COMP 424 Artificial Intelligence Course Project: A minimax approach to the Bohnenspiel game

Yuanhang Yang
McGill University

***Abstract*—In this report, I demonstrate my work on developing an Artificial Intelligence (AI) agent to play the Bohnenspiel game. This AI agent applies minimax search algorithm to make decisions, as well as $\alpha - \beta$ pruning to improve computation efficiency. Results and future improvements are surveyed in this report.**

I. Underlying algorithms

This section's explanation mostly comes from this course's textbook [3].

A. Minimax search

The minimax search algorithm assumes searching in game space against an adversarial opponent, with the goal of maximizing utility. With the opponent present, the agent A assumes the opponent O would try to minimize A's utility. A minimax search agent can optionally have a max level depth of search, on nodes deeper after which it would not keep exploring its successor nodes, but instead use an evaluation function to obtain a value of that state.

B. $\alpha - \beta$ pruning

The goal of $\alpha - \beta$ pruning is to decrease the number of nodes in the game tree that needs to be evaluated. It is based on an observation on the game tree: at a child node N of a MAX node, if the maximum utility possible is lower than another node of the same level, there is no point exploring the paths to follow N; similar argument can be made for a child node of a MIN node with expected utility higher than another node of the same level. This observation effectively marks many paths within the game tree "redundant" for the search algorithm.

II. The AI agent

A. Design choices

The AI agent developed for this project uses minimax search algorithm with a prescribed search depth of 7, together with a heuristic function for game states. In addition, $\alpha - \beta$ pruning techniques were also applied to improve efficiency. This combination yielded reasonable results, as discussed in section IV. Other approaches were also explored but they didn't make final submission; these are discussed in sub-section II-C. These algorithms were chosen because Bohnenspiel is a game with deterministic moves and a rather small branching factor.

B. Heuristic function selection

Once the algorithm is chosen, the most important task is to find a good heuristic evaluation function. If the search was complete, the game results can be directly used as an exact heuristic. But since this isn't possible, the agent faces the challenge of constructing a good heuristic from the information given by board states.

To empirically compare heuristic effects, I wrote a Simulator class that can play two players against each other. I evaluated the following heuristics by simulating a large number of games between agents with different evaluation functions:

1) My score.
2) My score ahead of my opponent's.
3) How close am I to winning.
4) How much closer am I to winning than my opponent.
5) How many moves can I choose from.
6) How many more moves I have to choose from than my opponent.
7) How many beans are on my side.
8) How many more beans are on my side than my opponent's side.

and I observed the following trends: firstly, using the "difference from opponent" versions almost always yield better gameplay results: that is, 2 outperforms 1, 4 outperforms 3, etc.; secondly, 1,3 and 2,4 seemed to generate highly similar results; my theory is that since in Bohnenspiel, a win is guaranteed if and only if 36 beans are captured, heuristics 3 and 4 are perfectly inversely correlated to heuristics 1 and 2.

Based on these observations and the results from simulated games, I ended up choosing 3 heuristics: numbers 2, 6 and 8. Of the three, heuristic 2 and 8 are suggested by [1], and heuristic 6 comes from my own understanding of the game. Heuristic 2 makes the agent try to stay ahead in the game. Heuristic 6 tries to always keep more options on the table for the agent possible. Heuristic 8 tries to keep more beans on the agent's side, as they are more likely going to convert into the agent's points.

The next step was to tune the weights of the three heuristics. This was done in two steps: firstly, multiple games were run between agents with only one heuristic weight different; secondly, assuming the heuristics are independent from each other, search near the combination of the best weights found

individually in the first step to obtain the final results. This assumption was necessary because I didn't have enough time to run the number of games required without it, so I had to pinpoint a weight value for each heuristic, and then refine locally. After extensive trial-and-error games, the evaluation function for a state S is set to be:

2.1 * (MyScore - OpponentScore) + 0.45 * (Beans on my side - beans on opponent's side) + 0.6 * (Non-empty pits on my side - non-empty pits on opponent's side).

