A Go Game Solver for 5x5 Board

Roujia Wen

1. Problem Definition

The problem I am tackling in this project is: For a given a state of a 5x5 Go game, what is the best move for a given player?

More specifically, I am focusing on late stage game states. There are a few reasons for this choice: (1) There are too many possibilities for an early stage game and therefore systematic search algorithms will not terminate in a reasonable amount of time. (2) A late stage game is more interesting for Go learners because they have developed patterns that human players usually recognize, and thus this program could serve as a teaching tool that provides strategic inspirations.

The basic rules of Go can be summarized into the following:

Rule 1 *Two players, Black and White, take turns to place a stone on a grid vertex of the game board. Unless captured, stones cannot change locations once they are placed.*

Rule 2 *Adjacency is only defined along vertical and horizontal lines.*

Rule 3 *A chain is either a single stone, or a group of adjacent stones.*

Rule 4 *The number of liberties of a chain is equal to the number of empty points adjacent to the chain.*

Rule 5 *When a chain has no liberty, it is captured and removed from the board.*

Rule 6 *When a stone is placed so that two chains from each side simultaneously lose all liberties, the passive side gets captured and the action side gets to keep its stones.*

Rule 7 *A stone cannot be placed at a spot if it will not have any liberty. That's called a suicide.*

Rule 8 *A player cannot make a move that returns the game to any previous position (the superko rule).*

Rule 9 *Scoring rule: A player's score is its number of stones on the board plus the enclosed areas.*

It is widely known that this game is extremely complex (at least PSPACE-hard under the superko rule). Besides, the seemingly simple set of rules of this game can generate many levels of implications and therefore heuristics, some of which are perfect ("logical entailments" are probably a better way to call them), and some are only approximate. That is why I find it a fascinating problem for AI, especially logic and search. For example, an implied rule derived from the basic rules above is that a group of stones that belongs to one player can survive if it has at least two "eyes"[1] – a term for an empty point surrounded surrounded by one or more chains which all have at least two liberties (See Figure 1). So now the decision problem for designing AI is whether to include this expert knowledge explicitly into the system, or letting the system infer the rule by itself. Choosing the former can reduce search steps but also bring extra cost of detecting those structures and make the system less "elegant." The tradeoff becomes more obvious when certain heuristics are not perfect – they give optimal solution most of the time, but in certain situations might lead the system to a local optimum.

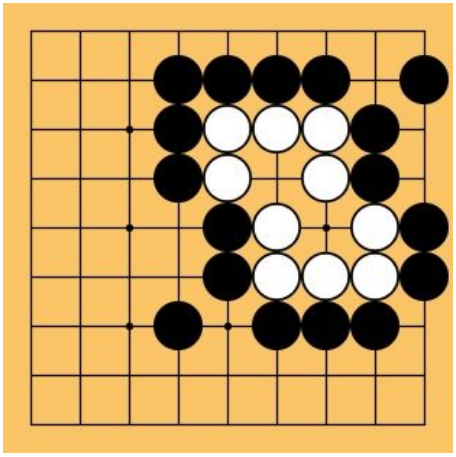

Figure 1 - A state of the game where White has two eyes and therefore stable. Black cannot put its stones into any of those eyes because that will be a suicide (Rule 7).

2. Solution Specification

2.1. Board Representation

There are three classes of objects:

Stone	Attributes color: black or white loc: a tuple indicating the location chain: a Chain object specifying which chain it belongs to.
Chain	Attributes

[1] More accurate way of putting it is that either it currently has at least two eyes, or it will have when challenged by a rational adversary.

	color: black or white stones: a list of Stone objects specifying the stones it consists of. liberty_set: a set of tuples indicating the locations of its liberties. liberty: the length of liberty_set (number of liberties).
Board	Attributes chains: a list of Chain objects, representing all chains that are currently on the board. stable_areas: a dictionary in which the keys are Black and White and values are 2D lists of 1 or 0 indicating whether the location has been secured for the player. history: a set of hashed board states which have been previous states of the current game. last_move: a tuple indicating location of the last move. Methods (omitting helper methods) put: put down a stone of given color at given location, if it's a legal move. is_nonstupid: determine whether a move is not stupid, therefore whether it should be included in the search. is_terminal: determine whether current board is at a terminal state. score: returns the difference between the score of Black and the score of White.

2.2. Rule Representation

Rule 1: This is enforced by the architecture of adversarial search (see next section).

Rule 2: This is enforced by defining neighbors as locations generated by adding (+1,0), (-1,0), (0,+1), or (0,-1) to current coordinate (x, y).

Rule 3: Each time when `put` is called, a Stone object will be initiated with its own chain. It then merges with any neighboring chains.

Rule 4: When a new Chain is instantiated (with only one stone), its empty neighbors are put into `liberty_set`. When chains are merged, the new `liberty_set` is the union of the original ones.

