Gobang AI Project

Hao Zheng 11610127

Department of Computer Science and Engineering

Southern University of Science and Technology

11610127@mail.sustc.edu.cn

1. Preliminaries

This project is implementing a Gobang AI using traditional decision tree algorithm. Gobang, also called *Five in a Row* or *Gomoku*, is an abstract strategy board game. In Gobang, the player who firstly link five pieces of cheese in a line win the game. Because the chess in Gobang cannot be removed, it can be played using only paper and pen. Thus, it's popular all over the world. This game has been proofed to be "the first player always win" on an empty 15 × 15 board without additional rules if the first player uses good enough searching algorithms [1]. However, it's also a challenge for us to apply AI algorithms to implement it in limited memory and time per step. In this report, I will describe the dormitory algorithms and personal tricks used in this project.

1.1 Software

This project is implemented in Python, invoking `Numpy` as external libraries. All of the coding section is finished using Pycharm. As for the analytics section, I take `pycallgraph` and `time` packages as external tools to visualize the function call relationship and performance bottleneck.

1.2 Algorithm

The most fundamental approach is Minimax algorithm, which is actually another form of depth-first-search. Based on Minimax, I also apply alpha-beta pruning to decrease the number of nodes that would be evaluated by Minimax in its search tree. To reduce searching node further, the heuristic evaluation function is used to sort the generated valid actions, which increases the cost of generating actions section. However, it also dramatically reduces the whole searching cost because I only reserve the head of the sorted actions. As for the evaluation function used in the leaf of the search tree as well as the other tricks in implementing this project, I will explain them together with these basic algorithms in the next section.

2. Methodology

The basic implementation of Minimax and alpha-beta pruning algorithms can be found everywhere, but the evaluation function and performance improvements should be implemented by myself. Limited by memory and time per go, the performance of the algorithm is emphasized extraordinarily. Thus, in this section, I will describe the architecture and detail of the algorithms applied in this project.

2.1 Architecture

There're two classes and one function in python file. One of the class `AI` is the API given by the Gomoku AI VS Platform in SUSTech. The function `alphabeta_search` is used to implement the bone of Minimax and alpha-beta pruning algorithm, which will internally do recursive invocation and finally return the searched step to `AI` as candidate step. The other class is `Game`, containing all the game-specific methods used in Gomoku, which will be invoked in `alphabeta_search`.

Here are all the functions in these three parts.

- `AI`
  - `go` put all the candidate steps searched by `alphabeta_search` to the candidate list for a given chessboard.
- `alphabeta_search`
  - **body:** implement the algorithm used in the first max layer, which invokes the `min_value` for each generated action and `yield` the action once the currently best choice is found.
  - `max_value` the algorithm used in the max layer, which invokes the `min_value` to select the child node that has the highest score, then return that score.
  - `min_value` the algorithm used in min layer, which invokes the `max_value` to select the child node that has the lowest score, then return that score.
- `Game`
  - `__init__` initialize the situations, which is the 'situations' to score dictionary.
  - `actions` get all the valid actions sorted by `heuristic_evaluate` according to the chessboard.
  - `has_neighbor` determine if the given position has neighbors 2 steps around it.
  - `heuristic_evaluate` approximately score the given position in the chessboard.
  - `max_kill_actions` get all the "kill actions" in the max layer, which is the rush 4 and alive 3 actions for AI player.
  - `min_kill_actions` get all the "kill actions" in the min layer, which is the rush 4 and alive 3 actions for both AI player and opponent player.
  - `terminal_test` determine if there's a winner in the given chessboard.
  - `eval_base` get the score according to the given chessboard only, which can be used as base_score in `evaluate`.
  - `evaluate` get the score according to the given chessboard, action, and base_score.
  - `get_lines_score` get the score of the given lines with given player color.
  - `get_lines` get the horizontal, vertical, diagonal, reverse_diagonal of given position

2.2 Detail of Algorithms

To apply less code while explaining the basic idea clearly, the "pseudocode-likely" python code is used in this section.

