Monte Carlo Tree Search in C

Jonathan Carter
University of Oxford
jcarter@robots.ox.ac.uk

December 3, 2020

Abstract

A command line version of the board game Connect4 and the Monte Carlo Tree Search reinforcement learning algorithm were implemented in C. The implementation, which can be found here[1] on GitHub, is able to play the game to a super-human level running in real time.

1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a planning algorithm for discrete state-action spaces that works by simulating future states and building up a search tree. At each iteration the state tree is traversed to a leaf node i.e. an unexplored state. The tree is then expanded by taking an unexplored action and then taking random actions until the episode finishes. The reward from the episode is then propagated back up the tree, updating the values of the traversed states. This process is illustrated in Figure 1 and explained in detail in Section 8.11 of Sutton and Barto's comprehensive introduction to Reinforcement learning [1].

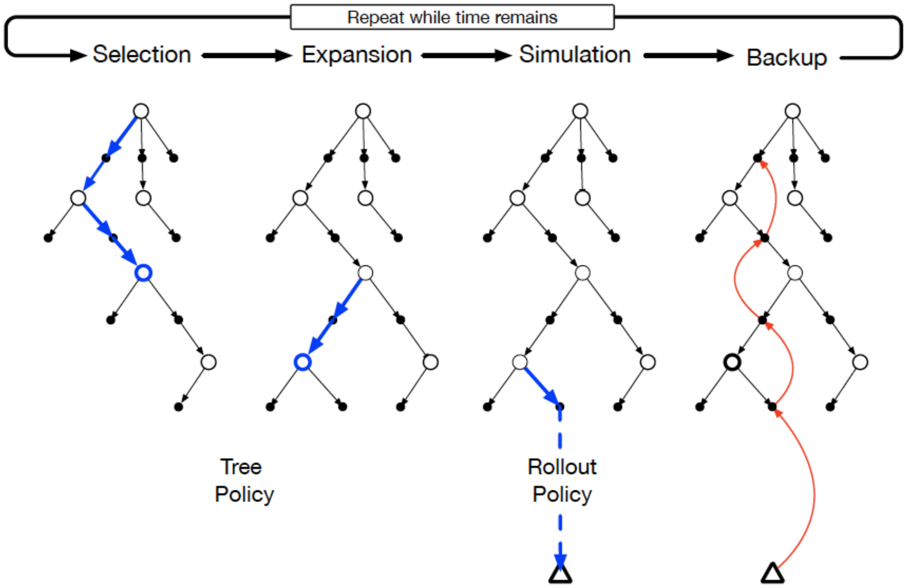


Figure 1: MCTS diagram from [1].

In simulation the tree is traversed using the UCB policy [1], at each node the action that maximises (1) is taken to select a child node.

$$a_{UCB1} = \max_{a_i \in \mathbb{A}} w_i/n_i + \sqrt{\frac{2 \log N}{n_i}} \tag{1}$$

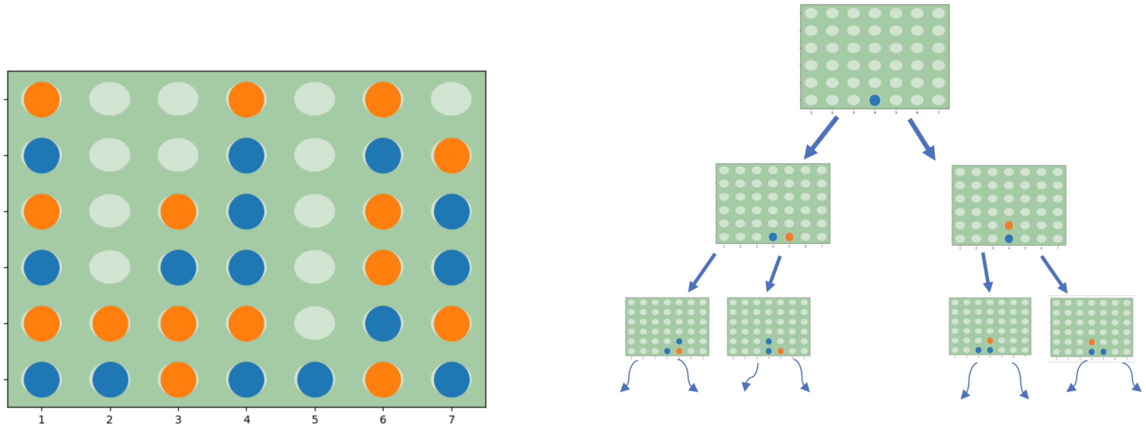
[1] https://github.com/joncarter1/MCTS-C

w_i is the total value of all games simulated from the i'th child node, n_i is the number of games simulated through the child node and N is the number of games simulated through the current node. The UCB policy balances exploitation and exploration in a principled manner; the second term in (1) encourages the exploration of unfrequented nodes.

