Tsumego Go - Life or Death

Leo Wong - wongleo7

Shayne Lin- linyihsi

**Roles:** The roles were split fairly evenly, as Shayne knew more about the game of Go (Leo has never really played Go), he encoded much of the game and rules, such as listing legal moves, encoding terminal states, successor states, and similar game mechanics. Leo looked into the alpha-beta search tree and the complementary class to get the go game working with the skeleton code we found. Both of us implemented one heuristics, which we combined at the end. Testing, analysis and find write-up was also split evenly.

**Type of Project:** Game Tree Search

**Project Motivation**

In this project, we are trying to solve the life and death problems (called Tsumego) that are often encountered in the game of Go. We created an application with predefined Tsumego problems where the AI will try to solve a problem by interacting with you. To solve this type of problems, we used Game Tree Search as the game of Go naturally fits all the requirements of a game tree (zero-sum, finite states, deterministic, perfect information, two-player, discrete values). Two heuristics were also implemented to improve the efficiency of pruning. The program was based off a piece of code that implements the alpha-beta pruning algorithm. The game itself, including problem formulation, state encoding and heuristics were developed by the team without using any reference documents or code.

**Methods**

Problem formulation:

Definitions:

1. Liberty: The number of liberties of an isolated is the number of adjacent positions (4-point connectivity) that are unoccupied. If a group of stones belonging to the same player are connected, all the pieces in the group share the same number of liberties, which is the union of all liberties of each individual stone. An example is shown in Fig. 1. At any point during the game, if the number of liberties of a stone (or group of stones) is reduced to 0, all stones within a group are “captured” and must be removed from the board. Multiple groups of stones may be removed as a result of one move.
2. Eye:An eye refers to an unoccupied position (or a group of connected unoccupied regions) in which all the liberties of the position(s) are occupied by the opponent. Eyes are real, false or undetermined depending on the number of diagonally adjacent positions (DAPs) that are occupied by the opponents. A real eye requires three or more DAPs occupied by its allies or belong to another eye. A false eye is an eye where at least two DAPs are occupied by the opponent. All other eyes are undetermined. For eyes of size greater than 1, an eye is real if all positions of the eye are real and is false if any position within an eye is false (see Fig. 2)

|  |  |
| --- | --- |
|  | Screenshot_20161207-213621.png |
| Figure 1. Liberties of a group labelled as triangles | Figure 2. Examples of real (left), false (middle) and undetermined (right) eye |

The Tsumego problem is defined as follows (assumptions are also stated):

* A Tsumego problem starts with a number of stones of one player (defender) surrounded by the opponent's stones (attacker).
* The players alternate to make a move by placing a stone at one of the legal positions on the board. A legal position is an unoccupied position that satisfies one of the following criteria:
  + The stone has at least 1 liberty right after it is played.
  + The stone is connected to other stones of the same team such that the resulting group contains at least one liberty.
  + The stone is placed in a position that results in capturing of at least one of the opponent’s pieces.
* An attempt is considered successful if the defender can form two real eyes and the pieces that make up each eye are connected. This it to make sure that the defender contains at least two liberties and the opponent cannot play in any way to capture these stones.
* The game ends if one of the following situations is encountered:
  + All moves within the boundary are exhausted and the defender fails to form two real eyes (unsuccessful)
  + All defending stones are captured by the attacking stones (unsuccessful)
  + Two connected real eyes are formed by the defender. Real eyes are required since unknown or false eyes can be destroyed. (successful)
  + The cut-off depth (set by user) is reached without forming two eyes (unsuccessful)
* Places where both players can place stones are pre-defined when problems are encoded.
* Since our solver deals with survival problems, it is the defender and always plays first.
* The program does not account for ‘seki’ or ‘ko’ situations. All test examples used is guaranteed to have a solution where the defender can form two real eyes. The ‘ko’ rule is also not enforced and its implication will be seen in the results section.
* All survival states are treated equally since the goal is simply to survive.

