Tic-Tac-Toe

Jiupeng Zhang, Sep 2018

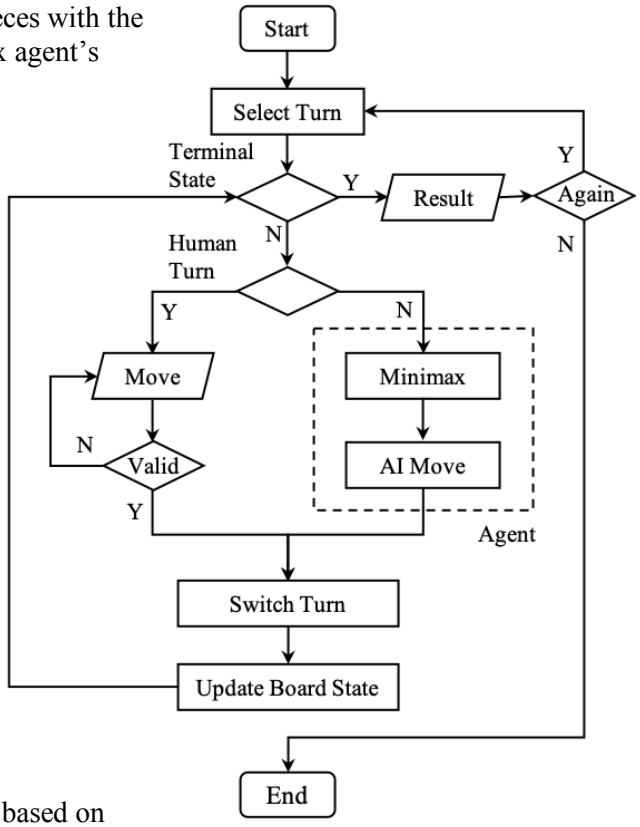
Part I: Basic Tic-Tac-Toe

- **Program Design**

  To build a object-oriented tic-tac-toe, I defined pieces with the board itself as class ***State***. Obviously, our minimax agent's goal is to forward each state and win a game.

  The basic processing flow is shown on the right. In order to reuse most of my code, I designed interfaces (***State*** and ***Agent***) with the granularity of my abstraction determined. The class ***PrunedMinimaxAgent*** is an implementation of interface ***Agent*** which expresses my core logic of calculating actions, utility values and decision making.

  A crucial part of the program is to examine terminal states. In order to reduce time, I decided to use local scan instead of full board checking and this was indicated in the ***state.update()*** method.



- **Concepts and Algorithms**

  Tic-tac-toe is a determinate zero-sum game model which is observable, so we can use the ***Minimax*** algorithm to passively find the best step based on the effort of state space DFS.

  However, the ***naive minimax*** performs awful when solving real world problems because of complete searching. Hence, optimizations are required to cut-off useless state searches.

  Although as the part one tic-tac-toe, a complete searching is available to have the result given, I introduced the ***alpha-beta pruning*** to speed up my program.

- **Implementation & Performance**

  I compared the actual enhancement of alpha-beta minimax by running each program 100 times, and record the average response time(in ns) in each program:

Step No.	1	3	5	7	9
Execution time(pure minimax)	205,756,048	1,696,931	74,729	12,802	6,709
Execution time(with α-β)	45,076,951	577,313	41,506	13,535	7,957

Percentage of improvement	78.09%	65.98%	44.46%	-5.73%	-18.60%

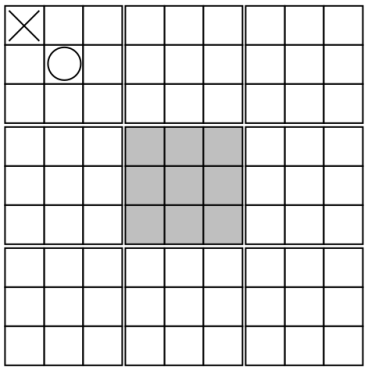
(This table above has five records on each row, for reason that the maximum possible count of steps for basic tic-tac-toe is nine with a result of draw, that is to say, if our agent goes first, there will be less or equal than five steps taken by agent per game.)

