Go - Life or Death

Leo Wong, wongleo7

\*Names, teach.cs-login ID for all members

\*Roles

* Problem encoding
* Heuristics
* Manuscript author
* Experimental Assessment
* More

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 will use 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). We have encoded several heuristics that will help maximally prune the alpha-beta tree.

**Methods**

Our problem was rather difficult as there are many branches for the alpha-beta tree and we had to maximize pruning as much as we could. In our initial tests, before encoding our heuristics, we were unable to go anywhere beyond a limit of a depth of 7 before encountering enormous processing times. To solve this problem, we had to….use magic

\*\*\*\*\*\*\*\*\*\*how did we solve it

\*\*\*what are our state variables and domains

Shayne: problem formulation, constraint encoding, state representation (var and doms, succesors),

Problem formulation:

Result

Leo: heuristics, admissibility, 2 real eyes as highest\*\*\*

Heuristics: eyeHeur, vitalpoint, check for survival (2 eyes)

Terminal: no more moves, wipedout, check for two eyes (completed problem)

--- pieces can get wiped out (more spaces)

Terminal States:

In order to call our game completed, we have encoded a couple terminal states. Our game is considered a loss when no more moves are available and the two eyes has yet been completed, or if the defending pieces is completely wiped off the board. The goal of Tsumego is to have the defender secure it’s position on the board by creating 2 eyes which are impenetrable and if a whole team has been wiped, it is definitely gone way off track and is an undesirable state. Our Tsumego problem is considered solved when two true eyes are created by the defender, this will in-turn return a heuristics value of 100 to the function which will be sent up the alpha-beta tree. 0 is returned by the function if it reaches the terminal depth without success.

Due to the nature of the game ‘Go’, we had to use very strong heuristics in order to prune the abundant amount of branches so that the AI will be able to finish running within a reasonable timeframe. We implemented a couple heuristics, all of which were able to significantly decrease our runtimes and the number of nodes visited. Our first heuristics score is returned when we reach our completion state, as described previously. Our second heuristic sorts the next available moves so that the move with the most influential vital spot (highest potential to create eyes) is pushed to the top of the queue for Beta nodes, and the probable least scoring values first on Alpha nodes. This coincides with the strategy of having decreasing values on Beta nodes and increasing values on alpha nodes to maximize pruning. As the goal of our game is to complete two eyes, our second sorting heuristic does exactly that. We sort our successor states based on the number of eyes they have. We give them weight for if they have ‘Real’ eyes, ‘False’ eyes, or ‘Unknown’ eyes. Each of these represent the stability of their eye, ‘Real’ being the least volatile while ‘False’ being the most volatile, Unknown is undetermined because the spots to determine if they are ‘Real’ or ‘False’ have not been occupied yet. Like our previous sorting algorithm, we have it increasing for alpha nodes and decreasing on Beta nodes. To use these two to their maximum potential, once our first sorting algorithm does not find any more vital spots, we use our second sorting heuristic instead. This tag team technique works well because our second sort is better suited for late games where the first one would most likely fail. Together with these few heuristics, we were able to visit less than half of the full tree and was able to increase the depth of the tree to be able to get stronger results.

\*\*Examples\*\*

For our state variables, we had ‘player’, ‘moves’, ‘board’, ‘connectedPieces’, and ‘survivalState’, ‘eyes’. Firstly, ‘player’ lets us know whose turn is it next so we could know for who we should calculate for next. Next, ‘moves’ is an array of legal moves for the current player, the player should only be able to pick from this list when selecting their next move. Naturally, ‘board’ is a dictionary containing the current state of the board which encodes the black and white pieces on the board and also the available spots. It may seem like we have redundant information as the board minus the pieces would give the available positions remaining, however, we have a predefined problem which may not always be played a square board. Our ‘connectedPieces’ is also a dictionary containing list of defender and attacker pieces that are connected as well as the number of liberties in each group.. This data helps us better understand the complex board arrangement as knowing this information we could easily tell when an area is enveloped by opposing pieces. This significantly reduces processing times as it means that we do not need to process this every time we generate a new successor state. Our heuristics mentioned above never overestimate the cost it takes to reach the desired state since they only help guide towards the goal and never explicitly suggest the estimate to reach the goal.

Example (black first to survive)

Player: ‘Def’

Moves: [(0,0), (0,2), (2,0), (2,1), (2,2), (1,1)]

Board: Contains attacker and defender pieces

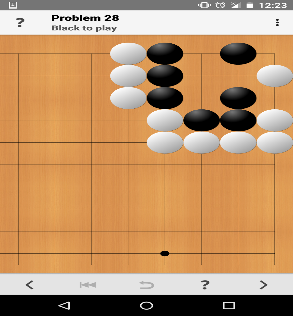
ConnectedPieces:

'Att': [([(0, 1)], [(0, 0), (0, 2), (1, 1)]), ([(4, 0), (4, 1), (4, 2)], [(5, 0), (5, 1), (4, 3), (5, 2)])...]

'Def': [([(1, 0)], [(1, 1), (2, 0), (0, 0)]), ([(1, 2), (1, 3), (2, 3)], [(1, 1), (2, 2), (0, 2)])...]

Survival state: False

Eyes: None



----player  
Att  
----moves  
[(1, 1), (2, 1), (2, 2)]  
----board  
{'Avail': [(1, 1), (2, 1), (2, 2)], 'Att': [(1, 0), (1, 2), (1, 3), (2, 3), (3, 0), (3, 1), (3, 2), (2, 0)], 'All': [(0, 5), (1, 1), (1, 5), (2, 1), (2, 2), (2, 5), (3, 5), (4, 3), (4, 4), (4, 5), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)], 'Def': [(0, 1), (0, 3), (0, 4), (1, 4), (2, 4), (3, 3), (3, 4), (4, 0), (4, 1), (4, 2), (0, 0), (0, 2)]}  
----connectedPieces  
{'Att': [([(1, 2), (1, 3), (2, 3)], [(1, 1), (2, 2)]), ([(1, 0), (2, 0), (3, 0), (3, 1), (3, 2)], [(1, 1), (2, 1), (2, 2)])], 'Def': [([(4, 0), (4, 1), (4, 2)], [(5, 0), (5, 1), (4, 3), (5, 2)]), ([(3, 3), (0, 1), (0, 0), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4)], [(4, 3), (1, 1), (0, 5), (1, 5), (2, 5), (3, 5), (4, 4)])]}  
----survivalState  
False  
----end

\*\* what are our successor functions and how did we code our constraints

\*\*What are our heuristics

\*\*\*include example

Show simplest case to demonstrate alpha beta search

Limitations and obstacles:

only dealt with living problem, not killing

Set depth to reduce search time, but has no effect on algorithm

Can use IDS in the future

Obstacles: determining real eyes is a pain...

Had to manually implement some constraints on search space

Ignore KO and tie

Reference:

alpha beta code

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