Automatic Move Pruning Revisited

Neil Burch
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(nburch@ualberta.ca)

Robert C. Holte
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(holte@cs.ualberta.ca)

Abstract

In this paper we show that the move pruning method we presented at SoCS last year sometimes prunes all the least-cost paths from one state to another. We present two examples exhibiting this erroneous behaviour–a simple, artificial example and a slightly more complex example that arose in last year's experiments. We then formally prove that a simple modification to our move pruning method makes it "safe", i.e., it will never prune all the least-cost paths between a pair of states. Finally, we present the results of rerunning last year's experiments with this provably safe version of move pruning and show that last year's main conclusions still hold, namely: (1) in domains where there are short redundant sequences move pruning produces substantial speedups in depth-first search, and (2) in domains where the only short redundant sequences are 2-cycles, move pruning is faster than parent pruning by a factor of two or more.

Introduction

Last year we (Burch and Holte 2011) presented a generalization of the method for eliminating redundant operator sequences ("move pruning") introduced by Taylor and Korf (1992; 1993) and showed that it could vastly reduce the size of a depth-first search tree in spaces containing short cycles or transpositions. We claimed our method could be safely applied to any state space represented using the `PSVN` language. The first contribution of the present paper is to show that this claim is false: there are state spaces representable in `PSVN` for which the method we presented last year produces incorrect results; in particular, that method sometimes removes all least-cost paths to the goal and can even render the goal unreachable when it is, in fact, reachable.

The second contribution of this paper is to formally prove that a slight modification of the method we presented last year is correct ("safe"): for every pair of states, the revised method is guaranteed to leave unpruned at least one least-cost path between them. This theorem also shows that Taylor and Korf's original results on Rubik's Cube are correct because of the lexical ordering on operators they used to choose between equivalent sequences. Finally, our theorem shows that using move pruning to eliminate cycles is always safe.

The third contribution of this paper is to rerun last year's experiments with the provably safe variation of our method to examine its pruning power.

Examples of Erroneous Move Pruning

The fact that move pruning, as we implemented it last year, could produce erroneous behaviour came to our attention after last year's symposium when we were testing our method on the Gripper domain from the International Planning Competition.[1] As we will see below, the problem also arises with the Work or Golf state space used in last year's paper: some of the results reported for that state space last year are incorrect.

From these counterexamples we distilled a simple, generic counterexample that challenges the fundamental principle underlying our system, and Taylor and Korf's, which we described last year as follows:

> following Taylor and Korf we can prune move sequence B if there exists a move sequence A such that *(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.

We say that B is redundant with A, denoted $B \geq A$, if A and B meet the three conditions just stated. We write $B \equiv A$ (B is equivalent to A) if $B \geq A$ and $A \geq B$, and write $B > A$ (strict inequality) if $B \geq A$ and $A \not\geq B$.

Our counterexample is as follows. 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.

[1] http://ipc.icaps-conference.org/

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 and is written as a vector of length three. The value of each variable is either 0, 1, 2, or 3. The operators are written in the form $LHS \rightarrow RHS$ where LHS is a vector of length three defining the operator's preconditions and RHS is a vector of length three defining its effects. The LHS may contain variable symbols (X_i in this example); when the operator is applied to a state, the variable symbols are bound to the value of the state in the corresponding position. Preconditions test either that the state has a specific value for a state variable or that the value of two or more state variables are equal (this is done by having the same X_i occur in all the positions that are being tested for equality). For example, operator a below can only be applied to states whose first state variable has the value 0 and whose second and third state variables are equal. A set of operators for which $ab > ac$ and $cd > bd$ is the following (all operators have a cost of 1).

$a : \langle 0, X_1, X_1 \rangle \rightarrow \langle 1, 0, X_1 \rangle$
$b : \langle 1, X_2, 0 \rangle \rightarrow \langle 2, 0, 0 \rangle$
$c : \langle 1, X_3, X_4 \rangle \rightarrow \langle 2, X_4, X_3 \rangle$
$d : \langle 2, 0, 0 \rangle \rightarrow \langle 3, 1, 1 \rangle$

The preconditions and net effects for ab and ac are:

$ab : \langle 0, 0, 0 \rangle \rightarrow \langle 2, 0, 0 \rangle$
$ac : \langle 0, X_1, X_1 \rangle \rightarrow \langle 2, X_1, 0 \rangle$

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, such as Work or Golf. For example, consider the two states of a 3x3 sliding-tile puzzle with 3 blanks shown in Figure 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.