State encoding:

Each problem is encoded as a Board object containing the following attributes:

1. ‘Att’, ‘Def’: List of tuples containing x,y coordinates of all attacker and defender pieces.
2. ‘Avail: List of unoccupied positions within the defined boundary.
   1. Usage: generate successor moves
3. ‘All’: List of unoccupied positions in the smallest rectangular board containing all pieces in the problem. It is a superset of ‘Avail’ since it contains unoccupied positions both inside and outside the trapped area. ‘All’ is used to calculate liberties of stones on the board.
4. ‘EyeSpots’: All possible positions that an eye can be formed within the defined boundary. Eyespots contain both occupied and empty positions since some stones might be captured turn into new eyes during the game. It is used to search for eyes in a given state.

|  |  |
| --- | --- |
|  |  |
| Figure 3. Example of ‘Att’ (square), ‘Def’ (triangle), ‘Avail’ (diamond) | Figure 4. Example of ‘All’ (diamond) and ‘EyeSpots’ (triangle) |

Each state during the game is represented by a GoGame object, which contains the following attributes:

1. Player: the player who will be playing next given the current state
2. Moves: all moves in Board[‘Avail’] that are legal moves
3. Board: explained above
4. ConnectedPieces: Groups of attacker/defender pieces that are adjacent to each other. Each connected group includes a list of all liberty spots belonging to that group.
5. Eyes: List of eyes in the current state. Each eye contains the list of positions within the eye as well as the type of eye, which can be ‘Real’, ‘Unknown’, or ‘False’.

Successor generation:

Successors are created using the GenerateSuccessor() function and performs the following:

1. Update the board variable by adding the new move on the board and remove any captured pieces if necessary. ‘Avail’ and ‘All’ are also updated.
2. Update on the newly connected pieces and any changes in liberties as a result of the move.
3. Change the player from ‘Att’ to ‘Def’ or vice versa.
4. Use list of positions in ‘Avail’ to determine possible next moves. The CheckLegal() function is called to exclude any illegal moves.
5. Determine all the eyes on the board as a result of the move.

Algorithms:

Three algorithms that built up the skeleton of the game is discussed.

1. Determine connected stones (Input: list of all stones from the same player)
   * Step 1: For each stone, if the stone is adjacent to an existing group, then add to that group. Else, create a new group.
   * Step 2: Depending on how the list is ordered, some groups might still be separated. Go through each group and merge neighboring groups until all groups are separate from each other.
2. Calculate liberties (Input: a group of connected stones stored as a list)
   * Step 1: For each stone in the list, find the 4 adjacent positions. Add the adjacent position to the liberty list if it is not occupied and does not exist in the list yet.
   * This algorithm works for groups of any sizes and shapes.
3. Find eyes (Input: list of unoccupied positions and all attacker pieces occupying an eye spot)
   * Step 1: A group of unoccupied space that are connected belong to the same eye, so use algorithm #1 again to generate “connected eye spots” from the input. Input includes attacker stones so that trapped stones in an eye do not separate an eye into two.
   * Step 2: For each connected space group, classify each group as an eye if all its adjacent positions are occupied by the defender.
   * Step 3: Ignore all attacker pieces trapped in eyes (not actually removed)
   * Step 4: Identify the eye type based on the definition in problem formulation.

Heuristics

Two heuristics were developed to improve alpha-beta pruning. The first heuristic, vital spot heuristic, was based on a strategy that looks for moves with the highest potential to form eyes (see reference link for details). It first looks through all the eye spots that are one or two moves away from forming an eye and record the positions that need to be occupied to form those eyes. The moves are then sorted by a weighted sum of how frequent they appear on the “one move” and “two moves” list. Different weights are used because forming an eye in one move is more crucial than forming an eye in two moves.The second heuristic was based on the number of eyes each state has. Different weights were assigned to each type of eye to reflect their relative importance. The score for a given state is given in eqn.1.

