Theodore Morley
260621657
11/4/2017

Das Bohnenspiel Project Report

Section 1: Explanation and Motivation

Das Bohnenspiel is a deterministic game with perfect information, and a fairly limited number of moves at each turn. This makes it an ideal case for my chosen approach, which was classic minimax search with alpha-beta pruning, implemented within a time-limited Iterative Deepening Search of the game tree, because the perfect information allows for a consistent and easily implemented evaluation function. However, there are a few additional optimizations which I implemented to improve my agent, so in order to describe them, I will now step through the actions my agent takes in order to evaluate a move, explaining each element as it arises.

Pre-Search Discussion

The first action taken by my agent is a very simple one which has surprisingly useful applications, and that is to play a pre-set opening move, specifically the second to last pit. This idea was drawn from expert knowledge, since we were linked to a clip of two expert Bohnenspiel players, and this was the first move played. I originally decided to choose an opening move by testing identical agents against each other with all combinations of opening moves, but I quickly realized that the problem was intractable. Instead, I realized that a much quicker way to gather appropriate opening moves was from the existing knowledge that we possessed of the game. This approach worked quite well, and was simple to implement. Further advantages and disadvantages of this, and other choices I made, are described in Section 3.

Now we'll examine my agent's choices for non-opening moves. The first action it takes on a non-first turn is inspired by the greedy agent. While my agent was less developed, I saw that the greedy agent was beating it frequently, and I wondered in what case the greedy move would be the optimal one. From this, I decided that the first action my agent would take would be to check whether any of the legal moves on this board state lead to a victory, and if one does, the agent takes it immediately. After all, there is no move more optimal than winning immediately.

Game Tree Exploration

Next, my agent enters the Iterative Deepening Search portion. One of the thorniest problems I tackled was the issue of time. In this tournament, our agents were limited to 700 ms in which to choose

a move, or else a random move would be picked. Due to this, just before the pseudo-greedy win-check earlier, a Starting Time is initialized, which is tracked throughout the search, and once the elapsed time is within a certain threshold of the time limit (Determined by testing), the search ends immediately and returns the best move it has found so far. This makes Iterative Deepening Search ideal for our application, because we are not guaranteed to be able to reach any given depth from a certain board position, and so we desire a way to maximize the depth reached within a certain time limit. Iterative Deepening achieves this well, because for each Depth Limited Search it performs, I store the chosen move before computing the search at the next depth.

This move is chosen using a Minimax Search augmented with alpha-beta pruning. The top level function for this search is fairly simple, it first checks if there is still time remaining, and if there is, it examines each legal move by calling the recursive part of the search, which computes the value of that move by exploring the subsequent game tree. The agent then returns the move corresponding to the highest value found. The recursive function which computes the value of a given state acts as follows. First, it checks whether the search has run out of time, and returns a special error value if it has. Next, it checks whether there are no legal moves remaining, or if the depth limit has been reached. In both these cases, the base case of the function is executed, in which an evaluation of the board state in question is calculated by finding the difference between the player's score and the enemy's score (Discussion of evaluation functions is found in Section 4).

In a non-terminal case, the agent first performs an ordering of the moves according to the evaluation function, motivated by the assumption that the most immediately promising move should be explored first, in hopes of producing more cutoffs from the other moves, and possibly saving computation time. The way in which it does this ordering is somewhat interesting, as I designed it to require as little overhead as possible. In order to explore a possible move, the agent must clone the current state, and apply the move to it to generate a new state which can be evaluated. Since this would occur once for each state even in the unordered case, I only introduced minimal overhead by using a function which takes a list of legal moves and the current state, and uses a lambda function to order the states generated from each move. Then, these generated states are explored in order, thereby exploring the moves in order.

The next step after ordering the possible states depends on whether it is the maximizing or minimizing player's turn, and is completely standard alpha-beta pruning, drawn from the textbook.[2]

Section 2: Theoretical Basis

No algorithms from other sources were used, as the algorithms I implemented were primarily drawn from the class slides and the textbook, though I did explore other options while considering what to use, none of these were implemented. Minimax was chosen as it guarantees optimal play against an optimal opponent (With respect to the evaluation function), but also makes sure that our agent hedges its bets in the case of a non-optimal opponent, since the non-optimal opponent has to choose a move resulting in a board position we find more favorable. Thus Minimax works quite well with a good evaluation function, as discussed in the textbook, but is beaten by agents with heuristics that perform better or capture greater information than the current evaluation.

