Mancalamax

Author: Ethan Mentzer (cometbeetle)

1. Introduction

Mancala—specifically the modern *Kalah* variant—is a deterministic, two-player, turn-based game with relatively simple rules. As a result, computer programs can be designed not only to play Mancala, but to win consistently against humans using the minimax algorithm. This report summarizes an implementation of minimax, which, coupled with alpha–beta pruning and a well-suited heuristic, performs well enough on average to beat random players and humans at Mancala even with limited computing resources.

2. Program Efficacy

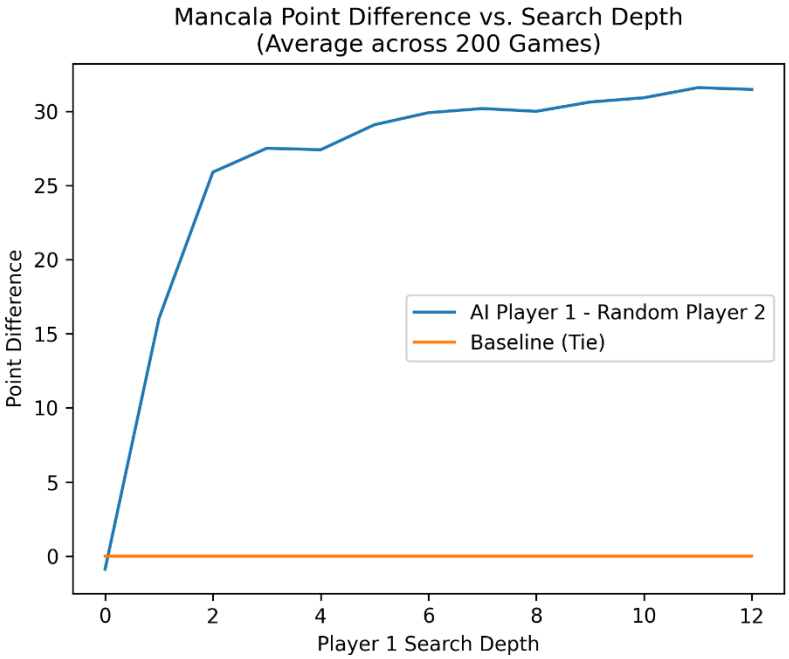

Fig. 1 | Average difference in point total at the end of a game vs. search depth. 200 samples were taken, where point difference is calculated by (Player 1 points – Player 2 points). Player 1 is the minimax algorithm with alpha-beta pruning; Player 2 is a random opponent. A depth of zero means moves are selected at random.

Against a random opponent, the minimax / alpha-beta pruning algorithm succeeds as soon as it can search at least one move ahead. From Fig. 1 and Fig. 2, we can see that as search depth increases, the average difference in scores at the end of the game increases in favor of the minimax player. Interestingly, it appears that the increase in score delta is initially large for depths 0 through 2, while for depths 3 through 12, there are diminishing returns. Still, by searching only four moves ahead, the minimax player manages to win by more than 25 points.

It is also worth noting that, when the minimax player is Player 1 (i.e., when the minimax player goes first), it tends to win by larger margins for each search depth. For example, at the maximum tested depth of 12, minimax

