Using Monte-Carlo Tree Search (MCTS) to generate moves in a two-player tactical game: Ataxx
A Report for CMPUT 657 Heuristic Search

Talha Ibn Aziz
Student ID: 1669108
Email: taziz@ualberta.ca
Course Instructor: Michael Buro
University of Alberta

April 16, 2021

Abstract

Monte-Carlo Tree Search (MCTS) is a stochastic algorithm based on the Multi-Armed Bandit (MAB) problem, which can be used to search for best moves in stochastic games. Tactical games, on the other hand, have deterministic search algorithms which find the best possible move - assuming that each player plays optimally. In this project, I try to apply MCTS to a tactical game domain and compare its performance with the deterministic search algorithm Minimax. The game domain of *Ataxx*, which is a two-player game like chess or checkers, is used to validate the results.

1 Design

The project is built using three primary components, where each component has a specific set of functionalities. The components interact with each other throughout the process, and each component has a set of variables in memory. Figure 1 shows the interactions among the components, and the following sections describe them.

1.1 Component: Interface

The *Interface* is the driver component which takes input from the user and dictates most of the interactions between the other two components. The Interface allows the control of the program by introducing menu options like:

- setting up a new board (sizes are fixed)
- performing a specific move and storing it in memory
- undoing a move by loading the previously performed move from memory
- changing turns explicitly (also done implicitly when doing/undoing moves)
- generating moves using search techniques
- setting search parameters for those techniques
- playing automated games by pitting the different search techniques against each other.

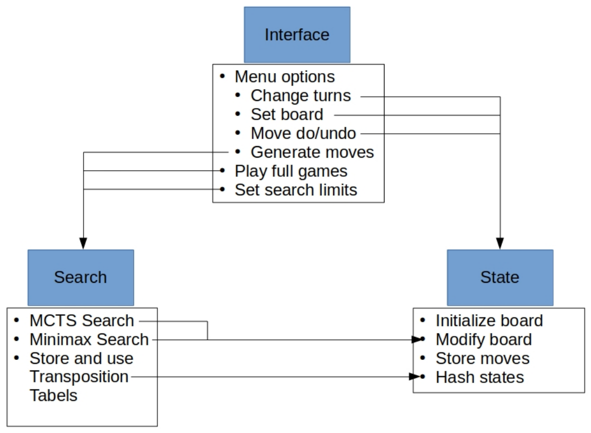


Figure 1: Design Diagram

1.2 Component: State

The *State* is the component which holds the game information and separates the game environment from the other components. This allows efficient modification of the board by other components without worrying about or interfering with low-level game details. The component also stores the variables containing the game information like - the number of pieces on the board, the possible sets of moves, the move history, etc., some of which are publicly available to other components.

State is the only independent component and is used by both the other components. The search algorithms complete their iterations by modifying the State component and therefore only one instance of the component is used by all the other components.

1.3 Component: Search

The *Search* component implements the search algorithms for comparison and maintains the other containers like the Transposition Tables and the Search Trees. The search algorithms used by the component are as follows:

1.3.1 Monte-Carlo Tree Search (MCTS)

The UCB1 (Upper Confidence Bound 1) variation of the MCTS algorithm is used, which is also known as UCT (Upper Confidence Bound 1 applied to trees) [Kocsis and Szepesvári(2006)]. There are four steps performed in the same order in this algorithm, as follows:

Step 1: Selection In this step, instead of randomly selecting moves at every node of the tree, the value v, to balance between exploitation and exploration, is used:

$$v = \frac{w}{n} + c \times \sqrt{\frac{ln(N)}{n}} \tag{1}$$

Here, w is the total number of games won by playing the move at the current node, n is the total number of games played using the same move, and N is the total number of games played using the move at the parent node. A constant c controls the randomness of the exploitation and exploration. The node with the highest value of v is selected as the best node in the selection step.

Step 2: Expansion This step expands the current node until all the moves possible at the node has been explored. The node is only expanded if it is selected in the previous step.

Step 3: Simulation After expanding the node, a random simulation from the current node to the end of the game, is performed to get a sample score.

Step 4: Back-propagation The resultant score is passed back to the parents, all the way upto the root of the tree. Along the way, the values of n and w, are updated at every node.

1.3.2 Minimax with Alpha-Beta pruning

The Minimax algorithm uses Alpha-Beta pruning to remove unnecessary branches and reduce the search tree significantly. It also used Transposition Tables to avoid duplicates as much as possible, and use previous search iterations to refine the search tree.

2 Implementation

The implementation can also be divided into three parts, each part corresponds to a separate component. The details are given below:

2.1 Main

The Interface component is implemented using the main function and the other components are driven by this function. The options available to the user to perform (according to the program order) are as follows:

- **i** - followed by a number, sets the size of the board
- **s** - followed by $i \times i$ characters, are used to set the board pieces
- **ft** - followed by a number sets the time limit per move when generating or searching moves
- **d** - followed by a number, sets the maximum depth for the Minimax search
- **g** - generates and performs a move using a search technique
- **p** - plays a full game between the two algorithms, from the current board state
- **m** - followed by the move locations, performs a specific and valid move (like 'ma3a4')
- **u, w, b, q** - are respectively used to undo a move, change turn to white, to black, and to quit the game
- **1, 2** - are used to change between the search modes, where 1 is for MCTS and 2 is for Minimax

The main function uses two classes - "ataxx_state" and "ataxx_search", which are detailed in the coming sections.