Alpha-beta pruning was used because it adds no major overhead, and potentially speeds up our computation drastically. This is because the additional tracking of upper and lower bounds can allow us to see that certain subtrees have no possibility of giving improving our result, so we don't bother searching them at all, and avoid wasting computation.

The theoretical and practical reasoning for IDS was described in depth above, so it will not be touched upon here.

Section 3: Interesting Advantages and Disadvantages

One element of my approach that I will analyze first is the fixed opening move drawn from the expert play. Without this fixed move, my agent loses to the greedy player whenever it goes first. This suggested to me that my evaluation function was choosing a poor opening move every time. Therefore the advantage of having the opening move selected in this way is quite apparent. As for the disadvantage, this choice limits the agent, since different agents might be developed in such a way as to counter exactly this opening move, and the fixed nature of my agent's movements means that it has no recourse against this. Some element of randomness may have been helpful here, but I did not implement any random elements to my approach.

Another primary weakness of my program is one shared by all Minimax agents. The Minimax algorithm assumes that the opponent evaluates states with the same heuristic function as the evaluating agent. This is a big assumption, and it means that my agent may take non-optimal moves. This could be problematic in the case of opponents who incorporate randomization, since it may lead to a game tree in which an unexplored state further down the tree could lead to loss.

One advantageous element is the incorporation of the timer into my search, which allows my agent to reach as deep into the tree as it can before returning a move, maximizing the return from the

resources. This likely leads to further depths being explored by my agent than by an agent using a depth limited search. The disadvantage of this, however, is the overhead created by IDS, which is required to use the timer, since we need to be able to return the best value found so far if time runs out during the search. There are a number of methods that could lessen the effect of this, such as Transposition Tables, but I ran into difficulties in their implementation.

Section 4: Other Approaches

The first and most relevant discussion in terms of other approaches which I took is the evaluation function, which went through a number of iterations throughout the project. The first evaluation function I tried was surprisingly effective, and it was in fact the simplest. This evaluation function, referred to as Simple Evaluation, only took into account the agent's score in a given state, making the Maximizing agent's goal equivalent to maximizing the player's score. However, this is missing out on several facets. Most pertinently, it does not consider the opponent's score at all. In order to counter this, I began to design more complex evaluation functions. To do this, I approached the problem from the high-level assumption that there were both desirable and undesirable components to a board position, and that to find a good board position, the agent should try to maximize the desirable components, and minimize the undesirable components. I identified a number of measurements in each of these categories, summarized below:

Desirable:

- Player Score increasing.
- Empty pits on opponent's side. (This is due to the rule in Das Bohnenspiel which causes a player to capture all remaining beans on the board if all the enemy's pits are emptied.)
- Beans on the player's side. (Preventing the opponent from emptying all the pits.)

Undesirable:

- Enemy score increasing.
- Empty pits on the player's side.
- More beans on the opponent's side.

I tested various combinations of these evaluation functions against each other, and found that for much of the more complicated ones, the overhead computation involved in gathering the relevant information outweighed the benefits. As such, I finally settled on the function which simply gathered the player's score and the enemy's score, and maximized the difference between them (Final Evaluate). Since the agent is deterministic, only two games need to be run to ascertain the properties of the

evaluation function. The differences in performance between these evaluation functions are compared below:

Simple Evaluate vs Final Evaluate: When played against each other, with all other factors equal, whichever agent moves second wins. However, when played against the greedy player, Final handily beats the greedy player every time, regardless of turn order, while Simple loses 50% of the time (Whenever going first).

Full Evaluate vs. Final Evaluate: The full evaluation is one which takes into account all desirable and undesirable aspects of the board, but is fairly computationally heavy by nature. This was the most difficult function for me to decide on, as it offers very interesting tradeoffs. The Full Evaluate beats the Final Evaluate 100% of the time. However, the Full Evaluate actually ***loses*** to the Greedy player 100% of the time as well! This makes it a difficult choice. I went with the Final function to be safe, because being beaten by simpler players seems to be a massive disadvantage. Similar results hold for various permutations of the full evaluate (Bean count but no empties, empties but no bean count).

