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

The game we are trying to solve is Go (also known as weiqi or baduk), but more specifically the life and death problems called Tsumego. We created a Go application that has predefined Tsumego problems where you will be able to play against the AI we built to try to beat him in a Tsumego match. 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 pruning efficiency. The program was built on 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:

Liberty: For a stone that is on the board, the number of liberties 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.

Eye: An eye refers to an unoccupied position (or group of connected unoccupied positions) in which all the liberties of the position(s) are occupied by the opponent. Eyes can be either real, false or undetermined depending on the number of diagonally adjacent positions (DAPs) that are occupied by the opponents. A real eye is an eye in which three or more DAPs are either occupied by the 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 or false if all positions within an eye meets the DAP requirement of a real or false eye, respectively. Fig. 2 shows an example of different types of eyes.

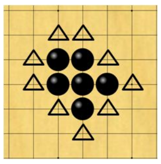
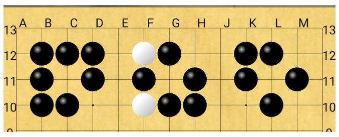
	

Figure 1. Liberties of a group labelled as triangles	Figure 2. Examples of real (left), false (middle) and undetermined (right) eye

The Tsumego problem will now be defined with certain assumptions that are adopted to narrow down the scope of the project.

- A Tsumego problem starts with a number of stones of one player (defender) surrounded by the opponent's stones (attacker). The defender stones do not have to be completely sealed by the attacker. However, these problems are designed so that the defender cannot escape from the trapped area and that the “walls” formed by the attacker are invincible.

- 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 contains at least 1 liberty when it’s played.
    - The stone is connected to other stones of the same team such that the resulting connected stones 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. This will guarantee that the stone that was placed will have at least one liberty and is entitled to stay on the board.
- An attempt is considered successful if the defender can form two real eyes and the pieces that make up each eye are connected. The rationale behind making two eyes is that a connected piece with two eyes contain at least two liberties. However, a player cannot place the stone in one of the two eyes because it will have no liberties and it will not be able to capture the surrounding stones. Therefore, these connected stones are guaranteed to survive. Real eyes are required since false and undetermined eyes can be destroyed.
- The game ends is reached if one of the following situations is encountered:
    - The attempt is unsuccessful if all moves within the boundary are exhausted and the defender fails to form two real eyes.
    - The attempt is unsuccessful if all defending stones are captured by the attacking stones.
    - The attempt is unsuccessful if two real eyes are formed by the defender.
    - A cutoff variable is used to limit the depth of the search. The attempt is unsuccessful if the maximum depth is reached without forming two eyes.
- In the real game of Go, the domain of the positions is the entire board. However, in Tsumego, a virtual boundary that encompass the trapped area is pre-defined so that both players can only play inside the boundary.
- Since we are only concerned with survival problems, the defender always makes the first move to try to survive.
- For this project, our program aims to survive inside the trapped area, so the computer is always the defender and always plays first.
- In this report, the term ‘defender’ and ‘attacker’ pieces are used instead of ‘black’ and ‘white’ to avoid ambiguity.
- 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.

<u>State encoding:</u>

Initialization:

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

1) ‘Att’: List of tuples containing x,y coordinates of all attacker pieces.
2) ‘Def’: Same as 1) except for defender pieces

3) ‘Avail: List of unoccupied positions within the defined boundary.
    a) Usage: generate successor moves
4) ‘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.
    a) Usage: calculate liberties of stones on the board.
5) ‘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.
    a) Usage: search for eyes in a given state.

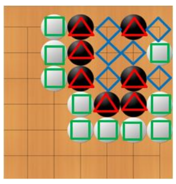
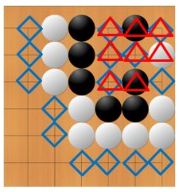
	

Figure 3. Example of ‘Att’ (square), ‘Def’ (triangle), ‘Avail’ (diamond)	Figure 4. Example of ‘All’ (diamond) and ‘EyeSpots’ (triangle)

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 has include a list of all liberty spot of 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. The successor function performs the following every time a successor node is generated:

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 was called to exclude any illegal moves.
5) Determine all the eyes on the board as a results of the move.

Algorithms:
Several algorithms were employed to ensure that the game works properly. Three major tasks that build up the skeleton of the game will be discussed.

1. Determine connected stone (Initialization):

    - 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 the that group. Else, create a new group.
    - Step 2: Depending on how the list is ordered, some groups might still be split into multiple group. Go through each group and check for connections 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 (for defenders only):
    - Input: list of unoccupied positions and all attacker pieces occupying an eye spot (see definition of board object above).
    - Step 1: A group of unoccupied space that are connected belong to the same eye, so use algorithm #1 again to generate "connected spaces" from the inputs.
    - 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:

Due to the large search tree, two heuristics were developed to improve alpha-beta pruning. Our first heuristic, vital spot heuristic, was based on a strategy for solving Tsumego problems. It values moves by their potential to create the most eyes with minimum number of moves (see reference link for details). Our second heuristics 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.

$$score\ =\ 20 * \#\ of\ real\ eyes\ +\ 10\ *\ \#\ of\ unknown\ eyes\ +\ 5\ *\ \#\ of\ false\ eyes$$

Our final version was a two stage approach that incorporates both heuristics. Specifically, we used the vital spot heuristics 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 provided a good heuristics throughout the entire search and should out-perform the individual heuristics. In order to maximize pruning, the alpha always chooses the move with the highest heuristic score while beta chooses the move with the lowest score

