Automatic Move Pruning in General Single-Player Games

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

Move pruning is a low-overhead technique for reducing the size of a depth first search tree. The existing algorithm for automatically discovering move pruning information is restricted to games where all moves can be applied to every state. This paper demonstrates an algorithm which handles a general class of single player games. It gives experimental results for our technique, demonstrating both the applicability to a range of games, and the reduction in search tree size. We also provide some conditions under which move pruning is safe, and when it may interfere with other search reduction techniques.

Introduction

Depth first search (DFS) of a tree is a common technique in graph search. Algorithms like IDA* (Korf 1985) use it, taking advantage of the low space requirements: it is linear with the tree depth, rather than in the number of vertices. Without other enhancements, however, a DFS may generate the same graph vertex multiple times. This is almost always wasted work. The search can even require exponential time to explore a polynomial space like an m by m grid. Eliminating duplicate states can mean very large savings.

Taylor and Korf (1992; 1993) introduce a method for automatically analysing a game to detect sequences of moves which are guaranteed to only visit duplicate states in a DFS. The offending move sequences are encoded in a finite state machine (FSM) with transitions based on the moves made during search. They use the technique in IDA* with two variants of the N-puzzle sliding tile puzzle (15 and 24 tiles) and Rubik's cube, getting large reductions in generated states for a very modest cost per state.

The combination of large savings, good performance, and automated analysis make it a compelling tool for a general single agent game system. If a description of the game is part of the input, only a general system can be used: there is no opportunity here for someone to hand code a game-specific duplicate detection method.

Taylor and Korf's algorithm (1993) works on a range of combinatorial games, but in the context of more general games there is an unresolved issue involving the preconditions of rules. The method assumes a generic state where all moves can be applied, which is not the case in many puzzles, including the N-puzzle. Taylor (1992) uses a workaround to this problem for the 15 and 24 puzzle (embedding them in a much larger 7x7 sliding tile grid), but this was done by hand and is not generalisable.

In this paper, we introduce an algorithm for automatically detecting sequences of moves in a general class of perfect information, non-stochastic, single player games with non-negative move costs. Given a game description in the `PSVN` language (described in the Appendix), we build and analyse a tree of "macro-rules" which describe the preconditions and results of sequences of multiple moves. We show that pruning based on this analysis is safe in conjunction with cycle detection and heuristic pruning, but may cause issues when combined with a transposition table. Experimental results are given for a range of puzzles, demonstrating both the utility and the generality of our method.

State and Rule Representation

A state is a vector $\langle v_1, v_2, ..., v_N \rangle$ 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 $\langle v_1, ..., v_N \rangle$ to another state $\langle v'_1, ..., v'_N \rangle$, with an associated cost c. In a single-player game, all the moves are chosen by the same player, and the objective is to transform an initial state into a state that satisfies a set of goal conditions.

Our running example of a single-player game is the N-Arrow puzzle (Korf 1980). A state consists of N "arrows", each of which is either pointing up or down. This is represented by a vector of N values all drawn from a single binary domain (we shall use 1 to represent "up" and 0 to represent "down"). There are $N - 1$ different moves that can all be applied to any state. Move i flips the arrows in positions i and $i + 1$ (for $i \in \{1, \ldots, N - 1\}$).

We deal with games 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 actions where $v'_i = d \in D_i$ or $v'_i = v_j$. All commonly used planning and search testbed problems 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 move. In the N-Arrow puzzle, for example, we described move 1 as flipping arrows 1 and 2. In our notation, the four rules shown in the Table 1

would be required to implement this move. The distinction made here between a conceptual move and rules in a formal game description is important, but we will henceforth only consider the formal game in our notation, and will use the terms rule and move interchangeably to describe a state transition governed by applying one of these formally described rules.