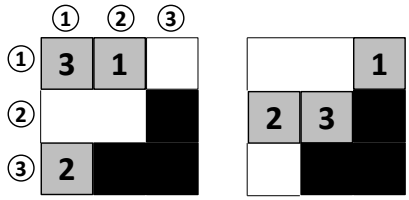

Figure 1: Two states of the 3x3 sliding-tile puzzle with 3 blanks. The black squares are tiles that are not moved in any of the shortest paths that transform the state on the left to the state on the right. In the circles are the row and column numbers.

Table 1 shows three of the 2- and 3-move operator sequences our method identified as redundant and the sequence with which each was redundant. To illustrate how our method identifies these redundancies, the derivation of Pruning Rule 2 is given in the Appendix. In Table 1 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 operator 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 1 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 1 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 Figure 2 represents an optimal (4-move) sequence to transform the left state in Figure 1 to the right state. Each one of these sequences contains one of the 2- or 3-move sequences identified as redundant in Table 1—the redundant subsequence is underlined. The arrows indicate the operator sequence that is produced when the redundant subsequence is replaced by the corresponding "Because of" sequence in Table 1. For example, the sequence in the upper

Pruning Rule #	Redundant Sequence		Because of
1	12R-11D	≡	11D-12R
2	11D-12R-21R	>	12R-11R-12D
3	11R-12D-31U	>	11D-21R-31U

Table 1: Pruning rules involved in Figure 2.

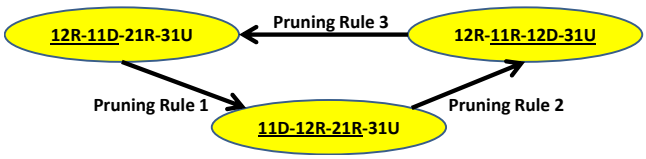

Figure 2: Each oval represents an optimal sequence transforming the state on the left of Figure 1 to the state on the right.

left oval (12R-11D-21R-31U) contains 12R-11D as its first two operators. Table 1 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 1, 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 optimal (4-move) sequences for transforming the left state in Figure 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.

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 ($cost(\varepsilon) = 0$), $pre(A)$ is the set of states to which A can be applied, and $A(s)$ is the state resulting from applying A to state $s \in pre(A)$.

The following is the formalization of the English at the beginning of this paper.

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

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

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

Lemma 1 *Let $B \geq A$ 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. $s \in pre(XAY)$.
   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 $pre(B) \subseteq pre(A)$ (because $B \geq A$). 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 $s \in pre(XAY)$.
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. $cost(XBY) = cost(XAY)$.
   Proof: $cost(XBY) \geq cost(XAY)$ follows from the cost of a sequence being additive ($cost(XBY) = cost(X) + cost(B) + cost(Y)$) and $cost(B) \geq cost(A)$ (because $B \geq A$). Since XBY is a least-cost path from s to t and XAY is also a path from s to t, $cost(XBY) = cost(XAY)$.

□

We now introduce a total ordering, $\mathcal{O}$, on operator sequences. We will use $B >_{\mathcal{O}} A$ to indicate that B is greater than A according to $\mathcal{O}$. $\mathcal{O}$ has no intrinsic connection to redundancy so it can easily happen that $B \geq A$ according to Definition 1 but $B <_{\mathcal{O}} A$.

