}_i^{\mathcal{A}}$ applicable. Thus, at some point in the plan, each operator in $\mathcal{A}$ is applicable, making it possible to replace all operators in $\mathcal{A}'$ with the operators in $\mathcal{A}$. Since $\mathcal{A}$ and $\mathcal{A}'$ begin and end with the same value for v_i, this does not compromise operators that change the value of v_i at earlier or later points in the plan. Since no operator in $\mathcal{A}'$ (except possibly the last) is in the support of the prevail-condition of any other operator in the plan, the substitution does not invalidate the plan.

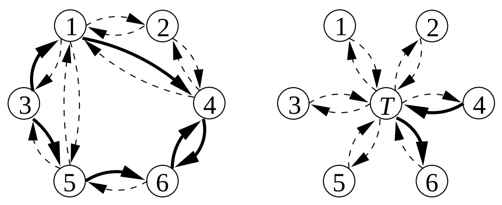

Fig. 1. Domain transition graphs for v_T and v_P

4 Pruning Operators

Using Theorem 1, we devise an algorithm for pruning operators from the planning domain. First, construct the domain transition graph for each state variable. The domain transition graph for state variable $v_i \in \mathbf{V}$ is a graph with one node per value in $\mathcal{D}(v_i)$. For each operator $a_j \in A_i$, add an edge in the domain transition graph from $\mathbf{pre}_j$ to $\mathbf{post}_j$. Go through the remaining operators of the planning domain to determine the subset of values $\mathcal{Q}_i \subseteq \mathcal{D}(v_i)$ that appear in the prevail-condition of any operator in A.

Starting from the projected initial state $\mathbf{s}_I[v_i] \in \mathcal{D}(v_i)$, do a breadth-first search over the domain transition graph. For each $d \in \mathcal{Q}_i$ visited during search, store the different operator sequences in A_i^* for reaching d from $\mathbf{s}_I[v_i]$. If the context path of one operator sequence subsumes the context path of another, it follows from Theorem 1 that we can replace the latter operator sequence with the former. If two context paths subsume each other (i.e., they are equal), we break ties by preferring the shorter of the two.

For each value $d \in \mathcal{Q}_i$ visited during the search from $\mathbf{s}_I[v_i]$, repeat breadth-first search starting at d. If the goal context $\mathbf{c}_G$ specifies a value for v_i, it is also necessary to determine the possible operator sequences for reaching the value $\mathbf{c}_G[v_i] \in \mathcal{D}(v_i)$. We are left with a minimal set of operator sequences necessary to move between any two values in $\mathcal{Q}_i$ (possibly starting in $\mathbf{s}_I[v_i]$ and/or ending in $\mathbf{c}_G[v_i]$). Any operator that does not appear in any of these operator sequences can be safely pruned.

To illustrate the algorithm for pruning operators we use a problem from the LOGISTICS domain. In LOGISTICS, trucks and airplanes are used to move one or several packages to designated locations. We choose a problem with one truck and one package, which can be modelled as a SAS$^+$ instance with two state variables v_T and v_P, representing the location of the truck and the package, respectively. The problem has six locations $1, \ldots, 6$, so $\mathcal{D}(v_T) = \{1, \ldots, 6\}$ and $\mathcal{D}(v_P) = \{1, \ldots, 6, T\}$, where T denotes that the package is inside the truck. Figure 1 shows the domain transition graphs of v_T and v_P. The initial state is $(v_T = 3, v_P = 4)$ and the goal context is $(v_P = 6)$.

First run the algorithm on the domain transition graph for v_P. The projected initial state is 4, and no operator of the problem has a prevail-condition on v_P, so the only target value is the projected goal context 6. One operator sequence for reaching 6 from 4 is $(pickup(4), putdown(6))$ with associated context path $((v_T = 4), (v_T = 6))$. Any other operator sequence for reaching 6 drops the package in an intermediate location along the way, so its associated context path is subsumed by $((v_T = 4), (v_T = 6))$. Thus, we can prune all operators other than $pickup(4)$ and $putdown(6)$.

