Theory of Computer Games 2022 – Project 3

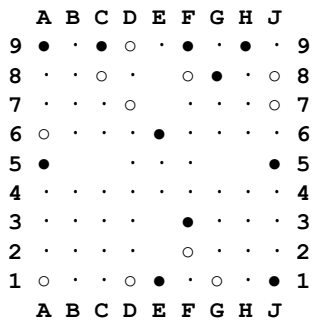
Overview: **Write an MCTS program to play *Hollow NoGo*, a variant of NoGo.**

1. Implement a Hollow NoGo framework on the 9x9 board.
2. Design a Hollow NoGo player with Monte Carlo tree search (MCTS).
3. Use Go Text Protocol (GTP) to communicate with other programs.

Specification:

1. Generally speaking, 9x9 Hollow NoGo has the same rules as 9x9 NoGo. The only difference is that some positions are cut out, which cannot be counted as liberty.
2. Two players (black and white) alternatively take turns to play. The same as in NoGo, **capturing, suicide, and pass are forbidden actions**. The player who has no action to play loses the game.
   a. Capturing: take the last liberty of an opponent block.
   b. Suicide: create a block that has no liberty.
   c. Pass: skip this turn without placing a stone.

   For example, in the right figure, the white player cannot play at A4 (capturing), G9 (suicide), E3 (illegal position).



3. The player should take actions based on MCTS and should be able to play as both sides.
   a. **The player's thinking time is 40 seconds for a game**, i.e., for the black player, the total time spent on black stones in a game should not exceed 40 seconds.
4. The Go Text Protocol (GTP) should be implemented using standard input/output.
5. Other implementation requirements:
   a. The program should be able to execute in the Linux environment.
      i. C/C++ is highly recommended for TCG projects since the methods involved are sensitive to CPU speed. If you need to use other programming languages (e.g., Python) for the projects, contact TAs for more details.
      ii. The makefile (or CMake) for the program should be provided.

Methodology:

1. **The framework of Hollow NoGo with GTP is provided**.
   a. It is similar to the 2048 framework; you should be able to get familiar with it quickly.
   b. It is recommended to follow the instructions below step by step.
2. **Make sure that you are familiar with the MCTS**. Think carefully about every implementation detail. You should implement the simplest version first.
   a. Use fixed simulation count, e.g., a total number of 100 MCTS cycles.
   b. At the selection phase, just use a simple UCB formula. Keep selecting child nodes until a leaf node, i.e., fully expanded node, is reached.
   c. At the expansion phase, expand the child node in random order. You may expand all the child nodes together, but only one child node is visited and updated, the other

child nodes are still unvisited (with visit count = 0). When the next time this node is selected, an unvisited child will be selected due to its highest UCB value.

   d. At the simulation phase, rollout by the random policy. Just place stones randomly until the end of the game and obtain the result. If the node is already a terminal state, just return its result.
   e. After the search, select the best action based on the visit count of child nodes.
3. With a correct MCTS implementation, the player with a few simulation counts (e.g., 100) should be able to outperform the random player, with a win rate of more than 90%.
4. Change the fixed simulation count to advanced time measurement.
5. Finally, try more improvements, e.g., more UCB variants, RAVE, heuristic rollout, move ordering, time management, parallel MCTS.

Scoring Criteria:

1. **Win rate against a weak player (70 points)**: Calculated by $\lceil \mathrm{WinRate}_{\mathrm{weak}} \times 70\% \rceil$.
   a. $\mathrm{WinRate}_{\mathrm{weak}}$ is the win rate against a weak sample player in 70 games. Your program will be black in 35 games and be white in another 35 games.
   b. **The weak sample player is provided**, you should check the correctness of your work by the sample program and the provided scripts before the deadline.
2. **Win rate against stronger players (30 points)**: Calculated by $\lceil \mathrm{WinRate}_{\mathrm{strong}} \times 30\% \rceil$.
   a. $\mathrm{WinRate}_{\mathrm{strong}}$ is the win rate against stronger sample players in 30 games. Your program will be black in 15 games and be white in another 15 games.
3. **Report (10 points, optional)**: Graded according to the completeness of the report.
   a. Summarize the methods used, improvements, and so on.
4. Penalties:
   a. **Time limit exceeded (lose that game)**.
   b. **Illegal action (lose that game)**.
   c. **Late submission (–30%)**: If the project requires any modifications after the deadline.
   d. **No version control (–30%)**: If there is no version control.
   e. **Failed demo (–30%~100%)**: If you cannot demo the program or cannot answer the asked question.
5. The final grade is the sum of the indicators minus the penalties.
   a. **The maximum grade is limited to 100 points**.
   b. **The grade is not counted if the demo is not passed**.

Submission:

1. The submission **should be archived as a ZIP file** and **named `ID.zip`**, where **`ID`** is your student ID, e.g., `0356168.zip`.
   a. Pack the **source files**, **makefile**, **report**, and other required files.
   b. Submit the archive **through the E3 platform**.
   c. Do not upload the version control hidden folder, e.g., the `.git` folder.
2. The program **should be able to run under the provided Linux workstations**.

a. Available hosts: tcglinux1.cs.nycu.edu.tw, …, tcglinux10.cs.nycu.edu.tw
    i. Use the NYCU CSIT account to log in via SSH.
    ii. Place project files in `/tcgdisk/ID`, where ID is your student ID. Note that you need to create the folder first. For example, suppose that your ID is 0356168:
    ```
    $ mkdir /tcgdisk/0356168 && chmod 700 /tcgdisk/0356168
    ```
b. The projects will be graded on the provided workstations.
    i. You may use your machine for development. The judge program should work on most Linux platforms.
    ii. Each workstation has 4 threads and 7.8GB of memory, which can be shared by 3–4 people. **Pay attention to the resources used by your program**.
c. Do not occupy the workstations. Contact TAs if the workstations are crowded.

3. Version control (e.g., GitHub or Bitbucket) is required during the development.