Rule	Preconditions	Actions
R1-00	$v_1 = 0, v_2 = 0$	$v_1' = 1, v_2' = 1$
R1-01	$v_1 = 0, v_2 = 1$	$v_1' = 1, v_2' = 1$
R1-10	$v_1 = 1, v_2 = 0$	$v_1' = 1, v_2' = 1$
R1-11	$v_1 = 1, v_2 = 1$	$v_1' = 0, v_2' = 0$

Table 1: 4-Arrow puzzle, rules representing move 1.

A rule of this type can be represented as a cost and two vectors of length N: $\vec{p} = \langle p_1, ..., p_N \rangle$ for the preconditions and $\vec{a} = \langle a_1, ..., a_N \rangle$ for the actions, 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 and 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 $p_i = x_j$, and the actions $v_i' = d$ 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 $D_j = D_i$.

Rule	Preconditions	$\rightarrow$	Actions	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}

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

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 actions, 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 action 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 action 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 action 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 action 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 actions.

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 actions: 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. A fact about the canonical representation of preconditions that we will use later is this: if $p_i = x_j$ and $i \neq j$ then it must be true that $p_j = x_j$.

Composition of Move Sequences

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

We will proceed by induction. As the base case of the induction, the empty sequence can be represented as a rule with cost 0, preconditions $\langle x_1, ..., x_N \rangle$ and actions $\langle x_1, ..., x_N \rangle$. Now assume that any valid sequence of k rules can be represented as a single macro-rule with cost c_1, preconditions $\vec{p}^1$, and actions $\vec{a}^1$ and consider how to compute the macro-rule for the extension of this sequence by one additional rule (the $(k+1)^{st}$ rule) , with cost c_2, preconditions $\vec{p}^2$ and actions $\vec{a}^2$.

The cost of the extended sequence is simply $c_1 + c_2$. The preconditions and actions of the extended sequence, $\vec{p}^*$ 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

[1] Our usage of this term is the same as in the literature on learning macro-rules (Finkelstein and Markovitch 1998), but in this paper, macro-rules are used only for analysis, not as new move options that are available at run time.

the preconditions one at a time. For each precondition, p_i^2, there are two main cases, depending on whether the 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. $\mathbf{a_i^1}$ **is a constant.**
   (a) $\mathbf{p_i^2}$ **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 i^{th} 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 i^{th} 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) $\mathbf{p_i^2 = x_j}$ **and** $\mathbf{a_j^1}$ **is a constant.** $p_i^2 = x_j$ means the i^{th} and j^{th} 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) $\mathbf{p_i^2 = x_j}$ **and** $\mathbf{a_j^1 = x_k}$**.** Again, $p_i^2 = x_j$ means the i^{th} and j^{th} 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. $\mathbf{a_i^1 = x_j}$.
   (a) $\mathbf{p_i^2}$ **is a constant.** All occurrences of x_j in $\vec{p}^*$ and $\vec{a}^*$ are replaced with the constant (p_i^2).
   (b) $\mathbf{p_i^2 = x_j}$. In this case the i^{th} 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) $\mathbf{p_i^2 = x_k}$ **for some** $\mathbf{k \neq j}$ **and** $\mathbf{a_k^1}$ **is a constant.** All occurrences of x_j in $\vec{p}^*$ and $\vec{a}^*$ are replaced with the constant (a_k^1).
   (d) $\mathbf{p_i^2 = x_k}$ **for some** $\mathbf{k \neq j}$ **and** $\mathbf{a_k^1 = x_t}$**.** 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 $k+1$ moves. $\vec{a}^*$ only describes how these states would be modified by the first k moves. We must modify $\vec{a}^*$ by applying $\vec{a}^2$ to it. This requires another copy of $\vec{a}^{copy} = \vec{a}^*$. 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 $a_i^* = a_j^{copy}$.

$\vec{p}^*$, $\vec{a}^*$, and cost $c_1 + c_2$ are now a rule with the preconditions and effects of the entire length $k+1$ move sequence. This completes the inductive proof that any valid sequence or rules of any length can be represented by a single macro-rule. The proof was constructive, and indeed, describes exactly how our algorithm for enumerating sequences operates.