2.2 Class: ataxx_state

The State component is implemented in the form of the "ataxx_state" class. This class implements the game environment, stores the hashed states using a Zobrist Hashing Table, and modifies the board when required. The modifications are made efficient in terms of both time and space, by compressing the moves and board states (Figure 2). The move values can have a maximum row and column value of 8 (maximum board size), and so each row or column

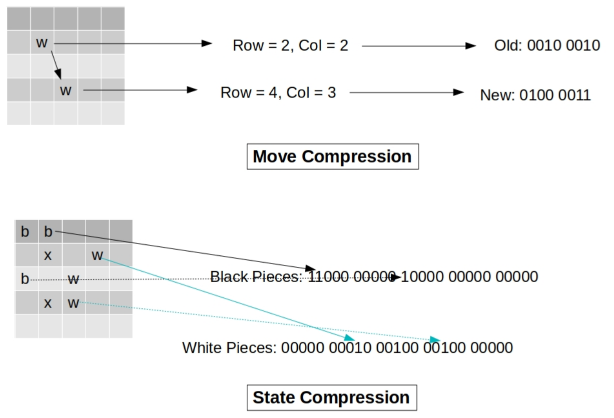


Figure 2: Compressed representation of moves and states

value can be stored in 4 bits. Two row and column values per location, and two locations can represent a move (except the capture information). A total of 16 bits are required for the compressed move storage, and the capture information can be stored in another 8 bits - each bit corresponding to an adjacent cell (Figure 3). A '1' means the piece was captured and '0' means otherwise.

The state representation is trickier as there can be at most 64 cells and considering two pieces, there are 2^{128} possible states, excluding the obstacles. However, as a 64-bit integer can be used to store piece information of one player - two such integers is used to store the entire state. The compressed moves are only used for storage, and the raw strings are used for faster printing and move processing.

All possible move generation is done using fixed arrays (or matrices) to store the changes - row and column-wise. The changes are considered for the center cell and applied accordingly - for example a move towards the bottom means a change in only the row value. The array can then be iterated to generate all possible moves.

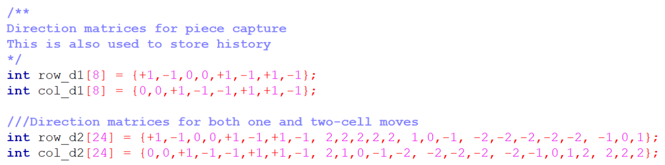
2.3 Class: ataxx_search

The Search component is implemented in the form of the "ataxx_search" class, which uses the MCTS algorithm or Minimax search to generate automated moves. The search algorithms modify a single instance of the State class ("ataxx_state") to traverse the search trees. This is done to reduce memory consumption and save time lost in passing large parameters. The transposition tables are also implemented in this class and are used by the Minimax search, locally. There are a few primary functions, which are long, but simple:

1. The 'MCTS' driver function which calls another recursive function. This is the driver function for the MCTS algorithm
2. The 'mcts_rec' recursive function (called by 'MCTS').
3. 'AlphaBeta', which uses Alpha-Beta pruning and is the Minimax recursive function.
4. The driver function for the Minimax search - 'Iterative_AlphaBeta'.

The MCTS driver function uses the UCB1 formula (Equation 1) to select the best node in the first step until a leaf node is reached. A leaf node is any node in the search tree that has not been fully explored yet. Initially, there is only a root node and after each iteration a new node is added to the tree in memory. After a fixed number of iterations, when a valid move is generated, the tree is discarded to save memory.

The second step adds a node from available memory - dynamic memory allocation is avoided for efficiency. A vector initialized at the start of the program with a fixed size, is used to allocate nodes to the tree until the available memory is exhausted. The third step does a random simulation from the selected leaf node to the end of the game, prioritizing clone moves at every state. The returned score is propagated to the root while changing the states back to the initial condition (as a single board state is used). In every iteration this is repeated, until time or memory runs out.


```
/**
Direction matrices for piece capture
This is also used to store history
*/
int row_d1[8] = {+1,-1,0,0,+1,-1,+1,-1};
int col_d1[8] = {0,0,+1,-1,-1,+1,+1,-1};

///Direction matrices for both one and two-cell moves
int row_d2[24] = {+1,-1,0,0,+1,-1,+1,-1, 2,2,2,2,2, 1,0,-1, -2,-2,-2,-2,-2, -1,0,1};
int col_d2[24] = {0,0,+1,-1,-1,+1,+1,-1, 2,1,0,-1,-2, -2,-2,-2, -2,-1,0,1,2, 2,2,2};
```

Figure 3: Generation of moves using direction arrays

The Minimax search is very similar to Assignment 1, except that the containers are changed and the algorithm is made modular. The details of the algorithm is mentioned in section 3.1.