Weighted Evaluate vs. Final Evaluate: In the weighted evaluation function, moves which empty certain pits are incentivized, since the function increases the desirability of a state if certain pits still have beans in them, depending on the amount of beans. This was based off the idea that it may be better to move from earlier pits most of the time, to leave more beans on your own side, or to move from later pits, to possibly capture more of the enemy's pits. Though this approach has a 100% win rate against Greedy, it only beats Final when moving first. This approach did not seem to add anything extra, while possibly adding more computation time, so I found it mostly non-useful. At higher weights, it also loses to Greedy.

From these results, I concluded that the Final evaluation function combined a low amount of computational overhead with solid results, allowing for greater depth to be reached while accurately assessing the board.

Section 5: Future Improvements

If I were to go about improving my player, the first step I would take would have been leveraging transposition tables. As explained in Russell and Norvig[2], a transposition table is a technique described as a hash table which stores the results of exploring certain states, in order to use the information later if the same board position is reached. Originally, I had planned to include transposition tables in order to decrease the overhead which Iterative Deepening Search creates. By storing the results (including the alpha and beta at that board position) from shallower searches in the game tree, one can easily

refer to the table when exploring the tree to a lower depth in order to quickly cut off branches, possibly saving computation time.

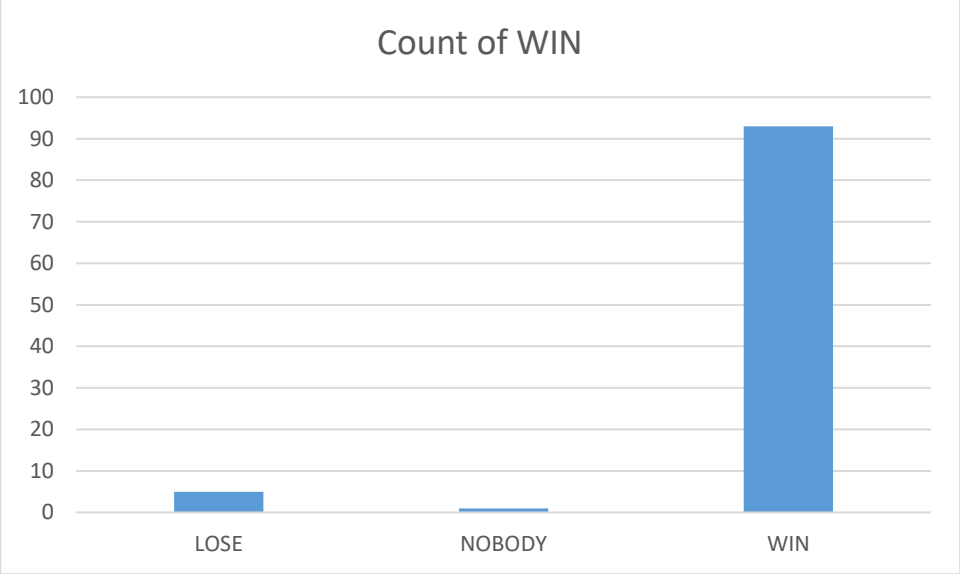
However, some differences from a regular transposition table would need to be implemented. For example, the standard hash function (used for storing board positions and their parameters in the hash table) for much of game playing is the Zobrist hash, but this hash relies on each move causing very little change to the overall game board, such as in Chess. [1] In their paper, Irving et al. solve another Mancala variant, called Kalah, and provide the following information on hash functions:

"The standard hash function for game playing is the Zobrist (1970) hash function, which has the advantage of incremental computation. In Kalah, this advantage disappears due to sowing that causes changes all over the board." (p. 144)

A different hash function, such as the Jenkins function, is needed to prevent frequent collisions. By encoding the board state, alpha, beta, and depth as a string, and implementing storing and lookup inside the alpha beta pruning, I could eliminate several sources of wasted computation in the algorithm, possibly allowing deeper search. However, I ran into difficulties in the implementation, especially with the hash function.

Appendices:

Chart 1: Performance against Random Player in 100 games:



Bibliography:

[1]: G. Irving et al., "*Solving Kalah,"* in *ICGA Journal*, vol 23, no. 3, pp. 139-147, September, 2000.

[2]: S. Russell, P. Norvig, "*Adversarial Search*" in *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2010.