Move Pruning the N-Arrow Puzzle

A Java Implementation

John Palomo
CS271: Introduction to Artificial Intelligence
University of California, Irvine
Irvine, CA
jmpalomo@uci.edu

Liangjun Feng
CS271: Introduction to Artificial Intelligence
University of California, Irvine
Irvine, CA
liangjuf@uci.edu

I. Abstract

In the context of single-player games and search, determining duplicate states has proven to be an invaluable tool in reducing the amount of search that is required of a particular algorithm. This paper deals with an implementation strategy that was introduced by Burch (2011) for minimizing the search-space that is created for single-player, non-random, fully-observable games in which moves have an associated non-negative cost by detecting sequences of moves that lead to duplicate states [Burch 2011]. More specifically, we validate the results that were presented in the Burch's algorithm for detecting and pruning duplicate states by encoding the algorithm in a Java application. Although details are discussed with respect to the N-Arrow puzzle [Korf 1980], this algorithm can be implemented in any general class of single-player games that exhibit the properties of non-random, fully-observable, and non-negative move costs to reduce the tree space that is searched.

II. Introduction

Depth first search is a common search technique that is used when searching large state-space trees. Its ability to search large trees using minimal memory O(bm) where b is the branching factor of the tree and m is the maximum depth of the tree is a highly desirable property when you compare it to other techniques, such as breadth first search which requires O(b^m) memory. One primary drawback of DFS is its inability to determine cyclic/duplicate states during its search. In the context of our single-player games, these moves provide no insight into achieving our goal since cyclic and duplicate nodes have identical states to some earlier state with a greater cost. It then becomes to be able to identify and remove these states that do not contribute to the search goal.

Detecting and eliminating duplicate states is a valuable tool, but as identified in Burch's algorithm [Burch, 2011], the vertices that are generated during a DFS search can be further reduced by identifying sequences of moves that lead to duplicate states through taking into account the preconditions of rules. For consistency with Burch's algorithm this paper will refer to game moves as rules which can be described as consisting of preconditions and actions. By using the PSVN notation, we can succinctly describe the moves and use this as input to the algorithm procedure to derive a set of *macro-rules* corresponding to a number of sequential moves based on these preconditions and actions [Burch 2011]. In this paper, we will describe the N-Arrow puzzle that will help in understanding the concrete details throughout the paper. Additionally, we will describe a notation, called PSVN [Hernádvölgyi 1993], and used to describe the N-Arrow puzzle, after which we will describe the move-pruning detection algorithm followed by our implementation, results, and conclusion.

III. N-Arrow Puzzle

The N-Arrow puzzle is a simple game that can be described using a vector of N arrows, which can either point up or down. The rules of the game are simple and concise: given a state of a sequence of arrows, a move can be applied by flipping two adjacent arrows. Consider the following 3-Arrow example: *down, up, down* where arrow in position 1 is down, position 2 is up, and position 3 is down. This can be represented more succinctly as a sequence of bits where 0 represents the arrow is down and 1 represents the arrow is up: 0, 1, 0. Given this state of the game, the valid rules can be depicted in a graph as in Figure 1. In the diagram the numbers on the edges represent the position of the bits (arrows) being flipped.

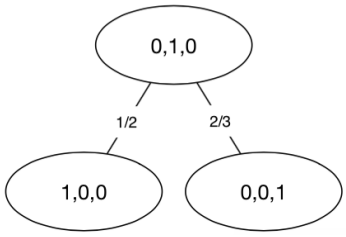

Figure 1: Valid Moves for 3-Arrow Puzzle with Initial State 0,1,0

The goal of the N-Arrow puzzle is to go from an initial state to an explicit goal state by applying a sequence of rules (bit flips). In our trivial example is easy to see that there are only two valid moves. However, any N-Arrow puzzle can have 2^N distinct states. Surely, being able to identify moves sequences such as flipping the bits in the same position consecutively would be very beneficial. An extended example from our 3-Arrow puzzle is given in Figure 2.

