Building Tic-Tac-Toe Bot

Game Algorithms and Optimization Techniques

Haeyoon Chang

June 7, 2019

Overview

The goal of this project is to explore game algorithms and optimization techniques, and tic-tac-toe is the game of pick for this project. Tic-tac-toe is a simple combinatorial game on a 3×3 board played by two players, which is zero-sum, deterministic, discrete and sequential. The winning and losing in the game depends on the terminal state of the game, and thus, each step of the game is usually modeled as trees of decisions. Basic game search algorithm used for this project is adversarial search, namely minimax and negamax search. Optimization tools implemented include alpha-beta pruning, moving the order of search, and using the transposition table.

Game Tree Search Algorithms and Analysis

Minimax search Minimax search is a depth-first search formulated for two-player zero-sum game, covering both cases where players take alternate moves; for tic-tac-toe game, it is a human vs. a bot. In minimax search, max search calls for min search and min search calls for max search, recursively computing minimax values of each successor state. When there are no more empty cells or it reaches the terminal state (win, lose or draw), it evaluates the state, then backpropagate to the root state or node where minimax algorithm is initially called.

In the pseudocode below, *state* stands for board status, *depth* represents the maximum number of turns two players can take (empty cells left), and *player* is either human (-1) or bot ($+1$)

```
MINIMAX(state, depth, player)
 1  if depth = 0 or state is a terminal state
 2      score = heuristic value of the state
 3      return score
 4  if player is computer:
 5      value = −∞
 6  else value = +∞
 7  for each cell in empty cells
 8      score = minimax(state, depth - 1, - player)
 9      if player is computer:
10          value = max(value, score)
11      else value = min(value, score)
12  return value
```

Negamax search Negamax search is a variant form of minimax search. This algorithm is based on the fact that $max(a, b) = -min(-a, -b)$ and it slightly simplifies minimax code above [4]. The pseudocode below is referenced from negamax wikipedia website.

NEGAMAX($state, depth, player$)

```
1  if depth = 0 or state is a terminal state
2        score = heuristic value of the state
3        return player x score
4  value = -∞
5  for each cell in empty cells
6        score = −negamax(state, depth - 1, - player)
7        value = max(value, score)
8  return value
```

Minimax and negamax search explore an extensive amount of states until they reach terminal states, and they can be expensive, especially for some games with large branching factor (e.g. chess). Alpha-beta pruning, one of optimization techniques, reduces time spent to search for the next move by avoiding branches of the game trees that are obviously not worth probing into.

Alpha-Beta Pruning Alpha-beta pruning removes (cut-off) obviously bad moves. During the depth-first search of the game tree, we keep track of two parameters, α and β. Alpha (α) is value of the best possible move the player can make (maximize *my* score), while beta (β) is value of the best possible move that the opponent can make (minimize *my* score). Logic of the algorithm stays the same as minimax/negamax search; however, this technique avoids evaluating the branch any further (lines 9-10 below) if alpha is greater or equal to beta, because we know that the opponent's move would force the player into a worse position than current status[7]. The pseudocode below is referenced from negamax wikipedia website.

NEGAMAX-ALPHA-BETA($state, depth, player, alpha, beta$)

```
 1  if depth = 0 or state is a terminal state
 2        score = heuristic value of the state
 3        return player × score
 4  value = −∞
 5  for each cell in empty cells
 6        score = −negamax(state, depth - 1, −player, −beta, −alpha)
 7        value = max(value, score)
 8        alpha = max(value, alpha)
 9  if alpha ≥ beta
10        break
11        return value

;initial call - a player is a bot(+1)
negamax(state, depth, 1, −∞, +∞)
```

Move Ordering In alpha-beta pruning, the algorithm attempts to prune as many states as possible based on the state it evaluates first, thus, the order of visiting nodes matters. To begin with, let's assign 0, 1 and 2 for x and y coordinates of 3×3 board, 1 is for the center for each

coordinate. Starting from nine available cells, widely known heuristics in tic-tac-toe is selecting cells in following order.

1. center cell (x, y) = (1, 1),
2. corner cells (x, y) = (0, 0), (0, 2), (2, 0), and (2, 2), and
3. all other remaining cells (x, y) = (0, 1), (1, 0), (1, 2), and (2, 1)

We can sort a list of empty cells according to the heuristics above, and explore branches in that order. Using this heuristics generally increases the effectiveness of alpha-beta pruning.