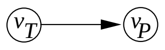

Fig. 2. Causal graph of the LOGISTICS domain

Next run the algorithm on the domain transition graph for v_T. After pruning operators for v_P, only $(v_T = 4)$ and $(v_T = 6)$ appear as prevail-conditions of other operators. Thus, we need to reach the values 4 and 6 from the projected initial state 3. Operators for moving the truck do not require any prevail-conditions, so each associated context path will by definition be empty. In this case, we break ties by selecting the shortest operator sequences: from 3 via 1 to 4, and from 3 via 5 to 6. We repeat the search from 4 and 6 to find the shortest operator sequences for reaching 6 and 4, respectively.

The pruned operators appear as broken edges in Figure 1. In this problem, our algorithm prunes 20 out of 28 operators. An advantage of the algorithm is that it handles each state variable separately. Typically, the domain transition graph of a single state variable has limited size, which bounds the complexity of the algorithm.

Note, however, that the order of pruning is important. If we start pruning operators for v_T, each value of v_T appears in the prevail-condition of other operators, since we can pick up and put down the package anywhere. Thus, we would not prune any operator for v_T. To determine the order of pruning operators, we construct the causal graph of the domain. The causal graph has one node per state variable v_i, and an edge between v_j and v_i indicates that there exists an operator in A_i with a prevail-condition on v_j.

To get rid of cycles in the causal graph, we compute the strongly connected components of the graph. We construct a component graph with the strongly connected components as nodes and edges mapped from the causal graph. The component graph is by definition acyclic. Figure 2 shows the causal graph of the LOGISTICS domain in our example. Since the causal graph is acyclic, each node is its own strongly connected component, so the component graph is identical to the causal graph in this case.

Operators that affect state variables in a strongly connected component have prevail-conditions on state variables in parent strongly connected components. Therefore, it makes sense to prune operators for the strongly connected components in inverse topological order. In the example, that means first pruning operators for v_P, then for v_T. Algorithm 2 describes the final algorithm for pruning operators.

Theorem 1 ensures that our algorithm preserves a valid solution. In the following theorem, we prove that in many cases our algorithm also preserves the optimal solution.

Theorem 2. *If redundant operator sequences are always at least as long as the operator sequences with which they can be replaced, the pruned operator set generated by our algorithm preserves the optimal solution.*

Proof. Assume that a pruned operator is part of a valid plan. To be pruned, this operator has to be part of a redundant operator sequence. From Theorem 1 it follows that we can replace the redundant operator sequence with a sequence of non-redundant operators without invalidating the plan. If the redundant operator sequence is at least as long as the replacing operator sequence, the transformed plan is shorter or equal in length to

Algorithm 2 PRUNE($\mathcal{P}$)

```
 1: construct the causal graph of the domain
 2: compute strongly connected components ($\mathbf{SCC}$) of graph
 3: construct component graph of $\mathbf{SCC}$s
 4: for each $\mathbf{SCC}$ in inverse topological order
 5:   repeat until convergence
 6:     for each $v_i \in \mathbf{SCC}$
 7:       determine set $\mathcal{Q}_i \subseteq \mathcal{D}(v_i)$ of prevail-condition values
 8:       let $L \leftarrow \{\mathbf{s}_I[v_i]\} \cup \mathcal{Q}_i$
 9:       for each $d \in L$
10:         do breadth-first search in domain transition graph
11:         find minimal operator sequences to $\mathcal{Q}_i \cup \{\mathbf{c}_G[v_i]\}$
12:       prune any operator not in a minimal operator sequence
```

the original plan. Thus, it is always possible to generate an optimal plan using only operators that are not pruned.

Note that the condition in Theorem 2 does not always hold. When we derive the context path of prevail-conditions associated with an operator sequence, we use the $\oplus$ operator to merge neighboring contexts that subsume each other. Thus, a long operator sequence may have a short associated context path. In particular, it is possible that a redundant operator sequence is shorter than the replacing operator sequence.

5 Extended Operators

When translating planning problems to multi-valued representations, it is sometimes possible to infer the truth value of certain predicates. The translated operators may be applicable for a subset of values of a state variable. We introduce the notion of extended operators, which are operators whose prevail-conditions specify sets of values on state variables. Extended operators can compactly represent activities that it would otherwise take many regular operators to represent.

