Go Project

Yilin Zheng 11510506
School of Computer Science and Engineering
Southern University of Science and Technology
Email: 11510506@mail.sustc.edu.cn

1. Preliminaries

This project is doing some simple implementations on Go game. Go is a an abstract strategy board game for two players, in which the aim is to surround more territory than the opponent [1]. As an old game, computer scientists try to use computer programs to compute every steps. However, the challenge thing is the oversize searching space which leads to large amount of computing. In this project, I just implement the judge of the rule, the steps to eat opponent chess and the possible steps next with the white hand. This part will describe the software and algorithm I used for this project.

1.1. Software

This project is written in Python using editor Sublime Text and IDE PyCharm. The libraries being used includes Tkinter, NumPy and Copy.

1.2. Algorithm

The algorithm being used in this project includes depth-first search. The method being used includes recursion. I define another three functions in order to modularize the codes. With the given functions, which includes three unfinished blocks, there are nine functions in the Python file go_test.

2. Methodology

Despite the go rule is so simple that can be describe in a sentence, the implementation is full for details handing and requires appropriate data structure to represent. This part describes the representation, the more detail of algorithm and the architecture I used in the codes.

2.1. Representation

This project contains some main data need to be maintain during the process: **chess board**, **chess status**, **step offset** and **output result**. Also some attached data needed during operation on some functions will be specified inside functions.

- **chess board**: nested list
- **chess status**:
  - COLOR_BLACK: black chess
  - COLOR_WHITE: white chess
  - COLOR_NONE: none chess
  - POINT_STATE_CHECKED: visited
  - POINT_STATE_UNCHECKED: unvisited
  - POINT_STATE_NOT_ ALIVE: dead chess
  - POINT_STATE_ALIVE: alive chess
  - POINT_STATE_EMPYT: none chess
- **step offset**: list of tuple
- **output result**: for offset, use list of tuple while for judgement outcome, use boolean value

2.2. Architecture

Here list all functions in the Python file go_test with their usage.

- Given:
  - ***read_go***: read the chess manual from txt file.
  - ***plot_go***: plot the GUI according to the reading result.
  - ***is_alive***: judge whether certain chess is alive.
  - ***go_judge***: judge whether the chess board follow go rule.
  - ***user_step_eat***: search the steps which leads to eat. opponent chess.
  - ***user_step_possible***: search the possible steps for white hand.
- Self define:
  - ***find_chess_block***: find the chess block with same color.
  - ***plot_possible_step***: plot the possible steps on the bard with read oval.
  - ***write_result***: Write results into txt file.

The last of the Python file, use *main()* to test all the codes.

2.3. Detail of Algorithm

Here describe the detail of algorithm in some vital functions.

- ***find_chess_block***: Use recursion, search the chess block with the same color, the direction includes the surrounded four directions.

Algorithm 1 find_chess_block

Input: $check\_state$, $chess\_board$, (i, j), $points$, $chess\_color$
Output: chess offset block $points$ //same color

$points \leftarrow points \cup (i, j)$ //points is the set of chess offset
for $offset \in (i-1, j), (i+1, j), (i, j-1), (i, j+1)$ **do**
  if $chess\_board[offset] == color\_type$ **and** $offset \notin points$ **then**
    $points \leftarrow$ find_chess_block( ) //recursion
  end if
end for
return $points$

- ***is_alive***: For an alive chess block, there must exist at least one vitality.

Algorithm 2 is_alive

Input: $check\_state$, $chess\_board$, (i, j), $chess\_color$
Output: $point\_status$

$points \leftarrow$ find_chess_block( ) //get chess block
for $offset \in points$ **do**
  for four directions of $offset$ **do**
    if find none chess in x **then**
      $check\_state[x] \leftarrow checked$ //x is also an offset
      return $point\_status \leftarrow is\_alive$ //find vitality
    end if
  end for
end for
return $points\_status \leftarrow is\_not\_alive$

- ***go_judge***: If find any dead chess, then the board must disobey go rule.

Algorithm 3 go_judge

Input: $chess\_board$
Output: $is\_fit\_go\_rule$

$is\_fit\_go\_rule \leftarrow True$
$check\_state \leftarrow$ all chess unchecked
for unchecked chess p **do**
  $tmp\_alive \leftarrow$ is_alive( ) //get the chess status
  if $tmp\_alive == is\_not\_alive$ **then**
    return $is\_fit\_go\_rule \leftarrow False$
  end if
end for
return $is\_fit\_go\_rule$

- ***user_step_eat***: Search the possible step for white hand so that any black chess will be eaten.

