AI Communications 27 (2014) 363–383
DOI 10.3233/AIC-140605
IOS Press

Automatic move pruning for single-agent search

Robert C. Holte * and Neil Burch
Computing Science Department, University of Alberta, Edmonton, AB, Canada
E-mails: {holte, nburch}@cs.ualberta.ca

Abstract. Move pruning is a low-overhead technique for reducing search cost in single-agent search problems by avoiding the generation of duplicate states. Redundant sequences of moves, where the effect of one sequence is provably identical to some other sequence of moves, are suppressed during search. We present an algorithm for automatically identifying redundant move sequences in a general class of single-agent search problems, and a method for selecting redundant move sequences to prune during search. We demonstrate that the redundant move sequences which are to be pruned must be chosen carefully, and give experimental results using our move pruning method which show a speedup of multiple orders of magnitude in a variety of domains. Finally, we give theoretical results on conditions where move pruning does, and does not, affect the correctness of different search algorithms.

Keywords: Heuristic search, singe-agent search problem, move pruning

1. Introduction

In a single-agent search problem (SASP), the aim is to find a least-cost sequence of moves that transforms a given state (the "start state") into a state that satisfies the given goal conditions, where a move consists of applying an operator to the current state. Most general-purpose algorithms for solving SASPs fall into one of two broad categories: linear-memory search algorithms, such as depth-first search and IDA* [21], and full-memory search algorithms, such as A* [14]. Full-memory search algorithms store all the states they generate, whereas linear-memory search algorithms store only the states on the path currently being explored.

Linear-memory algorithms, in their pure form, can detect cycles (sequences of moves that lead from a state back to itself) but not arbitrary "transpositions" (different paths leading to the same state). This means these algorithms may generate the same state numerous times and may even repeat the search beyond the state numerous times. This can cause these algorithms to require exponential time to explore a polynomial-sized space such as an $m \times m$ grid.

Eliminating the unnecessary work caused by transpositions is the aim of the research reported in this paper. Our approach builds on the work of Taylor and Korf [30,31]. They introduced a method for automatically analyzing an SASP to identify operator sequences that are guaranteed to be transpositions. The redundant sequences identified in this way were encoded in a finite state machine (FSM) which was used, during search, to avoid generating the redundant sequences. In particular, it disallowed ("pruned") the use of an operator after a specific sequence of operators had been executed. They used the technique in IDA* on Rubik's Cube and on two sizes of the sliding-tile puzzle (4×4 and 5×5), getting large reductions in the number of generated states for a very modest cost per state.

The combination of large savings and automated analysis make Taylor and Korf's method a compelling tool for improving the performance of linear-memory algorithms. There is, however, one shortcoming. The method they used to identify redundant operator sequences only applies to SASPs in which the operators have no preconditions (every operator can be applied to every state), such as Rubik's Cube. In order to apply their method to the sliding-tile puzzle, where operators have a precondition that tests the location of the blank, they devised a workaround that is entirely specific to the sliding-tile puzzle [30].

The main contribution of the present paper is to fully overcome this shortcoming. We present an algo-

*Corresponding author: Robert C. Holte, Computing Science Department, University of Alberta, Edmonton, AB T6G 2E8, Canada. E-mail: holte@cs.ualberta.ca.

rithm for automatically identifying redundant operator sequences in a general class of SASPs. Given an SASP description in the PSVN language (described in the Appendix), our method builds and analyzes a tree of "macro-rules", where each macro-rule describes the collective preconditions and net effects of an entire operator sequence. Using this tree, we generate a list of redundant operator sequences which can be encoded and used in exactly the same manner as Taylor and Korf's FSM. The effectiveness and generality of our method for speeding up depth-first search is demonstrated experimentally on a variety of domains whose encoding in PSVN, in most cases, requires operators with preconditions. The preprocessing required to identify redundant operator sequences is, in almost all cases, very rapid, and the use of move pruning often results in search being faster by orders of magnitude.

Pruning moves based on redundant operator sequences can be "unsafe" in some circumstances: if done incorrectly, or in combination with certain other pruning methods, it can eliminate all the least-cost paths between some pairs of states. We provide a proof that our method of choosing move sequences to prune is safe, so that for any pair of states, it is guaranteed to leave unpruned at least one least-cost path between them. We also show that our move pruning method is safe in conjunction with cycle detection and heuristic cutoff pruning.

Full-memory search algorithms have a simple mechanism for detecting that a state has been reached for a second time. For each state that is generated they record its distance from the start state. If the state is generated again by a path that is cheaper than the recorded distance, the distance is updated and the state is "re-opened" with a priority based on the new distance. Otherwise the new path to the state is ignored. We refer to this as "duplicate detection". It might seem that move-pruning has nothing to offer to systems that do duplicate detection, but, in fact, there are potential advantages for doing so. These are given in Section 6.1. Unfortunately, we will show that move pruning is, in general, not safe to use in conjunction with duplicate detection. In Section 6.3 we prove two conditions that must hold whenever move pruning is not safe to use with duplicate detection and discuss circumstances in which each of these conditions might not hold, i.e. circumstances in which it would be safe to use move pruning in conjunction with duplicate detection.

This paper combines three previously published papers. Our move pruning method was first described in 2011 [4]. After the publication of that paper we discovered that the method, as we had implemented it, was not safe: it would sometimes eliminate all least-cost paths from start to goal. This led to the second paper [5], which pointed out this error, formally proved the correctness of our new implementation, and provided experimental results for the revised system. The most recent paper investigated move pruning in the context of full-memory algorithms [18].

2. State and rule representation

A state is a vector $\langle v_1, v_2, \ldots, v_N \rangle$ of N values, where v_i is drawn from a finite set of possible values D_i (the domain for vector position i). A move is described by a rule (operator) that maps a state $\langle v_1, \ldots, v_N \rangle$ to another state $\langle v'_1, \ldots, v'_N \rangle$, with an associated cost. The cost of a rule is a non-negative number that does not depend on the state to which the rule is applied.

Our running example of an SASP is the n-Arrow puzzle [20]. A state consists of n "arrows", each of which is either pointing up or down. This is represented by a vector of $N = n$ values all drawn from a single binary domain (we shall use 1 to represent "up" and 0 to represent "down"). Conceptually, there are $n - 1$ different operators that can all be applied to any state: operator i flips the arrows in positions i and $i + 1$ (for $i \in \{1, \ldots, n - 1\}$).

We deal with state spaces where the rules have preconditions of the form $v_i = d \in D_i$ or $v_i = v_j$, and the resulting state can be described with a set of effects where $v'_i = d \in D_i$ or $v'_i = v_j$. All commonly used planning and search testbed domains can be expressed in rules of this form, although doing so often means that several rules will be required to express what is conceptually one operator. In the n-Arrow puzzle, for example, we described operator 1 as flipping arrows 1 and 2. In our notation, the four rules shown in Table 1 would be required to implement this operator. The distinction made here between a conceptual operator and rules in a formal problem description is important, but we will henceforth only consider the formal description in our notation, and will use the terms "rule" and "operator" interchangeably.

A rule of this type can be represented as a cost and two vectors of length N: $\vec{p} = \langle p_1, \ldots, p_N \rangle$ for the preconditions and $\vec{a} = \langle a_1, \ldots, a_N \rangle$ for the effects, where each p_i and a_i is either a constant $d \in D_i$ or a variable symbol drawn from the set $X = \{x_1, \ldots, x_N\}$. Variable symbol x_j is associated with domain D_j, therefore we only permit p_i or a_i to be x_j if $D_j = D_i$. In this notation, the precondition $v_i = d$ is represented by $p_i = d$, precondition $v_i = v_j$ is represented by

Table 1
4-Arrow puzzle, rules representing conceptual operator 1

Rule	Preconditions	Effects
R1-00	$v_1 = 0, v_2 = 0$	$v'_1 = 1, v'_2 = 1$
R1-01	$v_1 = 0, v_2 = 1$	$v'_1 = 1, v'_2 = 0$
R1-10	$v_1 = 1, v_2 = 0$	$v'_1 = 0, v'_2 = 1$
R1-11	$v_1 = 1, v_2 = 1$	$v'_1 = 0, v'_2 = 0$

Table 2
Rules for the 4-Arrow puzzle in vector notation

Rule	Preconditions $\rightarrow$ Effects	Cost
R1-00	$\langle 0, 0, x_3, x_4\rangle \rightarrow \langle 1, 1, x_3, x_4\rangle$	c_{100}
R1-01	$\langle 0, 1, x_3, x_4\rangle \rightarrow \langle 1, 0, x_3, x_4\rangle$	c_{101}
R1-10	$\langle 1, 0, x_3, x_4\rangle \rightarrow \langle 0, 1, x_3, x_4\rangle$	c_{110}
R1-11	$\langle 1, 1, x_3, x_4\rangle \rightarrow \langle 0, 0, x_3, x_4\rangle$	c_{111}
R2-00	$\langle x_1, 0, 0, x_4\rangle \rightarrow \langle x_1, 1, 1, x_4\rangle$	c_{200}
R2-01	$\langle x_1, 0, 1, x_4\rangle \rightarrow \langle x_1, 1, 0, x_4\rangle$	c_{201}
R2-10	$\langle x_1, 1, 0, x_4\rangle \rightarrow \langle x_1, 0, 1, x_4\rangle$	c_{210}
R2-11	$\langle x_1, 1, 1, x_4\rangle \rightarrow \langle x_1, 0, 0, x_4\rangle$	c_{211}
R3-00	$\langle x_1, x_2, 0, 0\rangle \rightarrow \langle x_1, x_2, 1, 1\rangle$	c_{300}
R3-01	$\langle x_1, x_2, 0, 1\rangle \rightarrow \langle x_1, x_2, 1, 0\rangle$	c_{301}
R3-10	$\langle x_1, x_2, 1, 0\rangle \rightarrow \langle x_1, x_2, 0, 1\rangle$	c_{310}
R3-11	$\langle x_1, x_2, 1, 1\rangle \rightarrow \langle x_1, x_2, 0, 0\rangle$	c_{311}

$p_i = x_j$, and the effects $v'_i = d$ and $v'_i = v_j$ are represented by $a_i = d$ and $a_i = x_j$, respectively. Table 2 shows the full set of rules for the 4-Arrow puzzle in this notation. We think of $\vec{p}$ and $\vec{a}$ as "augmented states", since they are precisely states over domains that have been augmented with the appropriate variable symbols: x_j is added to D_i if $D_j = D_i$.

To summarize the notation used in the rest of this paper: v_i refers to the value in position i of the state to which a rule is being applied, p_i refers to the symbol (either a constant from D_i or a variable symbol from X) in position i of a rule's preconditions, a_i refers to the symbol (either a constant from D_i or a variable symbol from X) in position i of a rule's effects, and x_i is a variable symbol from X that takes on the value v_i when the rule is applied.

In this notation, there can be several different, but equivalent, representations of a rule. For example, consider rule R1-01 in Table 2. Because its precondition requires $v_1 = 0$, its effects could be written as $\langle 1, x_1, x_3, x_4\rangle$, and because its precondition also requires $v_2 = 1$, there are two additional ways its effects could be written: $\langle x_2, 0, x_3, x_4\rangle$ and $\langle x_2, x_1, x_3, x_4\rangle$. We wish to have a unique canonical representation for a rule, so whenever we have a choice between writing an effect with a variable or writing it with a constant, we always choose the latter. The version of rule R1-01 shown in Table 2 is therefore the canonical representation. If we don't have a choice of writing an effect with a constant, but do have a choice of writing it with one of several variable symbols, we choose the variable symbol with the smallest index. By imposing these two constraints, there is a unique canonical representation for each rule's effects.