Time Complexity and Analysis For 3×3 tic-tac-toe game, the maximum number of legal moves at each point, namely branching factor, is 9 ($b = 9$) and the maximum depth of the tree is 9 ($d = 9$). Therefore, the time complexity of going through all possible outcomes would be:

$$\begin{aligned}\prod_{i=1}^{9} i &= 9 \times 8 \times 7 \times ... \times 2 \times 1 = 9! \\ &\in O(9!) \approx O(9^9)\end{aligned}$$

Let's assume generalized tic tac toe board with branching factor of b and depth of d. The worst case time compleity of simple negamax search with no optimization would be $O(b^d)$.

$$\begin{aligned}\prod_{i=1}^{b} x &= \underbrace{b \times (b-1) \times (b-2) \times ... \times 3 \times 2 \times 1}_{d \text{ times}} = b! \\ &\in O(b!) \approx O(b^d)\end{aligned}$$

Next, let's assume that we use alpha-beta pruning and that empty cells are examined in the optimal order. The best move is going to be searched first and the other branches in the same depth level will be pruned. This means the search algorithm will only scan full branch in every other level (depth), effectively halving the depth ($d/2$) and reduce the time complexity to $O(b^{d/2})$[1].

Alpha-Beta with Memory A notable feature of tic-tac-toe game is that one particular board state is evaluated several time because there are several ways to reach a particular state. If we can store the state as key and the value associated as value in the form of hash table (transposition table), search can become more efficient. Otherwise, each pass of function would re-explore most of these states [6]. This strategy is similar to memoization for dynamic programming.

Before saving the state and its value into the transposition table, we change the board state (a list of 3 lists, each with 3 elements) into a single unique number using zobrist hashing. In zobrist hashing, each cell has three unique randomly generated hash value (grids) for each case of nought (O), cross (X), and empty, and the hash value of a state (hash-boards) is the bit-wise exclusive disjunction (XOR) of hash values of each cells.

Transposition table access takes place in retrieve (look-up) and store calls. Retrieve function (lines 2-7) checks if a value is present in the table, and if so, stored information is used, instead of continuing the search. The store function (lines 19-28) fills up with values as they become available. The pseudocode below is referenced from MTD(f): A Minimax Algorithm faster than NegaScout website [6] and tweaked to align with negamax search.

NEGAMAX-ALPHA-BETA-TTABLE($state, depth, player, alpha, beta, grids, ttable$)

```
1  // **Look up transposition table first**
2  if state is in ttable
3        if ttable[state].lower-bound ≥ beta
4              return ttable[state].lower-bound
5        if ttable[state].upper-bound ≤ alpha
6              return ttable[state].upper-bound
7        alpha = max(alpha, ttable[state].lower-bound)

8  // **negamax with alpha beta pruning **
9  if depth = 0 or state is a terminal state
10       score = heuristic value of the state
11       return player × score
12 value = −∞
13 for each cell in empty cells
14       score = −negamax(state, depth - 1, −player, −beta, −alpha)
15       value = max(value, score)
16       alpha = max(value, alpha)
17 if alpha ≥ beta
18       break

19 // ** Store bounds to transposition table **
20 ttable[state].upper-bound = +∞
21 ttable[state].lower-bound = −∞
22 if value ≤ alpha
23       ttable[state].upper-bound = value
24 if value > alpha and value < beta
25       ttable[state].upper-bound = value
26       ttable[state].lower-bound = value
27 if value ≥ beta
28       ttable[state].lower-bound = value

29 return value
```

Experimental procedure

Basic tic-tac-toe board is used to conduct some experiments. In this process, I measured how many board states the function analyzed and time it takes to return the next move. Human makes the first move taking the middle cell, (1, 1), then computer makes next move based on the algorithms implemented. Performances of the following approaches have been measured:

1. negamax search *without alpha-beta pruning*

2. negamax search *with alpha-beta pruning* and *worst case move order*

3. negamax search with alpha-beta pruning and *best case move order*

4. negamax search with alpha-beta pruning, move ordering, and *using memory*

To ensure that garbage collection would not interfere with the process in python, timeit library is utilized. Also, timeit function is applied only to *bot-turn* so that it doesn't count the time spent on waiting for human input.

Result of experiments

Figure 1 shows a sample game of tic-tac-toe with the number of boards analyzed for each of four approaches. All four methods are based on negamax search and the sequence of human moves is unchanged across the four experiments. Therefore, the best move by a bot should also stay the same regardless of the optimization techniques used. In the experiment, the sequence of moves a bot chooses indeed remained unchanged throughout the experiments 2, 3 and 4.

