Midterm Project

Members : 林文智 (105265431)， 亞琳娜 (105065425)， 蔡潔詩 (x1050037)

1. Introduction

The project is to implement a computer program that uses MCTS (Monte Carlo Tree Search) to play five-in-a-line in a Go-game board of 15 by 15. A five-in-a-line Go-game is played on a Go-game board using black and white stones. The "quality score" of a "win"," tie", "loose" of a game is being considered.

1.1 Goal Formulation

Goal state : Getting an unbroken row of five stones horizontally, vertically, or diagonally on the Go-game board of 15x15 to win.

1.2 Problem Formulation

State : Position of stone on the Go-game board.

Initial State : Black starts. The player may place his stone on any intersection of two lines on the Go-game board which position is assigned according to the Row (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) and the Column (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14).

Actions : Play a stone of either black or white without spending more than 10 seconds.

Transition model : The initial state will run through the MCTS (Monte Carlo Tree Search) and add one stone in each turn to generate the child state.

Goal test : Winning the game by connecting stones in a connected line either horizontal or vertical or diagonal

Path cost : Time to obtain the solution

2. Monte Carlo Tree Search Steps

2.1 Selection

Starting from the root, successive child nodes are selected down to a leaf node, where the selected position is previously not added to the tree yet. For each node, the son with the highest sum of exploration and exploitation value (visit count) is chosen.

- Exploration : The square root of the ratio log(#simulations of the parent) / (#simulations of the son)
- Exploitation : The percentage of won simulations from the son

2.2 Expansion

One or more child nodes are created or choose per play-out to expand the tree, according to the available actions.

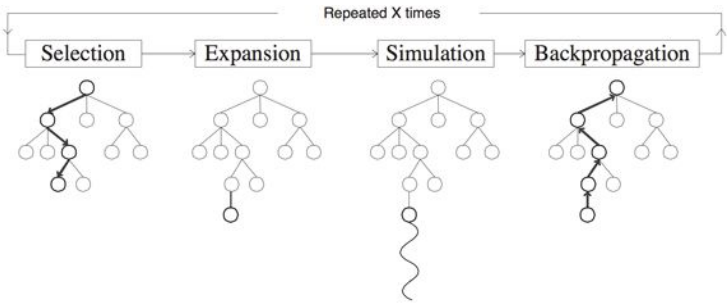
2.3 Simulation

A simulation is run from the new node to play a random play-out and use beyond the search frontier. Once a terminal node is achieved, simulation should return a score.

2.4 Back propagation

After reaching a terminal node, the result is used to update information in the nodes expanded in selection and expansion. The result is propagated back along the path from the selected node to the root node.

- Add reward value to node
- Increment node visit count



3. Heuristics

3.1 Smart Action List Selection

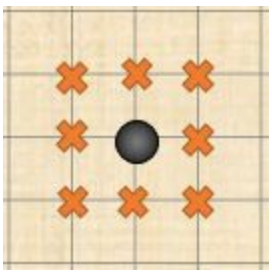


Figure 1

In order to execute both expansion and simulation steps, we should provide monte carlo with a list of legal moves. A naive approach would be to do a full scan of the board finding every open positions for the player, but this would be prohibitively expensive and would suggest actions that are not smart or with purpose. What we proposed was to only consider the closest positions to any movement previously done. For example, if a player adds a piece to position (i,j), we add the positions (i-1,j-1), (i,j-1), (i+1,j-1), (i-1,j), (i+1,j), (i+1,j+1), (i,j+1), (i+1,j+1) (as shown in the figure 1). This actions are stored in a set variable to make sure we are not generating duplicates. A check is executed as well to verify that the positions to be added are within the board bounds and don't have any existing pieces assigned to it.

3.2 Delayed Rewards

Delayed reward is a reinforcement learning concept in which the agent learn the most desirable action based on the amount of reward each action receives. The longer path to the goal state should receive less reward than those that are shorter.

We applied delayed reward in the simulation phase by keeping track of the simulation depth, penalizing each level up to a cap. This came out of the need to reinforce those actions that lead to faster winning terminal states, and thus we achieved an agent that is more aggressive in its action selection.