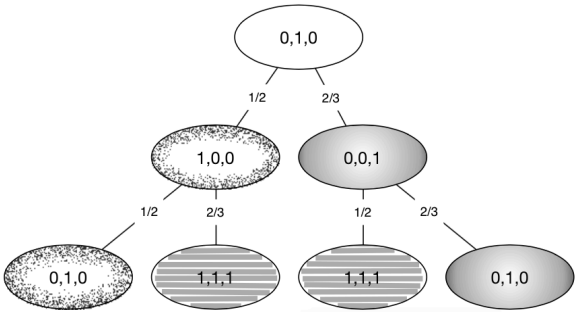

Figure 2: Duplicates Generate in 3-Arrow Puzzle with Initial State 0,1,0

As can be seen, by just applying two rules to the game tree, there are 6 vertices of only which 3 are distinct. It would be beneficial to identify these situations. Burch's move pruning algorithm accomplishes this.

IV. PSVN Notation

PSVN notation is a vector representation of states in a state space that uses fixed length vectors. The details and extensibility to a wide variety of AI problems are not concern in this paper. Abstractly, PSVN is:

> *...a state [of] fixed length vector of labels and abstractions [that] are generated by simply mapping the set of labels to another smaller set of labels (domain abstraction) [Hernádvölgyi 1993].*

In the case of the N-Arrow puzzle, our states can be represented as $< x_i, \dots, x_N >$ where $x_i = \{0,1\}$ for i =1 to N. More concretely and building off the running 3-Arrow example, the initial state could be represented as <0,1,0>. In the move pruning algorithm, the rules of the game are encoded in PSVN notation and given as input to the algorithm.

V. Move Pruning

In order to identify the duplicate states resulting from a sequence of moves that can be pruned, PSVN is used to describe the set of preconditions and actions applicable for a given state. As researched, all common planning and search problems can be represented in the form of preconditions and actions [Hernádvölgyi 1993] [Burch 2011]. Of course, the N-Arrow puzzle fits this class of problems. Before we can successfully and accurately prune sequences moves, we must describe a representation for the N-Arrow puzzle in terms of preconditions and actions and then efficiently describe a set of sequences as a more general representation of a game rule.

A. Preconditions and Actions in the N-Arrow Puzzle

The N-Arrow puzzle consists of rules that contain precondition and action vectors of length N and a non-negative cost associated. Abstractly, rules and actions have the form of $v_i = d \in D_i$ or $v_i = v_j$. As described by Burch:

> *...rule[s] of this type can be represented as a cost and two vectors of length N: $p^{\rightarrow} = < p_i, \dots, p_N >$ for the preconditions and $a^{\rightarrow} = < a_i, \dots, a_N >$ 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 $X = \{x_i, \dots, x_N\}$. Variable symbol x_j is associated with domain D_j and we only permit p_i or a_i to be x_j if $D_j = D_i$ [Burch 2011].*

In the N-Arrow puzzle, we can describe the constant domain, D_i, simply as the set {0,1} for all for i =1 to N. To make this a bit more concrete, the encoding rules for the 3-Arrow puzzle example is as follows:

Rules	Preconditions	Actions	Cost
R1	<0, 0, x_3>	<1, 1, x_3>	1
R2	<0, 1, x_3>	<1, 0, x_3>	2
R3	<1 0, x_3>	<0, 1, x_3>	3
R4	<1, 1, x_3>	<0, 0, x_3>	4
R5	<x_1, 0, 0>	< x_1, 1, 1>	5
R6	<x_1, 0, 1>	< x_1, 1, 0>	6
R7	<x_1, 1, 0>	< x_1, 0, 1>	7
R8	<x_1, 1, 1>	< x_1,0, 0>	8

Table 1: 3-Arrow Puzzle Rules with Preconditions and Actions in PSVN

As can be seen from the table, we do not need to represent states as an explicit state with concrete values. For example, in order to apply R1, we can be in either of two states: <0, 0, 0> or <0, 0, 1>. In other words, either state satisfies the preconditions of R1, thus R1 is a value rule (move) in these states.

