Move Pruning and Duplicate Detection

Robert C. Holte

Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
rholte@ualberta.ca

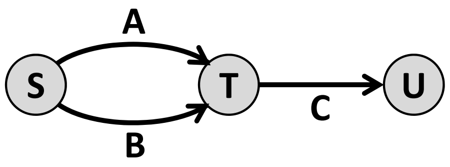
Abstract. This paper begins by showing that Burch and Holte's move pruning method is, in general, not safe to use in conjunction with the kind of duplicate detection done by standard heuristic search algorithms such as A*. It then investigates the interactions between move pruning and duplicate detection with the aim of elucidating conditions under which it is safe to use both techniques together. Conditions are derived under which simple interactions cannot possibly occur and it is shown that these conditions hold in many of the state spaces commonly used as research testbeds. Unfortunately, these conditions do not preclude more complex interactions from occurring. The paper then proves two conditions that must hold whenever move pruning is not safe to use with duplicate detection and discusses 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.

1 Introduction

Burch and Holte [1, 2] introduced a generalization of the method for eliminating redundant operator sequences introduced by Taylor and Korf [3, 4], proved its correctness, and showed that it could vastly reduce the size of a depth-first search tree in spaces containing short cycles or transpositions.[1] Both methods work by pruning moves, i.e. disallowing ("pruning") the use of an operator ("move") after a specific sequence of operators has been executed. Burch and Holte also showed that move pruning could not, in general, be safely used in conjunction with transposition tables [5]; i.e. there is a risk, if move pruning is used together with transposition tables, that all optimal paths from start to goal will be eliminated.

A*, breadth-first search, and many other search algorithms use a duplicate detection strategy that is simpler than the transposition tables considered by Burch and Holte. Such algorithms simply record each state that is generated and 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

[1] A "transposition" occurs when there are two distinct paths leading from one state to another.



$$\begin{tikzcd} S \arrow[r, bend left, "A"] \arrow[r, bend right, "B"'] & T \arrow[r, "C"'] & U \end{tikzcd}$$

Fig. 1. Example in which duplicate detection and move pruning interact to produce erroneous behaviour.

new path to the state is ignored. I shall refer to this as "duplicate detection" in the remainder of this paper.

Burch and Holte did not discuss whether it was safe to use their move pruning in conjunction with duplicate detection, but it was observed by Malte Helmert (personal communication) that it is not. Figure 1 shows the typical 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.

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) presented in the notation of the `PSVN` language (see [2]). A state in this example is described by three state variables and is written as a vector of length three. The value of each variable is either 0, 1, or 2. 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. The following operators behave like A, B, and C in Figure 1 when applied to state $S = \langle 0, 1, 1 \rangle$:

$A : \langle 0, X_1, X_2 \rangle \rightarrow \langle 1, 1, X_2 \rangle$
$B : \langle 0, X_3, X_4 \rangle \rightarrow \langle 1, X_3, 1 \rangle$
$C : \langle 1, X_5, X_6 \rangle \rightarrow \langle 2, 1, 1 \rangle$

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, $T = \langle 1, 1, 1 \rangle$.

This example motivates the study reported in this paper, whose aim is to understand the interactions between move pruning and duplicate detection and to identify conditions under which it is safe to use the techniques together. There are two main contributions of this paper. The first is to derive conditions under which simple interactions between move pruning and duplicate detection, such as the one depicted in Figure 1, cannot possibly occur. It turns out that these conditions hold in many of the state spaces commonly used as research testbeds (Rubik's Cube, TopSpin, etc.). Unfortunately, these conditions do not preclude more complex interactions from occurring. The second contribution of the paper is to derive conditions that must hold whenever there is a deleterious interaction between move pruning and duplicate detection.