Rule 5, 6, 7: When `put` is called, the following actions are taken. (1) Perform chain merging with neighboring stones of the same color. (2) Reduce liberties of neighboring chains of the opposite color. Check if any of those are killed. If so, remove them. (3) If no enemies are killed, check if it's a suicide. If so, reject the move.

Rule 8: To determine whether a move is legal, the board state resulting from the move is checked against `history`.

Rule 9: Due to the definition of terminal states (see next section), this rule has been simplified. When a location is empty, it's either an eye of Black or White, and therefore it is sufficient to only check its neighbors, rather than search for a surrounding border.

2.3. Adversarial Search with Alpha-Beta Pruning

This project uses the exact alpha-beta minimax algorithm specified in Russell & Norvig (1995), but adopts an iterative rather than recursive structure to circumvent maximum recursion threshold.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2 - The alpha-beta search algorithm (Russell & Norvig, 1995).

Criteria for a terminal state is specified in `is_terminal`. When no non-stupid move exists for both Black and White, the board is at its terminal state. However, this is only the lowest level criteria. An additional heuristic that has been included in the algorithm is that, when one player has an overwhelming amount of stable areas so that it becomes impossible for the other player to establish any territory (minimum requirement for that is an area of eight), the game terminates.

Utility is acquired through the `score` method already discussed.

Finally, the actions function generate all possible actions given a color and a Board object. For each empty location on the board, if it passes the `is_nonstupid` test, it will be included as a possible action. There are two ways to fail the `is_nonstupid` test: (1) The move is illegal. (2) The move involves filling an eye of oneself. The second scenario is a suboptimal in almost all possible cases[2] and therefore eliminated.

2.4. Iterative Deepening

In many problem instances, the search tree branches out exponentially and solutions cannot be found in a reasonable amount of time. Therefore, an iterative deepening algorithm is wrapped outside the adversarial search. The search starts with an initial `DEPTH_LIMIT`. It increases search depth by 1 each time, until either (1) search has been completed within the depth limit, or (2) number of search steps have exceeded `MAX_STEP` during that iteration.

If the algorithm cannot finish a complete search within reasonable amount of steps, it will return an approximate solution, which is the best move if we only consider games which terminate within `DEPTH_LIMIT` amount of steps (here "step" means a step in the game, not in the search algorithm).

3. Analysis of Test Cases and Solutions

3.1. Mostly Stable Configuration

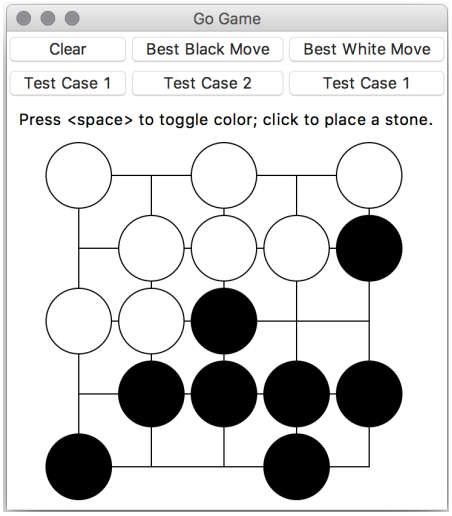

Figure 3 - Board configuration for test case 1.

[2] It turns out that there are a few scenarios in which it can be a better move (Holigor, 2009). However, to ensure the stability of the end game and prevent the search tree from exploding, this is a tradeoff to be made.

The above board state (Figure 3) is an example of a relatively stable game configuration, which means that no big blocks of Black or White can be captured. The main areas of potential gains are the top right and bottom left corners. The AI gives the optimal solution even though the search has been cutoff at a depth limit of 9.

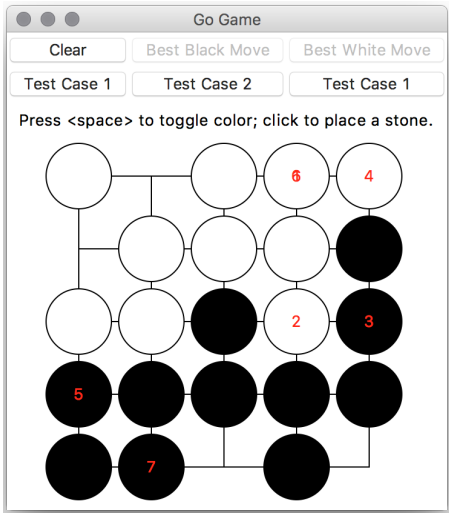

Figure 4 - Solution of the game when Black moves first.

