Term Project: Konane Agent Report

Authors: Aiden Lumley, Ayden Hogeveen, Davin Han, Kevin Ulliac

CMPT 355 / AS01

March 6, 2024

Course Instructor: Calin Anton

Context:
1. Started work on this report February 9, 2024
2. We used resources from the MacEwan Library database, as cited in Reference List
[02/16/2024: Konane game rules]
3. We did not utilize support from the MacEwan Writing Centre
4. Each group member critiqued, edited, and added to the report

Introduction

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

Playing Konane

The board alternates with black and white pieces from the top left (a8) to the bottom right (h1). Black pieces are always first to move, and must remove one piece from the middle (d5 or e4). Following this move, the player with white must remove any adjacent white piece from where the black piece was removed. Now the game continues: The player takes a piece (or pieces with a double jump), vertically or horizontally, by jumping over the oppositions'. The loser is the player who cannot make a move. [5]

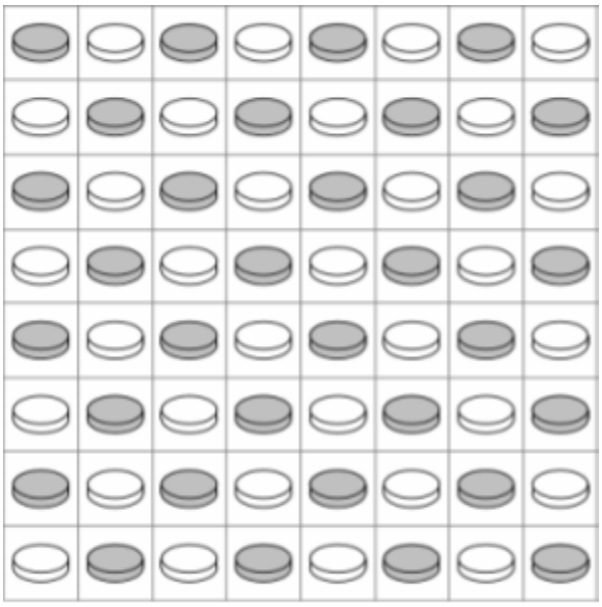

Figure 1: 8x8 Konane Board: Starting Position

Cultural Significance

Ancient Hawaiian Polynesians invented the game of Konane to practice strategic thinking. However, when European settlers arrived in Hawai'i, many aspects of culture were eroded and suppressed, including games like Konane. The subjugation and deficit understanding of native games are common to many cultures that have undergone colonization. This led to very few remembering the game, which was almost lost to history. The effort to revive games like Konane in Hawaiian schools started in the late 1990s, and moving the games online can further help that effort. Creating AI agents to play the game allows users to play easily and preserve the game into the future. [3]

Approach

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 evaluation 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. [2]

Utilizing the minimax search algorithm, the agent must 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. We will describe the reasoning and process of improving the evaluation and search below. [2]

Technical Considerations

The agent should:

- Represent the 8x8 Konane Board using an array of 8 arrays of size eight
- Represent the search space as a multi-way search tree for the minimax algorithm
- Take command line arguments [board_config] [B/W]
- Use algebraic notation

Improvements

Summary

There are likely to be several 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 heuristic to how the agent generates possible moves, such that the worst moves are ignored, and the agent can find more optimal moves sooner.

If we used the MCTS strategy, we would start with a random selection policy. Still, 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).

Search

The search function utilized the minimax algorithm, optimized with alpha-beta pruning (which will be discussed in more depth under Efficiency). The minimax algorithm allows an intelligent agent to minimize the maximum total loss and works by playing two sides of the game (maximizing and minimizing). In this way, the agent can avoid situations where the opposing player could make a choice that would give themselves a good position, as it follows that this would be a bad position for our agent. [2]

We decided to employ alpha-beta pruning for the Min-Max strategy, 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, reducing the total number of computations. We could also generate a better heuristic algorithm that considers some basic principles of Konane's “best practice” when generating possible moves. This is why we decided to utilize the Minimax strategy. [2]

Evaluation

The evaluation function is one of the most important parts of the move generation algorithm. To compare different moves, we need to be able to evaluate the board state resulting from the moves. We considered many ways to evaluate the board state.

Material: We can consider the number of pieces the player has left compared to the number of pieces the other player has left. The problem with this strategy is that the player could lose with more pieces on the board if they have no legal moves left.

```
// Initialize the counters
int blackCount = 0;
int whiteCount = 0;

// Loop through the board
for (int y = 8; y > 0; y--) {
        for (int x = 0; x < 8; x++) {
                // If the piece is black, increment the black counter
                if (game->board[y - 1][x] == BLACK) {
                        blackCount++;
                } else if (game->board[y - 1][x] == WHITE) {
                        whiteCount++;
                }
        }
}

// Return the difference between the two counters
if (game->maxPlayer == BLACK) {
        return blackCount - whiteCount;
} else if (game->maxPlayer == WHITE) {
        return whiteCount - blackCount;
} else {
        fprintf(stderr, "Error: Invalid max player!\n");
        exit(1);
```

Figure 2: Snippet from a Material-Based Evaluation Function

Moves: a better way to consider the boardstate is to use the win condition (being the last to make a move) to determine which side is winning. We can do this by counting the number of legal moves from the player, with the larger number being more valuable.