Similarly, there can be multiple ways to represent a rule's preconditions. Consider rule R1-00 in Table 2. Its preconditions require both v_1 and v_2 to be 0. Instead of representing this as $p_1 = 0$ and $p_2 = 0$, as in Table 2, it could instead have been written as $p_1 = 0$ and $p_2 = x_1$ or as $p_1 = x_2$ and $p_2 = 0$. To get a canonical representation for a rule's preconditions we impose the same constraints as for effects: if it is possible to use a constant, do so, and if that is not possible but there is a choice of variable symbols that could be used, use the variable symbol with the smallest index.

3. Rule composition

Restricting rules to have the types of preconditions and effects that we are using has the useful property that the preconditions required to execute an entire sequence of rules can be described in exactly the same notation as the preconditions for a single rule, and, likewise, the net effects of applying a sequence of rules can be described in exactly the same notation as the effects of a single rule. Hence, the collective preconditions and net effects of a sequence of rules can be represented by one rule (called a "macro-rule"[1]). In this section, we describe how to compute the collective preconditions and net effects for a sequence of rules.

The computation is an iterative one, starting with the macro-rule representing the empty rule sequence, namely, a rule with cost 0, preconditions $\langle x_1, \ldots, x_N\rangle$ and effects $\langle x_1, \ldots, x_N\rangle$. Now assume that, for $k \geqslant 0$, the first k rules in the sequence can be represented as a single macro-rule with cost c_1, preconditions $\vec{p}^{\,1}$, and effects $\vec{a}^{\,1}$ and consider how to compute the macro-rule for the extension of this sequence by one additional rule (the $(k+1)$st rule in the sequence) , with cost c_2, preconditions $\vec{p}^{\,2}$ and effects $\vec{a}^{\,2}$.

The cost of the extended sequence is simply $c_1 + c_2$. The preconditions and effects of the extended se-

[1] Our usage of this term is the same as in the literature on learning macro-rules [11], but in this paper, macro-rules are used only for analysis, not as new move options that are available at run time.

quence, $\vec{p}^*$ and $\vec{a}^*$, are constructed as follows. We start by setting $\vec{p}^* = \vec{p}^1$ and $\vec{a}^* = \vec{a}^1$. The next step is to update $\vec{p}^*$ and $\vec{a}^*$ to take into account the preconditions of the $(k+1)$st rule. We consider the preconditions one at a time. For each precondition, p_i^2, there are two main cases, depending on whether the net effect of the first k rules on position i (a_i^1) is a constant or a variable symbol; each case has several subcases.

(1) a_i^1 *is a constant.*

(a) p_i^2 *is a constant.* If the two constants (a_i^1 and p_i^2) are not the same, the extended sequence is not valid: the ith precondition of the $(k+1)$st rule is guaranteed *not* to be satisfied after executing the first k rules. If the two constants are the same, the ith precondition of the $(k+1)$st rule is guaranteed to be satisfied after executing the first k rules so no update to $\vec{p}^*$ or $\vec{a}^*$ is required.

(b) $p_i^2 = x_j$ *and* a_j^1 *is a constant.* $p_i^2 = x_j$ means the ith and jth positions must be the same, i.e., the two constants (a_i^1 and a_j^1) must be the same. This case is therefore the same as the previous case: if the two constants are not the same it is invalid to apply the $(k+1)$st rule after executing the first k rules, and if they are the same no update to $\vec{p}^*$ or $\vec{a}^*$ is required.

(c) $p_i^2 = x_j$ *and* $a_j^1 = x_k$. Again, $p_i^2 = x_j$ means the ith and jth positions must be the same, so for the sequence to be valid, x_k must equal the constant in a_i^1. All occurrences of x_k in $\vec{p}^*$ and $\vec{a}^*$ are replaced with this constant.

(2) $a_i^1 = x_j$.

(a) p_i^2 *is a constant.* All occurrences of x_j in $\vec{p}^*$ and $\vec{a}^*$ are replaced with the constant (p_i^2).

(b) $p_i^2 = x_j$. In this case the ith precondition of the $(k+1)$st rule is guaranteed to be satisfied after executing the first k rules, so no update to $\vec{p}^*$ or $\vec{a}^*$ is required.

(c) $p_i^2 = x_k$ *for some* $k \neq j$ *and* a_k^1 *is a constant.* All occurrences of x_j in $\vec{p}^*$ and $\vec{a}^*$ are replaced with the constant (a_k^1).

(d) $p_i^2 = x_k$ *for some* $k \neq j$ *and* $a_k^1 = x_t$. Let $y = \min(j,t)$, and $z = \max(j,t)$. All occurrences of x_z in $\vec{p}^*$ and $\vec{a}^*$ are replaced with x_y.

At this point, the validity of adding the $(k+1)$st rule has been determined and $\vec{p}^*$ represents the preconditions necessary to apply the entire sequence of $k+1$ rules, but $\vec{a}^*$ only describes the net effects of the first k rules. We must modify $\vec{a}^*$ by applying $\vec{a}^2$ to it. This requires a temporary copy, $\vec{a}^{\,\text{copy}}$, of $\vec{a}^*$. If a_i^2 is a constant d, we set $a_i^* = d$. Otherwise, $a_i^2 = x_j$ for some j and we set $a_i^* = a_j^{\text{copy}}$.

$\vec{p}^*$, $\vec{a}^*$, and cost $c_1 + c_2$ are now a rule with the preconditions and effects of the entire length $k+1$ move sequence. This process is repeated until all the rules in the original sequence have been taken into account. The end result will either be a proof that the given rule sequence is invalid or a single macro-rule representing the collective preconditions and net effects of the given rule sequence.

To illustrate this process, consider computing a macro-rule for the 4-Arrow puzzle to represent the sequence in which rule R1-00 is followed by rule R2-11. $\vec{p}^*$ and $\vec{a}^*$ for this macro-rule are initialized to be the precondition vector and effect vector for R1-00, respectively, i.e., $\vec{p}^* = \langle 0, 0, x_3, x_4 \rangle$ and $\vec{a}^* = \langle 1, 1, x_3, x_4 \rangle$. Next, we go through the precondition vector for R2-11 ($\langle x_1, 1, 1, x_4 \rangle$) one position at a time and update $\vec{p}^*$ and $\vec{a}^*$ according to which of the seven subcases above applies. For the first position, $i = 1$, case 1(b) applies, because position 1 of R1-00's effect vector is a constant (1) but position 1 of R2-11's precondition vector is a variable symbol (x_1). The conditions for validity are satisfied and no updates to $\vec{p}^*$ and $\vec{a}^*$ are made. For the next position ($i = 2$), case 1(a) applies because position 2 of R1-00's effect vector and R2-11's precondition vector are both constants. They are the same constant (1) so the conditions for validity are satisfied and again no updates to $\vec{p}^*$ and $\vec{a}^*$ are made. For $i = 3$ case 2(a) applies because position 3 of R1-00's effect vector is a variable symbol (x_3) but position 1 of R2-11's precondition vector is a constant (1). All occurrences of x_3 in $\vec{p}^*$ and $\vec{a}^*$ are changed to the constant 1 making $\vec{p}^* = \langle 0, 0, 1, x_4 \rangle$ and $\vec{a}^* = \langle 1, 1, 1, x_4 \rangle$. Finally, for $i = 4$ case 2(b) applies, leaving $\vec{p}^*$ and $\vec{a}^*$ unchanged. $\vec{p}^* = \langle 0, 0, 1, x_4 \rangle$ is the precondition for the sequence but there is one last step to derive the final $\vec{a}^*$ – it must be reconciled against the effect vector for R2-11.

The final step in calculating $\vec{a}^*$ is shown in Table 3. As described above, all the constants in R2-11's effect vector (positions 2 and 3) are directly copied into the same positions in $\vec{a}^*$, and, if position i of R2-11's effect vector is the variable symbol x_j, position i in $\vec{a}^*$ is set to be whatever was in position j of $\vec{a}^*$ before this last round of alterations began (this sets position 1 to be the constant 1 and position 4 to be the variable sym-

Table 3
Applying rule R2-11 after rule R1-00

$\vec{a}^*$ (before)	$= \langle 1,$	$1,$	$1,$	$x_4\rangle$
R2-11 effects	$= \langle x_1,$	$0,$	$0,$	$x_4\rangle$
		$\downarrow$	$\downarrow$	
$\vec{a}^*$ (after)	$= \langle 1,$	$0,$	$0,$	$x_4\rangle$

bol x_4). The final macro-rule in this example is thus $\langle 0, 0, 1, x_4\rangle \rightarrow \langle 1, 0, 0, x_4\rangle$.

4. Move pruning

Rule composition, described above, lets us build a tree of valid rule sequences, with a compact macro-rule representation of the preconditions and effects of each sequence. The root of the tree is the empty sequence, and each child extends the rule sequence of the parent with one additional rule. This tree gives us the set of potential rule sequences to prune. We chose to build a tree containing all rule sequences of up to L rules, but there are other ways to select a set of rule sequences of interest.

We can ignore rule sequence B if we can always use rule sequence A instead to reach the same state at no additional cost. More formally, following Taylor and Korf [31] we say rule sequence B is redundant with rule sequence A if (i) the cost of A is no greater than the cost of B, and, for any state s that satisfies the preconditions of B, both of the following hold: (ii) s satisfies the preconditions of A, and (iii) applying A and B to s leads to the same end state.

Condition (i) is trivial to check since the cost of each sequence is calculated by the rule composition process.

Because we have a unique macro-rule representation for each sequence, checking condition (ii) is a straightforward comparison of $\vec{p}^{\,A}$ and $\vec{p}^{\,B}$, the precondition vectors for sequences A and B, on a position-by-position basis. If $p_i^A = d \in D_i$, we require $p_i^B = d$. If $p_i^A = x_{j\neq i}$, we require $p_i^B = p_j^B$. Finally, if $p_i^A = x_i$, then p_i^B can have any value. Condition (ii) is satisfied if we pass these tests for all $i \in \{1, \ldots, N\}$.

Because we can treat precondition and effect vectors as augmented states, checking condition (iii) is also straightforward: we can simply apply the effects $\vec{a}^{\,A}$ to the preconditions $\vec{p}^{\,B}$. Condition (iii) holds if and only if the resulting augmented state is identical to $\vec{a}^{\,B}$.

These conditions are asymmetric: rule sequences A and B might be exactly equivalent so that either could be pruned, or it can be the case that A lets us prune B, but B does not let us prune A. For example, rule sequences A and B might both swap two variables and have the same cost, but A has no preconditions while B requires that the first variable be 1. Only B can be pruned.

As another example, consider another pair of rule sequences with the same cost. A swaps the first two variables. B turns $\langle 1, 2, 1, \ldots\rangle$ into $\langle 2, 1, 1, \ldots\rangle$. A does not, in general, always produce $\langle 2, 1, 1, \ldots\rangle$, but it will do so for any state that matches the preconditions of B, so we can again prune B. A is more general, and cannot be pruned. Note that if A happened to cost more than B, we could not prune either A or B, even though we know that A will generate duplicate children if the parent happens to match the preconditions of B.

For rule sequences A and B we write $B \geqslant A$, $B > A$, and $B \equiv A$ to denote that B is redundant with A, strictly redundant with A, or equivalent to A, respectively.

There are a number of implementation details to consider. We must check each sequence against all other sequences, which is O(n^2) for n sequences. A quadratic algorithm is not unreasonable, but because n here grows exponentially in the depth of the tree, and the branching factor of the tree can be large (over 100 in many state spaces), we are limited to fairly shallow trees (two or three rules).

Even with small trees, in the interest of efficiency, it is best to prune rule sequences as soon as possible. As Taylor and Korf [31] noted, sequences should be checked as the tree is built, rather than waiting to generate all sequences, and it is worthwhile building the tree in a breadth-first fashion.[2] That way, if we find any short rule sequences that can be pruned, we can use this information to immediately prune any longer rule sequences that include the shorter sequences we have previously pruned.