Definition 2 *A total ordering on operator sequences $\mathcal{O}$ is "nested" if $\varepsilon <_{\mathcal{O}} A$ for all $A \neq \varepsilon$ 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 left-most 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 $B \geq A$ 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 nested ordering will be "safe": it will not eliminate all the least-cost paths between any pair of states. In our `PSVN` implementation of move pruning we generate move sequences in increasing order according to a nested ordering (the one described in Example 1). What went wrong in the implementation reported last year is that when we generated a new move sequence B we did two redundancy checks against each previously generated, unpruned sequence A: we checked

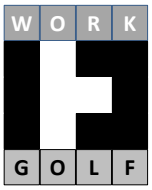

Figure 3: Goal state for the Work or Golf puzzle. It has eight 1x1 tiles with letters and 2 irregularly shaped tiles, shown in black. There are 4 empty locations.

both $B \geq A$ and $A \geq B$. If the latter occurred, we would prune A even though $B >_{\mathcal{O}} A$. Theorem 2 tells us that all we have to do to correct our implementation is to remove the check of $A \geq B$. This means we might do less pruning than before, but it guarantees the pruning will be safe. The experimental section below repeats our experiments from last year with this modification.

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.

Finally, Theorem 2 shows that cycle elimination (e.g., parent pruning) via move pruning is safe. This is captured in the following Corollary.

Corollary 3 *Let $\mathcal{O}$ be any nested ordering on operator sequences. For all B such that $B \geq \varepsilon$ according to Definition 1, B does not occur as a consecutive subsequence in $min(s,t)$ for any states s, t.*

Proof. By definition $B >_{\mathcal{O}} \varepsilon$ and therefore all the premises of Theorem 2 are satisfied. □

Experimental Results

In our 2011 paper we evaluated the effectiveness of move pruning on eight different state spaces: the 16-arrow puzzle, the 10-blocks blocks world, 4-peg Towers of Hanoi with 8 disks, the 9-pancake puzzle, the 3x3 sliding-tile puzzle (8-puzzle), 2x2x2 Rubik's cube, TopSpin with 14 tiles and a 3-tile turnstile, and the Work or Golf puzzle, whose goal state is shown in Figure 3. For each puzzle, we compared the number of nodes generated, and 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.

Here we repeat these experiments but with the modification to the move pruning algorithm described above, which guarantees that the move pruning is safe. The results are shown in the first eight rows of Table 2. All experiments were run on the same machine as last year (2.83GHz Core2 Q9550 CPU with 8GB of RAM). Node counts (in thousands of nodes) and computation times (in seconds) are totals (not averages) across 100 randomly generated start states. In a few cases, we had to generate a new set of start states instead of using exactly the same start states as last year, which means the results could be slightly different than last year's even if the effect of move pruning is the same. Also, changes have been made to the `PSVN` compiler, which has affected the execution times for some of the spaces. Less than 6 seconds were required to do the move pruning analysis for $L = 2$ and only three domains required more than 19 seconds for $L = 3$. The results in Table 2 lead to the same conclusions as last year. In domains where there are redundant short sequences beyond the 2-cycles detected by parent pruning, move pruning produces substantial speedups, ranging from 6.72x for Rubik's Cube to over 9200x for the Arrow puzzle. In do-

State Space	d	DFS+PP	DFS+MP L=2	DFS+MP L=3
16 arrow puzzle	15	? >3600s	3,277 0.39s 0.07s	3,277 0.39s 18.58s
10 blocks world	11	352,028 25.02s	352,028 12.23s 5.97s	352,028 12.53s 8m 27s
Towers of Hanoi	10	1,422,419 97.02s	31,673 1.45s 0.30s	9,060 0.49s 3m 43s
pancake puzzle	9	5,380,481 246.49s	5,380,481 115.22s 0.01s	5,288,231 111.18s 0.02s
8-puzzle	25	368,357 24.77s	368,357 10.40s 0.01s	368,357 10.40s 0.08s
Rubik's cube 2x2x2	6	2,715,477 132.74s	833,111 20.00s 0.02s	515,614 13.35s 1.10s
TopSpin	9	? >3600s	2,165,977 73.80s 0.00s	316,437 12.59s 1.11s
Work or Golf	13	? >3600s	209,501 16.44s 2.98s	58,712 5.14s 15m 4s
Gripper	14	9,794,961 544.85s	590,870 17.22s 0.08s	25,982 0.95s 0.85s

Table 2: 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.

mains where the only redundant short sequences are 2-cycles, move pruning is faster than parent pruning based on state comparison by a factor of two or more.