1.1 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 will be a duplicate. For example, the depth-first search system used in Burch and Holte's experiments [1] did "parent pruning," an elementary form of duplicate detection, and they reported that using move pruning to achieve the same effect as parent 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 undertaking this study is that it applies much more broadly than just to systems that use Burch and Holte's method for automatic move pruning. When problem domains with many obvious redundancies, such as TopSpin and Rubik's Cube, are coded by hand, the researchers writing the code manually do a simple version of the move pruning that Burch and Holte have automated. For example, here is a detailed description of the standard move pruning done by hand for Rubik's Cube [6]:

> 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. Further-

more, 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 Burch and Holte's method detects automatically. The correctness of the move pruning done manually has never been questioned, but the problem illustrated in Figure 1 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, I have verified that the manually encoded move pruning in the IDA* code written in my research group for TopSpin results in non-optimal solutions being produced if it is used in A*.

2 Essential Theory by Burch and Holte [1]

This section defines terminology and repeats the key theoretical ideas from [1].

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 $s \in pre(A)$. I assume the cost of each operator is non-negative. A prefix of A is a nonempty initial segment of A ($A_1 ... A_k$ for $1 \leq k \leq |A|$) and a suffix is a nonempty final segment of A ($A_k ... A_{|A|}$ for $1 \leq k \leq |A|$).

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,

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

(R1) $cost(B) \geq cost(A)$
(R2) $pre(B) \subseteq pre(A)$
(R3) $s \in pre(B) \Rightarrow B(s) = A(s)$

We write $B \geq A$ to denote that B is redundant with A.

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 $B \geq A$ 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.*

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 1. *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.*

As noted by Burch and Holte [1], from Theorem 1 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 between any pair of states. In Burch and Holte's implementation of move pruning, all operator sequences of length L or less are generated in an order defined by a fixed nested ordering, and each newly generated sequence is tested for redundancy against all the non-redundant sequences generated before it.

3 Conditions Precluding Simple Interactions

I will call the situation depicted in Figure 1 a "simple" interaction between duplicate detection and move pruning, by which I mean the interaction takes place between two optimal paths, AC and BC, that have a common suffix (C). In this section I derive commonly occurring conditions under which simple interactions cannot possibly happen. Throughout the rest of the paper I 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 $A'C' \geq B'C'$ 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 $A' \ngeq B'$.

Thus, a simple interaction requires an interesting situation: $A'C' \geq B'C'$ but $A' \ngeq B'$. There are natural conditions in which this combination is impossible because $(A'C' \geq B'C') \Rightarrow (A' \geq B')$ for all sequences A', B', and C'. To derive such conditions, recall that the definition of $X \geq Y$ has three requirements:

(R1) $cost(X) \geq cost(Y)$
(R2) $pre(X) \subseteq pre(Y)$
(R3) $s \in pre(X) \Rightarrow X(s) = Y(s)$

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

(R1) We require conditions under which $(cost(A'C') \geq cost(B'C')) \Rightarrow (cost(A') \geq cost(B'))$. In fact, no special conditions are needed, this is always true because the cost of an operator sequence is the sum of the costs of the operators in the sequence and operator costs are non-negative.

(R2) We require conditions under which $(pre(A'C') \subseteq pre(B'C')) \Rightarrow (pre(A') \subseteq pre(B'))$. This is often not true, but it certainly holds if $pre(XY) = pre(X)$ for all operator sequences X and Y (with X non-empty). There are at least two commonly occurring conditions in which this is true:
- 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 in the sequence are guaranteed by the effects and unchanged preconditions of the operators preceding it), as in the sliding-tile puzzles with one blank.

(R3) We require conditions under which $(s \in pre(A'C') \Rightarrow A'C'(s) = B'C'(s)) \Rightarrow (t \in pre(A') \Rightarrow A'(t) = B'(t))$. This follows if both of the following hold:
- $pre(XY) = pre(X)$ for all operator sequences X and Y (with X non-empty), the same condition discussed in connection with (R2); and
- all operators are 1-to-1 $((op(x)=op(y)) \Rightarrow (x=y))$.

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 with one blank, Rubik's Cube, Scanalyzer [7], TopSpin, and the Pancake puzzle. In all such spaces, simple interactions between move pruning and duplicate detection cannot occur.