The way we use the redundancy information generated by this analysis we call "move pruning". If sequence B, of length k, is redundant with sequence A and state t was reached by applying rules $B_1, \ldots, B_{k-1}$ to some state s, then we do not allow rule B_k to be applied to t because the resulting state will be reached by applying A to state s.

Taylor and Korf [31] encode the list of redundant rule sequences in a compact FSM, with transitions between states based on the last rule applied. The same algorithm could be used to generate a compact FSM for the move pruning information we generate, but

[2] A technical reason for generating the sequences in increasing order of length is given in Section 4.2.

because our trees are generally not as large, our implementation used a simple table-based method which generates a larger FSM. We start by assigning a unique integer tag to each non-redundant rule sequence in the interior of our tree. In our implementation, where we consider all sequences up to length L, this is all valid rule sequences of length $L-1$ that were not discovered to be redundant. This tag corresponds to the FSM state. If there are M rules defined in the SASP, and we used T tags, we construct a table with M entries for each of the T tags. The entry for a rule sequence P and subsequent rule r is set to -1 if the new rule sequence $P+r$ is redundant. Otherwise, the entry is the tag of the last $L-1$ rules in $P+r$. This encodes the transition rules between FSM states.

4.1. Interactions between multiple applications of move pruning

If we remove all the redundant sequences discovered in the manner described in the previous section, it can eliminate all the least-cost paths from one state to another. To see this, consider the following example. Suppose that abd and acd are the only least-cost paths from state s to state t. If $ab > ac$ then the principle above says we need not execute ab, which means abd will not be executed. That is fine because acd will be executed. But if we also have $cd > bd$, then the principle above says we need not execute cd, which means acd will not be executed. So if we apply the principle twice (once for each redundancy) neither abd nor acd will be executed and all least-cost paths from s to t will be pruned away.

To see that it is possible for a, b, c and d to exist such that $ab > ac$ and $cd > bd$, here is a very simple example. In this example a state is described by three state variables, all having the same domain ($\{0, 1, 2, 3\}$), and is written as a vector of length three. A set of operators for which $ab > ac$ and $cd > bd$ is the following (all operators have a cost of 1).

a: $\langle 0, x_2, x_2\rangle \rightarrow \langle 1, 0, x_2\rangle$,

b: $\langle 1, x_2, 0\rangle \rightarrow \langle 2, 0, 0\rangle$,

c: $\langle 1, x_2, x_3\rangle \rightarrow \langle 2, x_3, x_2\rangle$,

d: $\langle 2, 0, 0\rangle \rightarrow \langle 3, 1, 1\rangle$.

The preconditions and net effects for ab and ac are:

ab: $\langle 0, 0, 0\rangle \rightarrow \langle 2, 0, 0\rangle$,

ac: $\langle 0, x_2, x_2\rangle \rightarrow \langle 2, x_2, 0\rangle$.

Clearly, the preconditions of ab are more restrictive than those of ac and the effects and costs of the sequences are the same when the preconditions of ab are satisfied. Hence, $ab > ac$.

The preconditions and net effects for bd and cd are:

bd: $\langle 1, x_2, 0\rangle \rightarrow \langle 3, 1, 1\rangle$,

cd: $\langle 1, 0, 0\rangle \rightarrow \langle 3, 1, 1\rangle$.

The preconditions of cd are more restrictive than those of bd and the effects and costs of the sequences are the same when the preconditions of cd are satisfied. Hence, $cd > bd$. If the start state is $\langle 0, 0, 0\rangle$ and the goal state is $\langle 3, 1, 1\rangle$ the only least-cost paths from start to goal are abd and acd, both of which will be pruned away.

This is an artificial example created to be as simple as possible. Something very much like it arises in sliding-tile puzzles having more than one blank.[3] For example, consider the two states of a 3×3 sliding-tile puzzle with 3 blanks shown in Fig. 1. The shortest path transforming the state on the left to the state on the right has four moves. There are eight such paths. If move pruning is applied to all sequences of length 3 or less, all of these four-move paths are pruned.

Table 4 shows three of the 2- and 3-move operator sequences our method identified as redundant and the sequence with which each was redundant. In Table 4 operator names indicate the row and column of the tile to be moved with digits and the direction of movement with a letter. For example 12R is the oper-

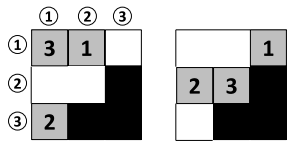

Fig. 1. Two states of the 3×3 sliding-tile puzzle with 3 blanks. The black squares are tiles that are not moved in any of the shortest paths that transform the state on the left to the state on the right. In the circles are the row and column numbers.

Table 4
Pruning rules involved in Fig. 2

Pruning rule #	Redundant sequence		Because of
1	12R-11D	$\equiv$	11D-12R
2	11D-12R-21R	$>$	12R-11R-12D
3	11R-12D-31U	$>$	11D-21R-31U

[3] This example actually occurred in our experiments with the "Work or Golf" SASP.

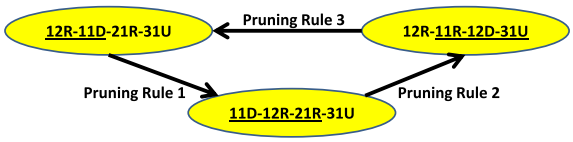

Fig. 2. Each oval represents a least-cost sequence transforming the state on the left of Fig. 1 to the state on the right. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140605.)

ator that moves the tile in Row 1 (top row), Column 2 (middle column) to the right (R). The first row of the table (Pruning Rule 1) indicates that the 2-move sequence 12R-11D is equivalent ("≡") to 11D-12R. Because they are equivalent, either one could be eliminated in favour of the other; our method eliminated 12R-11D because it is lexically "larger than" 11D-12R. The other two rows in Table 4 show operator sequences that are not equivalent (">"): the sequence in the "Redundant" column has the same effects but more restrictive preconditions than the sequence in the "Because of" column. In such cases there is no choice about which sequence to eliminate.

The three pruning rules in Table 4 interact in a way that is exactly analogous to how $ab > ac$ and $cd > bd$ interacted to prune both abd and acd in our simple example above. Each oval in Fig. 2 represents a least-cost (4-move) sequence to transform the left state in Fig. 1 to the right state. Each one of these sequences contains one of the 2- or 3-move sequences identified as redundant in Table 4 – the redundant subsequence is underlined. The arrows between ovals indicate the operator sequence that is produced when the redundant subsequence is replaced by the corresponding "Because of" sequence in Table 4. For example, the sequence in the upper left oval (12R-11D-21R-31U) contains 12R-11D as its first two operators. Table 4 indicates that this is redundant with 11D-12R. Applying this substitution to the sequence in the upper left oval produces the sequence in the bottom oval. This sequence contains the subsequence identified as redundant in the second row (Pruning Rule 2) of Table 4, and if that subsequence is replaced by the corresponding "Because of" sequence, the operator sequence in the upper right oval is produced. The last three operators in this sequence have been found redundant (Pruning Rule 3); replacing them with the alternative brings us literally full circle, back to the operator sequence in the upper left oval. Put another way, since all of these operator sequences contain a subsequence declared redundant by our move pruning system, all of them will be pruned. The other five least-cost (4-move) sequences for transforming the left state in Fig. 1 to the right state also all contain a subsequence declared redundant by our move pruning system, so all of them will be pruned too.

The important lesson from these examples is that even if the redundancy of each operator sequence is correctly assessed, a set of redundancies can interact to produce incorrect behaviour. We therefore need to develop a theory of when a set of redundancies does not prune all least-cost paths. In the next section we present one such theory.

4.2. Theory

The empty sequence is denoted ε. If A is a finite operator sequence then $|A|$ denotes the length of A (the number of operators in A, $|\varepsilon| = 0$), cost(A) is the sum of the costs of the operators in A (cost(ε) $= 0$), pre(A) is the set of states to which A can be applied, and $A(s)$ is the state resulting from applying A to state $s \in$ pre(A). We assume the cost of each operator is non-negative. A prefix of A is a nonempty initial segment of A ($A_1, \ldots, A_k$ for $1 \leqslant k \leqslant |A|$) and a suffix is a nonempty final segment of A ($A_k, \ldots, A_{|A|}$ for $1 \leqslant k \leqslant |A|$).

Recall that operator sequence B is redundant with operator sequence A if (i) the cost of A is no greater than the cost of B, and, for any state s that satisfies the preconditions of B, both of the following hold: (ii) s satisfies the preconditions of A, and (iii) applying A and B to s leads to the same end state. Formally, we have the following definition.

Definition 1. Operator sequence B is "redundant" with operator sequence A iff the following conditions hold:

(1) cost(B) $\geqslant$ cost(A),
(2) pre(B) $\subseteq$ pre(A),
(3) $s \in$ pre(B) $\Rightarrow B(s) = A(s)$.

As before, we write $B \geqslant A$, $B > A$ and $B \equiv A$ to denote that B is redundant with A, strictly redundant with A, or equivalent to A, respectively.

Lemma 1. *Let $B \geqslant A$ according to Definition* 1 *and let XBY be any least-cost path from any state s to state $t = XBY(s)$. Then XAY is also a least-cost path from s to t.*

Proof. There are three things to prove.

(1) $s \in \text{pre}(XAY)$.

Proof. X can be applied to s because XBY can be applied to s. A can be applied to $X(s)$ because B can be applied to $X(s)$ and $\text{pre}(B) \subseteq \text{pre}(A)$ (because $B \geqslant A$). Y can be applied to $A(X(s))$ because Y can be applied to $B(X(s))$ and $A(X(s)) = B(X(s))$. Therefore $s \in \text{pre}(XAY)$. □

(2) $XAY(s) = t$.

Proof. Since $t = Y(B(X(s)))$ and $B(X(s)) = A(X(s))$ (see the Proof of (1)), we get $t = Y(A(X(s))) = XAY(s)$. □

(3) $\text{cost}(XBY) = \text{cost}(XAY)$.

Proof. $\text{cost}(XBY) \geqslant \text{cost}(XAY)$ follows from the cost of a sequence being additive ($\text{cost}(XBY) = \text{cost}(X) + \text{cost}(B) + \text{cost}(Y)$) and $\text{cost}(B) \geqslant \text{cost}(A)$ (because $B \geqslant A$). Since XBY is a least-cost path from s to t and XAY is also a path from s to t, $\text{cost}(XBY) = \text{cost}(XAY)$. □

Let $\mathcal{O}$ be a total ordering on operator sequences. $B >_{\mathcal{O}} A$ indicates that B is greater than A according to $\mathcal{O}$. $\mathcal{O}$ has no intrinsic connection to redundancy so it can easily happen that $B \geqslant A$ according to Definition 1 but $B <_{\mathcal{O}} A$.

Definition 2. A total ordering on operator sequences $\mathcal{O}$ is "nested" if $\varepsilon <_{\mathcal{O}} Z$ for all $Z \neq \varepsilon$, and $B >_{\mathcal{O}} A$ implies $XBY >_{\mathcal{O}} XAY$ for all A, B, X, and Y.

Example 1. The most common nested ordering is "length-lexicographic order", which is based on a total order of the operators $o_1 <_{\mathcal{O}} o_2 <_{\mathcal{O}} \ldots$. For arbitrary operator sequences A and B, $B >_{\mathcal{O}} A$ iff $|B| > |A|$ or $|B| = |A|$ and $o_b >_{\mathcal{O}} o_a$ where o_b and o_a are the leftmost operators where B and A differ (o_b is in B and o_a is in the corresponding position in A).

Definition 3. Given a nested ordering $\mathcal{O}$, for any pair of states s, t define $\min(s, t)$ to be the least-cost path from s to t that is smallest according to $\mathcal{O}$ ($\min(s, t)$ is undefined if there is no path from s to t).

