Parker King-Fournier

260556983

**COMP 424 – Das Bohnenspiel Report**

There is a fascination among computer scientists, programmers, and developers with what would appear to the outside observer to be childhood games. Perhaps having never completely grown up, or a lack of socials skills explains the compulsion to stare, deep in thought, at the board, however it is much more likely that there is something inherently interesting about such games. Indeed, childhood games such as Checkers, Chess and Backgammon lend themselves to computational approaches: the rules are simple and easy to understand and the individual moves are not exceedingly complicated, but an entire game made up of a simple foundation proves itself to be complex and seemingly unpredictable. In this report, the variation of Mancala called Das Bohnenspiel will be the topic of discussion.

**Approach**

To correctly formulate a method by which to evaluate Das Bohnenspiel Games certain observations were gleaned from both the nature of the gameplay, the rules, and personal experience from playing the game. Immediately noted was that Das Bohnenspiel games contain no hidden information: at each step of the game each player knows the score, can see the entire board, all the pits, and the number and distribution of seeds on the board. A second critical observation was that at each turn of the game there are at most 6 possible moves that can be made[[1]](#footnote-1). A third observation was the fact that the progress of the game is easily measured by the score, or more specifically by the aggregate score,, of the form:

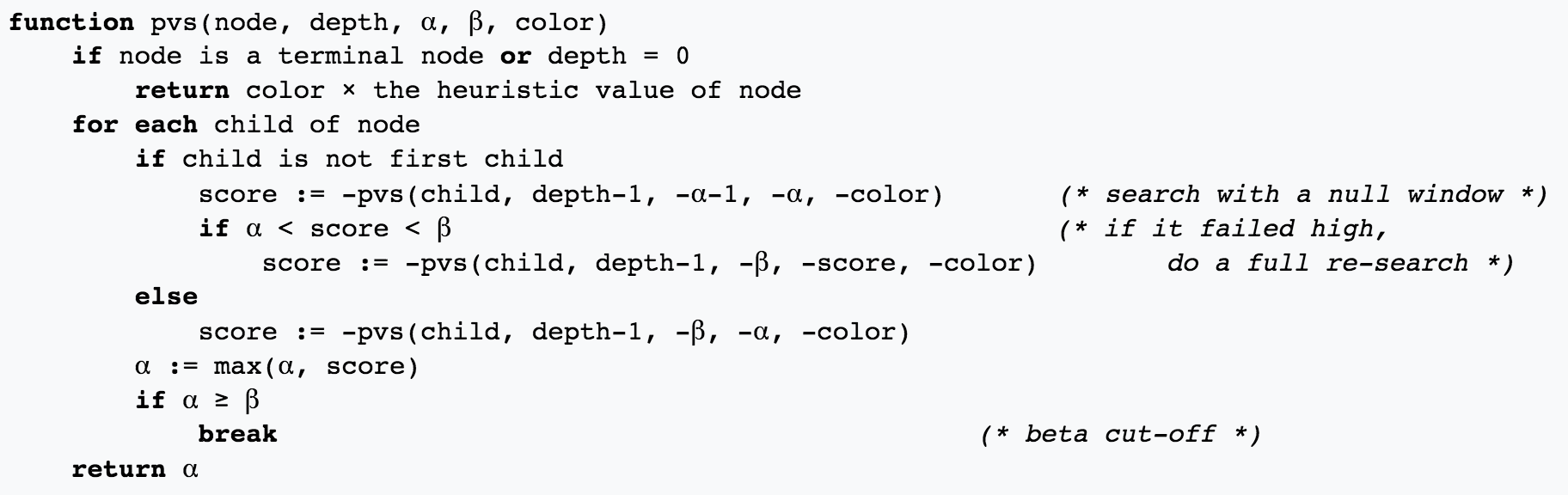
where is a board configuration, and is the score of player . Note that at each board state player is benefits from maximizing and player benefits from minimizing .

Due to the completely observable nature of the game, small branching factor, and adversarial playout of the game the decision to implement some variant of a minimax algorithm was obvious. By experimenting playing the game against a player making random moves it was discovered that the length of a game could be upwards of 40 moves and it became apparent that a naïve minimax algorithm would not search to sufficient depths. The Alpha – Beta Pruning algorithm was considered until further research revealed the Principal Variation Search algorithm, an improvement and slight modification of simple Alpha – Beta Pruning.