Unfortunately, simple interactions are not the only way that move pruning and duplicate detection can interact deleteriously, i.e., the situation in Figure 1 is not a necessary condition for move pruning to be unsafe in conjunction with duplicate detection. Figure 2 gives an example based on an actual run of A* on $(10, 4)$-TopSpin[2] 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 optimal 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 $N_5N_6 \geq J_5J_6$ and therefore prevents N_6 from being executed after N_5. It also detects that $K_3...K_6 \geq L_3...L_6$ and therefore prevents K_6 from being executed after $K_3...K_5$. Similarly, it detects that $L_2...L_5 \geq M_2...M_5$ and therefore prevents L_5 from being executed after $L_2...L_4$. 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.

[2] 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.

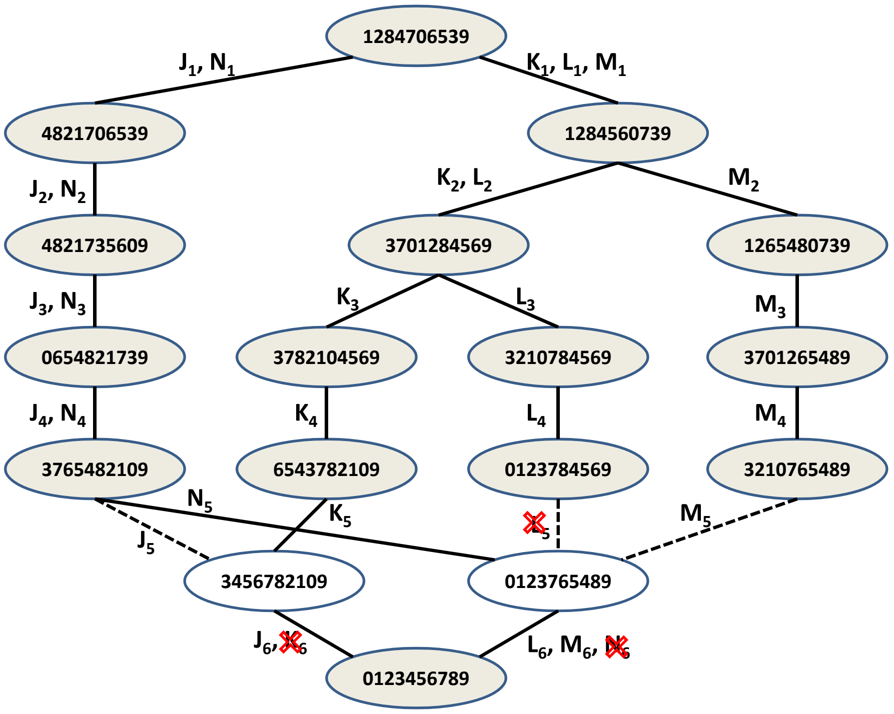


Fig. 2. Example from $(10, 4)$-TopSpin of move pruning and duplicate detection interacting to prevent the goal (bottom node) from being reached from the start (top node) by an optimal path.

The effects of duplicate detection are shown by drawing the edges entering the two states just above 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 $J_1 ... J_5$ is not cheaper than the first path to generate the state ($K_1 ... K_5$), it is ignored. Similarly, $M_1 ... M_5$ is not cheaper than the first path to generate state `0123765489` ($N_1 ... N_5$), so it too is ignored.

What makes this fundamentally different than Figure 1 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 optimal 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 optimal path (J). As I will show next, this represents the general situation in which move pruning and duplicate detection interact deleteriously.

4 Necessary Conditions for Move Pruning to be Unsafe

In this section I 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.

Let S be the start state, U any state that is reachable from S (U is the goal state), and BC any optimal path from S to U that contains no operator sequence considered redundant by move pruning (BC must exist because of Theorem 1).

Let *Alg* be a search algorithm that does neither duplicate detection nor move pruning and has the following properties.