B. Macro-Rules

Now that we have a concise representation of how to represent single rules in a standard notation, we can use this same notation to represent a sequence of rules, in other words, sequences of rules have the same representation as a single rule. This statement can be proved by induction [Burch 2011]. For the sake of the following discussion, we ignore the costs of the rules, and assume that each rule has a non-negative cost and no zero-cost rules exist. Therefore, applying rule, R3 to sequence (R1, R2) has a cost strictly greater than applying sequence (R1, R2), i.e. *cost*((R1, R2, R3)) > *cost* ((R1, R2)) and *cost*((R1, R2, R3)) = *cost*((R1, R2)) + *cost*((R3)).

In order to describe a sequence of rules, we take all the preconditions and the effects (actions) of the rules and represent this sequence as what is called a *macro-rule*. Since rules are represented as preconditions and actions (we ignore the cost element here), our macro-rule will also consist of preconditions and actions, specifically those that take into account the collective preconditions and net effects as aforementioned. Following the convention, we use a augmented vectors, $p^{\rightarrow *}$ and $a^{\rightarrow *}$ for the macro-rule representation.

C. Macro-Rule Generation Algorithm

The following description goes through the process of comprising these macro-rules [Burch 2011].

In order to generate the macro-rule representation for some sequence (R_x, R_y) we start by setting and augmented vector, p^* and a^*, equal to $p^{\rightarrow x}$ and $a^{\rightarrow x}$, respectively. It helps to view $p^{\rightarrow x}$ and $a^{\rightarrow x}$ as the starting rule of a sequence, but this statement can be relaxed and applied to general sequences of rules. Now, it must be deduced if this macro-rule can contain represent another arbitrary rule sequence, R_y. For the purposes

of this discussion, lets use $p^{\rightarrow 2}$ and $a^{\rightarrow 2}$ to represent R_x and $p^{\rightarrow 2}$ and $a^{\rightarrow 2}$ to represent R_y. In its most basic form, we are attempting to derive a macro-rule representation for the sequence of rules, (R1, R2). In practice, these can be any two arbitrary rules.

Now that the augmented vectors $p^{\rightarrow *}$ and $a^{\rightarrow *}$ are initialized, we need to go through each position, i to N, in p_i^2 precondition vector and determine if in fact a macro-rule can be generated by applying a_i^1, i.e. whether or not the net effect of a_i^1 can satisfy the preconditions of p_i^2. As we go through each position of the vectors, we either update/satisfy the $p^{\rightarrow *}$ vector or determine the move sequence to be invalid and conclude that the macro-rule cannot be generated from the sequence and terminate the processing. Using the PSVN notation, there are 3 possible domain values for any vector position, namely $\{x_i, 0, 1\}$. Therefore, we can generalized the possible scenarios that a_i^1 can be by noting that it can either be a constant or a variable symbol. Using pseudo-code, we have the general algorithmic form for resolving macro-rules:

```
function macro_rule(R1, R2)
 for i = 1 to N :
   if a_i^1 is a constant:
     use rules to determine if a_i^1 can satisfy p_i^1
     if no, return, sequence is not valid
     else update a^→* and p^→*
   if a_i^1 is a variable symbol:
     use rules to determine if a_i^1 can satisfy p_i^1
     update a^→* and p^→*
apply a^1 to p^→*
return macro-rule
```

Figure 3: Macro-Rule Generation Algorithm

The following list the rules described [Burch 2011] and give a brief reason why these rules work where needed. Referring to the pseudo-code above, we first go through the case in which *and* a_i is a constant and then a_i is a variable symbol:

1) $\boldsymbol{a_i^1}$ **is a constant** and p_i^2 is a constant, these two constant values must be the same in order for the move sequence to be valid. Consider the following example for a 3-Arrow Puzzle with *i*=1:

$$a^1 = < 1, x_2, x_3 > \qquad p^2 = < 0, x_2, x_3 >$$

This can be thought of as states, where the states are the preconditions and the actions are the mappings that take you from one state to the next. If we view p^1 as an initial state and the action as the resulting state, then clearly, the only resulting states are the one's in which $a_i^1 = p_i^2$. If the condition is satisfied, no update to the augmented vector is need.

