Name: Qin Liu Student ID: 260578790

Comp 424 AI Project Report

Introduction:

The goal of this project is to develop an AI agent to play the two-player competitive deterministic board game Bohnenspiel. By the nature of the game, the minimax algorithm is a reasonable approach. As we have time limits for choosing a move, we cannot afford to explore to the end of the game. Hence an evaluation function is carefully chosen and optimized. Furthermore, to increase the search depth of the minimax algorithm, alpha-beta pruning technique is employed. Overall, the AI agent is able to make sound decisions in choosing game moves and is aiming at getting a score as high as possible. Currently, the AI agent beats random player and greedy player by big advantage in score. It also wins over a benchmark minimax player (which uses score difference as the evaluation policy). Tables of performance comparison between different players and under different evaluation policies are exhibited later in part III. The possible improvements such as data-preprocessing or forward pruning, are briefly described in the last part of this report.

Part I: Algorithm and its theoretical basis

To develop a wise decision process for the agent to choose a game move, our chooseMove(.) method consists of two parts. First, we establish the decision structure by minimax algorithm, and second, we use alpha-beta pruning technique to efficiently compute the minimax value.

Minimax:

Minimax algorithm is a recursive tree-searching algorithm. The goal is to find the optimal move from the current state. Theoretically, from the current state, the depth-first search tree expands by adding and searching through its successor states. The process searches all the way down to the leaf nodes (which a win/loss has decided) or to a cutoff depth. At leaf nodes, its utility is calculated according to an evaluation policy, which is a heuristic as we cannot afford to search to the game end. The utility value is then backed up from the leaf to the root of the tree. As the game takes turns between the player and its opponent, we define max_player and min_player, which the former returns max utility among its successors and the latter returns minimum. An outline of minimax algorithm in pseudo-code, modified from slides Cheung 2017:

```
move Minimax(State S)
    For each legal move O from State S:
        Apply o and obtain new game state S'
        Value[O] = MinimaxValue(S')
    Return argmax_O Value[O]
double MinimaxValue(State S, int depth)
    If isLeaf or depth==0, return evaluate(S)
    For each state S' of successor(S)
        Value(S') = MinimaxValue(S', depth-1)
    If max_player, return max_S' Value(S')
    If min_player, return min_S' Value(S')
```

The minimax algorithm runs in $O(b^d)$ time complexity, where b is the branching factor and d is the depth. As we have a time limit of 700ms per move, running the minimax algorithm allows a depth of 5. To explore more of the tree and increase the depth, we employ alpha-beta pruning technique.

Alpha-beta pruning

Alpha-beta pruning aims at cutting off branches or even subtrees that would not going to influence the final decision. Define α = best choice so far for max_player, and β = best choice so far for min-player. We keep updating alpha and beta values as the search tree expands. Whenever the backed-out utility value of the current state is worse than alpha if max_player or beta if min_player, we pruned away the branch because this branch would never be selected given there's a better choice. On average, alpha-beta pruning gives a time complexity of $O(b^{3d/4})$. After exploiting alpha-beta pruning technique, we reached a depth of 9 before exceeding the 700ms time limit.

Here's the pseudo-code with alpha-beta pruning, modified from Wikipedia:

```
move Minimax(State S)
    For each legal move O from State S:
        Apply o and obtain new game state S'
        Value[O] = AlphaBeta(S', -inf, +inf, False)
double AlphaBeta(State S, double α , double β, int depth, Boolean max_player)
    If depth == 0 or isLeaf, return evaluate(S)
    If (max_player)
        V = -inf
        For each state S' of successor(S)
            v = max(v, AlphaBeta(S', depth – 1, α, β, FALSE))
            α = max(α, v)
            If β ≤ α break (* β prune *)
        return v
    elseif (min_player)
        V = +inf
        For each state S' of successor(S)
            v = min(v, AlphaBeta(S', depth – 1, α, β, FALSE))
```

```
        β = min( β , v)
        If β ≤ α break (* α prune *)
return v
```

Part II: Evaluation Policy

The evaluation policy is a heuristic measuring the advantage of the state. The performance of the agent, hence largely depends on a good evaluation policy. After studying the rules of the Bohnenspiel game, we find that at a state, three factors largely influence the winning chances. First, we would like to maximize the score of the player as it is the most direct reflect of current standing in the game. Second, we also would prefer to have more seeds in the player's pits as this would give our player more possible moves as well as future moves. Third, we would like to minimize the number of pits that has 1, 3, or 5 seeds as these pits has a large possibility of capturing seeds in the next turn, which is the opponent's turn. Also, as these factors will have very different values in earlier steps or in later steps of the game, we believe that we could use this evaluation policy throughout the game.

Factoring in the information we have on the opponent, and let w1, w2, w3 be the weight for each factor, we have: Evaluation(State) = w1* (player's score – opponent's score) + w2*(total number of seeds of player's pits - opponent's pits) – w3* the total number of pits that has 1, 3 or 5seeds.

We define w1+w2 =1, and w3 as a separate weight, as we believe that the first two are the advantages and the last one is the disadvantage. After trying out several players with different weights, we decided that w1 = 0.7, w2 = 0.3, w3 = 1.0 to be the optimal weight, as this beats the most of the other players. Please see next section for a detailed performance compare.

Part III Other Approaches tried and Performance Compare:

During the course of this project, we have come with the following different players, each with different algorithms or different evaluation function.

Minimax Player: this player uses minimax algorithm to make decisions. Without alpha-beta pruning, it searches to a depth of 5.