- *Alg* enumerates the paths (operator sequences) emanating from S in a fixed sequence, thereby imposing a total order on the paths ($p_1 <_{Alg} p_2$ denotes that path p_1 is enumerated by *Alg* before path p_2).
- If operator sequence p_1 is a prefix of operator sequence p_2 then $p_1 <_{Alg} p_2$.
- *Alg* is able to prove the optimality of any optimal path it generates.[3]

When *Alg* is used in conjunction with move pruning, the resulting system is called Alg^{MP}. The effect of move pruning is to remove paths from *Alg*'s enumeration sequence but not to change the order of those that remain. Path p_1 will be removed by move pruning if and only if it is determined that there exists another path p_2 such that $p_1 \geq p_2$ and $p_1 >_{\mathcal{O}} p_2$. Note that every such Alg^{MP} generates BC.

When *Alg* is used in conjunction with duplicate detection, the resulting system is called Alg_{DD}. The effect of duplicate detection is to remove paths from *Alg*'s enumeration sequence but not to change the order of those that remain. For a given start state S, duplicate detection removes path p_1 if and only if there exists a prefix of p_1, p' (possible p_1 itself), and a path p_2 such that $p'(S) = p_2(S)$, $p' >_{Alg} p_2$, and p_2 is not itself eliminated by duplicate detection. A* and breadth-first search are examples of such Alg_{DD} search algorithms.

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

I assume that the elimination of paths from *Alg*'s enumeration sequence (whether by move pruning or duplicate detection or both) does not adversely affect its ability to prove a path it generates is optimal among the paths that remain in the enumeration sequence. This is true of A* and breadth-first search.

Definition 4. *We say move pruning is "safe" to use in conjunction with duplicate detection if, for any algorithm Alg with the properties stated above,* Alg_{DD}^{MP} *generates an optimal path from* S *to* U *for all states* S *and all states* U *that are reachable from* S.

[3] Algorithms such as A* and breadth-first search accomplish this by enumerating all paths that might be cheaper than the current cheapest path from S to U.

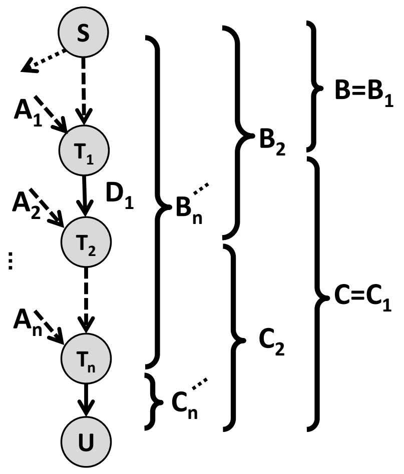


Fig. 3. Illustration of the Proof of REQ-2.

In other words, move pruning is unsafe to use in conjunction with duplicate detection only if some Alg_{DD}^{MP} fails to generate any optimal path from S to U. In particular if move pruning is unsafe, Alg_{DD}^{MP} will fail to generate BC. From this fact, I will now derive necessary conditions for move pruning to be unsafe to use in conjunction with duplicate detection.

Theorem 2. *Let Alg be any search algorithm with the properties stated above. Then* Alg_{DD}^{MP} *can only fail to generate BC if both of the following conditions hold:*

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

REQ-2 *There exists a state* T_n *on* $BC(S)$*, an alternative path* A_n *from* S *to* T_n*, and a suffix* C_n *of* C *such that* $C_n(T_n) = U$*,* A_nC_n *is an optimal 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 affect the generation of BC, which contradicts the premise that Alg_{DD}^{MP} fails to generate BC. A_1 cannot be cheaper than B because B is part of an optimal path to U and is therefore an optimal path to T_1. □

Proof of REQ-2. Figure 3 depicts the key ideas needed to prove this. A_1 here is as in REQ-1 and 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 $BC = B_iC_i$, $A_i(S) = B_i(S) = T_i$ and A_iC_i is an optimal path from S to U. D_i is the operator subsequence in BC that leads from T_i to T_{i+1}.

A_1C_1 is an optimal path from S to U, why did Alg_{DD}^{MP} not generate it in full? 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 $BC(S)$ 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^{MP}_{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 BC such that $B_2(S) = T_2$. Now repeat this reasoning for the path A_2C_2, which is an optimal 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 $BC(S)$ sequence than T_2, etc. This generates a sequence of states $T_1, T_2, ...$, each later in the $BC(S)$ 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 an optimal path from S to U. This path was not executed and it cannot have been eliminated by duplicate detection (because if it were 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.