2) $\boldsymbol{a_i^1}$ **is a constant** and $p_i^2 = x_k$ and a_k^1 is a constant, then the two constants a_i^1 and a_k^1 need to be the same if a macro-rule is to be generated. Again, we use the 3-Arrow Puzzle to illustrate an example in which this condition is not satisfied, with *i*=1:

$$a^1 = < 1, 0, x_3 > \qquad p^2 = < x_2, x_2, x_3 >$$

When we are resolving the precondition vector, p^2, with the action vector, a^1, we see that $p_1^2 = x_2 = 0$, however, $a_1^1 = 1$. If we again view these as states, the state that is realized when we apply a^1 to p^1, results in a state that has 1,0 in the first two positions of its new state (i.e. p^2). Clearly this is not a correct mapping and we need $p_1^2 = x_2 = x_1$. If the condition is satisfied, no update to the augmented vector is need.

3) $\boldsymbol{a_i^1}$ **is a constant** and $p_i^2 = x_k$ and $a_k^1 = x_l$. This is similar to the previous (2) situation, but is a little more relaxed since $a_k^1 = x_l$. Since $p_i^2 = x_k$ and $a_k^1 = x_l$, we can replace all instances of x_l with the constant from a_i^1 since the k[th] and l[th] position must be the same. Essentially, x_l needs to take on the value of a_i^1 to make the sequence valid. An illustration helps make this more clear:

$$a^1 = < 1, x_3, x_3 > \qquad p^2 = < x_2, x_2, x_3 >$$

Since we need the 1[st] position in p^2 to equal 1 to make this sequence valid, but $p_i^2 = x_2 = x_3$. The domain of $\boldsymbol{x_3}$ = {0,1}, the only valid value would be 1. So we set all instances of $x_3 = 1$ in $p^{\rightarrow *}$ and $a^{\rightarrow *}$.

4) $\boldsymbol{a_i^1}$ **is a a variable symbol** and p_i^2 is a constant allows us to replace all occurences of the variable symbol in $p^{\rightarrow *}$ and $a^{\rightarrow *}$ with the constant symbol of p_i^2. This is easy to see since we are trying to generate the macro-rule that satisfies this condition, the vairable symbol must be equal to the constant value for this rule to apply. Basically – if viewed as states – we want the state of $a_i^1 = p_i^2$ for position *i* for it to be valid.

5) $\boldsymbol{a_i^1}$ **is a variable symbol** and $p_i^2 = x_k$. Since both the action vector and the precondition vector have variable symbols, this will always be satisified and therefore no update is need to the augmented vectors.

6) $\boldsymbol{a_i^1}$ **is a variable symbol** and $p_i^2 = x_k$ and $x_k \neq a_i^1$ and $a_k^1 = \{0,1\}$ (constant). We replace x_k with the constant from the k[th] position of the action vector. Consider the following example when *i*=1:

$$a^1 = < x_2, 0, 1 > \qquad p^2 = < x_3, 0, 0 >$$

In order for this rule to be satisfied, x_2 must be equal to 1. So we update the augmented vectors with the value 1 for all variables x_2.

7) $\boldsymbol{a_i^1}$ **is a variable symbol** and and $p_i^2 = x_k$ and $x_k \neq a_i^1$ and $a_k^1 = x_t$ we let $y = \min(a_i^1\ variable\ index, t)$ and $z = \max(a_i^1\ variable\ index, t)$. All x_z are replaced with x_y in

the augmented action vectors. This is most easily seen by using an example as before:

$$a^1 = < x_2, x_3, 1 > \qquad p^2 = < x_3, 0,0 >$$
$$y = \min(2,3) = 2 \ \& z = \max(2,3) = 3$$

We replace all x_3 with x_2 in both augmented vectors. This makes sense, since both values x_2 and x_3 are variable symbols, we need them to be equal, so we replace the variable symbols that occur later in the sequence with the variable symbols that come before it, ensuring that any replacements done later in the processing of the vectors satisfies the rules that were previously resolved for the macro-rule.