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 action 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 action vector is a constant (0) 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 action 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 action 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 action 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 action vector (positions 2 and 3) are directly copied into the same positions in $\vec{a}^*$, and, if position i of R2-11's action vector is the variable symbol x_j, position i in $\vec{a}^*$ is 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 symbol x_4). The final macro-rule in this example is thus $\langle 0, 0, 1, x_4 \rangle \rightarrow \langle 1, 0, 0, x_4 \rangle$.

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

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

Move Pruning

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

We can prune a move sequence B if we can always use move sequence A instead to reach the same state at no additional cost. More formally, following Taylor and Korf (1993) 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.

Condition *(i)* is trivial to check since the cost of each sequence is calculated by the move 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 action vectors as augmented states, checking condition *(iii)* is also straightforward: we can simply apply the actions $\vec{a}^A$ 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: move sequences A and B might be exactly equivalent so that either could be pruned, but it can be the case that A lets us prune B, but B does not let us prune A. For example, move 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 move 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.

There are a number of implementation details to consider. We must check each sequence against all other sequences, which is $\Theta(N^2)$ 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 games), we are limited to fairly shallow trees (two or three moves).

Even with small trees, in the interest of efficiency, it is best to prune move sequences as soon as possible. As Taylor and Korf (1993) 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. That way, if we find any short move sequences that can be pruned, we can use this information to immediately prune any longer move sequences that include the shorter sequences we have previously pruned. Using breadth-first search also avoids a potential problem when combining move pruning with a separate cycle detection system in games with moves with cost 0 (see below).

Taylor and Korf (1993) encode the list of pruned move sequences in a compact FSM, with transitions between states based on the last move made. The same algorithm could be used to generate a compact FSM for the move pruning information we generate, but 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 un-pruned move sequence in the interior of the tree. In our implementation, where we consider all sequences up to length l, this is all move sequences of length $l-1$ that were not discovered to be redundant. This tag corresponds to the FSM state. If there are M moves defined in the game, and we used T tags, we construct a table with M entries for each of the T tags. The entry for a move sequence s and subsequent move m is set to -1 if the new move sequence $s + m$ is pruned. Otherwise, the entry is the tag of the last $l - 1$ moves of $s + m$. This encodes the transition rules between FSM states.

Mixing with Cycle Detection

Pruning move sequences up to length l does not guarantee that there will be no cycles. There may be cycles longer than l, which the move pruning will necessarily be unable to detect. There may also be cycles of length l or less which are not pruned because they only occur in certain circumstances. For example, consider a move 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 move is an identity operator (1-cycle). For arbitrary states, of course, this move is not an identity operator, so we cannot prune away all occurrences of it. It is possible, using a slight variant of our Move Composition algorithm, to derive the special conditions under which a move sequence is a cycle or is equivalent to another sequence, but doing so quickly generates an impracticably large number of special conditions. We therefore only prune a move sequence that is universally redundant with another move sequence, as defined by the three conditions for pruning identified by Taylor and Korf (1993) and reiterated in the previous section. We refer to move sequences that create cycles, or are redundant with other move sequences, under special circumstances that are not detected by our generic move pruning system as serendipitous cycles and redundancies.

If someone expected there to be many serendipitous cycles, they might wish to use run-time cycle detection in addition to move pruning, to terminate those cycles.[2] As we shall show below, in general, combining search reduction techniques is unsafe, even if one of them is as seemingly innocuous as cycle detection (Akagi, Kishimoto, and Fukunaga 2010). We now prove that it is safe to use both move pruning and cycle detection, first considering the case where there are no moves with a cost of 0.

We introduce some new notation for the proof. $Tree(S, b)$ is the set of all paths which are taken during a tree traversal starting at state S with an upper bound b on path cost. $Visited(S, b)$ is the set of states which are visited on this same traversal.

