CS303 Project1: Gomoku AI

Shijie Chen
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, Guangdong, China
Email: 11612028@mail.sustc.edu.cn

***Abstract*—Gomoku, also called five-in-a-row, is an abstract strategy board game. In this project, the author implemented an AI player of Gomoku that can play with either human or other AI players. This report describes the design and implementation of the AI player. It uses Minimax algorithm to search for best solutions each step. Due to the limited computational resources, the program is accelerated by α-β pruning and proper path-selection. Currently, the program is able to search 4 steps with 5 possible locations each step in 5 seconds.**

I. Introduction

The development of a Gomoku AI player includes understanding the rules of Gomoku, design and implement search algorithm, optimization and validation.

Gomoku, despite its simple rules, is very flexible. Brute force search algorithm requires too much computational power and is therefore abandoned. Techniques like Monte-Carlo Search and Neural Networks requires knowledge in Machine Learning, training data and some implementation skills.

This projects uses Minimax algorithm, which is simple and powerful, as the main search algorithm. Further optimization is done by adding α-β pruning and proper path-selection strategy.

II. Preliminaries

This project is developed in Python 3.6.5 under Ubuntu 18.04 ($Windows\ Subsystem\ for\ Linux$). Libraries used in this project includes Copy

A. Gomoku Rules

Typically, Gomoku is played on an 15x15 grid board like one that is used in Go games. Two players place stones of different colors (black/white) on empty intersections of the grid alternatively. A player wins if he obtains a consecutive 5 stones in any one of vertical, horizontal or the two diagonal directions.

B. GOMOKU Board Representation

The board is represented by a two-dimensional array as shown in figure 1. As in Python, its a *list of lists*. For each entry, 0 represents empty intersection, 1 represents a black stone and -1 represents a white stone. To make pattern matching easier, such matrix is converted to a matrix of 'a', 'b' and 'c' in which 'a' represents black stones, 'b' represents white stones and 'c' represents empty intersections.

TABLE I
Representation of Gomoku board

Name	Variable	Board	Pattern
Black Stone	COLOR_BLACK	-1	a
White Stone	COlOR_WHITE	1	b
Empty Intersection	COLOR_NONE	0	c

III. Method

The design and implementation of a Gomoku player involves several parts. Initially, design evaluation functions for both the global chessboard and a single blank position on the board that shows the grade, chance of winning, of the board or point. Then, implement Minimax algorithm using the previously designed evaluation functions.

However, Minimax algorithms explores a search tree whose size expands exponentially with search depth. Since search depth determines the "skill level" of the AI player, α-β pruning and proper path-selection are used to limit the size of search space in each step and increase the depth of search tree.

A. Evaluation Function for the Whole Board

This function scans the chessboard in four directions , do pattern-matching and assign scores for both colors according to the patterns.

1) Pattern Matching The evaluation involves multiple pattern matching. I use *AC automata* [1] in this project to match multiple patterns efficiently.
2) Pattern and Scores
Scores of more *dangerous* patterns must *dominates* scores of less *dangerous* ones. Table IV shows a score board for patterns in black('a').

TABLE II
Score board for global patterns

Pattern	Score
aaaaa	1000000000000
caaaac	3000000000
caaaab	1000000
baaaac	1000000
caaac	1000000

Algorithm 1 Global Evaluation

$AC \leftarrow new\ ACautomata()$
$AC.loadKeys(GlobalScoreBoard.keys$
$slices \leftarrow genStr(chessboard)$
for all $x \in slices$ **do**
  $patternCount.add(AC.match(x))$
end for
for all $x \in patternCount.keys$ **do**
  if $x \in PlayerPattern$ **then**
    $playerScore \leftarrow playerScore + GlobalScoreBoard.get(x) * patternCount.get(x)$
  else
    $opponentScore \leftarrow opponentScore + GlobalScoreBoard.get(x) * patternCount.get(x)$
  end if
end for
$Score \leftarrow playerScore - opponentScore$

B. Evaluation Function for Single Position

Similarly, we develop a score board for patterns that may appear when an empty position is assigned a stone of a color. The score is the sum of *score* it may get when inserting a stone in the player's color and the *danger* it may cause when inserting a stone in the opponent's color.

The score of a single point is used to sort the points in search set in *proper path-selection* Score Board for *score* and *danger*

TABLE III
SCORE VALUE FOR SINGLE POSITION

Pattern	Score
aaaaa	1000000000000
caaaac	3000000000
caaaab	410000000
baaaac	410000000
acaaaa	410000000
aacaa	410000000
aaaca	410000000
caaacc	1000000
ccaaac	1000000
acaca	1000000
caacac	1000000
cacaac	1000000
ccaacc	1500
cacacc	1000
ccacac	1000
cccacc	10
ccaccc	10

TABLE IV
DANGER VALUE FOR SINGLE POSITION

Pattern	Score
bbbbb	900000000000
cbbbbc	2100000000

For a single position *pos*, get the strings that contains *pos* in four directions. Then apply pattern matching to decide its score and danger value. Final score is the sum of score and danger value.