Theorem 2. *Let $\mathcal{O}$ be any nested ordering on operator sequences and B any operator sequence. If there exists an operator sequence A such that $B \geqslant A$ according to Definition 1 and $B >_{\mathcal{O}} A$, then B does not occur as a consecutive subsequence in $\min(s, t)$ for any states s, t.*

Proof. By contradiction. Suppose there exist s, t such that $\min(s, t) = XBY$ for some such B and some X and Y. Then by Lemma 1 XAY is also a least-cost path from s to t. But $XBY >_{\mathcal{O}} XAY$ (because $\mathcal{O}$ is a nested ordering and $B >_{\mathcal{O}} A$), contradicting XBY being the smallest (according to $\mathcal{O}$) least-cost path from s to t. □

From this theorem it immediately follows that a move pruning system that restricts itself to pruning only operator sequences B that are redundant with some operator sequence A *and* greater than A according to a fixed nested ordering will be "safe", i.e. it will not eliminate all the least-cost paths from the start state (s) to any other state (t). The choice of $\mathcal{O}$ might well affect the amount of pruning that is done, but any choice of $\mathcal{O}$ will guarantee safety. In our implementation of move pruning all operator sequences of length L or less are generated in the order defined by the fixed nested ordering described in Example 1, and each newly generated sequence is tested for redundancy against all the non-redundant sequences generated before it.

Theorem 2 also proves that Taylor and Korf's results on Rubik's Cube are correct. They used the nested ordering described in Example 1 and tested the equality of the net effects of two operator sequences, the only part of Definition 1 that needs to be tested when operators have no preconditions.

4.3. Interaction with cycle detection

Checking for redundancy among operator sequences up to length L does not guarantee that there will be no cycles. There may be cycles longer than L, which the redundancy analysis will necessarily be unable to detect. There may also be cycles of length L or less which are not considered redundant because they only occur in certain circumstances. For example, consider a rule that swaps two variables (i.e., $\langle x_1, x_2 \rangle$ becomes $\langle x_2, x_1 \rangle$). There are some states, such as $\langle 1, 1 \rangle$, for which this rule is an identity operator (1-cycle). For arbitrary states, of course, this rule is not an identity operator, so we cannot prune away all occurrences of it. It is possible, using a slight variant of our Rule Com-

position algorithm, to derive the special conditions under which a rule sequence is a cycle or is equivalent to another sequence, but doing so might generate an impracticably large number of special conditions. We therefore only consider a rule sequence redundant if it is universally redundant with another rule sequence, as defined by Definition 1. We refer to rule sequences that create cycles, or are redundant with other rule sequences, under special circumstances, but not universally, as serendipitous cycles and redundancies.

If one expected there to be many long and/or serendipitous cycles, they might wish to use run-time cycle detection in addition to move pruning, to eliminate those cycles.[4] In general, combining different search reduction techniques is unsafe, even if one of them is as seemingly innocuous as cycle detection [1]. We now prove that it is safe to use move pruning in conjunction with cycle detection. In the following $\mathcal{O}$ is any fixed nested ordering on operator sequences used for move pruning.

Our proof will apply to any search algorithm that implements the pseudocode in Algorithm 1. Typical implementations will use a fixed enumeration order for operator sequences, rather than accept the order as an input parameter. Familiar examples implementing this pseudocode are breadth-first search,[5] Dijkstra's algorithm, A*, IDA*, and depth-first branch and bound. A "filter" in this algorithm is a rule for eliminating a path from consideration. In this section we are examining two filters: one based on move pruning (MP), and one based on cycle detection (CD). Filters based on heuristic cutoffs (HC) and duplicate detection (DD) will be discussed in Sections 4.4 and 6, respectively.

Practical systems usually cannot ascertain that the stopping condition (line 19) holds the instant that the first least-cost solution path is enumerated. They typically continue to enumerate paths until they are certain that no paths remain in the enumeration sequence that would improve the best known solution path, p_{opt}. Note that the stopping condition is testing for the existence of a path with certain properties in the original, unfiltered enumeration sequence. It is therefore unaffected by the use of filters.

We will refer to the basic search algorithm, without any of the filters, as *Alg*.

[4] Another option in this situation is for the person to "split" (reformulate) the general rules in which cycles sometimes occur into more specialized rules so that the exact conditions under which the cycles occur are preconditions of sequences involving the specialized rules.

[5] Breadth-first search is only guaranteed to return a least-cost path if all operators have the same cost.

Algorithm 1. Algorithm *Alg* uses none of the optional filters. The filters are based on move pruning (MP), cycle detection (CD), heuristic cutoff (HC), and duplicate detection (DD), as described in the text. "skip p_i" means to skip the rest of the body of the *for* loop.

1: Given: a start state s, a goal predicate *goal*, an enumeration sequence $\{p_1, p_2, \ldots\}$ of all paths starting at s (including the empty path ε), and a path-cost function cost(p). For the HC filter, also given are an admissible heuristic function h and a cost bound b.
2: Return: a least-cost path p_{opt} in the enumeration sequence such that $goal(p_{\text{opt}}(s))$ is true.
3: $p_{\text{opt}} =$ undefined
4: *bestCost* $= \infty$
5: **for** $i = 1$ to ∞ **do**
6: // Apply optional filters.
7: **MP:** skip p_i if move pruning deems it redundant (defined in the text).
8: **CD:** skip p_i if it contains a cycle.
9: **HC:** skip p_i if it contains a prefix p (possibly p_i itself) such that cost(p) + $h(p(s)) > b$.
10: **DD:** skip p_i if it contains a prefix p (possibly p_i itself) such that $p(s)$ has previously been generated by a path q and cost(q) $\leqslant$ cost(p).
11:
12: Generate state $s_i = p_i(s)$.
13: // check if p_i is the best solution so far
14: **if** (cost(p_i) $<$ *bestCost*) **and** $goal(s_i)$ **then**
15: $bestCost =$ cost(p_i)
16: $p_{\text{opt}} = p_i$
17: **end if**
18: // check stopping condition
19: **if** there does not exist $j > i$ such that (cost(p_j) $<$ *bestCost*) **and** $goal(p_j(s))$ **then**
20: **return** p_{opt}
21: **end if**
22: **end for**

When *Alg* is used in conjunction with move pruning, the resulting system is called Alg^{MP}. Move pruning based on operator sequences of length L or less eliminates path p_i if and only if p_i contains a consecutive subsequence p' of length L or less and there exists an operator sequence q of length L or less such that $p' \geqslant q$ and $p' >_{\mathcal{O}} q$. This is the "MP" filter in Algorithm 1. Note that, for all t reachable from start state s, min(s, t) is not eliminated by the MP filter.

When *Alg* is used in conjunction with cycle detection, the resulting system is called Alg_{CD}. For a given

start state s, cycle detection eliminates path p_i if and only if p_i contains a non-empty operator sequence C such that $p_i = XCY$ and $XC(s) = X(s)$. This is the "CD" filter in Algorithm 1. Note that the CD filter eliminates paths containing serendipitous cycles as well as those containing universal cycles.

When *Alg* is used in conjunction with both move pruning and duplicate detection, the resulting system is called $Alg^{\text{MP}}_{\text{CD}}$.

Definition 4. For a given start state s, we say a *goal* predicate is "reachable" if there exists a goal state that is reachable from s, and we say a goal state t is "nearest to s" if cost(min(s,t)) $\leqslant$ cost(min(s,u)) for every goal state u.

Definition 5. We say move pruning is "safe" to use in conjunction with filter X (where X is one of CD, HC or DD) if, for any start state s and any reachable *goal* predicate, Alg^{MP}_{X} returns a least-cost path from s to a goal state that is nearest to s.

In other words, move pruning is unsafe to use in conjunction with cycle detection only if, for some start state s and reachable *goal*, $Alg^{\text{MP}}_{\text{CD}}$ fails to return a least-cost path to a goal state that is nearest to s. We will now show that move pruning is safe to use in conjunction with cycle detection. The proof has two steps:

(1) For all t reachable from s, min(s,t) is not eliminated by the combination of MP and CD filters.
(2) In particular, for each goal state t nearest to s, at least one least-cost path from s to t remains after the MP and CD filters have been applied. $Alg^{\text{MP}}_{\text{CD}}$ will return the first such path in its enumeration sequence.

Theorem 3. *Move pruning is safe to use in conjunction with cycle detection.*

Proof. Let s be any start state and *goal* any reachable goal predicate. First, we show that, for any state t reachable from s, min(s,t) is not eliminated by the combination of MP and CD filters. We show this by contradiction. min(s,t) is not eliminated by move pruning, so if it is eliminated it must be eliminated by the CD filter. This will only happen if min$(s,t) = XCY$ and $XC(s) = X(s)$ for a non-empty C. If such a C existed, then XY would be a path from s to t such that cost(XY) $\leqslant$ cost(min(s,t)) and $XY <_{\mathcal{O}}$ min(s,t), which contradicts the definition of min(s,t). Therefore no such C exists, so cycle detection does not eliminate min(s,t) from *Alg*'s enumeration sequence.

For each goal state t nearest to s define i_t to be the smallest index of a path in *Alg*'s enumeration sequence such that p_{i_t} is not eliminated by the MP and CD filters, $p_{i_t}(s) = t$, and cost(p_{i_t}) = cost(min(s,t)). We know such an index exists because, as we have just seen, min(s,t) is not eliminated by the MP and CD filters. Define m to be the minimum i_t.

We will now show that $Alg^{\text{MP}}_{\text{CD}}$ returns p_m. $Alg^{\text{MP}}_{\text{CD}}$ cannot terminate before enumerating p_m because, by the way p_m was defined, all paths with smaller indices in *Alg*'s enumeration sequence that reach the stopping condition (line 19 in Algorithm 1) are not least-cost paths to a nearest goal state, and therefore the stopping condition is not satisfied. Therefore p_m will be enumerated by $Alg^{\text{MP}}_{\text{CD}}$, it will pass the MP and CD filters, and p_{opt} will be set equal to p_m in line 16 of Algorithm 1. Because cost(p_m) is the least cost of any path that leads to a goal state, subsequently enumerated paths will fail the test in line 14 of Algorithm 1, so p_{opt} will never be changed, it will remain equal to p_m until termination. □

4.4. Interaction with heuristic cutoff

Many search algorithms use a bound b and eliminate all paths, p, for which cost$(p) + h(p(s)) > b$, where s is the start state and h is an admissible heuristic function (a lower bound on the cost to reach a goal state nearest to s from state $p(s)$). We call this method for eliminating paths heuristic cutoff.

When *Alg* (see Algorithm 1) is used in conjunction with heuristic cutoff, the resulting system is called Alg_{HC}. Note that the HC filter in Algorithm 1 eliminates all paths that have a prefix p such that cost(p) + $h(p(s)) > b$; this means that if a path (p) is eliminated by HC then so are all extensions of it. When *Alg* is used in conjunction with both move pruning and heuristic cutoff, the resulting system is called Alg^{MP}_{HC}.

Of course, heuristic cutoff eliminates all paths from s to *goal* if $b < f^*$, the cost of a least-cost path from s to *goal*, so we are only interested in its interaction with move pruning when $b \geqslant f^*$, as in the last iteration of IDA*, depth-first branch and bound, or implementations of A* that do "trimming" and/or "screening" [22]. We now show that move pruning is safe to use in conjunction with heuristic cutoff when $b \geqslant f^*$ and the heuristic being used is admissible. The proof has the same structure as the proof for Theorem 3.

Theorem 4. *Move pruning is safe to use in conjunction with heuristic cutoff for any cost bound* $b \geqslant f^*$, *and any admissible heuristic* h.