Move pruning has the property that $Visited_{MP}(S, b) = Visited(S, b)$, and for all T in $Visited(S, b)$ the least cost path to T in $Tree_{MP}(S, b)$ is no more expensive than the least cost path to T in $Tree(S, b)$. We would like to show something similar for move pruning and cycle detection.

[2] 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.

$Visited_{MP+cycle}(S,b) \subset Visited_{MP}(S,b)$ is trivially true: we are only removing more nodes from the traversed tree by adding a second pruning method. To show the other direction and the cost constraint, consider $T \in Visited_{MP}(S,b)$. There is some minimum cost path p in $Tree_{MP}(S,b)$ which leads to T. p must contain no cycles: otherwise, there would be some shorter path in $Tree(S,b)$ which removes the cycle, contradicting the assumption that p has minimum cost.

Because there are no cycles in p, cycle detection does not remove it, p is in $Tree_{MP+cycle}(S,b)$ and T is in $Visited_{MP+cycle}(S,b)$. Therefore, the least cost path to T is preserved, $Visited_{MP}(S,b) \subset Visited_{MP+cycle}(S,b)$ and we are finished.

If there are moves with a cost of 0, the assumption that a minimum cost path p in $Tree_{MP}(S,b)$ must have no cycles may not be true. We must use the fact that we generate move sequences in a breadth-first fashion, or otherwise consider modifying the cost test so that p_1 is considered to have lower cost than p_2 if $cost(p_1) < cost(p_2)$, or if $cost(p_1) = cost(p_2)$ and $length(p_1) < length(p_2)$. In either case, we have the property that p must have no cycles, and the proof continues as before.

Mixing with Heuristic Cutoffs

Using a heuristic to do cutoffs during search with move pruning is safe. In this context, we have a more specific tree traversal: we are looking for goal states. If we are at state T with a path of cost c, an admissible heuristic value h and a bound $b < c+h$, we have a guarantee that $Visited(T, b-c)$ does not contain any goal states. If we cut off search at T, we have not explicitly performed the search below T, but we can reason about every state we would have visited in an un-pruned tree and say that they would not have been goal states. Neither move pruning, cycle detection, transposition tables, nor the search for a goal itself require that we explicitly visit these states. Even if we haven't done so by visiting the states, we have effectively done the entire search below T, just as if we had not done the cutoff.

Mixing with Transposition Tables

If we use a transposition table (TT) (Reinefeld and Marsland 1994) and move pruning, IDA* may run into the graph history interaction (GHI) problem (Akagi, Kishimoto, and Fukunaga 2010). The TT assumes that everything is described by the state. This assumption is false when there is additional information, like the current history, which may affect the search result.

Figures 1(a) to 1(c) demonstrate an example of this. To keep the example as simple as possible, we are using a contrived game, described by the rules in Figure 1(a). All moves have unit cost, and the goal state is $\langle 1,1,0,1 \rangle$. The vector before the arrow is the augmented state giving the preconditions, and the augmented state after the arrow describes the actions.

Building a length 2 move sequence tree, we create two new macro-rules by appending $R3$ to $R1$ and $R2$. $R3(R2(s))$ cannot be pruned: it can be applied to the state $\langle 0,1,0,0 \rangle$ to get $\langle 1,1,0,1 \rangle$, which does not hold for any of the other move sequences. We can, however, prune $R3(R1(s))$. Both sequences have a cost of 2. The only state we can use for $R3(R1(s))$ is $\langle 0,0,0,0 \rangle$, which can be used for $R3(R2(s))$. Finally, $R3(R2(\langle 0,0,0,0 \rangle))$ produces the same result as $R3(R1(\langle 0,0,0,0 \rangle))$.