We can see that the minimax enhanced program performs much better in first several steps. An interesting thing is that with step goes further, the improvement of alpha-beta pruning becomes trivial and even worse. This is because when the algorithm goes deeper into a state space tree, fewer states are needed to be pruned, and the effort of alpha-beta value checking (those if-else statements) becomes costly then.

Part II: Advanced Tic-Tac-Toe

- **Program Design**

The main difference between the basic and the advanced tic-tac-toe is on the scale of "actions" states. In this problem, the next possible positions to take is no more limited in a specific 3x3 board, which means that we must take care of all available places on another board. For example (on the graph below), if an opponent took a circle in position #5 in board #1, we need to take positions on the whole board #5 as my next actions. That is to say, nine grey states need to be added in "actions" with this example.



Additionally, the depth of the state space is significantly increased because of the enlargement of the whole board. In this case, we can hardly search till the end of game. At least on my own laptop, I cannot get an answer by use of alpha-beta pruned minimax algorithm in an acceptable period, so further optimizations are needed.

To avoid sinking into terminate states to get a real value, we can rate a state based on a heuristic judgment instead, and that's the reason I introduce the heuristic function in this part II program.

- **Heuristic Strategies**

At the start, I wrote a naïve heuristic strategy, to let the agent avoid taking positions to correspond to opponent's advantage board [#1]. Specifically, I sent a larger penalty to a board with more opponent's pieces.

In our team testing play, we found the agent was lost when facing adversary's assault, so we offered a large penalty when my agent give up and let his rival win [#2], In addition, we fed another heuristic to him: "make more troubles to your opponent (create more checkmate situations (denoted by "check states" in the following paragraphs) [#3])", that means I will encourage him when he did that (after backtracking, this encouragement, a.k.a. larger state value, might lead him to this advantage state in future).

My team had an intense discussion on this heuristic strategy, after that, an idea came out, why we won't also encourage him to break the opponent's check states? Finally, we gave this advice to our

agent, not only let him increase his check count, but also reduce the check count of his opponent's [#4].
This strategy let him automatically resolve the opponent's checks, which makes my agent ever smarter!

It turns out that at the beginning of a game, we need more delicate handling on each state, so we add several low-weighted heuristics [#5], one of them is let him know the position advantage (the center position on a board has a chance of 4 to win a game which the corner position only has three, and so on)

Then is the most suffering procedure - optimization.

- **Optimization**

  1. **Heuristics optimization**
     The first thing we do is to adjust our weight of each heuristic strategy, so we run the program to let it play with itself (but with different versions of heuristic function), and then analyze their performance by key step review. Finally, we decided to keep heuristic #2, #3 and #5 and found the #2 is a super essential strategy with a super intuitive expectation.

  2. **Programing optimization**
     In addition to my local scan method to save terminal checking time, I replace multi-loops as a single one to shrink overhead of frequently visited key procedures.

     Furthermore, I buffered many useful parameters in my class ***State*** to reduce method re-querying as well as tried to avoid using too much high-level packaging collection types to simplify my code logic. In fact, this type of optimization helps a lot (I can see much further if I would able to dig deeper in searching, which will lead my agent get a better solution).

  3. **Algorithm optimization**
     In tests, I found the first step almost take the largest amount of time, this is because the actions space for the first step is the biggest of the whole (because my agent must access every position on the board). However, we don't actually need to search for all possible positions because of the initial board is naturally symmetrical (I tried to use this logic to optimize this algorithm in other steps but I found it is costly to detect symmetric itself ☹), hence, the only positions I need to search in my next step are positions on board #1, #2 and #5. This trick really saved my time.