Proof. Let s be any start state and *goal* any reachable goal predicate. First, we show that, for any goal state t nearest to s, $\min(s,t)$ is not eliminated by the combination of MP and HC filters. $\min(s,t)$ is not eliminated by move pruning, so it could only be eliminated by heuristic cutoff. Let p be any prefix of $\min(s,t)$ (including $\min(s,t)$ itself). Then $\text{cost}(p) + h(p(s)) \leqslant \text{cost}(\min(s,t)) = f^* \leqslant b$ and therefore p will not be eliminated by heuristic cutoff. Hence, $\min(s,t)$ will not be eliminated by the combination of MP and CD filters.

The remainder of the proof is identical to the second part of the proof of Theorem 3, so we do not repeat it here ("For each goal state t nearest to s define i_t"... define $m \ldots p_{\text{opt}}$ is set to p_m and never changed). □

5. Experimental results for depth-first search

To test our move pruning in a linear-memory setting, we implemented the move pruning algorithm in the PSVN programming environment that we have developed. This toolkit takes an SASP description in the PSVN language (described in the Appendix), and generates C code for state manipulation. We modified the code generator to build a move sequence tree and print out a table of pruning information (as described in Section 4) and wrote depth-first search code that could use this pruning information.

We used 9 different SASPs in our experiments: the 16-Arrow puzzle [20], the Blocks World [27] with 10 blocks, the 4-peg Towers of Hanoi [17] with 8 disks, the Pancake puzzle [7] with 9 pancakes, the 3×3 sliding-tile puzzle (8-puzzle) [28], $2 \times 2 \times 2$ Rubik's Cube [31], TopSpin [6] with 14 tiles and a 3-tile turnstile, the Work or Golf puzzle, which is a sliding-tile puzzle variant with two irregular tiles described in Fig. 3, and the Gripper domain with 10 balls.

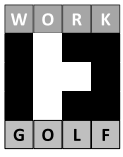

Fig. 3. Goal state for the Work or Golf puzzle. It has eight 1×1 tiles with letters and 2 irregularly shaped tiles, shown in black. There are 4 empty locations.

In our PSVN SASP descriptions, the Arrow puzzle had 60 rules, Blocks World had 200 rules, Towers of Hanoi had 96 rules, the Pancake puzzle had 8 rules, the 8-puzzle had 24 rules, the $2 \times 2 \times 2$ Rubik's Cube had 18 rules, TopSpin had 14 rules, Work or Golf had 93 rules, and Gripper had 22 rules.

As written in PSVN, the Arrow puzzle, Blocks World, Towers of Hanoi, 8-puzzle, Work or Golf, and Gripper all had rules with preconditions, which precludes using the method of Taylor and Korf [31].

For each SASP, we compared the total number of nodes generated, and total execution time, of three variations of depth-first search (DFS) with a depth bound:

(1) DFS with parent pruning (length 2 cycle detection) performed by comparing a generated state to the parent of the state from which it was generated,
(2) DFS with our move pruning method applied to sequences of length $L = 2$ or less, and
(3) DFS with our move pruning method applied to sequences of length $L = 3$ or less.

The SASPs have different branching factors, so we used a different depth bound for the DFS in each SASP.

The results are shown in Table 5. All experiments were run on a 2.83 GHz Core2 Q9550 CPU with 8 GB of RAM. Node counts (in thousands of nodes) and computation times (in seconds) are totals (not averages) across 100 randomly generated start states. Less than 6 s were required to do the move pruning analysis for $L = 2$ and only three domains required more than 19 s for $L = 3$.

As was seen by Taylor and Korf [31], we see here that using DFS with move pruning can generate many fewer nodes than DFS with parent pruning. Even with shallow move pruning ($L = 2$), we can do at least as well parent pruning, and in those cases where move pruning ($L = 2$) and DFS with parent pruning generate exactly the same number of nodes, move pruning requires less time than parent pruning. This holds across a broad range of SASPs.

The n-Arrow puzzle has an interesting property. Any move sequence i, j has the same result as j, i and i, i has no effect. Our move pruning algorithm with $L = 2$ will enforce an ordering on the moves. That is, if we just made move i, all moves 1 to $i - 1$ are pruned by our system. In this special case, we have removed all duplicate states from the search tree. It is easy to see this: consider an arbitrary path between two states. Because the moves commute and a move is its own in-

Table 5
Experimental evaluation of move pruning

State space	d	DFS + PP	DFS + MP $L = 2$	DFS + MP $L = 3$
16-Arrow puzzle	15	?	3,277	3,277
		>3600 s	0.39 s	0.39 s
			0.07 s	18.58 s
10 Blocks world	11	352,028	352,028	352,028
		25.02 s	12.23 s	12.53 s
			5.97 s	8 m 27 s
8-puzzle	25	368,357	368,357	368,357
		24.77 s	10.40 s	10.40 s
			0.01 s	0.08 s
Pancake puzzle	9	5,380,481	5,380,481	5,288,231
		246.49 s	115.22 s	111.18 s
			0.01 s	0.02 s
Towers of Hanoi	10	1,422,419	31,673	9,060
		97.02 s	1.45 s	0.49 s
			0.30 s	3 m 43 s
$2 \times 2 \times 2$ Rubik's Cube	6	2,715,477	833,111	515,614
		132.74 s	20.00 s	13.35 s
			0.02 s	1.10 s
TopSpin	9	?	2,165,977	316,437
		>3600 s	73.80 s	12.59 s
			0.00 s	1.11 s
Work or Golf	13	?	209,501	58,712
		>3600 s	16.44 s	5.14 s
			2.98 s	15 m 4 s
Gripper	14	9,794,961	590,870	25,982
		544.85 s	17.22 s	0.95 s
			0.08 s	0.85 s

Notes: The first two columns indicate the state space and the depth bound used. The other columns give results for each DFS variation. In each results cell the top number is the total number of nodes generated, in thousands, to solve all the test problems. The number below that is the total time, in seconds, to solve all the test problems. In the DFS + MP columns the bottom number is the time ("m" for minutes, "s" for seconds) needed for the move pruning analysis.

verse, we can sort the moves by index and remove pairs of identical moves. The resulting path is unique, and is not pruned by our system. Strictly ordering the moves also means the depth 15 searches we used explore the entire reachable space of the 16-Arrow puzzle.

We can see related behaviour with the Blocks World and 8-puzzle: increasing L from 2 to 3 did not change the number of nodes generated. In this case, there are no transpositions using paths of length 3 which are not found by looking at paths of length 2. Unlike the n-Arrow puzzle, redundancy analysis with sufficiently large values of $L > 2$ on these SASPs will find additional redundant sequences. In the 8-puzzle, for example, we will see additional pruning if we consider rule sequences of length 6.

In Blocks World, 8-puzzle, and Pancake puzzle, we see that the number of generated nodes is the same whether we use parent pruning or our move pruning system with $L = 2$. In these SASPs, the only transpositions that occur within a sequence of two moves are cycles of length 2. Looking at the total time, we see that parent pruning is more expensive. Our move pruning system only needs to look up a single integer in a table. DFS with parent pruning is more than twice as slow because in the general case it requires generating the child states and then comparing them to the parent state, which is much more expensive. For exactly this reason, hand-implemented systems for a specific SASP generally do parent pruning by skipping moves that undo the previous move (a form of move pruning) if the SASP supports this.

The Blocks World results have one final strange feature: the total time for $L = 3$ is larger than for $L = 2$. Ideally, we would expect this to be the same. In this case, because we did not implement the FSM of Taylor and Korf [31], the move pruning table grows as L increases. The Blocks World had 200 rules, so the $L = 3$ table was fairly large (around 30 MB) and we suspect cache performance suffered when using this larger table.

The last row in Table 5 is for the Gripper domain. In this domain, there are two rooms ($Room_1$ and $Room_2$) and B balls. In the canonical start and goal states the balls start in $Room_1$ and the goal is to move them all to $Room_2$. Movement is done by a robot that has two hands. The operators are "pickup ball$_k$ with hand h" (where h is either left or right), "change room" and "put down the ball in hand h". It is an interesting domain in which to study move pruning because of the large number of alternative least-cost solutions: the balls can be moved in any order, any pair of balls can be carried together, and there are eight different operator sequences for moving a specific pair of balls between rooms. The results in Table 5 are for $B = 10$, which has 68,608 reachable states, and are for only one start state (all the balls and the robot in $Room_1$), which has a least-cost solution length of 29. As can be seen, even just pruning short redundant sequences ($L = 2$ or 3) has a very large effect (over 500× speedup for $L = 3$). For $B = 10$ it is possible to discover redundant sequences up to length $L = 6$ in just under nine hours using less than 40 MB of space to store the results. If this is done, the number of nodes generated to depth 14 is 368 thousand and the time drops to

0.02 s, over 23,000 times faster than DFS with parent-pruning. With $L = 6$ move pruning, depth-first search can fully explore to depth 29 (the solution depth for $B = 10$) in 46 s. The number of nodes generated is 157,568,860.

At the end of Section 4.2, we mentioned that the restriction we imposed on move pruning to make it safe – i.e. testing a newly generated sequence for redundancy only against sequences that precede it in the given nested ordering – might substantially reduce the amount of pruning that was done. This can easily be measured by comparing the results shown in Table 5 to analogous results using the unsafe version of move pruning, which removes all redundant sequences. In Gripper, the unsafe pruning method generates substantially fewer nodes, but it also fails to visit many reachable states and the goal state becomes unreachable. Using unsafe move pruning with $L = 3$ in Work or Golf generates 56,986 thousand nodes (in total) compared to 58,712 thousand nodes with safe pruning, and for both $L = 2$ and $L = 3$ there are pairs of states for which all least-cost paths have been pruned (e.g. see Fig. 1). For all the remaining SASPs, including the relatively complicated Blocks World and Towers of Hanoi, the safe pruning method produces identical results to the unsafe pruning method.

6. Move pruning and duplicate detection

Full-memory search algorithms, such as A* and breadth-first search, avoid re-expanding nodes unnecessarily by doing duplicate detection – testing if each state they generate has previously been generated by a path of equal or smaller cost. It was observed by Malte Helmert (personal communication) that move pruning is not, in general, safe to use in conjunction with duplicate detection. Figure 4 shows a simple situation in which a problem arises. A and B are operators or operator sequences that are not redundant with each other in general, but happen to produce the same state, T, when applied to state S. AC and BC are the only two paths from S to U, and move pruning determines that AC is redundant with BC and decides to prohibit C from being applied after A. However, the search generates T via path A first, and records this fact using the usual backpointer method found in A* implementations. When the search later generates T via path B it notices that T has already been generated by a path of the same cost and therefore ignores B. Since the only recorded path from S to T is A, move pruning prevents C from being applied to T and state U is never reached.

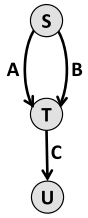

Fig. 4. Characteristic situation in which duplicate detection and move pruning interact to produce erroneous behaviour.

To see that it is possible for such A, B and C to exist, with AC and BC being redundant with each other but A and B not being redundant with one another, here is a very simple example (also due to Malte Helmert). A state in this example is described by three state variables, all with the same domain ($\{0, 1, 2\}$), and is written as a vector of length three. The following operators behave like A, B, and C in Fig. 4 when applied to state $S = \langle 0, 1, 1\rangle$:

$$A\text{: } \langle 0, x_2, x_3\rangle \rightarrow \langle 1, 1, x_3\rangle,$$
$$B\text{: } \langle 0, x_2, x_3\rangle \rightarrow \langle 1, x_2, 1\rangle,$$
$$C\text{: } \langle 1, x_2, x_3\rangle \rightarrow \langle 2, 1, 1\rangle.$$

A and B are not redundant with one another, in general, but both can be applied to state $S = \langle 0, 1, 1\rangle$ and doing so produces the same state, $T = \langle 1, 1, 1\rangle$.