$\square$

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.

4.1 Discussion of REQ-1.

REQ-1 states that there must be an alternative optimal path, A_1, to $T_1 = B(S)$ that is generated before B. 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 $BC(S)$ (namely, the path that is part of BC). This would happen, for example, if move pruning eliminated all alternative paths, as it does in the Arrow Puzzle [2]. 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 state in the state space.

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 optimal paths to a state $T_1 = B(S)$ but none of them is generated before B. For example, consider the special case depicted in Figure 1, 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 Figure 1 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 Figure 1 are open problems at present.

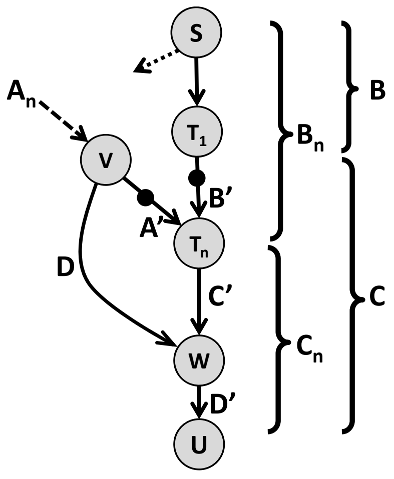


Fig. 4. General Case for Requirement 2.

4.2 Discussion of REQ-2.

REQ-2 says that there must exist optimal 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 Figure 1, 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., $AC \geq BC$. In the general case we are now considering we do not require $AC \geq BC$, 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. The first possibility, which is what we saw in Figure 1, is that D is part of BC, i.e., there exists a suffix B' of B_n such that $A'C' \geq B'C'$ and $A'C' >_{\mathcal{O}} B'C'$. Circumstances in which this cannot possibly happen have been discussed in Sections 3 and 4.1 above.

The other possibility for D is shown in Figure 4. Here D is a sequence entirely distinct from BC. In this case, we have another optimal 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 Figure 2. In that example optimal 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.

5 Summary and Conclusions

This paper has investigated the interactions between move pruning and duplicate detection with the aim of elucidating conditions under which it is safe to use both techniques together. I have derived conditions under which simple interactions cannot possibly occur and shown that these conditions hold in many of the state spaces commonly used as research testbeds (Rubik's Cube, TopSpin, etc.). Unfortunately, these conditions do not preclude more complex interactions from occurring and an example was given where A* fails to find an optimal solution in TopSpin because of these more complex interactions. I then derived conditions that must hold whenever there is a deleterious interaction between move pruning and duplicate detection.

6 Acknowledgements

Thanks to Malte Helmert for his insights. I gratefully acknowledge the financial support of Canada's Natural Sciences and Engineering Research Council (NSERC).

References

1. Burch, N., Holte, R.C.: Automatic move pruning revisted. In: Proceedings of the 5th Symposium on Combinatorial Search (SoCS). (2012)
2. Burch, N., Holte, R.C.: Automatic move pruning in general single-player games. In: Proceedings of the 4th Symposium on Combinatorial Search (SoCS). (2011)
3. Taylor, L.A.: Pruning duplicate nodes in depth-first search. Technical Report CSD-920049, UCLA Computer Science Department (1992)
4. Taylor, L.A., Korf, R.E.: Pruning duplicate nodes in depth-first search. In: AAAI. (1993) 756–761
5. Reinefeld, A., Marsland, T.A.: Enhanced iterative-deepening search. IEEE Trans. Pattern Anal. Mach. Intell. **16**(7) (1994) 701–710
6. Korf, R.E.: Finding optimal solutions to Rubik's Cube using pattern databases. In: AAAI. (1997) 700–705
7. Helmert, M., Lasinger, H.: The scanalyzer domain: Greenhouse logistics as a planning problem. In: ICAPS. (2010) 234–237