As we mentioned at the beginning of the section, we want to determine if the sequence (R_1, R_2) is valid. Once we have gone through the previous rules for each precondition, the final step is to apply the action vector, a^2 to the augmented precondition vector, $p^{\rightarrow *}$ resulting in a modification of $p^{\rightarrow *}$, namely a new rule representing a sequence of rules.

Now that we have defined the rules for generating macro-rules, we can use the rules to generate a tree of valid move sequences with each node having the macro-rule representation of as previously described.

D. Moving Pruning Tree Generation and Algorithm

From the previous discussion of generating macro rules we can build a tree of valid move sequences. In general, we start with an empty move sequence, which consists of only variable symbols for both vectors, at the root of the tree. When generating this tree we identify move sequences to prune as described by [Taylor 1993] in the pruning algorithm as follows:

> *…we can prune a move sequence B if we can use 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 and (iii) applying A and B to s leads to the same end state.*

For simplicity, we ignore the costs of the move sequences and assume that adding an additional move to our sequence results in a higher cost since our costs are non-negative and no zero-cost moves. Our macro-rule representation helps us determine if a specific move sequence can be pruned for condition (ii) and (iii).

For condition (ii) we just compare the precondition vectors of both A and B.

a) if p_i^A is a constant, p_i^B must be that same constant.

b) If p_i^A is equal to some variable symbol that is not x_i (e.g. x_k), then p_i^B should equal p_k^B

For condition (iii), we can ensure that by applying the action vector of A to the preconditions of B results in the action vector B. This is easy to see, since we are basically checking to see that the net effect of the action vector of A, applied to B, will result in the same action vector of B.

VI. Java Implementation

In order to validate the results that were described in [Burch 2011], we constructed a Java implementation of a move pruning tree and ran a DFS on the resulting tree. All source code for this implementation can be found at https://github.com/jpalomo/f_14_cs71.

The project is divided up into two logical parts, one for the move-pruned tree and the other for the data model, which is simply an object-oriented representation of the data needed to generate the tree.

A. Move Pruning Tree Implementation

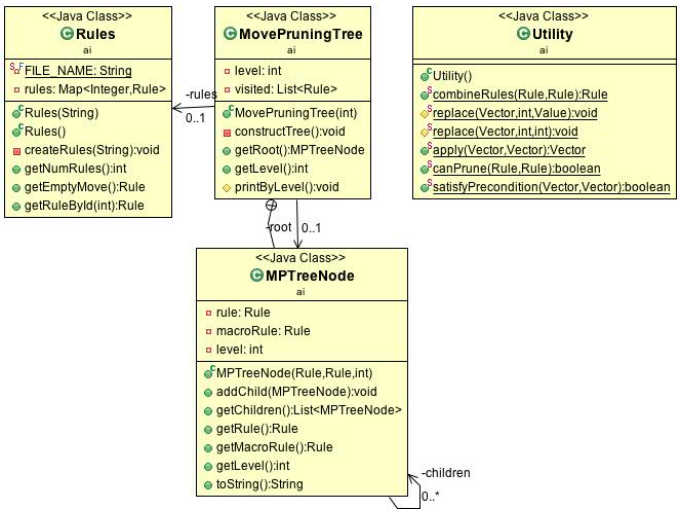

Figure 4: Classes Used for Creating Move Pruning Tree

The *Rules* class is simply a representation of the N-Arrow Puzzle rules. The *Rules* class is given an input file in PSVN notation and parses this file to generating a mapping from rule number to the vector notation of the rule. It maps each PSVN to an internal *Rule* object, which is part of the data model.

The *MovePruningTree* class is responsible for generating the macro-rule move pruning tree. Each node – *MPTreeNode* – of tree is a representation of a unique rule and macro-rule. Within this class is uses our *Utility* class that provides the implementation of the algorithm to generate macro-rules given two rules. Additionally, it provides support to update components of the data model, such as iterating over the preconditions of a rule and resolving whether or not a macro-rule can be generated as well as supplying the output vector representation of applying action vectors to precondition vectors. One thing to note in the *MovePruningTree* class is the *level* field. In the experiments conducted by [Burch 2011], a cut-off level was used which corresponds to the maximum depth of move-pruned tree that should be generated. This will be elaborated on more in the results of our experiments.