Let $2^{\mathbf{C}} = \times_{v_i \in \mathbf{C}} 2^{\mathcal{D}(v_i)}$ be the joint domain power set of a subset $\mathbf{C} \subseteq \mathbf{V}$ of state variables. An *extended* context $\mathbf{c}^e \in 2^{\mathbf{C}}$ assigns a subset of values $\mathbf{c}^e[v_i] \subseteq \mathcal{D}(v_i)$ to each state variable $v_i \in \mathbf{C}$. An extended operator $a_j^e = \langle \mathbf{pre}_j, \mathbf{post}_j, \mathbf{prv}_j^e \rangle$ has an extended context $\mathbf{prv}_j^e$ describing the prevail-condition. The extended operator a_j^e is applicable in any state $\mathbf{s}$ such that for each $v_i \in \mathbf{Prv}_j^e$, $\mathbf{s}[v_i] \in \mathbf{prv}_j^e[v_i]$.

We illustrate the benefit of using extended operators using the BLOCKSWORLD domain. In BLOCKSWORLD a robot hand has to rearrange a group of blocks to achieve a designated target configuration. Helmert [7] showed how to translate planning problems from PDDL to multi-valued formulations. The idea is to identify invariants, which are sets of predicates such that precisely one predicate in each set is true at any point. Table 1 shows the invariants of an instance of BLOCKSWORLD with four blocks.

To obtain a multi-valued planning problem, define a state variable for each invariant with the set of predicates as its domain. Once a predicate has been included in the domain of a variable, it is excluded from all other invariants. In the example, this creates

Table 1. Invariants in BLOCKSWORLD with four blocks

1. $\mathsf{holding(a), ontable(a), on(a,a), on(a,b), on(a,c), on(a,d)}$
2. $\mathsf{holding(b), ontable(b), on(b,a), on(b,b), on(b,c), on(b,d)}$
3. $\mathsf{holding(c), ontable(c), on(c,a), on(c,b), on(c,c), on(c,d)}$
4. $\mathsf{holding(d), ontable(d), on(d,a), on(d,b), on(d,c), on(d,d)}$
5. $\mathsf{holding(a), clear(a), on(a,a), on(b,a), on(c,a), on(d,a)}$
6. $\mathsf{holding(b), clear(b), on(a,b), on(b,b), on(c,b), on(d,b)}$
7. $\mathsf{holding(c), clear(c), on(a,c), on(b,c), on(c,c), on(d,c)}$
8. $\mathsf{holding(d), clear(d), on(a,d), on(b,d), on(c,d), on(d,d)}$
9. $\mathsf{holding(a), holding(b), holding(c), holding(d), handempty}$

four state variables v_1 through v_4 whose domains equal invariants 1-4. The remaining five invariants now contain a single predicate ($\mathsf{clear(x)}$ and $\mathsf{handempty}$, respectively), since all others have been excluded.

Helmert introduces five binary state variables corresponding to invariants 5-9. However, it is possible to infer the true predicate of these invariants from the first four state variables. For example, if $\mathsf{on(b,a)}$ holds in invariant 2, $\mathsf{clear(a)}$ is false in invariant 5. If no block is on top of c and c is not held, $\mathsf{clear(c)}$ is true in invariant 7. Thus, the problem is completely specified by state variables v_1 through v_4.

When translating a PDDL operator to the multi-valued representation, we can simply ignore add and delete effects on inferred predicates. However, we cannot ignore inferred predicates in the pre-condition. As an example, consider the operator $\mathsf{stack(a,b)}$ with pre-condition $\mathsf{holding(a)}$ and $\mathsf{clear(b)}$. The operator deletes $\mathsf{holding(a)}$ and $\mathsf{clear(b)}$ and adds $\mathsf{on(a,b)}$, $\mathsf{clear(a)}$ and $\mathsf{handempty}$. Delete and add effects on $\mathsf{clear(b)}$, $\mathsf{clear(a)}$ and $\mathsf{handempty}$ can be ignored since they are inferred. Consequently, the pre-condition of the translated operator is $v_1 = \mathsf{holding(a)}$ and the post-condition is $v_1 = \mathsf{on(a,b)}$.

In contrast, the inferred predicate $\mathsf{clear(b)}$ in the pre-condition cannot be ignored. Since $\mathsf{clear(b)}$ is true, it follows from invariant 6 that $\mathsf{holding(b)}$ is false and no block is on top of b. Since $\mathsf{holding(a)}$ is true, it follows from invariant 5 that no block is on top of a, and from invariant 9 that no other block is held. Thus, the prevail-condition of the translated operator is an extended context on $\{v_2, v_3, v_4\}$ such that $v_2 \in \{\mathsf{ontable(b), on(b,c), on(b,d)}\}$, $v_3 \in \{\mathsf{ontable(c), on(c,c), on(c,d)}\}$ and $v_4 \in \{\mathsf{ontable(d), on(d,c), on(d,d)}\}$. This operator could be represented using $3^3 = 27$ regular SAS$^+$ operators, but the extended operator is clearly more compact.