Principal Variation Search, which will be discussed in depth in the next section, is advantageous to Alpha – Beta Pruning given the efficacy of a move ordering function, which predicts which moves are better than others. Through trial and error, it was found that most the time the aggregate score,, was a sufficient for ordering moves, but that in the case two board states had the same the function failed. It was also noted that certain attributes of a board state other than the score were advantageous. These attributes, to be discussed in the next section, were used to then break ties between moves whose resulting board states had the same . The following section will discuss in detail the exact formulation of the Principal Variation Search algorithm with respect to the game Das Bohnenspiel.

**Implementation**

**Principal Variation Search**

 Principle Variation Search is a variant of Alpha – Beta pruning that only fully searches the Principal Variation, or hypothesized list of optimal moves. More precisely, at each turn in the game Principal Variation Search searches through its children moves in order of best to worst move, as is determined by an ordering function. Only for the hypothesized best move does Principal Variation Search complete a full Alpha – Beta search. The algorithm assumes the remaining moves are worse than the best move and searches simply to see if this assumption was wrong by searching with a null window, which will always fail. The algorithm expects to see the score of all non-best moves to fail below the alpha value of the best move: if it finds a new move that fails above the alpha value of the best move it realizes that the best move was indeed not the best move, and searches the new move with a full alpha beta window (fig. 1)[[2]](#footnote-2)[[3]](#footnote-3).

Figure

**Implementation**

The implementation of Principal Variation Search with respect to Das Bohnenspiel followed the above pseudocode closely with a couple modifications specific to game play. This implementation of Principal Variation Search uses a custom Java object called a Node. Each node represents a board state of the game, and has an associated value, and stores the best move from that board state as found from Principal Variation Search.

Before the search is performed the algorithm handles game-specific cases. The algorithm first checks to see if the board state of a node in the game tree shows that the game is over. If it sees the game is over, and the favored player has won it sets the value of that node to ∞, meaning that this state is extremely desirable. If it finds that the board state shows that the opponent has won, it sets the value of that node to -∞ to signify that this board state is undesirable. The algorithm returns these end-game nodes with a null move of “skip” which will never be played. The algorithm next checks to see if the board state association with a node of the game tree has any legal moves: if it does not have any legal moves it sets the best move to be skipping the move in hopes of the opponent creating more moves. Finally, the algorithm checks to see if the node it is evaluating is a terminal node. A terminal node is defined by being at the maximum search depth which is specified when calling the Principal Variation Search algorithm (fig. 1). In the last two cases the algorithm sets the value of the node appropriately by, sets the best move to be a null move of “skip” and returns the node.

After these game-specific cases have been dealt with the algorithm proceeds in a similar fashion to the pseudocode (fig. 1). First, all legal moves from the current node are found. These moves are then ordered per , which acts as the primary heuristic. The algorithm searches the first, and assumed best, move with a full alpha – beta window, recording that nodes score as the resulting alpha value and setting the bets move field of the node to be the move from which the alpha value came from. The remaining moves, assumed to be suboptimal, are searched with a null window, with the upper bound being and the lower bound being . This null window will always result in a failure. Most of the time the returned valuesatisfies but in the case that the algorithm has learned that its assumption that of which move was best was wrong. The algorithm then fully searches whichever moves returned such that with an alpha – beta search with a lower bound of .

The case where represents two moves which result in the same value. The values of , , and are dependent on only the scores of the players and take into no account the potential of the board state. The potential of the board state may be a function of many different attributes of the board state. In this implementation, the attributes of the board state that were chosen to represent the potential of that board state were the number of legal moves at that board state, the number of seeds on the side of the player, and the number of seeds on the side of the opponent, where is a node in the game tree. These attributes were guessed from personal game play experience.

Each time a best move was returned from the Principal Variation Algorithm, excluding the end game and terminal nodes, a value of 1 or 0 was given to each for that node by the function called *findStatistics*. For example, a returned node that had the most number of available moves, the most number of seeds on the side of the player, but did not have the most number of seeds on the side of the opponent as compared to all sibling nodes would be updated as:

If the primary statistic of two nodes and was determined to be equal, a more accurate score, , for each node was calculated by summing all and normalizing the resulting vector entries by the number of best moves that had been returned throughout the entire game. Thus, these percentages become more accurate of the true percentages of each attribute as the game evolves. Each of these percentages represent a weight used to create a weighted sum of and per the equation:

where is a node. Ties were then broken per the values of and .