6.1. Motivation

The motivation for adding move pruning to a system that does duplicate detection is computational – move pruning is faster than duplicate detection. This is because with duplicate detection a state must be generated and looked up in a data structure to determine if it is a duplicate. Move pruning saves the time needed for duplicate detection because it avoids generating states when it knows (by analysis in a preprocessing step) the resulting state is certain to be a duplicate. For example, in the experiments in Section 5, in those SASPs where move pruning ($L = 2$) achieved exactly the same effects as parent pruning (an elementary form of duplicate detection), move pruning was more than twice as fast as doing parent pruning by explicit duplicate detection. In addition, if suboptimal paths to a state are generated before optimal ones, duplicate detection will involve updating the data structure that stores the distance-from-start information. This can be relatively

expensive – updating the priority queue used by A*, for example. Move pruning will avoid some of these updates by not generating some of the suboptimal paths at all.

On the other hand, duplicate detection is useful to add to a system that does move pruning because move pruning, in general, is incomplete: it only detects short sequences that are redundant (in the current implementation move pruning considers all and only sequences of length L or less) and it only detects "universal" redundancy, as opposed to "serendipitous" redundancy, as illustrated in the example above, where sequences A and B are redundant when applied to certain states but are not redundant in general. Duplicate detection is complete, unless there is not enough memory to store all the generated states.

The final motivation for studying the interactions between move pruning and duplicate detection is that it applies much more broadly than just to systems that use our method for automatic move pruning. When SASPs with many obvious redundancies, such as TopSpin and Rubik's Cube, are coded by hand, the person writing the code often manually does a simple version of the move pruning that we have automated. For example, here is a detailed description of the standard move pruning done by hand for Rubik's Cube [19]:

> Since twisting the same face twice in a row is redundant, ruling out such moves reduces the branching factor to 15 after the first move. Furthermore, twists of opposite faces of the cube are independent and commutative. For example, twisting the front face, then twisting the back face, leads to the same state as performing the same twists in the opposite order. Thus, for each pair of opposite faces we arbitrarily chose an order, and forbid moves that twist the two faces consecutively in the opposite order.

These are precisely the kinds of redundant operator sequences that our method detects automatically. The correctness of the move pruning done manually has never been questioned, but the problem illustrated in Fig. 4 applies regardless of whether the move pruning was inferred by an automatic method or by hand. Thus it brings into question the correctness of the standard encodings of testbeds such as Rubik's Cube and TopSpin if they are used in a system that does duplicate detection. In fact, we have verified that the manually encoded move pruning in the IDA* code written in our research group for TopSpin results in non-optimal solutions being produced if it is used in A*.

6.2. Conditions precluding simple interactions

We call the situation depicted in Fig. 4 a "simple" interaction between duplicate detection and move pruning, by which we mean the interaction takes place between two least-cost paths, AC and BC, that have a common suffix (C). In this section we derive commonly occurring conditions under which simple interactions cannot possibly happen. Throughout the rest of the section we assume there is a fixed nested ordering on operator sequences, $\mathcal{O}$, used for move pruning.

Because AC and/or BC can be longer than the sequences that move pruning considers, define A' to be the suffix of A, B' to be the suffix of B, and C' to be the prefix of C such that move pruning determines that $A'C' \geqslant B'C'$ and $A'C' >_{\mathcal{O}} B'C'$. The latter implies $A' >_{\mathcal{O}} B'$. This, together with the fact that A' is not pruned by move pruning (A' is fully executed) implies that $A' \not\geqslant B'$.

Thus, a simple interaction requires an interesting situation: $A'C' \geqslant B'C'$ but $A' \not\geqslant B'$. There are natural conditions in which this combination is impossible because $(A'C' \geqslant B'C') \Rightarrow (A' \geqslant B')$ for all sequences A', B' and C'. To derive such conditions, recall that the definition of $X \geqslant Y$ has three requirements:

(R1) $\text{cost}(X) \geqslant \text{cost}(Y)$,
(R2) $\text{pre}(X) \subseteq \text{pre}(Y)$,
(R3) $s \in \text{pre}(X) \Rightarrow X(s) = Y(s)$.

In order to derive conditions under which $(A'C' \geqslant B'C') \Rightarrow (A' \geqslant B')$ we need to consider each of these in turn.

(R1) We require conditions under which $(\text{cost}(A'C') \geqslant \text{cost}(B'C')) \Rightarrow (\text{cost}(A') \geqslant \text{cost}(B'))$. In fact, no special conditions are needed, this is always true because the cost of a sequence is the sum of the costs of the operators in that sequence.

(R2) We require conditions under which $(\text{pre}(A'C') \subseteq \text{pre}(B'C')) \Rightarrow (\text{pre}(A') \subseteq \text{pre}(B'))$. This is often not true, but it certainly holds if $\text{pre}(XY) = \text{pre}(X)$ for all sequences X and Y (with X nonempty). There are two commonly occurring conditions in which this holds:

- operators have no preconditions (every operator is applicable to every state) as in Rubik's Cube;
- the precondition of any sequence is the precondition of the first operator in the sequence (because the preconditions of the next operator

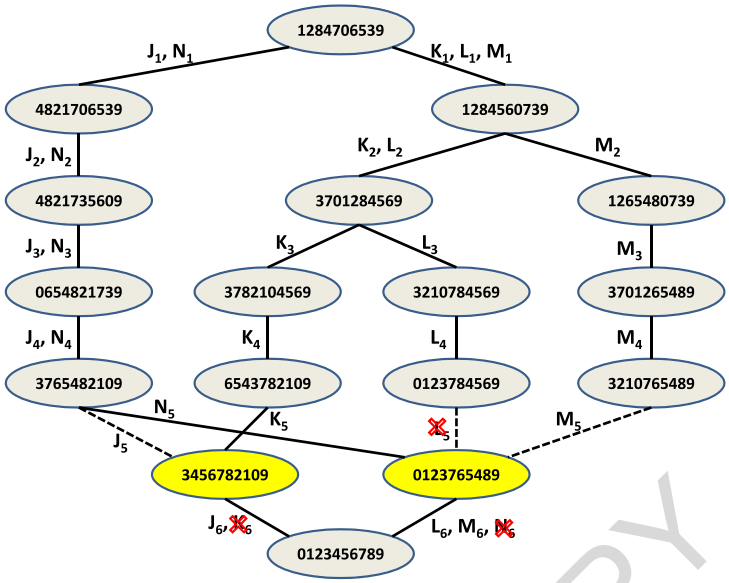

Fig. 5. Example from (10, 4)-TopSpin of move pruning and duplicate detection interacting to prevent the goal (bottom node) from being reached from the start (top node) by a least-cost path. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-140605.)

in the sequence are guaranteed by the effects and unchanged preconditions of the operators preceding it), as in the sliding-tile puzzles with only one blank.

(R3) We require conditions under which $(s \in \text{pre}(A'C') \Rightarrow A'C'(s) = B'C'(s)) \Rightarrow (t \in \text{pre}(A') \Rightarrow A'(t) = B'(t))$. This follows directly if both of the following hold:

- $\text{pre}(XY) = \text{pre}(X)$ for all sequences X and Y (with X non-empty), the same condition discussed in connection with (R2); and
- all operators are 1-to-1 ($\text{op}(x) = \text{op}(y) \Rightarrow x = y$ for all states x and y, and all operators op).

The two conditions listed under (R3) are thus sufficient to prevent simple interactions from occurring. These conditions hold in many commonly used state spaces: the sliding-tile puzzle when there is just one blank, Rubik's Cube, Scanalyzer [15], and any permutation state space such as TopSpin and the Pancake puzzle. In all such spaces, there cannot be a simple interaction between duplicate detection and move pruning.

Unfortunately, simple interactions are not the only way that move pruning and duplicate detection can interact deleteriously, i.e., the situation in Fig. 4 is not a necessary condition for move pruning to be unsafe in conjunction with duplicate detection. Figure 5 gives an example based on an actual run of A* on (10, 4)-TopSpin[6] when move pruning is applied to sequences of length 4 or less. The start state is at the top of the figure, the goal state is at the bottom. Move pruning eliminates all but two of the least-cost paths from start to goal; those two paths are labelled J (the leftmost path) and M (the rightmost path) in the figure; the individual operators in a path are indicated by a subscript (e.g. J_2 is the second operator in path J).

Three additional paths (K, L and N) are shown because they play a role in preventing J and M from being fully executed even though they themselves cannot be fully executed because of move pruning. The move pruning that eliminates K, L and N is shown in the figure by an X through operators K_6, L_5 and N_6. The reasons for these are as follows. Move pruning detects that $N_5 N_6 \geqslant J_5 J_6$ and therefore prevents N_6 from being executed after N_5. It also detects that $K_3, \ldots, K_6 \geqslant L_3, \ldots, L_6$ and therefore prevents K_6 from being executed after $K_3, \ldots, K_5$. Similarly, it detects that $L_2, \ldots, L_5 \geqslant M_2, \ldots, M_5$ and therefore prevents L_5 from being executed after $L_2, \ldots, L_4$. These can all be seen in the figure as paths of length 4 or less that branch apart at some particular state and later rejoin.

The effects of duplicate detection are shown by drawing the edges entering the two states just above

[6] In (10, 4)-TopSpin there are 10 tokens (numbers 0 to 9) in a circle and there are operators that reverse the order of any 4 adjacent tokens. Because only the cyclic order matters and not the absolute location within the circle, in the figure a state is written as a vector with token 9 always placed at the end.

the goal as either solid or broken. A solid edge indicates the path by which the state was first generated; a broken edge indicates an alternative path to the state that is generated later (or not at all in the case of L_5). For example, state `3456782109` is first generated by path K (operator K_5) and is later generated by path J (operator J_5). Since the path $J_1, \ldots, J_5$ is not cheaper than the first path to generate the state $(K_1, \ldots, K_5)$, it is ignored. Similarly, $M_1, \ldots, M_5$ is not cheaper than the first path to generate state `0123765489` $(N_1, \ldots, N_5)$, so it too is ignored.

What makes this fundamentally different than Fig. 4 is that the path (K) that blocks J because of duplicate detection is not itself blocked by J because of move pruning, it is blocked by a different least-cost path (L, which in turn is blocked by M because of move pruning). Likewise, the path (N) that blocks M because of duplicate detection is not itself blocked by M because of move pruning, it is blocked by a different least-cost path (J). As we will show next, this represents the general situation in which move pruning and duplicate detection interact deleteriously.

6.3. Necessary conditions for move pruning to be unsafe in conjunction with duplicate detection

In this section we state and prove conditions that must hold if move pruning is unsafe to use in conjunction with duplicate detection. The importance of identifying these "necessary" conditions is that one can then consider whether there are specific circumstances in which one or more of the necessary conditions are guaranteed not to hold. Move pruning is safe to use in such circumstances.

As in Section 4.3 our discussion focuses on search algorithms implementing Algorithm 1, *Alg* is Algorithm 1 without any filters, and Alg^{MP} refers to *Alg* augmented with the move pruning filter (MP).

When *Alg* is used in conjunction with duplicate detection, the resulting system is called Alg_{DD}. For a given start state s, duplicate detection eliminates path p_i if and only if there exists a prefix p of p_i (possibly p_i itself), such that the state $p(s)$ has been previously generated (line 12 in Algorithm 1) via a path $q \neq p$ and $\text{cost}(q) \leqslant \text{cost}(p)$. This is the "DD" filter in Algorithm 1. Note that the fact that $p(s)$ was generated by q means that q was not itself eliminated by duplicate detection. A* and breadth-first search are examples of Alg_{DD} search algorithms.[7]

[7] Breadth-first search is only guaranteed to return a least-cost path if all operators have the same cost.