2 Connect4

Connect4 is a popular children's board game for two players each player takes turns to drop a counter into a selected column. A player wins if they create a line of 4 counters of their own colour in any direction. At each turn each player has a maximum of 7 moves, and there is a maximum of 42 moves in a game, making it a sensibly sized problem to tackle on desktop-level hardware.[2]

The possible game states can be represented as a tree as illustrated in Figure 2b, with each node in the search tree storing the mean of the values of all game simulations that passed through that node: +1 for a win for player 1, +0.5 for a draw and 0 for a win for player 2. This gives an estimate of the value of that game state for either player. A policy may therefore be found by choosing the child node with the largest or smallest value, depending on whether the agent is playing from the perspective of player 1 or 2.



(a) Winning position for player 2 (orange)

(b) Subset of game tree after initial move.

Figure 2: Connect4 - Example Winning Position and Game Tree. Each board position is a node in the tree, edges are the possible moves from each position.

3 Program Design

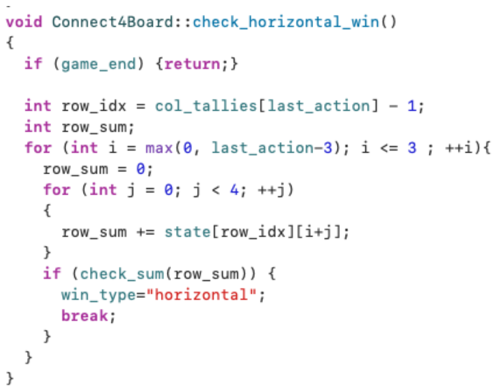
As previously mentioned, the code can be found at the following repository: https://github.com/joncarter1/MCTS-C. This section gives a brief overview of the program's functionality.

The code is divided between three files: main.cpp, which imports instantiates the relevant objects and runs the game; mcts.cpp, which implements the MCTS algorithm including the search tree; and connect4.cpp, which implements the game itself.

3.1 Connect4 Implementation

The Connect4Board class represents the board using a matrix. Adding counters is mimicked by updating relevant matrix values to ± 1 dependent on which players turn it is. An additional vector contains the number of counters already placed in a column to index into the matrix correctly. Using this formulation checking for a win is equivalent to checking that the sum of four consecutive elements in any direction is equal to ± 4. To further reduce computation only the rows and columns around the last counter placed need to be checked. The code used to check for a horizontal win is shown in Figure 3.

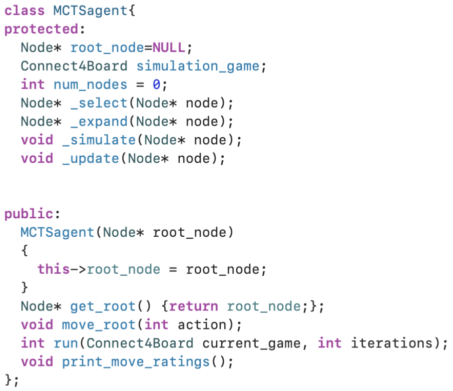
[2] As opposed to Chess or Go!



```
void Connect4Board::check_horizontal_win()
{
  if (game_end) {return;}

  int row_idx = col_tallies[last_action] - 1;
  int row_sum;
  for (int i = max(0, last_action-3); i <= 3 ; ++i){
    row_sum = 0;
    for (int j = 0; j < 4; ++j)
    {
      row_sum += state[row_idx][i+j];
    }
    if (check_sum(row_sum)) {
      win_type="horizontal";
      break;
    }
  }
}
```

Figure 3: Checking for a horizontal win



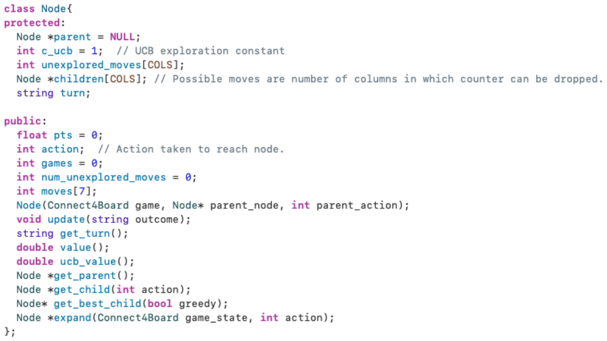
```
class MCTSagent{
protected:
  Node* root_node=NULL;
  Connect4Board simulation_game;
  int num_nodes = 0;
  Node* _select(Node* node);
  Node* _expand(Node* node);
  void _simulate(Node* node);
  void _update(Node* node);


public:
  MCTSagent(Node* root_node)
  {
    this->root_node = root_node;
  }
  Node* get_root() {return root_node;};
  void move_root(int action);
  int run(Connect4Board current_game, int iterations);
  void print_move_ratings();
};
```