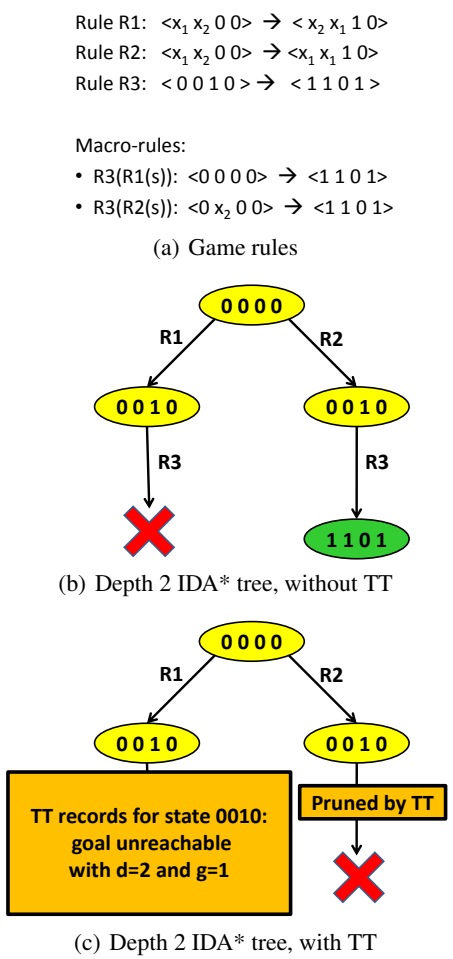

Figure 1: GHI example

Figure 1(b) shows the behaviour of IDA* with a start state of $\langle 0,0,0,0 \rangle$ when the maximum depth is 2. At depth 0, we apply $R1$. After this, the only applicable rule is $R3$, but we have pruned $R3(R1(s))$. With no valid children, we return failure back up to the root of the tree. Back at depth 0, we now apply $R2$. From here, the only applicable rule is $R3$. $R3(R2(s))$ is valid, so we reach the goal and return a length 2 path.

Figure 1(c) shows the IDA* behaviour if we use a TT. After making move $R1$ we first check if the state is in the TT. It is not, so we proceed and find no valid children. We note the depth information and failure to find a goal in the TT, using the current state $\langle 0,0,1,0 \rangle$ as a key. We then return failure. Back at the root, we try $R2$ and get the state $\langle 0,0,1,0 \rangle$. We look in the TT for this state and discover an entry. The depth information is valid, so we re-use the entry and return fail-

ure. We get back to the root, where there are no moves. The iteration has finished, and we have failed to find a legitimate path to the goal.

Experimental Results

Our algorithm applies to general single player games. To test it, we implemented the move pruning algorithm in a PSVN programming environment that we have developed. This tool kit takes a game 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 (functionally similar to the FSM of Taylor and Korf (1993)). We then modified the provided generic DFS code to use this pruning information.

We used 7 different puzzles: the N-arrow puzzle (Korf 1980) with 16 arrows, the blocks world (Slaney and Thiébaux 2001) with 10 blocks, 4-peg Towers of Hanoi (Hinz 1997) with 8 disks, the pancake puzzle (Dweighter 1975) with 9 pancakes, the 8-puzzle sliding tile puzzle (Slocum and Sonneveld 2006), 2x2x2 Rubik's cube (Taylor and Korf 1993), TopSpin (Chen and Skiena 1996) with 14 tiles and a 3 tile turnstile, and the Work or Golf puzzle, which is a sliding tile puzzle variant with some irregular pieces, described by Figure 2. In our PSVN game 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 small Rubik's cube puzzle had 18 rules, TopSpin had 14 rules, and Work or Golf had 93 rules.

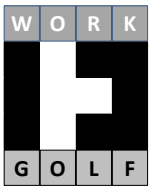

Figure 2: The Work or Golf puzzle. It has eight 1x1 pieces with letters and two irregularly shaped pieces, shown in black. All ten pieces are movable. There are 4 empty 1x1 locations.