**Advantages and Disadvantages**

Principal Variation Search is, in theory, an improvement on Alpha – Beta Pruning. Given that the ordering function of the nodes is effective, the algorithm will search less nodes with a full window than Alpha – Beta Pruning. It has been found that with a well-constructed ordering function, Principal Variation Search performs better than Alpha – Beta Pruning.

The efficacy of the algorithm then falls into the arms of the ordering function. A random ordering function will result in Principal Variation Search being worse than Alpha – Beta Pruning as the algorithm searches every node first with a null window, then with a proper alpha – beta window. Note that even though Principal Variation Search will take longer, it will not search any nodes that would not be search by Alpha – Beta Pruning.

The ordering function described above is advantageous in the fact that it finds trends in player’s decisions based on the attributes of the board states. In general, this approach is advantageous when an opponent decides moves based on consistent board attributes: these decisions will show up as high attribute percentages ( and can be used to better predict how the opponent will play.

In implementation, the choice of attributes is extremely important and will play a major role in how well the ordering function, and thus Principal Variation Search, performs. A player that decides moves on an attribute that is not considered will have a huge advantage as their move choices may seem unpredictable. A random player may also decrease the utility of the ordering function as each attribute will be randomly selected: the resulting calculated weights will thus be random and largely unhelpful. The more attributes considered, the more accurate the ordering function. This creates a clear and familiar tradeoff between accuracy and algorithmic efficiency.

This implementation of Principal Variation Search clones the current board state at each iteration, and creates objects that are memory intensive for all nodes in the search tree. This increases both computational complexity and memory requirements and ultimately takes away from the efficiency of the algorithm.

**Other Approaches**

During work on this project, no other approaches were applied in practice. As mentioned earlier other algorithms were considered, but decided against. All algorithms considered were minimax algorithms. Using Alpha – Beta Pruning was originally guessed to be the best method, but was abandoned upon discovery of Principal Variation Search. Monte Carlo Search was also considered for a short amount of time but abandoned since the given time constraint would most likely not allow enough playouts to predict the best move of a given board state.

**Improvements**

Due to the length of games and the branching factor it is almost impossible to solve Das Bohnenspiel in any reasonable amount of time. The question then becomes how to allow deeper searches while still maintaining a fast decision time. There are multiple ways to do this, two of which will be discussed.

The first way to improve search depth is to improve the ordering function. This could naïvely be done by simulating game play before the first move, thus training the attribute estimations to be more accurate at the start of the game. Alternatively, the ordering function could be further improved by selecting attributes of a board state that are truly representative of how much potential a board state has. While simple in theory, this is difficult in practice as the role of many attributes is unclear. For example, the greedy approach, which takes the move that captures the most seeds each turn may be advantageous in most situations, but will result in a loss against a very smart player. The determination of quality attributes by observation requires much research and proves to be quite difficult. A machine learning algorithm which infers the best attributes given past choices would be an ideal solution and could be implemented using neural networks to infer a hidden structure among move choices.

The second way to improve search depth would be to maintain the principal variation , where each is a move. At each step of the game, the algorithm could check if the last move was part of the principal variation. This allows a constant time move decision, but more importantly allows a deeper search depth as the algorithm could then search from the last move of the principal variation. Any time a move not in the principal variation is found, a normal Principal Variation Search would be conducted on the board state resulting from that move and the principal variation would be updated.

A third and final option would be to implement an Iterative Deepening version of Principal Variation search which would allow variable depth searches. This would allow the algorithm to take advantage of its correct guesses and search to a greater depth.

1. This excludes skipping a turn which, per the project specifications, should happen at most twice. Thus, there will be two turns that have 7 possible moves, but the majority will contain at most 6. [↑](#footnote-ref-1)
2. Note that pseudocode shows that at each step the algorithm takes the maximum score value. The astute observe will note that the change in alpha and beta values at each iteration results from the fact that . The parameter ‘color’ of the algorithm represents whether the player is the Max or Min player and will have a value of 1 or -1, respectively. Multiplying the heuristic value by the color allows the maximum score to always be taken. [↑](#footnote-ref-2)
3. Pseudocode from courtesy of Wikipedia and is available under the Creative Commons Attribution-ShareAlike License [↑](#footnote-ref-3)