When *Alg* is used in conjunction with both move pruning and duplicate detection, the resulting system is called $Alg^{\text{MP}}_{\text{DD}}$.

We say that $Alg^{\text{MP}}_{\text{DD}}$ "generates" path p if it does not terminate before p is enumerated and p is not eliminated by the combination of MP and DD filters. We use the notation $p <_{Alg} q$ to indicate that path p is before path q in *Alg*'s enumeration sequence. We write $p <_{Alg^{\text{MP}}_{\text{DD}}} q$ if p is generated by $Alg^{\text{MP}}_{\text{DD}}$ and $p <_{Alg} q$ (we do not require q to be generated by $Alg^{\text{MP}}_{\text{DD}}$).

Specializing Definition 5 for duplicate detection, we have the following definition.

Definition 6. We say move pruning is "safe" to use in conjunction with duplicate detection if, for any start state s and any reachable *goal* predicate, $Alg^{\text{MP}}_{\text{DD}}$ returns a least-cost path from s to a goal state that is nearest to s.

In other words, move pruning is unsafe to use in conjunction with duplicate detection only if $Alg^{\text{MP}}_{\text{DD}}$ fails to return a least-cost path from s to *goal*. In particular, if move pruning is unsafe, $Alg^{\text{MP}}_{\text{DD}}$ will fail to generate every least-cost path to every goal state nearest to s. From this fact, we will now derive necessary conditions for move pruning to be unsafe to use in conjunction with duplicate detection.

Theorem 5. *Let S be any start state, goal any goal predicate reachable from S, U any goal state nearest to S and P any least-cost path from S to U that is not eliminated by move pruning* (*e.g.* $\min(S, U)$). *Then $Alg^{\text{MP}}_{\text{DD}}$ will fail to generate P only if there exist operator sequences B, C, B_n and C_n such that $P = BC = B_nC_n$ and both of the following conditions hold*:

REQ-1: *There exists an alternative path A_1 from S to state $T_1 = B(S)$ such that* $\text{cost}(A_1) = \text{cost}(B)$ *and T_1 was generated by $Alg^{\text{MP}}_{\text{DD}}$ via A_1 prior to B being enumerated* (*i.e.* $A_1 <_{Alg^{\text{MP}}_{\text{DD}}} B$).

REQ-2: *There exists an alternative path A_n from S to state $T_n = B_n(S)$, A_nC_n is a least-cost path from S to U, and move pruning prohibits C_n from being applied after A_n.*

Proof of REQ-1. This is necessary because if no such T_1 and A_1 existed duplicate detection would not eliminate P, which contradicts the premise that $Alg^{\text{MP}}_{\text{DD}}$ fails to generate P. A_1 cannot be cheaper than B because B is part of an least-cost path to U and is therefore a least-cost path to T_1. □

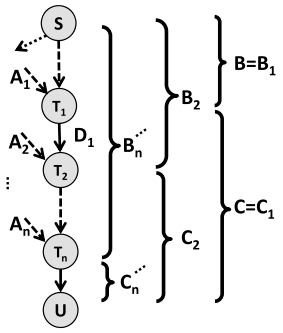

Fig. 6. Illustration of the Proof of REQ-2.

Proof of REQ-2. Figure 6 depicts the key ideas needed to prove this. A_1 here is as in REQ-1 and operator sequences B and C from above are renamed here B_1 and C_1. As the proof proceeds, they are replaced by A_i, B_i, and C_i for larger values of i, with B_i increasing in length as i increases and C_i decreasing in length. In all cases $P = B_iC_i$, $A_i(S) = B_i(S) = T_i$ and A_iC_i is a least-cost path from S to U. D_i is the operator subsequence in P that leads from T_i to T_{i+1}.

A_1C_1 is a least-cost path from S to U, why did $Alg^{\text{MP}}_{\text{DD}}$ not generate it? Either because A_1C_1 was eliminated by move pruning or because it was eliminated by duplicate detection. If it was eliminated by move pruning then we are done, with $n = 1$ ($T_n = T_1$, $A_n = A_1$, and $C_n = C_1 = C$). If it was eliminated by duplicate detection then there must be a state T_2 later in the $P(S)$ sequence and alternative path A_2 from S to T_2 such that cost(A_2) = cost(A_1D_1) and T_2 was generated by $Alg^{\text{MP}}_{\text{DD}}$ via A_2 prior to being generated via A_1D_1. Let C_2 be the suffix of C such that $C_2(T_2) = U$ and B_2 be the prefix of P such that $B_2(S) = T_2$. Now repeat this reasoning for the path A_2C_2, which is a least-cost path from S to U. If it was eliminated because of move pruning we are done with $n = 2$, and if it was eliminated because of duplicate detection, there must exist a T_3, A_3, B_3, and C_3 such that T_3 is later in the $P(S)$ sequence than T_2, etc. Repeating this reasoning defines a sequence of states $T_1, T_2, \ldots$, each later in the $P(S)$ sequence than the one before, and therefore there must be a final state in this sequence, T_n, with a corresponding A_n, B_n, and C_n, with A_nC_n being a least cost path from S to U. This path was not generated and it cannot have been eliminated by duplicate detection (because if it had been there would be a T_{n+1}), therefore it must have been eliminated because move pruning did not allow C_n to be executed after A_n. □

Because both of these requirements are necessary for move pruning to be unsafe, if one of them does not hold, move pruning is safe to use in conjunction with duplicate detection. The remainder of this section considers each of them in turn.

6.4. Discussion of REQ-1

REQ-1 states that there must be an alternative least-cost path, A_1, to $T_1 = B(S)$ that is generated before B is enumerated. This could fail to hold in at least three different ways. First, it would fail to hold if there was only one path to each of the states on $P(S)$ (namely, the appropriate prefix of P). This would happen, for example, if move pruning eliminated all alternative paths, as it does in the Arrow Puzzle [4]. In such cases, no duplicate is ever generated so duplicate detection is obviously safe to use with move pruning. Secondly, it would fail to hold if there were alternative paths to one or more states $T_1 = B(S)$ generated prior to B, but all of them were suboptimal. This is not impossible; for example, it would happen if there was a unique shortest path from S to each reachable state.

The third way that REQ-1 could fail to hold, and perhaps the most interesting from a practical point of view, is that there are indeed alternative least-cost paths to a state $T_1 = B(S)$ but none of them is generated before B. For example, consider the special case depicted in Fig. 4, where $A_1 = A$ is generated before B (i.e. $A <_{Alg} B$) but $A >_{\mathcal{O}} B$. In other words there is a disagreement between how *Alg* orders the sequences and how they are ordered by $\mathcal{O}$. If the two orderings $>_{\mathcal{O}}$ and $>_{Alg}$ were chosen so that such a disagreement did not occur then the special case depicted in Fig. 4 could not arise. Whether this can be done in practice, and whether it solves the general problem and not just the special case depicted in Fig. 4 are open problems at present.

6.4.1. Discussion of REQ-2

REQ-2 says that there must exist least-cost paths A_nC_n and B_nC_n such that move pruning prohibits C_n from being executed after A_n but allows it after B_n. This is very similar to the special case depicted in Fig. 4, but with one important difference. In the special case, $C = C_n$ is prohibited after $A = A_n$ because of $B = B_n$, i.e., $AC \geqslant BC$. In the general case we are now considering we do not require $AC \geqslant BC$, we just require that AC is redundant with some path.

Let A' be the suffix of A_n and C' be the prefix of C_n such that $A'C'$ is the sequence within A_nC_n that move pruning determines to be redundant with some other sequence D. There are two possibilities for D.

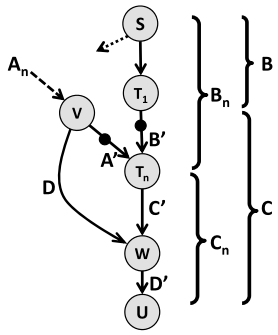

Fig. 7. General case for Requirement 2.

The first possibility, which is what we saw in Fig. 4, is that D is part of BC, i.e., there exists a suffix B' of B_n such that $A'C' \geqslant B'C'$ and $A'C' >_{\mathcal{O}} B'C'$. Circumstances in which this cannot possibly happen have been discussed in Sections 6.2 and 6.4 above.

The other possibility for D is shown in Fig. 7. Here D is a sequence entirely distinct from BC. In this case, we have another least-cost path from S to U – one that follows A_n to state V, then executes D, which leads to state W on the BC path from which the goal is reached by sequence D'. This, in fact, is precisely what we saw in the TopSpin example in Fig. 5. In that example least-cost solution J was blocked by duplicate detection by another sequence, K, which in turn was blocked by move pruning by a sequence, L, that had nothing in common with J.

There is, however, one special circumstance in which REQ-2 cannot possibly occur and therefore move pruning is safe to use in conjunction with duplicate detection, and that is if move pruning is restricted to considering only sequences of length 1, i.e. redundancy among individual operators considered in a fixed order. If this restriction is imposed, move pruning cannot prohibit C_n from being executed after A_n but allow it after B_n since no "history" is taken into account.

7. Related work

Wehrle and Helmert [32] have recently analyzed techniques from computer-aided verification and planning that are closely related to move pruning. They divide the techniques into two categories. *State reduction techniques* reduce the number of states that are reachable while still guaranteeing the cost to reach the goal from the start state remains unchanged. *Transition reduction techniques* reduce the number of state transitions ("moves") considered during search without changing the set of reachable states or the cost to reach a state from the start state. Move pruning is a transition reduction technique.

The most powerful transition reduction technique discussed by Wehrle and Helmert is the "sleep sets" method [12]. Sleep sets exploit the commutativity of operators.[8] To illustrate the key idea, suppose move sequence A contains an operator c that commutes with all the other operators in A. Then c can be placed anywhere in the sequence, with each different placement creating a different sequence that is equivalent to A. For example, if A is o_1o_2c and c commutes with o_1 and o_2 then sequences o_1co_2 and co_1o_2 are both equivalent to A. The sleep set of a node is the set of operators that do not have to be applied at that node because of this commutativity principle.

Sleep sets are less powerful than move pruning in some ways and more powerful in others. They are less powerful because they consider only one special kind of redundancy – commutativity of individual operators. Move pruning with $L = 2$ will detect all such commutative relations, but will also detect relations between sequences that do not involve the same operators, such as $o_1o_2 \equiv o_3o_4$, and strict redundancies such as $o_1o_2 > o_2o_1$. In addition, move pruning can be applied with $L > 2$.

On the other hand, sleep sets can eliminate sequences that are not eliminated by move pruning, as we have implemented it, because sleep sets can prune arbitrarily long sequences even if not all the operators in the sequence commute with one another. Continuing the above example, if the ordering on operators used by move pruning had $o_2 <_{\mathcal{O}} c <_{\mathcal{O}} o_1$ and move pruning considered only sequences of length $L = 2$ or less, then it would permit both o_1o_2c and co_1o_2 to be executed whereas the sleep set method would only execute one of them.[9]

A different, but related approach to avoiding redundant move sequences is state space "factoring" [2,3,9,13,23,24]. The ideal situation for factoring is when the given state space is literally the product of two (or more) smaller state spaces; a solution in the original state space can then be constructed by independently finding solutions in the smaller state spaces and interleaving those solutions in any manner whatsoever. The aim of a factoring system is to identify the smaller state spaces given the definition of the original state space. If the smaller spaces are "loosely coupled" (i.e., not

[8] Operators o_1 and o_2 are commutative if $o_1o_2 \equiv o_2o_1$.

[9] This is Wehrle and Helmert's Example 1 adapted to our notation.

perfectly independent but nearly so) factoring can still be useful, but requires techniques that take into the account the interactions between the spaces. Techniques similar to state space factoring have been developed for multiagent pathfinding [26,29] and additive abstractions (*cf.* the independent abstraction sets defined by Edelkamp [8] and the Factor method by Prieditis [25]).