- `heuristic_evaluate`

  I evaluate the position in two ways. One is the score that the AI color chess is in the position, the other is the opponent color chess in the position. The addition of them is the final score of the position. I consider both of them because the position that benefits opponent also should be taken into account.

  ```
  score = 0
  x, y = position

  state[x, y] = color # state is the chessboard
  lines = get_lines(state, position)
  score += get_lines_score(lines, color)

  state[x, y] = opponent_color
  lines = get_lines(state, position)
  score += get_lines_score(lines, oponent_color)

  return score
  ```

- `actions`

  The heuristic evaluation function cost a lot, but it's precise enough to cut down 2/3 of the tail to reduce the search space. To prevent overdoing this cutting, I set the minimum to 15, which can guarantee the accuracy of chosen actions at the end of chess playing.

  ```
  valid_move = all the move that is 0 in chessboard
  valid_move = [move for move in valid_move if has_neighbor(state, move)]

  sort_key = lambda m: self.heuristic_evaluate(state, m, color)
  valid_move = sorted(valid_move, key=sort_key, reverse=True)

  return valid_move[:max(15, (len(valid_move) // 3) + 1)]
  ```

- `evaluate_base`

  This function is used only at the first max-layer. Then the `evaluate` can use these base_score to calculate the next score.

  ```
  score = 0
  lines = all horizontal, vertical, diagonal, reverse_diagonal lines in chessboard
  for line in lines:
      score += get_lines_score(lines)

  return score
  ```

- `evaluate`

  According to this function:

$$score_{next} = score_{base} - score_{originlines} + score_{currentlines}$$

  The score of lines is the line value of horizontal, vertical, diagonal, and reverse diagonal at the position that would be put chess on. The difference between origin lines and current lines is that there is a chess in that position for current lines while there isn't for origin lines.

  According to my experience, the position that is surrounded by more chess of the same color is relatively more valuable. Thus, the **link coefficient** is introduced to be `0.015 * surrounding number`. Finally, the total score is:

$$score_{finally} = (1 + linkcoe) \times score_{next}$$

```
    x, y = move

    origin_lines = get_lines(state, position)
    origin_score = get_lines_score(origin_lines, color)

    state[x, y] = color
    current_lines = get_lines(state, position)
    current_score = get_lines_score(current_lines, color)

    surrounding_num = number of same color chess surrounding the position (3 x 3 - 1)
    link_coe = 0.015 * surrounding_num

    return (base_score - origin_score + current_score) * (1 + link_coe)
```

- `get_lines_score`

  The situations score (set in `__init__`) is shown below:

```
  self.situations = {
              'win5': 100000,             # 连5
              'alive4': 10000,            # 连4
              'continue-rush4': 900,      # 冲4
              'jump-rush4': 800,          # 跳冲4
              'alive3': 600,              # 活3
              'continue-sleep3': 100,     # 眠3
              'jump-sleep3': 75,          # 眠3
              'special_sleep3': 60,       # 特型眠3
              'fake-alive3': 60,          # 假眠3
              'alive2': 50,               # 活2
              'sleep2': 10,               # 眠2
              'alive1': 10,               # 活1
              'sleep1': 2,                # 眠1
          }
```

  To simplify the scoring section, for each line, I only detect if there is a highest-score situation in a line.

Firstly, the '-1' is changed into '2' for each line, which decreases the cost of detecting situations using string matching algorithm. As for the specific method, I use `if substring in string` in python to make it. In Performance Analytics section, I will compare these string matching algorithm given by python to explain why I choose this one.

In each line, the score of the AI player and opponent player are calculated. To pay more attention to the defensive, I give the opponent part external weight:

$$score_{final} = score_{AI} - 1.1 \times score_{opponent}$$

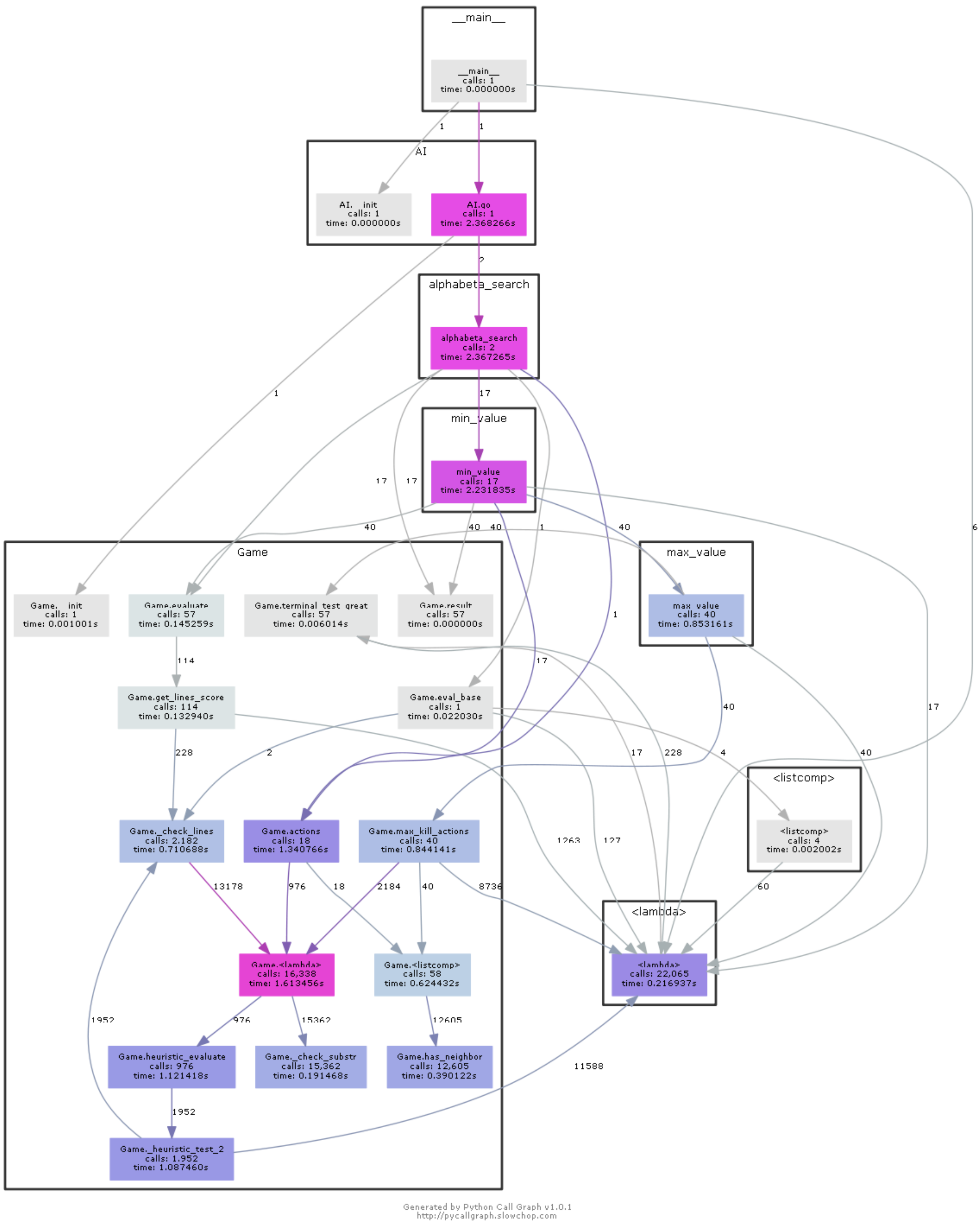
- Check Kill: `max_kill_action` & `min_kill_action`

  After the terminal depth, I use "check kill" to deepen my search depth with acceptably external cost. In these depths, if there's no winner in the chessboard, I will consider the "rush 4" and "alive 3" for AI player in the max layer, and the "rush 4" and "alive 3" for both AI and opponent player in the min layer.

  Because the "kill actions" are much less than the normal actions in the previous searching tree, these external actions actually don't hurt the performance.

3. Performance Analytics

3.1 Call relationship



According to this page, the `actions` take 1.33 s, which is 56.4% time spent in searching. However, if I fetch the actions randomly, the total time would be increased up to 5 s for the same chessboard. Thus, it's necessary to spend this actions time.

3.2 String matching

The test bench code:

```python
from time import time


string = '001121021111010'
substr = '211110'

test_range = range(1000000)
result = None

tic = time()
for x in test_range:
    result = substr in string
print('in:', time() - tic, 's')

tic = time()
for x in test_range:
    result = string.__contains__(substr)
print('contains:', time() - tic, 's')

tic = time()
for x in test_range:
    result = string.find(substr)
print('find:', time() - tic, 's')
```

result:

- `in` : 0.1446690559387207 s
- `contains` : 0.22919988632202148 s
- `find` : 0.28012681007385254 s

Acknowledgments

I would thank Yijing Huang for exchanging ideas among the performance optimization and score strategy. Also, the `AI VS Platform` helps a lot for determining the performance of Gobang my AI. The development team and all the students who spend time improving the platform is highly appreciated.

References

1. L. Victor Allis (1994). *Searching for Solutions in Games and Artificial Intelligence* (PDF). Ph.D. thesis, University of Limburg, The Netherlands. p. 124. ISBN 90-900748-8-0.