The last row in Table 2 is new, it 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 optimal 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 2 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 an optimal solution length of 29. As can be seen, even just pruning short redundant sequences ($L = 2$ or 3) has a very large effect (over 500x 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 40MB 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 seconds, 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 seconds. The number of nodes generated is $157,568,860$.

Related Work

Wehrle and Helmert (2012) 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 (Godefroid 1996). Sleep sets exploit the commutativity of operators.[2] 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.[3]

A different, but related approach to avoiding redundant move sequences is state space "factoring" (Lansky and Getoor 1995; Lansky 1998; Amir and Engelhardt 2003; Brafman and Domshlak 2006; Fabre et al. 2010; Guenther, Schiffel, and Thielscher 2009). 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 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 (Sharon et al. 2011; Standley 2010) and additive abstractions (*cf.* the independent abstraction sets defined by Edelkamp (2001) and the Factor method by Prieditis (1993)).

Transposition tables (Akagi, Kishimoto, and Fukunaga 2010) 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 dupli-

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

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

cate states.[4] 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 $\Theta(m^L)$, 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 $\Theta(a^2)$ 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 $\Theta(n^2)$ 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 seconds (see Table 2). If our transposition table implementation is restricted to use 7 kilobytes, depth-first search is unable to finish depth 13 in 6 minutes. 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 showed in last year's paper that the two methods can interact with each other to produce erroneous behaviour (rendering some reachable states unreachable). The modification we have presented in this paper to make move pruning safe does not make it safe to use in combination with transposition tables.

Conclusions

In this paper we have shown that the move pruning method presented last year would, in some circumstances, prune all the least-cost paths from one state to another. We have presented a simple, abstract example in which move pruning will behave erroneously and a slightly more complex example that arose in our experiments last year with the Work or Golf domain. We then formally proved that a simple modification—requiring move pruning to respect a fixed nested ordering on operator sequences—was "safe", i.e., would never prune all the least-cost paths between a pair of states. Imposing this restriction did not noticeably reduce the amount of pruning done in our experiments compared to last year, 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.

[4]The efficiency gained by avoiding generating unneeded nodes, as opposed to generating and testing them, is the entire motivation for Enhanced Partial Expansion A* (Felner et al. 2012), 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.

Acknowledgements

Thanks to Shahab Jabbari Arfaee for encoding the Gripper domain in PSVN and running the initial experiment that exposed the problem, to Sandra Zilles for her thoughtful feedback on early versions of the manuscript and help with the theoretical section of the paper, and to Malte Helmert and Martin Wehrle for discussions about the partial order reduction methods described in (Wehrle and Helmert 2012). We gratefully acknowledge the financial support of NSERC and Alberta Innovates Technology Futures.

Appendix. Derivation of Pruning Rule 2

In PSVN states are represented by vectors of a fixed length. In this example, we will use vectors of length 6, where the first 3 positions of the vector represent Row 1 (Columns 1, 2, and 3) and the next 3 represent Row 2. In reality, there would be 3 additional vector positions for the third row, but since they are not needed in this example they are omitted for brevity.

An operator in PSVN is of the form `LHS => RHS` where `LHS` and `RHS` are both vectors the same length as a state vector. `LHS` specifies the operator's preconditions, `RHS` specifies its effects. Figure 4 shows the PSVN definitions for the operators involved in Pruning Rule 2. A capital letter (`V, W,`...) is a variable symbol that will be bound to the corresponding value in the state vector when the operator is applied to a state. `b` is a constant used to represent that the corresponding puzzle position is blank. A dash (`-`) in the `LHS` indicates that the corresponding state position is irrelevant, a dash in the `RHS` indicates that the operator does not change the value in that position.