The number of boards analyzed varied across the four approaches, especially for the first two turns. For example, the number of searches for the first turn in the example ranged from 256 (negamax search with alpha-beta pruning, optimal move order, and memory) to 25,872 (simple negamax). This shows that optimization techniques discussed above work efficiently.

Theoretically, if we rearrange the order of states to be optimal (case 3), we can expect the number of searches to drop from b^d to $b^{d/2}$ at maximum. Likewise, if the states are ordered in the worst sequence (case 2), the number of search should be close to simple negamax case, b^d. However, when we use simple negamax as our base case, it shows that case 3 underperformed our expectation ($815 > \sqrt{25872} \approx 161$) and that case 2 outperformed our expectation ($1053 < 25872$). The difference between the expectation and actual results are presumably related to the nature of tic-tac-toe game. In this game, the value returned from the terminate states is one of three values (win: 1, lose: -1, and draw: 0), thus several states yield to same value.

Figure 2 shows the time it takes to make the next move in each of four cases. I ran 15 games in each methods, then average the time it takes to proceed to the next move for each depth. Provided that the execution time of simple negamax search is set at 100, negamax search with alpha beta pruning with worst order (ab worst) for the first move showed 87.6, the search with alpha beta pruning with best order (ab best) had 85.3, and the search with alpha beta pruning with best order and with memory had 87.1. Although this tic-tac-toe has a small board expansion and it limits accurate evaluation of efficiencies in each methods, all three optimized scenarios show some reduction in running time, especially in the first stage of the game where branching factor and depth level are still relatively large.

It can be seen that negamax with alpha beta pruning and with transposition table does not show significant improvement in terms of running time. This result may have several reasons, it is expected that the overhead of looking up the table took just as much time as evaluating the next states and limits the performance, for the game with such small branching factor and depth.