Our final version was a two stage approach that incorporates both heuristics. Specifically, we used the vital spot heuristic in the beginning of the game to occupy the crucial spots that has the highest chance of making new eyes. However, when there are no more places to make new eyes, we shift to the second heuristic to ensure that incomplete eyes are formed and secure (e.g. turning from unknown into real eyes). This dynamic approach provides a good heuristics throughout the entire search by employing the stronger heuristic at different stages of the game. In order to maximize pruning, alpha always chooses the move with the highest heuristic score while beta chooses the move with the lowest score.

Alpha-beta search:

Since we were only concerned with finding a path that will guarantee to survive, our alpha and beta values were 1 or 0 (live or die) and we exit the search when alpha becomes 1 at the root node. There may be other paths to the solution in other parts of the tree, but it will not make a difference since all survival states are treated equally. To make the search even more efficient, we applied the vital spot heuristics at the top level as well to minimize the number of nodes that will be visited for the first move. This has a significant impact on efficiency since nodes are being pruned at the root. Another change that was made was to decrease the search depth by 2 every time both players make one move. This method is not guaranteed to work since some solutions that require longer paths may not have been visited due to pruning. However, since our heuristics usually make beta return the strongest moves (usually the longest path to reach survival state), the nodes that are visited often correspond to the max depth to the entire solution space where alpha makes rational moves. Therefore, the subsequent moves can most likely be solved with depth d-2 in all cases. This method can greatly reduce the time and space complexity towards the end of the problem.

**Evaluations and Results:**

Problems of varying breadths and depths were used to evaluate the capability of our solver. Breadth referred to the number of possible moves in a given state while depth referred to the number of steps required to reach the solution. For each problem, the cut-off depth was initially set to be the number of steps required in the solution provided by the apps. However, it was later realized that the depth should be set higher because some survival states may require more steps depending on the moves beta chooses. The solution given by the app was merely one of the many possible ways beta could respond. Therefore, the depth was incremented manually until a solution was found (similar to IDS). In the end, 15 problems were tested and the depths ranged from 5 to 10 steps.

Our final solver was evaluated by two measures: accuracy and heuristic performance. Accuracy was determined by the percentage of time our program solves the problem correctly. To determine the correctness of a solution, we entered the moves that our solver played in the app and played the corresponding move the app returned back into the program. A solution was deemed correct if the solver could play the game until the app indicated that the problem was solved correctly. The effectiveness of our heuristics was measured by solving the same problems with combined heuristics and without heuristics. In both cases, the same cut-off depth and the depth decrement strategy was used. The performance was quantified by comparing nodes visited as well as the search time for the first move only since the search tree is the biggest compared to subsequent moves. Out of the 15 problems tested, 14 were successfully solved using combined heuristics and 13 were solved without heuristics. The search time with and without heuristics is shown in Fig 5. Overall, the total search time was reduced by 5.8 times (163 vs. 949 seconds in all problems combined). The total number of nodes visited was also significantly reduced (Fig. 6). It may seem weird that the percentage of nodes pruned is higher when heuristics are not used. However, with heuristic present, the nodes that were visited were states where both alpha and beta play strong moves, so the heuristics may not be perfect. In the case without heuristics, lots of nodes are pruned in trivial cases where many moves can lead to survive state, especially toward the bottom of the tree. Therefore, the chances of finding a survival state early and pruning the rest of the moves is higher.

|  |  |
| --- | --- |
|  |  |
| Figure 5. Search time comparison between cases where heuristics were used and those that were not used. | Figure 6. Nodes visited in cases where heuristics were used and those that were not used. |