Algorithm 2 Single Position Evaluation

$AC \leftarrow new\ ACautomata()$
$AC.loadKeys(SingleScoreBoard.keys)$
$AC.loadKeys(SingleDangerBoard.keys)$
$scoreSlices \leftarrow getStr(chessboard, pos, color)$
$dangerSlices \leftarrow getStr(chessboard, pos, opponentColor)$
for all $x \in scoreSlices \bigcup dangerSlices$ **do**
  $patternCount.add(AC.match(x))$
end for
for all $x \in patternCount.keys$ **do**
  if $x \in PlayerPattern$ **then**
    $score \leftarrow score + SingleScoreBoard.get(x) * patternCount.get(x)$
  else
    $danger \leftarrow danger + SingleDangerBoard.get(x) * patternCount.get(x)$
  end if
end for
$Score \leftarrow score + danger$

C. Proper Path Selection

This step will generate a $SearchSet$ for each non-leaf node of the search tree. The $SearchSet$ contains positions with at least 1 non-empty neighbor.

To limit the size of $SearchSet$, the program will sort the positions according to their single-point evaluation and pick 5 positions. This path-selection increases depth of search tree from 2 to 4 within time limit of 5s.

Algorithm 3 Proper Path Selection

function GENSET(chessboard)
  $SearchSet \leftarrow \emptyset$
  $set \leftarrow \emptyset$
  for all $empty\ pos \in chessboard$ **do**
    if pos has non-empty neighbor **then**
      $set.add(pos)$
    end if
  end for
  $set.sort(SinglePositionEvaluation())$
  $SearchSet \leftarrow set[0:5]$ **return** SearchSet
end function

D. α-β Pruning

In order to increase the depth of the search tree in Minimax algorithm, we must limit the size of search space in each level. α-β pruning works by skipping obviously un-optimal points while searching. Since Max Player will choose the step with score greater than current max (α), steps with score less than current max can be skipped when the Min Player plays(Next level is Max Player's turn). Therefore the step is no longer expanded. Same thing happens to when the Max Player plays.

E. The Minimax Search Algorithm

Algorithm 4 Minimax Search Algorithm (with $\alpha - \beta pruning$)

```
function MINIMAX(chessboard, depth)
    α ← ∞
    β ← −∞
    SearchSet = genSet(chessboard)
    Max ← ∞
    Solutions ← ∅
    for all pos ∈ SearchSet do
        subBoard ← chessboard.setStone(pos)
        score ← minPlayer(subBoard, depth − 1, α, β)
        if score = Max then
            Solutions.add(pos)
        else if score > Max then
            Max ← score
            Solutions.clear()
        end if
    end for
end function
function MINPLAYER(chessboard, depth, α, β)
    score = getGlobalScore(chessboard)
    if depth = 0 or score ≥ winningScore then
      return score
    end if
    SearchSet = genSet(chessboard)
    value ← ∞
    for all pos ∈ SearchSet do
        subBoard ← chessboard.setStone(pos)
        value ← min(value, maxPlayer(subBoard,
depth − 1, α, β))
        β ← min(β, value)
        if α ≥ β then
            break
        end if
    end for
      return value
end function
function MAXPLAYER(chessboard, depth, α, β)
    score = getGlobalScore(chessboard)
    if depth = 0 or score ≥ winningScore then
      return score
    end if
    SearchSet = genSet(chessboard)
    value ← ∞
    for all pos ∈ SearchSet do
        subBoard ← chessboard.setStone(pos)
        value ← max(Max, minPlayer(subBoard,
depth − 1))
        α ← max(α, value)
        if α ≥ β then
            break
        end if
    end for
      return value
end function
```

- The pseudocode below has included $\alpha - \beta pruning$
- $chessboard.setStone(pos)$ sets pos as color of AI player in $maxPlayer$, vice versa.

Minimax Algorithm is used in this project as a main search algorithm. The idea is based on adversary search. Since the global evaluation function shows the extent to which the situation is good for the AI player, the AI player will try to maximize the evaluation score. On the contrary, the opponent will try to minimize the score. Minimax algorithm works by simulating this process. It uses strategy of the Max Player (the AI) and the Min Player (the opponent) alternatively on each level of search and finally obtain an optimal solution. [2]

More details of this algorithm can be found on the Internet.

IV. Validation

The player is tested with test cases provided by CS303 Artificial Intelligence course as well as some scenarios encountered during games with other AIs and human players.

V. Discussion

This project can be improved in several aspects.

- Evaluation Function V.S. Search Depth
  In practice, due to time limit, programs with better evaluation functions may outperform programs with simpler evaluation functions but deeper search trees.
  A *better* evaluation function takes into account dangerous situations and is able to construct traps for the opponent. However, such ability can be achieved with a search tree with depth of more than 8(i.e. 4 steps ahead). It's quite challenging to implement an AI that explores a 8-level search tree within 5 seconds.
- SearchSet Generation
  This step is designed to limit size of SearchSet. Currently it just includes empty positions with non-empty neighbors.
  Also, the order of subBoards has great impact on the efficiency of $\alpha - \beta$ *pruning* Improvement can be made in 2 ways:
  1) consider a broader range of empty positions.
  2) Improve single position evaluation function, so that optimal position is included.
- Improve Single Position Evaluation
  Due to the limited time and complexity of single position evaluation, I didn't employ AC automata in it. Otherwise the performance should be noticeably improved. Also, more dangerous situations can be considered and constructed.

VI. Conclusion

In project, I implemented an AI player of Gomoku. Minimax algorithm as well as $\alpha - \beta$ *pruning* is used. The design process gave me a deeper understanding of Gomoku and showed me the magic of artificial intelligence.

Acknowledgment

The authors would like to thank the TAs for their hint in Lab and maintaining a online Gomoku platform.

REFERENCES

[1] "Implement ac automata in python3." https://www.ctolib.com/topics-106266.html. Accessed October 25, 2018.

[2] Wikipedia contributors, "Minimax," 2018. [Online; accessed 29-October-2018].