ChecknGo

Chaitanya Mitash

February 7, 2017

Compiling and starting the game

1. Download the code repository.
2. cd path-to-repository/AIGame
3. mkdir build
4. cmake ..
5. make
6. ./ChecknGo

Troubleshooting

- The code has 3 package dependencies : cmake, OpenGL and GLUT. In case any of the package is not present, cmake command will give an error. You can install the packages with apt-get.
- setup has been tested on Ubuntu 14.04.

Game Modes

There are 5 modes in the game. Press 1-5 keys on your keyboard to select one of the following modes.

1. to play against another player press '1'.
    - Blue starts the game.
    - select one of your pieces.
    - if you selected a valid cell, you are shown options to move in RED.
    - click on one of the RED positions to move there.
    - when you move to a position, all opponent neighbors turn to your color.

- game ends when a player has no where to move.
- winner is decided based on the count of cells of respective colors.

2. to play against a simple random AI press '2'.
   - this is the place to practice your moves, against a simple random AI.
   - the player starts the game in Blue.
   - AI move :

$$a \leftarrow rand(\{legal(s, a)\})$$

3. to play against a MINMAX AI press '3'.
   - given enough time and memory, AI can play an optimal game by backtracking the moves from leaves (final game states) to current state. However, the complexity of this search increases exponentially with the breadth of the search tree (possible actions in each state).
   - This makes the search intractable, so we restrict ourselves to play optimally by looking forward a few steps. This demands that we define goodness values to intermediate states. In this case this could be as simple as

$$goodness \leftarrow (\#Color1 - \#Color2)$$

     . However, other heuristics can be added to goodness, such as number of "safe pieces", "attacking pieces" and so on.
   - then the AI choses

$$\begin{aligned} a &\leftarrow argmax_a\{MINVAL(s, a)\} \\ MINVAL(s, a) &\leftarrow min_a\{MAXVAL(s, a)\} \\ MAXVAL(s, a) &\leftarrow min_a\{MINVAL(s, a)\} \end{aligned}$$

     this is the basic recursive structure of the algorithm, however the code takes care of the boundary cases and other intricacies. The algorithm stops at a pre-specified depth ($THRES\_DEPTH = 2(default)$)
   - the AI already is a very "Smart" competitor. Increasing the threshold to a depth more than 2, makes it perform even better but at a slower speed.

4. to play against a Monte Carlo Search Tree (MCTS) AI, press '4'.
   - This is the basic Monte Carlo Tree search implementation. It runs multiple simulations at each step to figure out the best action, based on the outcome of the simulations.
   - As the simulation is playing against a random opponent, it does not really get the best choices always. However, increasing the number of simulation steps (NUM_SIMULATIONS) and simulation depth (THRES_MOVES) makes the choices better.

5. to simply watch a game between MINMAX and MCTS, press '5'.

    - MINMAX beats MCTS with the default threshold values. This is expected as MINMAX is an exhaustive search (even though depth limited in this case) compared to MCTS which averages out the the behavior of several random games.
    - MINMAX becomes really slow as we increase the depth, because of the exponential branching factor. However, the speeds are comparable with the default parameters and MINMAX performs better for that speed.

What more could be done ?

- MINMAX can be made faster by alpha-beta pruning and then we can increase the depth of the search.
- Use Upper Confidence bounds on MCST. This would require hashing of state values.
- Maybe also learn an evaluation function with a lot of simulations from a slow but accurate algorithm and then replace the function in the online version. (Maybe a deep network as used by AlphaGo).