It is easy to modify our pruning algorithm to planning problems with extended operators. First, we modify the definition of subsumption to include extended contexts:

Definition 4. *An extended context* $\mathbf{c}^e$ *subsumes an extended context* $\mathbf{z}^e$*, which we denote* $\mathbf{c}^e \sqsupseteq \mathbf{z}^e$*, if and only if* $\mathbf{C} \subseteq \mathbf{Z}$ *and for each* $v_i \in \mathbf{C}$*,* $\mathbf{z}^e[v_i] \subseteq \mathbf{c}^e[v_i]$.

As before, if $\mathbf{c}^e \sqsupseteq \mathbf{z}^e$, any state $\mathbf{s}$ that satisfies $\mathbf{z}^e$ also satisfies $\mathbf{c}^e$. The definitions of extended context paths and subsumption of extended context paths are analogous to Definitions 2 and 3.

In the domain transition graph for a state variable, the only difference is that prevail-conditions are now sets of nodes. Starting from the projected initial state, determine the

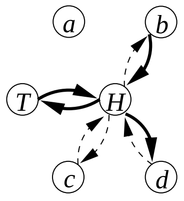

Fig. 3. Domain transition graph for BLOCKSWORLD

minimal operator sequences for reaching any node in each prevail-condition. Store each node reached this way and repeat the process from each such node.

Figure 3 shows the domain transition graph of state variable v_1 in BLOCKSWORLD. Block a can either be held (H), on the table (T) or on top of blocks $a - d$. Suppose that the initial state is $v_1 = \mathsf{on}(\mathsf{a}, \mathsf{b})$ and that the goal context specifies $v_1 = \mathsf{on}(\mathsf{a}, \mathsf{d})$. It turns out that $\mathsf{ontable}(\mathsf{a})$ is part of every extended prevail-condition on variable v_1. In addition, the prevail-condition of the operator for putting block a on the table subsumes the prevail-condition of any operator that stacks block a on top of another block.

The modified algorithm finds that it is always possible to put block a on the table instead of stacking a on top of another block. In addition, putting a on the table satisfies the prevail-condition of each extended operator of the domain. Thus, the operator sequences for stacking block a on an intermediate block are redundant. The minimal operator sequences only include the solid edges in the figure. All broken edges correspond to operators that are pruned by the algorithm.

6 Experimental Results

We ran experiments with our algorithm in three domains taken from the International Planning Competition: LOGISTICS, BLOCKSWORLD and DRIVERLOG. DRIVERLOG is an extension of LOGISTICS that includes the problem of allocating drivers to trucks. All three domains can be translated to SAS$^+$ with extended unary operators. In each domain, we ran the algorithm across a range of problem instances. We used the translator of Helmert [7] to identify invariants, and wrote our own code to translate the PDDL code into SAS$^+$ with extended operators. Table 2 shows the number of state variables, operators, and operators pruned in each of the problem instances.

In LOGISTICS, the algorithm pruned about half of the operators. In BLOCKSWORLD, the algorithm did very well, pruning up to 95% of the operators. Finally, in DRIVERLOG, the algorithm did not do as well, pruning 10-30% of the operators. For comparison, Haslum and Jonsson [4] reported a minimal reduced set of 420 operators in Blocks15, whereas our algorithm found a reduced set of 48 operators in the same problem instance, guaranteed to preserve solution existence.

