Das Bohnenspiel Minimax with Alpha-Beta Pruning AI in Java

Matthew Cooke

McGill University

I. Algorithm and Motivation

For the artificial intelligence project of COMP 424 I was tasked with developing an intelligent player of the game Das Bohnenspiel (the bean game). This is a two player game in which each player moves beans around different pits, trying to capture as many beans as possible. An artificially intelligent agent would, therefore, have to calculate the optimal, or near optimal, move to make for each turn.

To accomplish this, I chose to use a Minimax algorithm with alpha-beta pruning. Minimax works by generating a search tree with MIN and MAX nodes alternating by depth. One player is assigned a "max" assignment and the other a "min" assignment. Typically, the MAX player is defined as the AI agent. Minimax works by assigning the maximum possible value from its children to the MAX nodes, and the minimum value from its children to the MIN nodes. Moves are then chosen by the best possible node. Minimax with infinite depth is optimal if the opponent also plays optimally, or always choose the correct MIN node. A–β pruning was used in order to ensure I was not computing paths worse than ones that had already been computed.

My Minimax algorithm has some slight variations than the standard Minimax definition. Firstly, because I was constrained on computation time to <700ms per turn, I could not compute a full Minimax tree. In order to accommodate this constraint, I chose a tree depth of 10, and added a leaf node heuristic. This heuristic is used when we are at depth 10 in the tree. My heuristic is as follows:

$$value = [(my\ score) - (opponents\ score)] + [(all\ my\ beans) - (all\ opponents\ beans)] / 4 + [(any\ pits\ I\ have\ of\ 2,4,6) - (any\ pits\ opponent\ has\ of\ 2,4,6)] * 4$$

I also implemented a random error in my computations in order to throw off other optimal agents. I assume that other agents will also use Minimax or other "optimal when opponent plays optimally" algorithms, and therefore, chose not to play optimally. Every 1/50 MAX or MIN nodes chooses a random child, instead of the MAX/MIN child.

Lastly, I added a catch in case the time limit was approached. Although I have a defined depth, we cannot control the breadth of the tree. Minimax is O(b^m). If I surpass 700ms computation time, our turn is ignored. Therefore, I only calculate Minimax for at most 500ms, and if this is surpassed, it plays randomly.

II. Theory

Minimax with A–β pruning is the algorithm of choice for artificially intelligent agents in deterministic, perfect information games (Knuth, 1975). Das Bohnenspiel fits this definition. I chose Minimax with A–β pruning because it is a somewhat efficient algorithm. Also, because our agents are playing against the agents of classmates, it can be assumed that the opponent is playing pseudo-optimally.

```
function alphabeta(node, depth, α, β, maximizingPlayer)
     if depth = 0 or node is a terminal node
         return the heuristic value of node
     if maximizingPlayer
         v := -∞
         for each child of node
             v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
             α := max(α, v)
             if β ≤ α
                 break (* β cut-off *)
         return v
     else
         v := +∞
         for each child of node
             v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
             β := min(β, v)
             if β ≤ α
                 break (* α cut-off *)
         return v
```

Figure 1. Minimax with A–β pruning pseudocode
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

III. Advantages and Disadvantages

My approach beats the RandomBohnenspielPlayer with $p>0.99$, $n=100$, and GreedyBohnenspielPlayer with $p>0.95$, $n=100$. The approach, although exponential in time complexity, is limited in computational time, and therefore runs quickly. A–β pruning ensures computation time is not wasted on nodes with sub-optimal paths. By playing randomly ($p=0.02$),

I can ‘confuse’ other AI agents that expect me to play optimally, without hampering my own performance drastically.

Although this approach works well, it is not perfect. It is limited to a computation time of 700ms and therefore cannot compute the entire tree; I chose a depth of 10. This means that my agent is only ‘looking’ 10 moves into the future, and is not considering moves after this. Games in Das Bohnenspiel are typically more than 10 moves. Playing randomly also decreases the optimality of my algorithm. If the opponent is using a strategy that plays optimally, independent from our play, we will be guaranteed to lose if we chose a random move that is sub-optimal ($p=0.02$ per node). Statistically it is likely we will lose ($p>0.95$) against an optimal agent.

IV. Other approaches

Originally, I had implemented a Monte-Carlo-Tree-Search algorithm. MCTS is a generic best-first-search algorithm, that uses random simulation as an evaluation scheme (Jakl, 2011). MCTS performed well (win versus GreedyBohnenspielPlayer $p>0.90$, $n=100$), but lost to the Minimax algorithm when played against each other. Monte-Carlo-Tree-Search was also much harder to implement and more difficult to improve. I, therefore, decided to use and improve the Minimax with A–β algorithm.

MCTS had one major advantage over my Minimax approach. It did not require a predefined depth, and could compute indefinitely in the given time constraint. Therefore, it always used the most possible time allowed to compute its result. If given more time, I would like to improve my MCTS approach.

V. Future Work

There are many improvements that can be made to optimize the Minimax algorithm (Stockman, 1979). Continuing to improve my heuristic function and improving the utilisation of all allotted time is a future goal. Currently, I ignore the extra 30 seconds given for initialization. I would also like to rework and improve my Monte Carlo Tree Search approach. MCTS seems like it should be the dominant algorithm, and with more work, I believe it will beat my Minimax (A–β), and do so in less time.

Lastly, I would love to attempt a machine learning approach to this problem, allowing the agent to learn how to play on its own. Machine learning would allow for the agent to predict different strategies used by opponents and respond with counter-strategies.

VI. References

A minimax algorithm better than alpha-beta?. G.C. Stockman. 1979.

Alpha–beta Pruning. Wikipedia. Wikimedia Foundation. 2017.

Arimaa challenge – comparison study of MCTS versus alpha-beta methods. Tomas Jakl. 2011.

An analysis of alpha-beta pruning. Donald E. Knuth, and Ronald W. Moore. 1975.

Minimax. Wikipedia. Wikimedia Foundation. 2017.