One of the problems that could not be solved is shown in Fig. 7.1. In this case, the white stone at (0,1) was identified as a trapped stone and a real eye with 4 spots ((1,0), (0,0), (0,1), (0,2)) was identified. Therefore, the solver made the move (the black stone labelled with a circle) to complete the second real eye. However, the attacker could actually play at the red dot to start a ‘ko’. Since our solver does not account for ‘ko’ situation, it simply removes the stone and reaches the survival state with three eyes (Fig. 7.2). The correct solution is shown in Fig. 7.3 where the defender can survive without a encountering a ‘ko’, which is often preferred since ‘ko’ is not a guaranteed win in the real game. It is also worth mentioning that the correct first move was not predicted by the vital spot heuristics because it was quite an unintuitive move. However, if the correct first move is given, the solver could indeed produce the right sequence.

|  |  |  |
| --- | --- | --- |
| Screenshot_20161209-143915.png | Screenshot_20161209-172807.png |  |
| Figure 7. Problem that the solver failed to produce the right answer. | | |

There was another problem that could be solved only when heuristics were used, so we ran the solver without heuristics again and increased the depth from 8 to 10, but no solution was found. Depth beyond 11 took too long to search so we did not wait for the search to complete. From the observation, we realized that the second heuristic was also driving the moves towards the minimum number of steps to reach the solution. During the search, there were plenty of occasions where the attacker plays a less desirable move such that the defender is not forced to make the final move to survive. In that case, the defender can play any moves until the attacker plays the forcing move. However, the eye heuristic awards any moves that progress towards the survival state at any point during the search. Therefore, the heuristic will converge to the solution state more rapidly.

Optimality of Heuristics

The vital spot heuristic attempts to make the best move in the beginning of the problem. While it is a reasonable approach, it is not always optimal since the defender often has to save stones with small liberties left and gets distracted from forming eyes. The combination of liberties and eye formation is what makes the best move difficult to judge and sometimes unintuitive. On the other hand, our second heuristic is almost always optimal due to the high weight assigned to real eyes. A survival state with two real eyes is guaranteed to have a minimum score of 40. The only other way the score can be 40 (with three eyes only) is if there is one real eye and two unknown eyes, in which case there is a high chance that one of the unknown eyes can turn into the second real eye. Situations where 4 eyes are present are rare. Therefore, the survival state will almost always receive the highest priority.

**Limitations and obstacles:**

* We did not enforce the ‘ko’ rule and did not account for ‘ko’ and ‘seki’ states. Therefore, we could only solve problems in which two real eyes are formed and when ‘ko’ was not encountered in any intermediate steps.
* We confirmed that combined heuristics worked better than individual heuristics, but the results were not officially documented.
* The cut-off depth was set manually to save search time. Ideally, an iterative deepening search should be used to find the solution systematically.
* Obstacles: It was quite challenging to simply make the game work. We did not find any relevant code on go that suits our purpose, but we took the challenge anyways and it was rewarding to see that our program actually works. The hardest part of the project was to determine the eyes since they appear in multiple configurations. There were a lot of trial and errors involved to deal with special cases. However, once it was implemented correctly, It could theoretically solve any Tsumego problems regardless of the size.

**Conclusion**

In summary, we implemented a Tsumego solver that can solve non-trivial questions that cannot be solved at a first glance. However, there are still several improvement that can be made in the future:

* Check for dead state to avoid playing until a terminal node is reached.
* Implement ‘ko’ rule and ‘seki’ state so that the solver can tackle any Tsumego problems.
* Make the state generation process more efficient. For example, the process of counting liberties can be simplified by only updating the liberties of groups that are affected by the new move instead of recalculating the liberties for the entire board.

**References**

<http://teaching.csse.uwa.edu.au/units/CITS3001/resources/gameAI.py>

<http://teaching.csse.uwa.edu.au/units/CITS3001/resources/utils.py>

<http://senseis.xmp.net/?ApproachInTsumego#toc6>

<https://play.google.com/store/apps/details?id=net.lrstudios.android.tsumego_workshop&hl=en>