Figure 4: MCTS Class declaration

3.2 Monte Carlo Tree Search Implementation

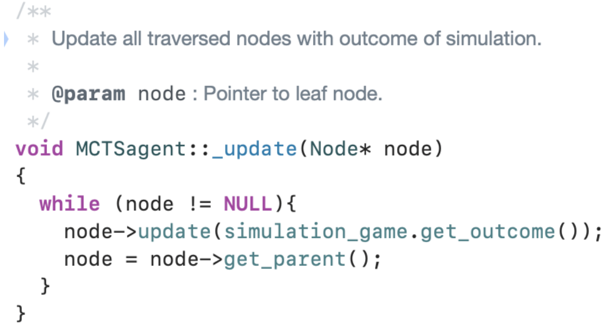
The MCTSagent class declaration is shown in Figure 4. The class has private methods that implement each step of the MCTS algorithm (Figure 1). The agent stores a copy of the current game through the variable 'simulation_game'. At each turn this copy of the game is simulated to completion for a set number of iterations to build up the search tree, before taking an action in the real game. The 'root_node' pointer stores the address of the root node of the search tree, represented by the Node class. The 'move_root' method is used to move this pointer down the tree after an action is taken in the real game, so that the existing search tree is built upon at each step.



```
class Node{
protected:
  Node *parent = NULL;
  int c_ucb = 1;  // UCB exploration constant
  int unexplored_moves[COLS];
  Node *children[COLS]; // Possible moves are number of columns in which counter can be dropped.
  string turn;

public:
  float pts = 0;
  int action;  // Action taken to reach node.
  int games = 0;
  int num_unexplored_moves = 0;
  int moves[7];
  Node(Connect4Board game, Node* parent_node, int parent_action);
  void update(string outcome);
  string get_turn();
  double value();
  double ucb_value();
  Node *get_parent();
  Node *get_child(int action);
  Node* get_best_child(bool greedy);
  Node *expand(Connect4Board game_state, int action);
};
```

Figure 5: Node Class declaration



```
/**
 * Update all traversed nodes with outcome of simulation.
 *
 * @param node : Pointer to leaf node.
 */
void MCTSagent::_update(Node* node)
{
  while (node != NULL){
    node->update(simulation_game.get_outcome());
    node = node->get_parent();
  }
}
```

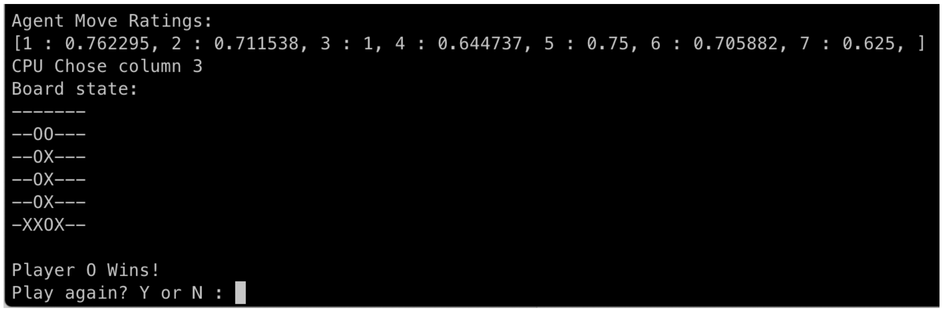
Figure 6: Node update method

The Node class declaration is shown in Figure 5. It stores the addresses of its parent node and all known child nodes. When initialised it uses the current game to assess all possible actions for exploration. In simulation, the 'ucb_value' method is used to guide the search through the tree using the UCB1 policy (1). In the real game against an opponent the agent chooses the action that maximises the empirical mean value of games simulated that passed through the node, using the 'value' method. The 'get_best_child' function is used to find the optimal action in both simulation and the real game, using the 'greedy' argument to differentiate between the real and simulation game respectively.

After simulating a game the '_update' method of the MCTSagent class is used to update the value of each node. The code for this method is shown in Figure 6. The function traverses back up the tree from the leaf node of the simulation, updating the value of each node along the way to the root node.

3.3 User Interface

A screenshot from the game is shown in Figure 7. The user is able to select either player 1 or 2 and the number of search iterations allowed by their computer opponent. At each turn the computer displays its empirical value of each move (points/games).



```
Agent Move Ratings:
[1 : 0.762295, 2 : 0.711538, 3 : 1, 4 : 0.644737, 5 : 0.75, 6 : 0.705882, 7 : 0.625, ]
CPU Chose column 3
Board state:
-------
--OO---
--OX---
--OX---
--OX---
-XXOX--

Player O Wins!
Play again? Y or N :
```

Figure 7: Command line Connect4.

4 Remarks

The algorithm runs remarkably quickly. The number of simulations per turn can be set as high as 25000 before the runtime between turns becomes noticeable. This is far higher than that required for super-human performance; I was unable to beat the computer as either player 1 or 2 with the agent allowed more than 500 iterations per turn. Theory indicates that there is a forced win for player 1 given they place their first counter in the middle column. Given over 500 iterations and the first turn the agent never picks anything other than the middle column on the first turn. The MCTS algorithm is remarkably effective at learning close to optimal behaviour despite only searching a tiny fraction of all possible states.

References

[1] Richard S Sutton and others. *Introduction to reinforcement learning*, volume 2.