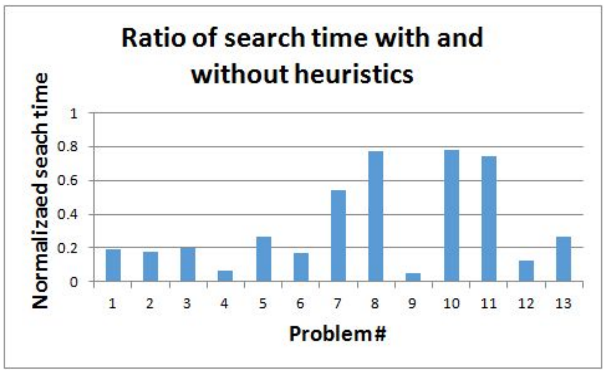
Alpha-beta search:

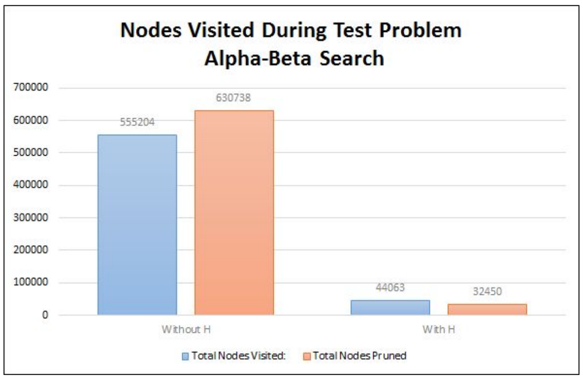
Since we were only concerned with finding a path that will guarantee to survive, so our alpha and beta values were 1 or 0 (live or die) and we exit the search when alpha becomes 1 at the root node. There may be other paths to the solution in other parts of the tree, but it will not make a difference 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 near 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 drive beta to return the strongest moves (i.e. longest path to reach survival state), the nodes that are visited often correspond to the max depth to the entire solution space. 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 efficiency of our solver. Breadth referred to the number of possible moves in a given state while depth reflected the number of steps required to reach the solution. For each problem, the cutoff 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 enter the moves that our solver plays in the app and play the corresponding move the app returns back into the program. A solution is deemed correct if the our solver can play the game until the app tells us that the problem is solved correctly. The effectiveness of our heuristics was measured by solving the same problems with the combined heuristics and without heuristics. In both cases, the same cutoff depth was used 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 2. 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 is also reduced by a significant amount. It may seem weird that the percentage of nodes pruned is higher when heuristics are not used. However, with heuristic present, the node that visited are 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. Therefore, the chances of finding a survival state early and pruning the rest of the moves is higher.





One of the problem that could not be solved is shown in Fig. 4.1. 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 defender made the move (the black stone labelled with a circle) to complete the second real eye. However, the attacker can actually play at the red dot to start a 'ko', but since our solver does not account for 'ko' situation, it simply removes the stone and reaches the survival state with three eyes (Fig. 4.2). The correct solution is shown in Fig. 4.3 where the defender can survive without a encountering a 'ko', which is often desired 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 can indeed produce the right sequence.

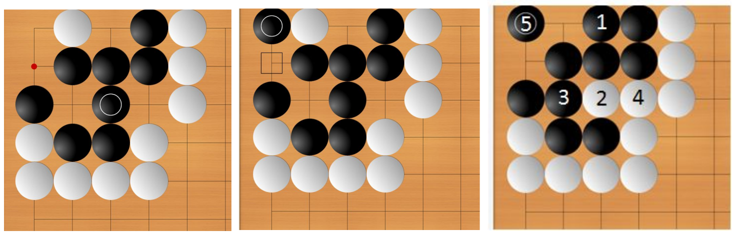


Figure 4.

There was another problem that could only be solved 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 actually drives the moves towards the minimum number of steps to reach the solution. During the search, there are 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 (assuming the number of steps has not reach the cutoff depth yet). However, with the heuristic, the moves that reaches the survival state will be rewarded compared to other unfinished states at any point during the search. Therefore, the heuristic will converge to the solution state more rapidly.

The vital spot heuristic attempts to make a move in the beginning of the problem. While it is a reasonable approach, it is not always optimal since the defender often have to save stones with small liberties left and are 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 is very rare in Tsumego problems. 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 observed that combined heuristics worked better than individual heuristics, but the results were not officially documented.
- The cutoff 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 code is to determine the eyes since they can appear in multiple configurations. There was a lot of trial and errors involved to deal 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 (speaking from someone who has learned the game for 7 years). However, there are still several improvement that can be made in the future:

- Implement iterative deepening search to find the solution systematically in the shortest amount of time.
- Implement ‘ko’ rule and ‘seki’ state so that the solver can tackle any Tsumego problems.
- The search itself takes a long time because a lot of computations are needed to simply generate the successor states. Future improvement can be made to make this process more efficient. For example, the process of counting liberties can be simplified by updating the liberties that are affected by the new move instead of recalculating the liberties for the entire board.

References

Hollosi, A., & Pahle, M. (n.d.). Front Page at Sensei's Library. Retrieved December 01, 2016, from http://senseis.xmp.net/?ApproachInTsumego#toc6

Russell, S. J., & Norvig, P. (2014). *Artificial intelligence: A modern approach*. Harlow: Pearson. Retrieved from http://teaching.csse.uwa.edu.au/units/CITS3001/resources/gameAI.py, http://teaching.csse.uwa.edu.au/units/CITS3001/resources/utils.py

Emmanuel Mathis. (2016). Tsumego Pro (Go problems) (Version 2.09). (2016). [Mobile application software]. Retrieved from https://itunes.apple.com/ca/app/tsumego-pro-go-problems/id892041876?mt=8