CMPT 355 Term Project (Group 3)

Konane Agent

Authors: Aiden, Ayden, Davin, Kevin Ulliac

Introduction

Konane is a Hawaiian two-player board game. For the purpose of this project, the board size will be 8x8, and we must implement a playing agent based on Min-Max Search or Monte Carlo Tree Search. We've decided to code the agent in the C programming language, primarily because of its computational efficiency and speed.

How Konane is played

The board is alternately filled with black and white pieces starting from top left (a8) to bottom right (h1). Black pieces are always first to move, and must remove one piece from either the middle (d5 or e4) or the two corners (a8 or h1). Following this move, white must remove any adjacent white piece from where the black piece was removed. Now the game continues as such: Player takes a piece (or pieces with a double jump), vertically or horizontally, by jumping over the oppositions'. The game ends when a player can no longer move – implicating them as the loser.

Approach

We will explore both Min-Max Search and Monte Carlo Tree Search algorithms and choose the strategy that performs the best, and/or if they perform equally well then we'll choose the one that's simpler to implement. For the purpose of testing, we'll assume that our agent has infinite time and memory to fully search/explore for the best possible moves for both strategies.

Min-Max Strategy

For the Min-Max strategy, the agent starts by generating a heuristic or scoring to generate the best moves. We'll denote the agent as "max" and the opponent as "min." To generate the heuristic, the agent must generate the initial state of the game and compute all possible moves by "max" and "min," and compute the heuristic value for every branch and at every depth value (i.e., for each successive move by "max" and "min"). In short, "max" aims to maximize the number of possible moves (jumping their piece over the opponent's, thus "capturing" one of the opponent's pieces), while "min" aims to reduce the total number of possible moves that "max" can play.

By utilizing the minimax search algorithm, the agent needs to know how to make moves and then build a tree for Min-Max to pick the best move. So, the agent will generate played games with each move to construct a tree. And by applying the Min-Max algorithm to that tree, it will determine the best move possible.

Monte Carlo Tree Search (MCTS) Strategy

For the Monte Carlo Tree Search strategy (MCTS), the agent must first represent the initial state of the game, and instead of generating all possible moves all at once, the MCTS uses four main components to progressively explore possible moves and to rank them based on simulated results.

The first step is selection, which starts from the root node and selects a successive child node until it reaches a leaf node. Assuming that leaf node is not a terminal node (i.e. the game is not yet over with a definite winner) then it would move on to the next step, the expansion phase, which expands/creates one or more child nodes and selects one of them based on the chosen exploration/exploitation selection policy. Then it runs a simulation, from the currently selected child node it plays the game using a default random policy until it reaches a terminal node. And lastly, it uses back-propagation to update the simulation results in all the traveled nodes accordingly.

Improvements

There are likely to be a number of improvements that we could implement in both strategies. One type of improvement is reducing the total number of computations the agent needs to perform without decreasing its odds of winning the game. Another obvious improvement is applying some kind of heuristic to how the agent generates possible moves, such that the worst moves are ignored, and the agent can find more optimal moves sooner.

For the Min-Max strategy, we could employ alpha-beta pruning which will reduce the total number of computations to determine the best move. In the early stages of the game, it’s more difficult to determine the exact values for each possible game (in this case, a possible game is the exact sequence of moves that make it unique from every other game). We could limit the tree search to a certain depth, which will reduce the total number of computations. We could also generate a better heuristic algorithm that takes into account some basic principles of Konane “best-practice” when it comes to generating possible moves.

For the MCTS strategy, we will start with a random selection policy, but there’s likely to be a more efficient and effective selection policy, perhaps even as simple as keeping track of symmetric moves and not allowing for equivalent symmetric moves to be simulated as opposed to all possible moves. This would eliminate some redundancy and ensure that each simulation is a unique game (i.e. a unique sequence of moves).

Technical Considerations

The agent should:

- Represent the 8x8 Konane Board using an array of 8 arrays of size 8
- Represent the search space:
    - Min-Max: Binary Search Tree (at most two child nodes)
    - MCTS: N-ary/Generic Search Tree (each node can have more than two child nodes)
- Take command line arguments [board_config] [B/W]
- Use algebraic notation