As written in PSVN, the arrow puzzle, blocks world, Towers of Hanoi, the 8-puzzle and the Work or Golf puzzle had rules with preconditions, which precludes using the method of Taylor and Korf (Taylor and Korf 1993). While there may be other formulations of a game or game-specific workarounds that get around this for the arrow puzzle and 8-puzzle, this is not possible in a general framework, and blocks world and Towers of Hanoi will always have rules with preconditions.

For each puzzle, we ran three searches: (1) DFS with parent pruning (length 2 cycle detection), (2) DFS using a move sequence tree limited to sequences of length 2, and (3) DFS with a length 3 move sequence tree. The games have different branching factors, so we used a different depth limit for the DFS in each puzzle.

All length 2 move sequence trees took a few seconds to build: the slowest game was blocks world, at 3 seconds. Blocks world was also the slowest game for the length 3 tree, taking 13 minutes. The 8-puzzle took less than a second for both trees. This is a one time cost: once the tree is built and the table has been output, the generated code has all necessary information and we do not need to analyse the game again.

The results are given in Table 4. All experiments were run on a machine with a 2.83GHz Core2 Q9550 CPU and 8GB of RAM. Node counts (in thousands of nodes) and computation times are totals across 100 randomly generated start states.

Game	d	DFS+PP	DFS+MP l=2	DFS+MP l=3
16 arrow puzzle	15	? >3600s	3,277 0.39s	3,277 0.39s
10 blocks world	11	594,371 43.10s	594,371 22.22s	594,371 22.53s
Towers of Hanoi	10	1,423,401 97.05s	31,652 1.52s	9,041 0.51s
pancake puzzle	9	5,380,481 251.11s	5,380,481 66.10s	5,288,231 65.24s
8-puzzle	25	359,254 23.70s	359,254 10.49s	359,254 10.49s
Rubik's cube 2x2x2	6	2,715,477 136.58s	833,111 20.32s	515,614 13.58s
TopSpin	9	? >3600s	2,165,977 53.60s	316,437 9.96s
Work or Golf	13	? >3600s	232,524 19.76s	59,905 5.70s

Table 4: Total work, in thousands of generated nodes and seconds, to search 100 random problem instances to the given depth d.

As was seen by Taylor and Korf (1993), we see here that using DFS with move pruning results can result in much smaller search trees than DFS with parent pruning. Even with shallow move pruning trees, we can do at least as well parent pruning, and in those cases where the tree is no smaller, move pruning requires less time than parent pruning. This holds across a broad range of puzzles.

The N-arrow puzzle, which we have been using as a running example, has an interesting property. Any move sequence i, j has the same result as j, i, and i, i has no effect. If we build a move sequence tree with 2 move sequences, the move pruning algorithm 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 inverse, 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 puzzle and the 8-puzzle: increasing the maximum length in the move sequence tree may not discover additional duplicates. 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, these puzzles will eventually find additional duplicate states with larger trees. In the 8-puzzle we will see additional pruning if we consider move sequences of length 6.

In blocks world, the pancake puzzle, and the 8-puzzle, we see that the number of generated nodes is the same whether we use parent pruning or our move pruning system with a length 2 move sequence tree. In these puzzles, 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 that we compare two states, which is much more expensive. For exactly this reason, hand-implemented systems for a specific puzzle generally do parent pruning by skipping moves which undo the previous action (a form of move pruning) if the domain supports this.

The blocks world results have one final strange feature: the total time for the length 3 tree is larger than for the length 2 tree. Ideally, we would expect this to be the same. In this case, because we did not implement the FSM of Taylor and Korf (1993), the move pruning table grows as $O(R^l)$, where R is the number of rules and l is the maximum sequence length in the tree. The blocks world puzzle had 200 rules, so the $l = 3$ table was fairly large (around 30MB) and we suspect cache performance suffered when using this larger table.