Algorithm 4 user_step_eat

Input: $chess\_board$
Output: steps to eat eat_step, changed board $changed\_board$

$eat\_step \leftarrow \emptyset$ //eat step offset sets
$changed\_board \leftarrow chess\_board$
$dead\_chess \leftarrow \emptyset$ //dead chess offset set
$vitality \leftarrow 0$ //count the vitality
$check\_state \leftarrow$ all chess unchecked
for black chess $p \in changed\_board$ **and** ($dead\_chess == \emptyset$ **or** $offset \notin dead\_chess$) **do**
  $dead\_chess \leftarrow \emptyset$ //clear the set
  $dead\_chess \leftarrow$ find_chess_block( )
  for $offset \in dead\_chess$ **do**
    for four directions d of $offset$ **do**
      if find none chess **and** direction $d \notin vitality$ **then**
        $vitality \leftarrow vitality \cup d$
      end if
    end for
  end for
  if $vitality.length == 1$ **then**
    $eat\_step \leftarrow eat\_step \cup vitality$
    for $offset \in dead\_chess$ **do**
      $changed\_board[offset] \leftarrow$ none chess
    end for
    $vitality \leftarrow 0$ //clear vitality set
  end if
end for
if $eat\_step \neq \emptyset$ **then**
  for $offset \in eat\_step$ **do**
    $changed\_board[offset] \leftarrow white$
  end for
end if
return eat_step, $changed\_board$

- ***user_step_possible***: Search the possible steps for white hand including eating steps.

Algorithm 5 user_step_possible

Input: $chess\_board$
Output: possible step set $possible\_steps$
$possible\_steps \leftarrow \emptyset$
for $offset \in (i-1,j),(i+1,j),(i,j-1),(i,j+1)$ **do**
  if $chess\_board[offset]$ $==$ none chess **and** $offset \notin points$ **then**
    $chess\_board[offset] \leftarrow white$
  end if
  $is\_fit\_go\_rule \leftarrow$ go_judge( )
  if $is\_fit\_go\_rule == True$ **then**
    $possible\_steps \leftarrow possible\_steps \cup offset$
    $chess\_board[offset] \leftarrow$ none chess
  end if
end for
$eaten\_chess \leftarrow$ user_step_eat() //consider the steps which can eat opponent
$possible\_steps \leftarrow possible\_steps \cup eaten\_chess$
return $possible\_steps$

3. Empirical Verification

Empirical verification is compared with the given ans in the txt file: **answer_for_train.txt**. The results are the same and can be output to file with the same format.

3.1. Design

The project is specified by certain targets. I define another three functions to assist myself meeting the demand of finished the given functions during coding. One must be mentioned, beyond the requirements, to indicate the possible steps for white hand, I add function ***plot_possible_step*** with possible steps marked with red oval on chess board.

3.2. Data and data structure

Data being used in this project is five txt files for test the codes.

The data structures used in this project include stack, list, tuple, and array.

3.3. Performance

Since this is a project with interaction process with programmers, so the performance cannot be simply measured by actual running time. So, theoretic running time, computed by time complexity of functions is about $O(n^4)$, which might not be a good performance if tested on large board such as original chess board with $19*19$, will be a reference.

3.4. Result

See output file **answer_for_train.txt**.

3.5. Analysis

This project is an simple implementation of go game program. If compared with Google AlphaGO which is based on neural networks and Monte Carlo tree with the support of technologies of deep learning and big data, we will know that for oversize computing now we have no enough computing resource and running time, though it might be the lack of efficient discriminate algorithms. Since the algorithms in this project are all discriminate, and they actually do nothing with artificial intelligence, but for most AI programs, how to design the architecture and algorithms are also crucial. All the above is just a simple beginning.

Acknowledgments

I would like to thank Ziqiang Li for exchanged some thoughts with me on some vital algorithms which finally leads me to successfully finished thie project. And I also want to thanks for TA Yao Zhao who clarify my confusion on some representations of data in the given codes. Last I would like to thank forward to all the student assistances who will assess my codes and reports.

References

[1] Wikipedia contributors, [Online]. Available:https://en.wikipedia.org/wiki/Go_(game), [Accessed: 03- Oct- 2017]