Transposition tables [1] represent an entirely different approach to eliminating redundant operator sequences: they store states that have been previously generated and test each newly generated state to see if it is one of the stored states. Like parent pruning, transposition tables are slower at eliminating redundant sequences than move pruning because they involve generating a state and then processing it to determine if it is a duplicate, whereas move pruning simply avoids generating duplicate states.[10] Transposition tables are not strictly more powerful than move pruning because, with transpositions tables, a state might be generated by a suboptimal path before being generated by an optimal path. If the two paths are length L or less, this will not happen with move pruning. More importantly, in large state spaces there is not enough memory available to store all the distinct generated states. This forces transposition tables to be incomplete, i.e., to detect only a subset of the duplicate states. The memory required by move pruning, by contrast, is O(m^L), where m is the number of operators, which is usually logarithmic in the number of states in a combinatorial state space, and L is the maximum length of the sequences being considered. The memory required for move pruning will therefore usually be small compared to the number of distinct generated states. For example, for the Arrow puzzle the move pruning table when $L = 2$ is O(a^2) in size, where a is the number of arrows, whereas the number of reachable states is 2^{a-1} and for the ($n \times n$)-sliding tile puzzle, the move pruning table when $L = 2$ is O(n^2) in size whereas the number of reachable states is $(n^2)!/2$. This means move pruning can be effective in situations where transposition tables would be too small to be of much use. For example, in the Gripper domain ($B = 10$) the move pruning table when $L = 2$ requires 7 kilobytes and allows a complete depth-first search to depth 14 to finish in 17.22 s (see Table 5). If our transposition table implementation is restricted to use 7 kilobytes, depth-first search is unable to finish depth 13 in 6 min. In general, the duplicates eliminated by move pruning and by incomplete transposition tables will be different and one would like to use both together. However, we have seen that move pruning is not, in general, safe to use with duplicate detection and transposition tables include duplicate detection among their functionality [4].

[10] The efficiency gained by avoiding generating unneeded nodes, as opposed to generating and testing them, is the entire motivation for Enhanced Partial Expansion A* [10], which uses a data structure (the "OSF") specifying when an operator should be applied that is much like the move pruning table in our system.

8. Conclusions

In this paper, we introduced an algorithm for automatically analyzing a general single-agent search problem (SASP) to identify redundant operator sequences. We have shown that it is not generally safe to remove all redundant sequences identified, but if move pruning is required to respect a fixed nested ordering on operator sequences then move pruning is always safe. Imposing this restriction did not noticeably reduce the amount of pruning done in our experiments, but in principle an unlucky choice of the ordering could substantially reduce the amount of pruning done – in the extreme case, the ordering could prevent any move pruning from being done (this would happen if $B <_{\mathcal{O}} A$ for every pair of operator sequences A and B such that $B > A$). There is, therefore, more research to be done on how to maximize the amount of pruning that can be done while remaining safe.

We applied our automatic move pruning analysis to a variety of SASPs and experimentally demonstrated that it could speed up depth-first search by orders of magnitude.

We showed that move pruning is safe to use in conjunction with cycle detection and heuristic cutoffs, but that it is not safe, in general, to use in conjunction with the duplicate detection method found in breadth-first search, A*, and, indeed, virtually all full-memory search algorithms.

Acknowledgements

Thanks to Shahab Jabbari Arfaee for encoding the Gripper domain in PSVN and running the initial experiment that exposed the problem of unsafe pruning, to Sandra Zilles for her thoughtful feedback on early versions of the manuscript, and to Malte Helmert and Martin Wehrle for discussions about the partial order reduction methods they analyzed [32]. We gratefully acknowledge the funding sources whose support has made this research possible: the Alberta Ingenuity

Centre for Machine Learning (AICML), Alberta's Informatics Circle of Research Excellence (iCORE), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Appendix. The PSVN Language

Our PSVN is a slight extension of the language with the same name introduced by Hernádvölgyi and Holte [16]. It directly implements the state and rule representations described in Section 2.

A state is a vector of fixed length, N. The entry in position i is drawn from a finite set of possible values called its domain, D_i. In many state spaces every position of the vector has the same domain, but in principle they could all be different. Different domains in PSVN are entirely distinct; the same symbols can appear in different domains for user convenience, but our PSVN compiler will internally treat them as distinct.

The transitions in the state space are specified by a set of rules. Each rule has a left-hand side (LHS) specifying its preconditions and a right-hand side (RHS) specifying its effects. The LHS and RHS are each a vector of length N. In both the LHS and the RHS, position i is either a constant from D_i or a variable symbol. Any number of positions in these vectors (LHS and RHS) can contain the same variable symbol as long as their domains are all the same.

State $s = \langle s_1, \ldots, s_N \rangle$ matches LHS $= \langle L_1, \ldots, L_N \rangle$ if and only if $s_i = L_i$ for every L_i that is a constant and $s_i = s_j$ for every i and j such that L_i and L_j are the same variable symbol.

A rule is "deterministic" if every variable symbol in its RHS is also in its LHS. The effect of a deterministic rule when it is applied to state $s = \langle s_1, \ldots, s_N \rangle$ matching its LHS is to create a state $s' = \langle s'_1, \ldots, s'_N \rangle$ such that: (i) if position j of the RHS is the constant $c \in D_j$ then $s'_j = c$; (ii) if position j of the RHS is the variable symbol that occurs in the position i of the LHS then $s'_j = s_i$.

A rule is "non-deterministic" if one or more of the variable symbols in its RHS do not occur in its LHS. We call such variable symbols "unbound". The effect of a non-deterministic rule when it is applied to state $s = \langle s_1, \ldots, s_N \rangle$ matching its LHS is to create a set of successor states. There is one successor for every possible combination of values of the unbound variables (if the unbound variable is in position i, its values are drawn from D_i). Each of the other positions of these successors will be the same in all the successors and are determined by the rules for calculating the effects of deterministic rules. For example, if $N = 4$ and all positions have domain $\{1, 2\}$ then the rule $\langle 1, A, B, C \rangle \rightarrow \langle E, 1, D, E \rangle$ would create four successors when applied to state $\langle 1, 2, 1, 2 \rangle$, namely, $\langle 1, 1, 1, 1 \rangle$, $\langle 2, 1, 1, 2 \rangle$, $\langle 1, 1, 2, 1 \rangle$, and $\langle 2, 1, 2, 2 \rangle$.

As a concession to human readability PSVN allows one additional symbol, the dash ("-"), to be used in any position in the LHS or RHS. A dash in position i of the LHS means there is no precondition on the value in position i, and a dash in position i of the RHS means that the value in position i does not change when the operator is applied.

The final feature of PSVN is that the goal of a search is not required to be a single state. Goal conditions are specified by writing one or more special operators of the form "GOAL Condition", where Condition takes the same form as the LHS of a normal operator. If there are several such operators they are interpreted disjunctively.

References

[1] Y. Akagi, A. Kishimoto and A. Fukunaga, On transposition tables for single-agent search and planning: Summary of results, in: *Proc. Third Annual Symposium on Combinatorial Search (SOCS-10)*, 2010.

[2] E. Amir and B. Engelhardt, Factored planning, in: *IJCAI*, 2003, pp. 929–935.

[3] R.I. Brafman and C. Domshlak, Factored planning: How, when and when not, in: *AAAI*, 2006, pp. 809–814.

[4] N. Burch and R.C. Holte, Automatic move pruning in general single-player games, in: *Proceedings of the 4th Symposium on Combinatorial Search (SoCS)*, 2011.

[5] N. Burch and R.C. Holte, Automatic move pruning revisited, in: *Proceedings of the 5th Symposium on Combinatorial Search (SoCS)*, 2012.

[6] T. Chen and S.S. Skiena, Sorting with fixed-length reversals, *Discrete Applied Mathematics* **71** (1996), 269–295.

[7] H. Dweighter, Problem E2569, *American Mathematical Monthly* **82** (1975), 1010.

[8] S. Edelkamp, Planning with pattern databases, in: *Proc. European Conference on Planning*, 2001, pp. 13–24.

[9] E. Fabre, L. Jezequel, P. Haslum and S. Thiébaux, Cost-optimal factored planning: Promises and pitfalls, in: *Proc. 20th International Conference on Automated Planning and Scheduling*, 2010, pp. 65–72.

[10] A. Felner, M. Goldenberg, G. Sharon, R. Stern, N. Sturtevant, J. Schaeffer and R.C. Holte, Partial-expansion A* with selective node generation, in: *AAAI*, 2012, pp. 471–477.

[11] L. Finkelstein and S. Markovitch, A selective macro-learning algorithm and its application to the $n \times n$ sliding-tile puzzle, *Journal of Artificial Intelligence Research* **8** (1998), 223–263.

[12] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, Lecture Notes in Computer Science, Vol. 1032, Springer, 1996.
[13] M. Guenther, S. Schiffel and M. Thielscher, Factoring general games, in: *IJCAI Workshop on General Game Playing (GIGA'09)*, 2009.
[14] P.E. Hart, N.J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Systems Science and Cybernetics* **4**(2) (1968), 100–107.
[15] M. Helmert and H. Lasinger, The Scanalyzer domain: Greenhouse logistics as a planning problem, in: *ICAPS*, 2010, pp. 234–237.
[16] I. Hernádvölgyi and R. Holte, PSVN: A vector representation for production systems, Technical Report TR-99-04, Department of Computer Science, University of Ottawa, 1999.
[17] A.M. Hinz, The Tower of Hanoi, in: *Algebras and Combinatorics: Proceedings of ICAC'97*, Hong Kong, Springer-Verlag, 1997, pp. 277–289.
[18] R.C. Holte, Move pruning and duplicate detection, in: *Proceedings of the 26th Canadian Conference on Artificial Intelligence*, 2013, pp. 40–51.
[19] R. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: *Proceedings of the 14th AAAI Conference on Artificial Intelligence*, 1997, pp. 700–705.
[20] R.E. Korf, Towards a model of representation changes, *Artificial Intelligence* **14**(1) (1980), 41–78.
[21] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence* **27**(1) (1985), 97–109.
[22] J.B.H. Kwa, BS*: An admissible bidirectional staged heuristic search algorithm, *Artificial Intelligence* **38**(1) (1989), 95–109.
[23] A.L. Lansky, Localized planning with diverse plan construction methods, *Artificial Intelligence* **98**(1,2) (1998), 49–136.
[24] A.L. Lansky and L. Getoor, Scope and abstraction: Two criteria for localized planning, in: *IJCAI*, 1995, pp. 1612–1619.
[25] A. Prieditis, Machine discovery of effective admissible heuristics, *Machine Learning* **12** (1993), 117–141.
[26] G. Sharon, R. Stern, M. Goldenberg and A. Felner, The increasing cost tree search for optimal multi-agent pathfinding, in: *IJCAI*, 2011, pp. 662–667.
[27] J. Slaney and S. Thiébaux, Blocks world revisited, *Artificial Intelligence* **125** (2001), 119–153.
[28] J. Slocum and D. Sonneveld, *The 15 Puzzle*, Slocum Puzzle Foundation, 2006.
[29] T.S. Standley, Finding optimal solutions to cooperative pathfinding problems, in: *AAAI*, 2010, pp. 173–178.
[30] L.A. Taylor, Pruning duplicate nodes in depth-first search, Technical Report CSD-920049, UCLA Computer Science Department, 1992.
[31] L.A. Taylor and R.E. Korf, Pruning duplicate nodes in depth-first search, in: *AAAI*, 1993, pp. 756–761.
[32] M. Wehrle and M. Helmert, About partial order reduction in planning and computer aided verification, in: *Proceedings of ICAPS*, 2012.