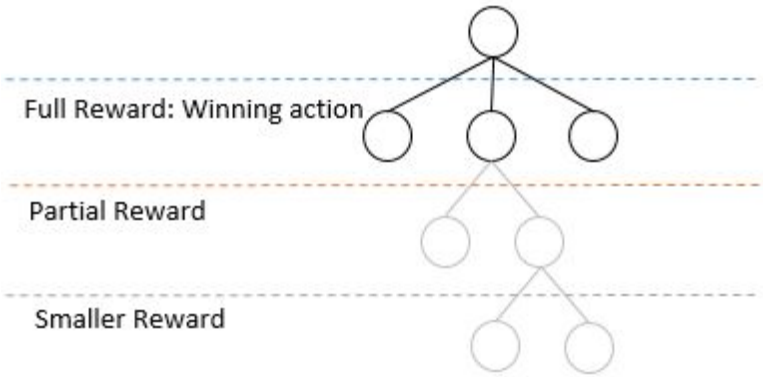


Figure 2

3.3 Thread Execution

Due to the time execution for our monte carlo, we tried several ways to improve our execution time and implement parallel execution. It has been proven (3) that one of the best approach to thread execution is to run independent monte carlo instances and make them vote for the best possible move. We select the movement with the most votes or, in case of tie, the highest score. The agent execution was greatly benefitted by this strategy since it reduces the risk of missing the best move due to the randomness inherent in monte carlo.

3.4 Terminal Node Identification

One of the biggest objectives during the agent implementation was to optimize every part of the monte carlo execution, with special focus on the simulation phase as this is the one that is repeated the most. Terminal node identification should be done in each simulation level to verify if a player won, lost or tied with the recently tried action. The naive approach of doing a full matrix scan would have been too slow. In order to resolve this, we keep a record of the latest piece inserted, and scan the nearest 5 positions around it to verify if the latest action lead to terminal node. This check is done every time a player inserts a piece as well and so we are certain that once a terminal node is reached, the game will be over.

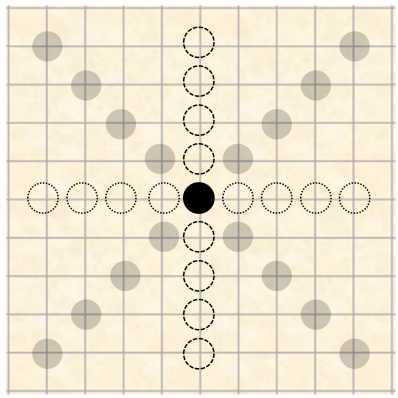


Figure 3

3.5 Threat Management

In general, we expect our agent to behave accordingly to the following logic:

```
If next_best_move leads to a winning state then
        Execute next_best_move
else if threads_exists then
        Address threat by blocking enemy
else
        Execute next_best_move
```

A threat is identified by executing a check on the positions vertically, horizontally and diagonally to the last enemy movement (similar to Figure 3). If 3 or more enemy pieces are found in a line with no obstacles, we should block them to avoid the threat to get worse. In order to identify any threat pattern dynamically, we implemented a regex dictionary to deal with the threat identification.

4. Conclusion

Monte Carlo Tree Search is a powerful approach to gaming agents since it allows an agent with little knowledge about the game rules to generate possible moves and use reinforce learning to improve its selection. unfortunately, due to the Gomoku's board size, we couldn't just rely on Monte Carlo's result and further heuristics were implemented to improve the agent 'intelligence' as seen in section 3. We were able, at the end, to create a semi intelligent agent that tries to execute the best possible movements but is careful regarding enemy threats. It still would be easily beat by an player with gomoku strategy knowledge and by creating threat forks (1), but is definitely better than random selection and avoids basic losing scenarios.

5. References

1. What is Renju? (2016) Renju International Federation retrieved from http://www.renju.net/study/rules.php
2. Allis, L. V., H. J. van den Herik, and M. P. H. Huntjens. "Go-Moku and Threat-Space Search."
3. Alkhafaji, Hiba. "Monte Carlo Tree Search."
4. Go-moku rules (2016) Your Turn My Turn retrieved from http://www.yourturnmyturn.com/rules/go-moku.php