For each problem instance, we tested how the reduced operator set affects search. We ran the Fast Downward planner [8] on the original problem instance and the problem instance with pruned operators. Since Fast Downward cannot handle extended opera-

Table 2. Results of operator pruning and Fast Downward search

	Operator pruning			Fast Downward search			
				Original		Pruned	
Problem	Variables	Operators	Pruned	Time	Length	Time	Length
Logistics5	9	78	50	0.01	32	0.00	32
Logistics10	17	308	254	0.05	55	0.02	55
Logistics15	22	650	370	0.21	97	0.10	97
Logistics20	30	1274	714	0.53	135	0.24	137
Logistics25	39	2664	1300	1.85	190	0.94	181
Logistics30	43	3290	1452	2.92	230	1.60	237
Logistics35	51	4740	2420	5.12	233	2.63	232
Blocks5	5	50	36	0.01	12	0.00	12
Blocks10	10	200	166	0.07	48	0.03	34
Blocks15	15	450	402	0.45	228	0.09	52
Blocks20	20	800	728	0.60	192	0.12	74
Blocks25	25	1250	1160	1.95	326	0.41	84
Blocks30	30	1800	1696	3.27	284	1.04	104
Blocks35	35	2450	2322	10.67	404	1.95	134
Blocks40	40	3200	3054	unsolved		1.35	138
DriverLog3	8	120	24	0.01	15	0.01	15
DriverLog6	11	222	78	0.05	13	0.04	13
DriverLog9	11	384	108	0.13	63	0.10	39
DriverLog12	11	948	90	1.61	108	1.62	102

tors, we reverted to Fast Downward's multi-valued translation prior to search. Table 2 shows the search time (in seconds) and the resulting plan length in the two cases.

Unsurprisingly, the speedup in search time was largest in BLOCKSWORLD. In the largest instance of BLOCKSWORLD, Fast Downward failed to solve the problem with the original operator set, but solved the problem quickly with the pruned operator set. In LOGISTICS, the pruned operator set cut the search time roughly in half, while in DRIVERLOG it did not have a significant effect. Overall, the reduction in search time seems proportional to the number of operators pruned. This is to be expected since the planner spends most of its time evaluating heuristics along different branches. Reducing the branching factor should reduce search time by an equivalent factor. More surprising was the fact that the resulting plan length in BLOCKSWORLD was significantly reduced, sometimes by as much as 75%.

7 Conclusion

We have presented a novel algorithm for identifying and pruning operators in planning problems prior to search. The algorithm constructs domain transition graphs of multi-valued state variables and performs search in the graphs to identify redundant operator sequences. The pruned operator set generated by the algorithm is solution-preserving, and under certain conditions it also preserves the optimal solution. We modified the algorithm to allow for extended operators whose prevail-conditions specify sets of values on state variables. Experimental results indicate that our approach can significantly speed up search in some planning problems.

In the future, we would like to extend the algorithm to planning problems with non-unary operators. Most planning problems cannot be translated to a multi-valued representation with unary operators, even when extended operators are used. The trick is to handle non-unary operators without significantly increasing the complexity of the algorithm. Ideally, the algorithm should still be able to identify redundant operator sequences using the domain transition graphs of individual state variables.

Another interesting approach to explore is the notion of objects in multi-valued representations. For example, in LOGISTICS, two trucks that operate within the same area are perceived as two different objects, so our algorithm will consider their operator sequences to be different. However, transporting a package using one truck is functionally equivalent to transporting the packing using the other truck. Even though the PDDL language includes a notion of objects, this notion is not preserved during the translation to multi-valued representations. If the SAS$^+$ formalism included a notion of objects, our algorithm could potentially prune even more operators.

References

1. Bäckström, C., Nebel, B.: Complexity results for SAS$^+$ planning. Computational Intelligence **11**(4) (1995) 625–655
2. Nebel, B., Dimopoulos, Y., Koehler, J.: Ignoring irrelevant facts and operators in plan generation. In: Proceedings of the 4th European Conference on Planning. (1997) 338–350
3. Scholz, U.: Action constraints for planning. In: Proceedings of the 5th European Conference on Planning. (1999) 148–158
4. Haslum, P., Jonsson, P.: Planning with Reduced Operator Sets. In: Proceedings of the 5th International Conference on Automated Planning and Scheduling. (2000) 150–158
5. Scholz, U.: Reducing Planning Problems by Path Reduction. Ph.D Thesis, Darmstadt University of Technology, Darmstadt, Germany (2004)
6. Vidal, V., Geffner, H.: Solving Simple Planning Problems with More Inference and No Search. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming. (2005) 682–696
7. Helmert, M.: Solving Planning Tasks in Theory and Practice. Ph.D Thesis, Albert-Ludwigs-Universität, Freiburg, Germany (2006)
8. Helmert, M.: The Fast Downward Planning System. Journal of Artificial Intelligence Research **26** (2006) 191–246