B. Data Model

The data model of the implementation represents the object-oriented approach to the move pruning algorithm. In other words, we wanted to be able to model the different components of the N-Arrow Puzzle and move pruning algorithm as distinct logical entities. Modeling the problem like this provides for more clarity and reduced the complexity of debugging the initial implementation and validating the results.

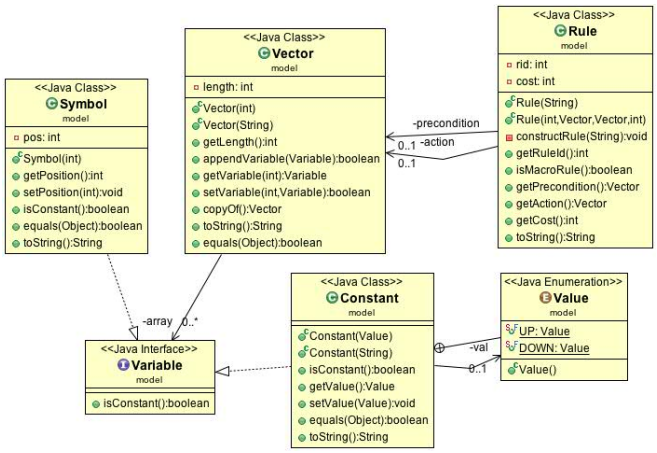

Figure 5: Data Model Classes

The *Rule* class is responsible for representing a single rule (one input line of PSVN file) for a specific N-Arrow Puzzle instance. Additionally, we extended the functionality of this class to represent a macro-rule on the move-pruned tree. This was a logical extension since a macro-rule has the same format and representation as a single rule, but captures a sequence of rules/moves as described in previous sections.

The *Vector* class was our representation of precondition and action vectors, including augmented vectors. In this class, a vector is represented as a list of *Variable.* This *Variable* interface was created as a means of storing one type of object in our list – in terms of object orientation. As can be seen both the *Constant* class and the *Symbol* class implement this interface. So, as we are iterating over the *Variable*s in the list (vector), we can determine whether a particular *Variable* is a either a constant or symbol, by invoking the implemented method of the interface. As can be seen, the *Constant* and *Symbol* classes both represent the values a particular PSVN vector can take on, namely, $\{0,1,x_i \mid i = 1 \text{ to } N\}$. Lastly, a simple class of an enumerated type was created to represent the values of {0,1} (up/down) as needed in an N-Arrow Puzzle.

VII. Experimental Results

In order to validate the approach taken by [Burch 2011], we ran our Java implementation on various inputs and analyzed both the number of nodes visited in two different depth first search algorithms: DFS with Move Pruning (DFS+MP) and DFS with Parent Pruning (DFS+PP). DFS+PP is a simple implementation where we detect cycles of length 2. Put simply, if were at node B with parent A, we do not visit node A if it can be generated by B. In other words DFS+PP avoids inverse operations as illustrated in Figure 6.

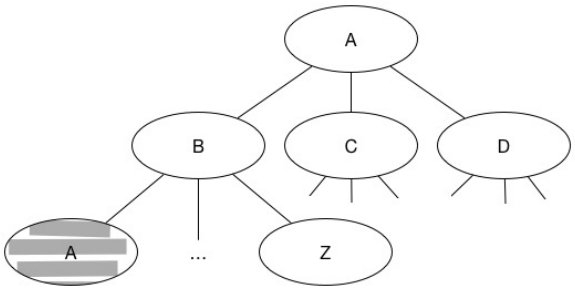

Figure 6: DFS+PP Cycle Detection of Length 2

Our goal – given an initial start state and a goal state – was to determine that the number of nodes visited in DFS+MP search was in fact considerably less than the number of nodes generated by a DFS+PP search in order to reach a goal state. It should be noted that we used sequence depth (denoted as *i*) similar to those used by [Burch 2011]. Namely, sequence depth levels were typically 2 or 3 when using DFS+MP. In other words, sequences that could be generated with more than 2 or 3 rules/moves were not considered.