Score difference player: this player uses minimax algorithm with alpha-beta pruning. It searches to a depth of 9. It's using an evaluation policy that simply calculates current score difference between the player and the opponent. This is a benchmark player, as it successfully beats Random and

Greedy player. (Random and Greedy player are set up as given.) It also beats the Minimax player, which shows that the performance largely depends on search depth and thus alpha-beta pruning is preferred. See table below for a detailed match results:

Player 0	Score	Player 1	Score	Winner	Player 0	Score	Player 1	Score	Winner
scoreDiff	58	Random	14	scoreDiff	Random	6	scoreDiff	66	scoreDiff
scoreDiff	50	Greedy	22	scoreDiff	Greedy	26	scoreDiff	46	scoreDiff
scoreDiff	52	minimax	20	scoreDiff	minimax	22	scoreDiff	50	scoreDiff

WeightTest player: our player with weights needed to be supplied.

To figure out an optimal weight function, we decide to iterate over all possible weights, and play each weightTest player against scoreDiff player, Random, and Greedy player for multiple times to obtain winning rates. See the table below on the left for winning rates at w1 = 0: 1.0: 0.1, w2 = 1-w1, w3 = 0:1.0:0.1. Now for the seven candidate optimal weights which win over the other three players, we play a tournament between them. For each match, the winner gets 1 point and the loser gets -1 point. The results are below on the right:

	Win rate for weightTest Player		
<w1, w2, w3>	Random	Greedy	ScoreDiff
<0.1, 0.9, 0.0>	1	1	0
<0.1, 0.9, 0.1>	1	0.5	0
…			
<0.6, 0.4, 0.3>	1	1	1
<0.6, 0.4, 1.0>	1	1	1
<0.7, 0.3, 0.4>	1	1	1
<0.7, 0.3, 1.0>	1	1	1
<0.8, 0.2, 0.3>	1	1	1
<0.8, 0.2, 0.6>	1	1	1
<0.9, 0.1, 0.7>	1	1	1
…			
<1.0, 0.0, 1.0>	1	1	0.5

First round	
weightTest	**cumulated points**
<0.6,0.4, 0.3>	0
<0.6,0.4, 1.0>	4
<0.7, 0.3, 0.4>	-6
<0.7,0.3,1.0>	2
<0.8,0.2,0.3>	2
<0.8,0.2,0.6>	-4
<0.9,0.1,0.7>	2

Second round	
weightTest	**cumulated points**
<0.6,0.4, 1.0>	2
<0.7,0.3,1.0>	0
<0.8,0.2,0.3>	-2

Therefore, to be conservative on the correctness of our heuristic, we pick <0.7, 0.3, 1.0> to be our final weights. To sum up, we conclude that the agent performs the best is the one employs the minimax algorithm with alpha-beta pruning technique using evaluation function = 0.7(difference in score) + 0.3(difference in the number of seeds) – 1.0(number of pits have 1, 3, 5 seeds). The overall time complexity is $O(b^{3d/4})$ and time complexity O(bd), where b is the branching factor, d is the depth of the search tree.

Drawbacks

The major drawback of our algorithm is that it assumes that the opponent employs the same heuristic as the player. If the assumption is violated, for example, the opponent uses a more accurate evaluation function or plays completely differently, then the algorithm is not guaranteed to give the optimal move. This is very possible as our evaluation function is not perfect. There would be other winning factors that we overlooked and the relationship might not be linear too. Another weakness of the algorithm is that its depth is limited because of timeout constraints. The performance of our algorithm largely depends on search depth, and depth is often more influential than the evaluation function. If we face an opponent who uses the same strategy but manages to search deeper, our player fails.

Part IV: Improvements

As the search depth is crucial for algorithms like minimax, we could further look into pruning techniques like forward pruning. In general, forward pruning trades off between number of moves explored and depth of tree searched. Using forward pruning technique like ProbCut algorithm by Buro(1995), we might be able to explore to the end of the game. However, there are certain possibilities that we missed the optimal move as we might have pruned it away in earlier stages.

We could also try to improve our alpha-beta algorithm. First, we could order the moves so that the pruning would happen as soon as possible without exploring unnecessary branches. The ordering could decrease the time complexity to $O(b^{d/2})$ and would allow us to explore more deeply within the time constraint.

We could also explore Monte Carlo tree search, however, without a huge amount of games played and back propagating the winning probabilities, Monte Carlo does not yield promising results. Given our 700ms time constraint, this may not be feasible.

Another approach is that we could supply our algorithm with some prior knowledge. We could create a transposition table and store it using hash table in the data folder. For each game state, the transposition table would store its score, best move previously searched, the search depth, and possible upper bound and lower bound of scores. This way when the same state appears again during the game, it gives the agent advantage to act based on previous knowledge.

References:

Alpha–beta pruning. (2017, April 10). In *Wikipedia, The Free Encyclopedia*. Retrieved 05:50, April 11, 2017, from https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=774807224

Buro, M. (1995). ProbCut: An effective selective extension of the alpha-beta algorithm. *J. International Computer Chess Association*, 18(2), 71–76.

Cheung, J. (2017). Lecture 6: Game Playing *Mycourses COMP 424 Artificial Intelligence, McGill* Slide 12.

Kato, H., Fazekas, S. Z., Takaya, M., & Yamamura, A. (2015). Comparative Study of Monte-Carlo Tree Search and Alpha-Beta Pruning in Amazons. *Information and Communication Technology Lecture Notes in Computer Science*, 139-148. Doi:10.1007/978-3-319-24315-3_14

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence, 6*(4), 293-326. Doi:10.1016/0004-3702(75)90019-3

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence a modern approach*. Boston: Pearson.

Winands, M. H., Herik, H. J., Uiterwijk, J. W., & Werf, E. C. (2005). Enhanced forward pruning. *Information Sciences, 175*(4), 315-329. doi:10.1016/j.ins.2004.04.011