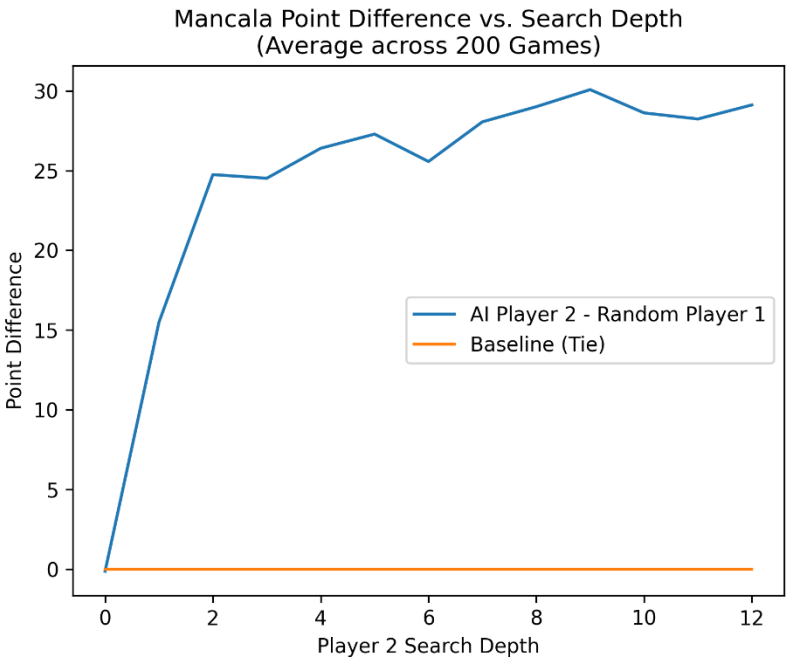

Fig. 2 | Average difference in point total at the end of a game vs. search depth. 200 samples were taken, where point difference is calculated by (Player 2 points – Player 1 points). Player 2 is the minimax algorithm with alpha-beta pruning; Player 1 is a random opponent. A depth of zero means moves are selected at random.

as Player 1 wins by more than 31 points on average, whereas minimax as Player 2 wins by approximately 29 points. This may indicate some sort of advantage for Player 1, at least when playing by the rules followed by my Mancala game simulator.

As for the program's efficacy against human opponents, both I and another person could not beat the machine, even when we considered different move options for much longer than the machine did. It is not clear from our testing whether someone with significant experience playing Mancala would beat the algorithm.

3. Implementation Efficiency

In order to simplify the software development process, I initially wrote the Mancala simulator and minimax algorithm in Python 3.12. However, after verifying the functionality of the program, it quickly became clear that the lackluster performance of the Python interpreter was a major limiting factor. So, to achieve better efficiency, I rewrote the program in C, which reduced overhead and significantly increased performance. Specifically, while the Python implementation could, on average, reach a search depth of 11 within one second of computing time, the C implementation consistently reaches a depth of 14. Time-limited depth testing was performed using an initial Mancala board state, on a Windows machine equipped with an Intel Core i7-14700K CPU.

Additionally, before implementing alpha-beta pruning in the Python version, I tested a standard minimax algorithm. Anecdotally, standard minimax was significantly slower; once I added alpha-beta pruning, the program began to return results in less time, given a fixed search depth limit.

Move ordering also played a significant role in performance. Specifically, while testing the C implementation of minimax with alpha-beta pruning, I observed that if the algorithm checked moves from highest pit to lowest pit (i.e., if moves 1 through 6 are available, checking 6 first, then 5, then 4, etc.), the performance was significantly better. I do not currently have a hypothesis as to why this specific ordering is preferred, however it must the case that checking moves in a descending order allows the algorithm to prune branches of the search tree sooner.

4. Heuristic and Utility Functions

To evaluate terminal game states, my algorithm simply takes the difference in points between the current player and the opponent. An optimal move should maximize the player's point total, while simultaneously minimizing the opponent's total.

To evaluate intermediate states, such as when a search depth or time limit has been reached, my algorithm uses a heuristic function that evaluates states in the same manner as the utility function. While there are many possible heuristic functions—some likely better than others—I found that I had no need to add complexity to the program by adopting a sophisticated heuristic. This is partially because taking the delta between player scores is already guaranteed to be an admissible heuristic (given our utility function), and because more advanced heuristics would almost certainly require more computing power. Ultimately, I found it more worthwhile to spend time optimizing the algorithm to search deeper rather than to create a specialized heuristic.

5. Additional Information

Below are some final additional facts about the program, summarized in a bulleted list:

- Iterative deepening is implemented in the `minimaxIterDep` function, where a time limit can be set for the computer to return an optimal move.
- The algorithm will work on any board size (simply specify a number of pits other than 6).
- When running the implementation through Valgrind, no memory leaks or errors are detected.