The first step in deriving a pruning rule is to compute the macro-operators for the two sequences (in this example, 11D-12R-21R and 12R-11R-12D). A macro-operator for an operator sequence has exactly the same form as an ordinary operator but its `LHS` specifies the

```
V - -  b - -  =>  b - -  V - -   LABEL 11D
- W b  - - -  =>  - b W  - - -   LABEL 12R
- - -  X b -  =>  - - -  b X -   LABEL 21R
Y b -  - - -  =>  b Y -  - - -   LABEL 11R
- Z -  - b -  =>  - b -  - Z -   LABEL 12D
```

Figure 4: PSVN rules involved in Pruning Rule 2.

preconditions required to execute the entire sequence and its RHS specifies the net effect of executing the entire sequence.

The macro-operator is built up through a "move composition" process starting with the identity operator

```
A B C   D E F  =>  A B C  D E F
```

and updating it with one operator at a time from the sequence to create the macro-operator for the entire sequence. The update process involves two steps. The first is to "unify" the RHS of the current macro-operator with the LHS of the next operator. This determines whether the sequence can be executed at all and, if so, what constraints there are on the variables (A, B,...) in the LHS of the macro-operator. In our example, unifying the RHS of the current macro-operator (the identity operator) with operator 11D, adds the constraint that D=b. The second step in the update process is to update the effects of the macro-operator. This is simple once the first step has been done, because that step has bound symbols in the macro-operator's RHS to the variable symbols in the operator's LHS. Transferring these to the operator's RHS gives the RHS for the updated macro-operator. In our example, the update process based on 11D produces the macro-operator

```
A B C   b E F  =>  b B C  A E F.
```

The process is then repeated using this updated macro-operator and the next operator in the sequence, 12R in this example. First the RHS of the macro-operator, b B C A E F, is unified with the LHS of 12R. This produces the constraint C=b and bindings which, when transferred to the RHS of 12R, result in the following macro-operator:

```
A B b   b E F  =>  b b B  A E F.
```

The same process is applied with this macro-operator and the third operator in the sequence, 21R, to produce the final macro-operator for 11D-12R-21R:

```
A B b   b b F  =>  b b B  b A F.
```

Applying the move composition process to operator sequence 12R-11R-12D produces this macro-operator:

```
A B b   D b F  =>  b b B  D A F.
```

The two macro-operators can now be compared, using Definition 1, to determine if one is redundant with the other. First, their costs are the same (3). Second, it is easy for the PSVN compiler to determine that any state that matches the LHS of the macro-operator for 11D-12R-21R will also match the LHS of the macro-operator for 12R-11R-12D, since the LHS's are identical except that the former requires the fourth vector position to be b whereas the latter allows it to be any value. Finally, it is easy for the PSVN compiler to determine that the net effect of both sequences (the RHS of both macro-operators) is identical for any state that matches the LHS of the macro-operator for 11D-12R-21R. By this means the PSVN compiler establishes that 11D-12R-21R > 12R-11R-12D. Last year's system immediately declared 11D-12R-21R redundant. As we have shown in this paper, this is not safe to do, in general. The correction we propose in this paper is to only declare 11D-12R-21R redundant if it follows 12R-11R-12D in the fixed nested order being used.

References

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

Amir, E., and Engelhardt, B. 2003. Factored planning. In *IJCAI*, 929–935.

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when and when not. In *AAAI*, 809–814.

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

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. European Conference on Planning*, 13–24.

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

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Sturtevant, N.; Schaeffer, J.; and Holte, R. C. 2012. Partial-expansion A* with selective node generation. In *AAAI*.

Godefroid, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer.

Guenther, M.; Schiffel, S.; and Thielscher, M. 2009. Factoring general games. In *IJCAI Workshop on General Game Playing (GIGA'09)*.

Lansky, A. L., and Getoor, L. 1995. Scope and abstraction: Two criteria for localized planning. In *IJCAI*, 1612–1619.

Lansky, A. L. 1998. Localized planning with diverse plan construction methods. *Artificial Intelligence* 98(1-2):49–136.

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011. The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI*, 662–667.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 173–178.

Taylor, L. A., and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *AAAI*, 756–761.

Taylor, L. A. 1992. Pruning duplicate nodes in depth-first search. Technical Report CSD-920049, UCLA Computer Science Department.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In *Proceedings of ICAPS*.