3 Results

The result section has two sub-sections - section 3.1 describes the simulation environment of the experiments and section 3.2 analyzes the performance of MCTS in different scenarios.

	a	b	c	d	e	f	g	h
8	w							
7		x	x	x	x	x	x	
6		x	x	x	x	x	x	
5		x	x			x	x	
4		x	x			x	x	
3		x	x	x	x	x	x	
2		x	x	x	x	x	x	
1								b

Table 1: Board scenario 1: Linear map

3.1 Simulation Environment

The algorithms are implemented using the high-level language C++. A brief description of the rules of the game *Ataxx*, which is used as the experimental domain, is given below:

1. There are two players - *black* and *white*. Each player alternatively gets a turn to play a move.

2. Each player starts with a fixed number of pieces on a square board. The board sizes used in the experiments range from 4×4 to 8×8.

3. There might be obstacles in the board, where board pieces cannot be placed. Furthermore, a piece cannot be placed where there is already another existing piece (same or opposite).

4. A player can move a piece one-square in any direction, i.e., to any adjacent location, as long as it is not outside the board. This move is called *cloning*, as the piece is duplicated - after the move is done. Therefore, there are two pieces - one in the old location and another in the new location.

5. A player can also move two-squares in any direction which is called *jumping*. However, in this case, after the move is done, the piece is removed from the old location.

6. A *pass* move is not playing any move, and can only be done if no other moves are available.

7. The game ends when there are no moves left for both the players. The player which has more pieces compared to the opponent, wins the game. If both players have the same number of pieces, it is a draw. The game also ends if a player performs more than 50 jump moves in a row.

The algorithm used to validate and compare with MCTS is a variant of the Minimax Search algorithm, and is described as follows:

- In each state of the search tree, each player selects the best move from an available set of moves, according to an evaluation function.

- The evaluation function is the difference between the white and black pieces. For example, the best move for a white player is the one which results in the most number of white pieces.

- *Alpha-Beta* pruning is used to discard sub-trees which are not essential for finding the optimal move. It uses a lower bound *alpha* and an upper bound *beta* to reduce the window where the search needs to be performed.

- Transposition Tables are used to avoid repetition of moves, to select the best move of the previous search (if any), and to update the *alpha-beta* window.

3.2 Experimental Analysis

The MCTS algorithm focuses on cloning when performing the third step. The prioritization is done using a random value r_p, which forces a clone move probability $\frac{r_p-1}{r_p}$. For $r_p = 2$, cloning and jumping moves have equal probability of occurring. For

	a	b	c	d	e	f	g	h
8	w							
7		x	x	x	x	x	x	
6		x					x	
5		x					x	
4		x					x	
3		x					x	
2		x	x	x	x	x	x	
1								b

Table 2: Board scenario 2: Non-linear map

	a	b	c	d	e	f	g	h
8	w							
7								
6		x					x	
5								
4								
3		x					x	
2								
1								b

Table 3: Board scenario 3: Typical open map

$r_p = 1$, each possible move has equal probability of occurring, which in most cases increases the probability of jump moves. This is because the number of available jump moves is typically twice the number of available clone moves.

The Minimax search dominates MCTS in most board matches, which can be understood if a few scenarios are studied. The scenarios used are such that the obstacles are monotonously decreasing in each initial board, to study the effect of the search, with various available options. For example, the board in Table 1 places obstacles in such a way that the map becomes linear. The best strategy here is to keep cloning until a jump move can be made adjacent to an opponent piece. The player which performs even a single jump move before meeting the opponent, loses the match. The gap is further increased if more jump moves are made. The MCTS algorithm, which has a lower (but not zero) probability of jumping, still loses to Minimax. This is because Minimax is able to determine that cloning increases it's score in each step. Contrarily, MCTS still makes jump moves, albeit at a lower rate compared to clone moves. MCTS loses having 9-12 pieces less, compared to Minimax.

The second map (Table 2) forces the players to jump into the center and therefore the winning strategy is more complex compared to before. In this map, MCTS is very quickly defeated by Minimax as in order to create a stronghold at its own corner, MCTS needs to continuously clone itself for the first couple of moves. This is not possible as MCTS is programmed to randomly jump at a certain interval. Conversely, Minimax forms a group of its own pieces, and then shoots for the opponent as soon as it finds the opponent (a fixed depth away) while keeping backup at its own corner. The simulations done in this map show that initially the score is more or less even. However, as soon as Minimax pieces reach the opposite corner, MCTS loses in a battle of attrition as Minimax can look further ahead almost like a perfect player (at least upto a fixed depth).

The third map (Table 3) is the most difficult map, where all manners of strategies are possible. The more deterministic algorithm Minimax, finds the best strategies upto a fixed depth and executes it efficiently. MCTS is again slow to find the best move in the same time as much of the time is taken in randomizing the moves according to a selection function. The total number of iterations completed by MCTS in one second (in my local computer) is about 1600. Minimax on the other hand, traverses about 200,000 moves in the same time. If MCTS is made efficient and fast enough to look ahead like Minimax (upto a finite distance), it's performance can improve greatly.

References

[Kocsis and Szepesvári(2006)] Kocsis, L.; and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.