3.2. Critical Point Configuration

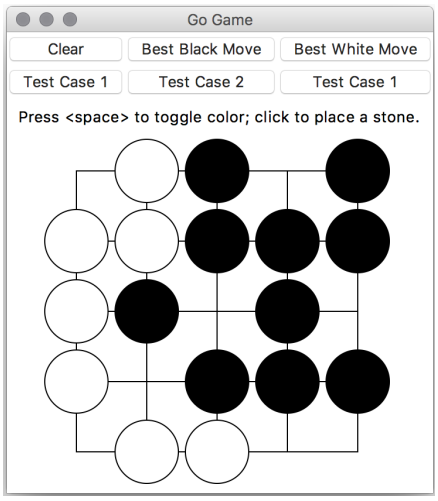

Figure 5 - Board configuration for test case 2.

This is an example of a game in which a critical point in the next move can determine the life and death of White. If White places a stone at (3, 1)[3], the whole block of stones survived. Otherwise Black takes all the territory on the board. The AI is able to solve this game, thanks to the pruning provided by

[3] Index starts at 0; row number first.

the heuristic in `is_terminal`. When Black establishes enough stable territories, White has no chance of winning and therefore the game terminates.

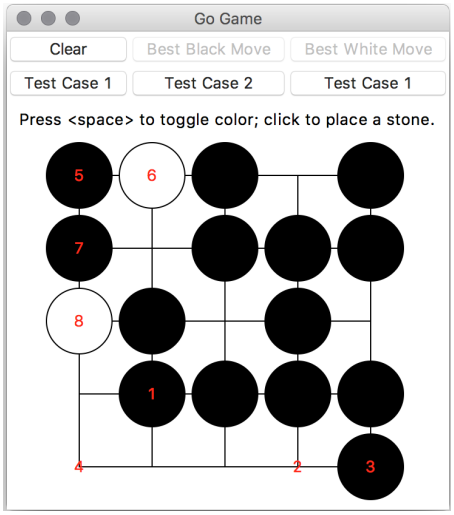

Figure 6 - Solution of the game when Black moves first. The AI is able to find the critical point and capture White.

3.4. Unstable Critical Point Configuration

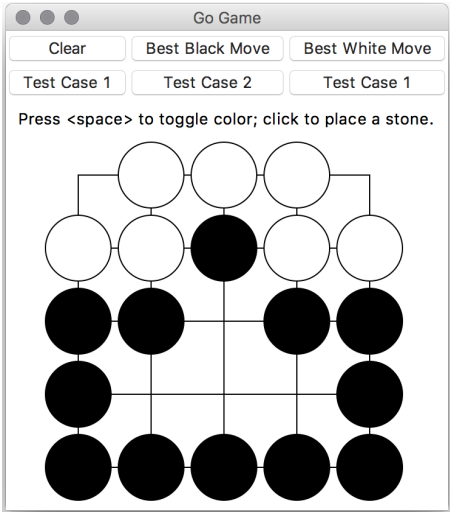

Figure 7 - Board configuration for test case 3.

This is a game configuration in which a large block of stones can be captured and therefore unpredictable in its dynamics afterwards. In this test case, the AI does not give the optimal solution because the length of the path to the optimal solution is larger than the `DEPTH_LIMIT` when the search stops. When White moves first, the optimal solution should be (3, 2), which will lead to the capture of all Black stones. However, the AI took a conservative route (see Figure 8).

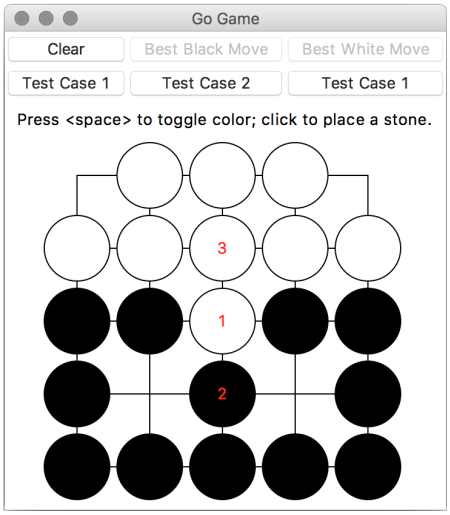

Figure 8 - Suboptimal solution of the game when White moves first. Black remained uncaptured.

In all test cases, the performance and behavior of the AI is consistent with my intuitions and predictions. It is able to find an optimal solution when the possible changes in the game is local (e.g., when no huge chunks of stones can be taken out from the board). It has limited capability when the number of possible states of a game gets too large and cannot be easily pruned.

References

Russell, S., Norvig, P., & Intelligence, A. (1995). A modern approach. Artificial Intelligence. *Prentice-Hall, Egnlewood Cliffs, 25, 27.*

Holigor. (2009). Holigor fills his own eye. *Sensei's Library*. Retrieved from https://senseis.xmp.net/?HoligorFillsHisOwnEye