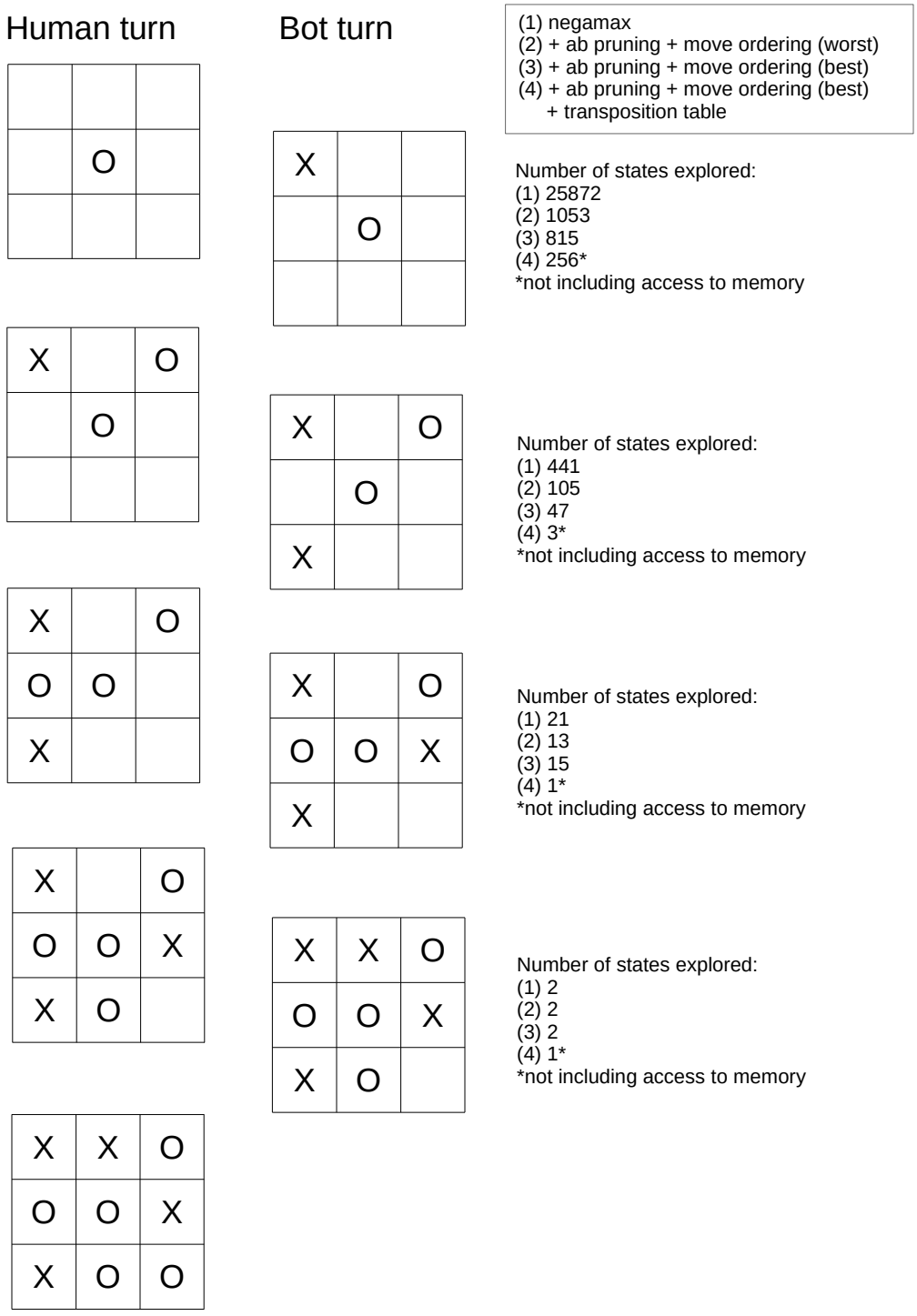

Figure 1: A sample demonstration of tic-tac-toe game

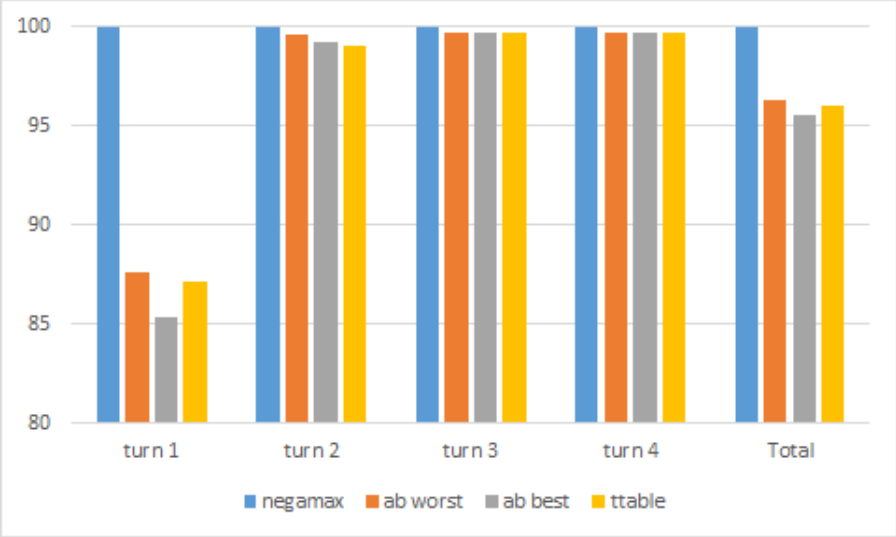

Figure 2: Execution Time to Make the Next Move

Conclusion

Focus of this project was to improve the game tree search algorithm such as minimax and negamax so that it finds the optimal move more efficiently.

Amongst the three optimization techniques discussed, alpha beta pruning should always be used over simple adversarial search for more time efficient search. Move-ordering is a strong tool that can maximize effect of the alpha-beta pruning optimization; however, it requires game-specific heuristics and unless players have an idea of which order they will conduct the search, finding out optimal order itself will be difficult, especially for the games like Go with high branching factor and depth levels. Finally, it is observed that the transposition table allows us to avoid doing the same calculation over and over at the expense of memory.

On a personal note, I have been interested in game AI, but did not have an opportunity to dig into it due to lack of backgrounds. I am glad that I had a chance to implement full tic-tac-toe as well as some of game search algorithms with optimization tools including building a hash table and memoization.

References

[1] M.Tim Jones. *Artificial Intelligence: A Systems Approach.* Infinity Science Press LLC, Hingham, Massachusetts, 2008

[2] Anurag Bhatt, Pratul Varshney, and Kalyanmoy Deb *Evolution of No-loss Strategy for the Game of Tic-Tac-Toe* KanGAL Report Number 2007002

[3] Stephan Schiffel. *Symmetry Detection in General Game Playing* Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010.

[4] Negamax wikipedia
`https://en.wikipedia.org/wiki/Negamax`

[5] Vasileios Megalooikonomou, The CIS603-Artificial Intelligence,
`https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html`

[6] MTD(f): A Minimax Algorithm faster than NegaScout
`http://people.csail.mit.edu/plaat/mtdf.html#abme`

[7] Prof. Melanie Mitchell's website on Artificial Intelligence
`http://web.cecs.pdx.edu/~mm/AIFall2011/`