Readers looking at Taylor and Korf's work (1993) will notice that we used smaller trees. They build a depth 7 tree for the 2x2x2 Rubik's cube puzzle, and a depth 14 tree for the 15-puzzle. There are two factors which affect us, related to the size of the move sequence tree. A length l move sequence tree will have at most $\sum_{i=0}^{l} R^i$ move sequences. We must generate all of these sequences, barring any sequences which contain subsequences we have already pruned. This is the same in Taylor and Korf's work.

The first difference is that Taylor and Korf (1993) had 4 rules for both the 8-puzzle and 15-puzzle. The `PSVN` description of the 8-puzzle has 24 rules. This explains the especially large discrepancy for the 8-puzzle: 24^{14} is many orders of magnitude larger than 4^{14}.

The second difference is that by keeping track of a state, Taylor and Korf (1993) were able to speed up duplicate detection by sorting generated states. Within our environment of general games, we must generate macro-rules, and the pruning check is more complicated than a simple state comparison. This precludes the same sorting technique, and we are reduced to comparing each newly generated macro-rule against all previously generated macro-rules. Ignoring the move sequences that get pruned, the 2x2x2 Rubik's cube puzzle has around $650M$ move sequences in a length 7 move sequence tree. Doing on the order of $650M * log_2(650M)$ comparisons by using a sorted list is reasonable, but doing around $650M^2$ comparisons is not. As is often the case, there is a cost to handling a more generic class of problems.

Conclusions

In this paper, we introduced an algorithm for automatically analysing a general single player game and detecting sequences of moves which lead to duplicate states in DFS. We applied the algorithm to a number of domains which are not handled by the existing move pruning algorithm, and experimentally demonstrated speedups in these games ranging from two to hundreds of times faster than using parent pruning. Comparing the one-time analysis cost to the existing method, we are asymptotically slower. Looking at ways to improve this cost is an area for future work.

Mixing multiple techniques which prune nodes is potentially unsafe, in the sense that minimum cost paths between states may not be discovered. Using cycle detection and a TT is known to be unsafe, and we give an example showing that using move pruning and a TT is unsafe as well. We prove that in DFS, move pruning is safe when combined with cycle detection of any length, and that adding pruning based on a heuristic is safe.

Acknowledgements

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).

References

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

Chen, T., and Skiena, S. S. 1996. Sorting with fixed-length reversals. *Discrete Applied Mathematics* 71:269–295.

Dweighter, H. 1975. Problem E2569. *American Mathematical Monthly* 82:1010.

Finkelstein, L., and Markovitch, S. 1998. A selective macro-learning algorithm and its application to the nxn sliding-tile puzzle. *Journal of Artificial Intelligence Research* 8:223–263.

Hernádvölgyi, I., and Holte, R. 1999. PSVN: A vector representation for production systems. Technical Report TR-99-04, Department of Computer Science, University of Ottawa.

Hinz, A. M. 1997. The tower of hanoi. In *Algebras and Combinatorics: Proceedings of ICAC97*, 277289. Hong Kong: Springer-Verlag.

Korf, R. E. 1980. Towards a model of representation changes. *Artificial Intelligence* 14(1):41–78.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.* 16(7):701–710.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125:119–153.

Slocum, J., and Sonneveld, D. 2006. *The 15 Puzzle*. Slocum Puzzle Foundation.

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.

Appendix: The `PSVN` Language.

Our `PSVN` is a slight extension of the language with the same name introduced by Hernádvölgyi and Holte (1999). It very directly implements the state and rule representations described above.

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 $s = \langle s_1 ... s_n \rangle$ matches $LHS = \langle L_1 ... L_n \rangle$ 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 $s = \langle s_1 ... s_n \rangle$ matching its LHS is to create a state $s' = \langle s'_1 ... s'_n \rangle$ 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". A non-deterministic rule applied to state $s = \langle s_1 ... s_n \rangle$ creates a set of states, with one successor for every possible combination of values for 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 $\langle 2, 1, 2, 2 \rangle$.

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.