A. Experiemnt I

In this experiment, we generated a static 12-Arrow Puzzle in PSVN format. Our initial state to begin the search start state was <1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1> and our goal state to terminate search was <1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1>. We ran the DFS+MP with i=2, i=3 and DFS+PP

Num_Arrows	Seq_Depth (i)	Search Depth (d)	Nodes Visited
DFS+PP	-	11	113,850
DFS+MP	2	11	38,820
DFS+MP	3	11	224,973

Table 2: Results of 12-Arrow Puzzle Experiment

B. Experiemnt II

In this experiment, we pseudo-randomly generated 100 instances of a 10-Arrow Puzzle in PSVN format and used this as input into our program. For all 100 instances we ran the DFS+MP with i=2, i=3 and DFS+PP.

Num_Arrows	Seq_Depth (i)	Search Depth (d)	Nodes Visited
DFS+PP	-	9	11,877,644
DFS+MP	2	9	4,591,605
DFS+MP	3	9	224,973

Table 3: Results of 10-Arrow Puzzle Experiment on 100 instances

C. Experiemnt III

In this experiment, we again pseudo-randomly generated 100 instances, but with a 12-Arrow Puzzle in PSVN format and used this as input into our program. For all 100 instances we ran the DFS+MP with i=2, i=3 and DFS+PP.

Num_Arrows	Seq_Depth (i)	Search Depth (d)	Nodes Visited
DFS+PP	-	11	662,298,412
DFS+MP	2	11	117,794,604
DFS+MP	3	11	1,1547,978

Table 4: Results of 12-Arrow Puzzle Experiment on 100 instances

VIII. Conclusion

In all of our experiments, our results validated the claims made by [Burch 2011]. DFS+MP consistently out-performed DFS+PP in all experiments. Additionally, we can see from the results that as we increase the sequence depth, i.e. consider more sequences when generating macro-rules, we get even better performance in terms of the number of nodes visited in a DFS+MP. It is interesting to note that in both the random 100 instance experiments, the DFS+PP suffered a 50-60% decrease in performance when juxtaposed with DFS+MP (i=3). Although we were limited in the number of arrows we could include in our puzzle, due mainly to time and memory constraints, we believe our results due align closely with the figures that were obtained in [Burch 2011].

It would be careless for us not to mention that we did in fact intentionally leave out an important metric that was considered in the original paper, that of time. We chose this for a variety of reasons. The first is that the hardware that we were performing our results on was notably different. That aside, the main reason we opted to remove time from our consideration is due to our choice of language for the implementation. We used a high-level, object-oriented, language that removes the burden of memory management from the programmer. In the original paper, the implementation language was C. It is well known that C outperforms many languages when it comes to speed. Additionally, C is not impeded from the overhead such garbage collection in Java. When creating large search trees as in DFS+PP, many references to unused objects may exist, prompting garbage collection to run and hindering the time results of experiments. Most of all, our intent was to validate the general usefulness of the move pruning algorithm for single-player games, specifically the N-Arrow Puzzle.

Additionally, with respect to the N-Arrow Puzzle, we should note that the time to generate move-pruned tree increases greatly as the number of arrows and set of distinct rules increases. However, the costs of generating these trees can be amortized over the lifetime of the tree and the number of uses, that is, if a move-pruned tree is used frequently, the cost to build the tree becomes negligible over time. Therefore, we urge the users of the algorithm to understand the usability of the tree that will be generated and offset that with the cost to build it, particularly for large N and large number of rules. If a search is performed once, DFS+PP may outperform the DFS+MP since the initial move prune tree does not to be generated.

We conclude with saying that DFS+MP is a useful technique when applied to single-player, non-random, fully observable games in which moves have an associated non-negative cost. We have validated the results by [Burch 2011] and have shown that in our results, DFS+MP consistently outperforms traditional DFS (even if parent pruning is implemented).

References

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

Taylor, L.A., and Korf, R.E. 1980. Towards a model of representation changes. Artificial Intelligence, 14(1): 41-78

Taylor, L.A., and Korf, R.E. 1993. Pruning duplicate nodes in depth-first search. In AAAI, 657-671

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