C. Other approaches explored

A few other methods were explored in my attempt to improve the AI agent's performance. However, the agent showed no noticeable improvement with any of them, compared to using the heuristic described in II-B.

Firstly, I attempted to adjust the heuristics' weights over the progress of each game. The intuition behind this is that information in heuristics may have different values in different stages of the game. Specifically, I suspected that the "beans on my side - beans on opponent side" should have relatively increasing weights during gameplay compared to the other two, because the closer we are to the end of a game, the more likely that beans on our side would be directly captured by us. However, after trying a few models of such a trend, including linear, square root and quadratic functions, no noticeable pattern of improvement was found.

Secondly, seeing that many peer students are implementing minimax-based agents, I observed that when two agents play against each other, the outcomes never vary. Hence I thought about allowing the agent to perform random moves with a decreasing probability over the gameplay, in order to escape local decision traps - an idea similar to the simulated annealing algorithm. I attempted to find research works on whether or not this is possible for incomplete minimax searches (because, as we have learned in course tutorials, playing random moves against a complete minimax search agent is not a good idea). In the end, this attempt yielded no positive results.

III. Results

In the end, the agent with the 3-factor heuristic evaluation function showed reasonable performance. I wrote a program to play my agent against Random and Greedy players 1000 times each, and the agent achieved full victory records against both.

On the other hand, as I was tuning the weights of the 3 heuristics, I observed what I would call the "paper-scissor-stone effect", where 3 sets of different weights A, B and C would result in A beating B, B beating C and C beating A. This was also observed when I use different weights in my agent to play my friends' agents. This effect leads me to believe that plain heuristics can only take you so far, and further improvements ought to be done by techniques other than tuning heuristic weights.

IV. Discussion

A. Strengths and weaknesses

My approach has the advantage of being intuitive and straightforward: the algorithm used has no surprise element in it; in the contrary, most of my efforts were spent on improving the heuristic evaluation function. Because I ran extensive searches to tune the weights of the 3 heuristics, my agent outperforms others using the same plain heuristic model, especially those with less than 3 heuristics.

On the other hand, this approach does not take advantage of the structure of Bohnenspiel game; that is, it is a general-purpose approach that doesn't read any pre-trained data regarding this game tree. This not only makes state evaluation potentially inaccurate, but also decreases efficiency, so it cannot foresee too far into the game: with a search depth of only 7, other agents with a larger search depth can easily outperform it, even with much simpler heuristics. One way to mitigate this is discussed in the next section, IV-B.

B. Future works

To conclude this report, I propose the following possible approaches to improve the works presented. First of all, the tournament allowed the first move to be made in 30 seconds, a generous time window my agent isn't currently taking advantage of. The way I see it, even with the same underlying algorithm, the 30 seconds can lead to a great improvement in agent performance. As an example, consider writing a program to explore all possible states in the game and build the complete search tree, serializing the results into a disk file, and use the 30 seconds to read the file, de-serialize the search tree, and use that pre-constructed (perfect) information to achieve the optimal utilities in all circumstances.

Secondly, another option is to use a completely different set of algorithms. A good example is the Monte Carlo Tree Search (MCTS) algorithms. Although hard to implement, MCTS algorithms have shown very pleasant performances in game spaces searches. In particular, researches have shown that, in partial game settings, decisions made by MCTS family algorithms can be superior to those of minimax search in regions of the search space with no or few terminal nodes, such as Mancala or Bohnenspiel, where there is only one terminal node: a board with 12 empty pits [2]. This suggests that MCTS may eventually be a better solution to the Bohnenspiel game, under computation budget constraints.

References

[1] Chris Gifford, James Bley, Dayo Ajayi, and Zach Thompson. Searching & game playing: An artificial intelligence approach to mancala. Technical report, Technical Report ITTC-FY2009-TR-03050-03, Information Telecommunication and Technology Center, University of Kansas, Lawrence, KS, 2008.

[2] Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In *ICAPS*, pages 202–209, 2011.

[3] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.