Efficiency

The search algorithm is made more efficient through alpha-beta pruning. In this technique, the search is optimized not to go down paths that will not improve the evaluation. The goal is to minimize the number of nodes visited in the tree search by keeping track of the lowest possible value the maximizing player can assume and the highest possible value the minimizing player can assume (starting at negative infinity and positive infinity, respectively).

```
// Get the number of valid moves for the current player
int currentValidMoves = node->size;

// Generate a temporary node for the other player
Node* tempNode = malloc(sizeof(Node));

//      Check if memory was allocated
if (tempNode == NULL) {
        fprintf(stderr, "Error: Memory not allocated for tempNode\n");
        exit(1);
}

// Initialize the temporary node
tempNode->game = node->game;
tempNode->capacity = 10;
tempNode->size = 0;

// Allocate memory for the children array
tempNode->children = malloc(tempNode->capacity * sizeof(Node*));

// Check if memory was allocated
if (tempNode->children == NULL) {
        fprintf(stderr, "Error: Memory not allocated for children\n");
        exit(1);
}

// Switch the turn
togglePlayer(&tempNode->game);

// Generate the children
generateChildren(tempNode, 1);

// Get the number of valid moves for the other player
int otherValidMoves = tempNode->size;
```

Figure 3: Snippet of a Move-Based Evaluation Function

Memory Allocation and Time Limit

The agent must return a move before the twenty-second move time limit. To do this, the agent should not consider every move or evaluate the full depth of the search tree. In order to do this, we need to limit the agent's processing to return a move before the time limit is reached. There are a few ways to do this. We decided to go with the time-limiting strategy, but others are possible.

Time limiting: We can include the <time.h> library to keep track of the time since receiving the move and return immediately when the time reaches twenty seconds. The problem with this is that it will inevitably lead to worse moves being returned to immediately return a move and stay within the limit.

Depth limiting: We could set a short depth limit so that the agent will not consider too many moves and exceed the time limit. In this case, the agent will play worse than those considering more nodes. Another issue is that this one does not consider the time limit and could potentially go over.

Iterative deepening: This technique allows the use of the timer and depth limit to get the best of both strategies and find the best move in the amount of time. Iterative deepening involves starting at a depth limit of 0, finding the best node, checking the time, then increasing the depth limit and repeating the process. We can also consider the result of the previous depth to determine the order of depth-limited BFS. This would allow us to increase the depth limit to correspond to the time limit and, ideally, get the most out of the time. [2]

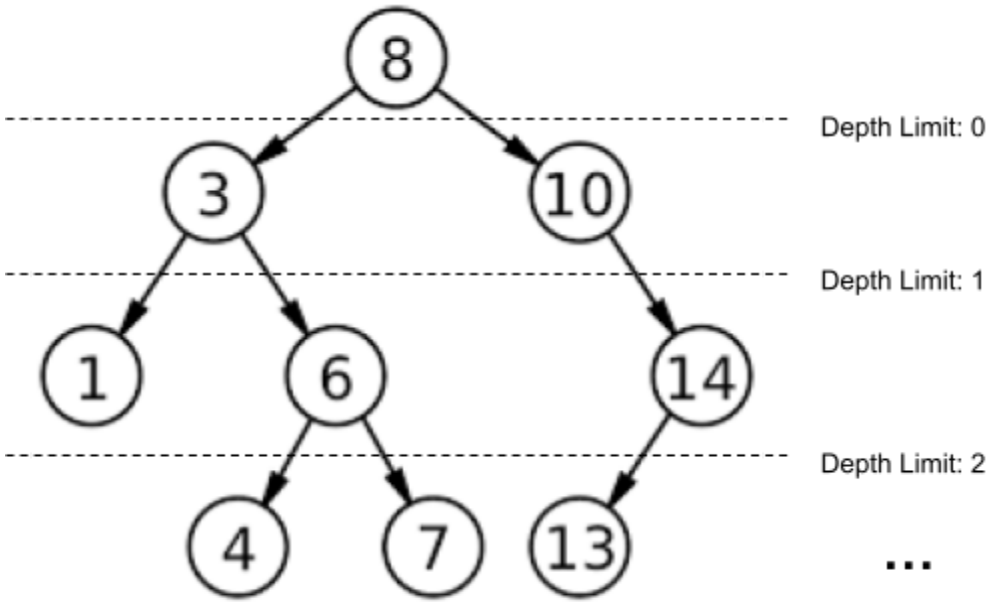

Figure 4: Iterative Deepening Depth-Limited Search Tree

Future Work

Pthreading

$$\pi = \frac{\prod_{n=1}^{\infty}\left(1+\frac{1}{4n^2-1}\right)}{\sum_{n=1}^{\infty}\left(\frac{1}{4n^2-1}\right)}$$

Figure 5: Example calculation of pi using two independent loops to calculate the numerator and denominator in parallel.

Allowing the computational power to be split among different threads can allow the agent to perform independent operations simultaneously. [3]

Strategy

Beyond the evaluation of the boardstate, the agent's play can be improved by implementing some basic konane strategies. A basic example of this is the agent's ability to play a first move involving a different mode than the rest. Another way we can implement strategy is to give moves from the edge of the board a lower multiplier, so the agent will not move edge pieces until later in the game. For now, implementing strategy is a future goal, and we focused on improving the evaluation and search first.

Reference List

[1] Anton, C. (2024, Jan 21-28) *Heuristic Search* [Slide Presentation]. CMPT 355: Introduction to Artificial Intelligence, Edmonton, AB. Canada

[2] Anton, C. (2024, Feb 1-7) *Adversarial Search* [Slide Presentation]. CMPT 355: Introduction to Artificial Intelligence, Edmonton, AB. Canada

[3] Ernst, M. D. (1995). Playing Konane Mathematically: A Combinatorial Game-Theoretic Analysis. *UMAP Journal*, *16*(2). https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=532fc4499a110b79b190e62e23de49c1c51b3f6f

[4] *pthreads explained*. (n.d.). ai-jobs.net. https://ai-jobs.net/insights/pthreads-explained/

[5] Ting, G., & Winters, K. (1991). Konane. *Cricket*, *18*(5), 21.