- **Implementation & Performance**

  I printed some status parameters in my output which might helped me notice some feedbacks and status of my program:

  Distinction: the subtraction of my best and worst action values.
  Confidence: the optimal value for my best choice.
  Piece count: piece count on each board.
  Max Piece count: piece count for the max player.
  Max check: checkmate count in each board for max player.
  Min check: checkmate count in each board for min player.
  Last step: the last step the opponent taken.
  Depth/Status: the total step taken/max or min player's turn.

```
Distinction:     [1990.0]
Confidence:      [2030]
HeuristicPrunedMinimaxAgent used 1346376680 nano seconds
+-----------+-----------+-----------+
| -  -  -   | X  -  -   | -  -  -   |
| 0  -  -   | -  -  -   | -  -  -   |
| -  -  -   | X  -  0   | -  -  -   |
+-----------+-----------+-----------+
| -  -  -   | -  -  -   | -  0  -   |
| -  -  X   | -  -  -   | -  -  -   |
| -  -  X   | -  -  -   | -  -  -   |
+-----------+-----------+-----------+
| -  -  -   | -  -  -   | -  X  X   |
| -  -  -   | 0  -  -   | -  -  -   |
| 0  X  -   | -  -  -   | -  -  0   |
+-----------+-----------+-----------+
Piece count:     [1, 3, 0, 2, 0, 1, 2, 1, 3]
Max Piece count: [0, 2, 0, 2, 0, 0, 1, 0, 2]
Max check:       [0, 1, 0, 1, 0, 0, 0, 0, 1][3]
Min check:       [0, 0, 0, 0, 0, 0, 0, 0, 0][0]
Last step:       [9, 3]
Depth/Status:    [13][MAX]

#3>
```

The last line shows the next board number you'll take over.
We can reach depth 11 in a not-bad time, and time for depth 12 is also acceptable:

Cut-off depth	11	10	9	8	7
Avg. time to take first max step(ns)	11,354,604,239	4,086,604,229	954,841,359	434,483,832	195,997,212
Avg. time to take first min step(ns)	2,698,224,244	782,260,614	310,984,196	176,519,648	72,030,806

Part III: Ultimate Tic-Tac-Toe

- **Program Design**

  The board is the same as the advanced tic-tac-toe, so the good news is that we can reuse most of the previous version of ***State*** and only add a few extra parameters to it. Following with that good news, we need to change some logic in several other components, like ***state.update()***, ***agent.actions()***, ***agent.heuristic()***, etc. Besides, additional global/local terminal detect is needed to implement this game program correctly.

  However, the most valuable part of ultimate tic-tac-toe is to come up with new heuristic strategies.

- **Heuristic Strategies**

  Drawing on the experience of the previous heuristic strategies, we chose four strategies for this game:

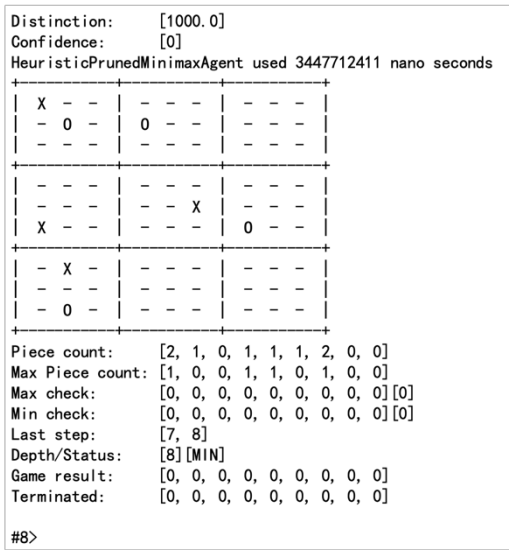
  1. Do not lose a local game: penalty the agent if he loses a local game.
  2. Pick a better board to win: encourage more when I can win a more valuable board.
  3. Avoid let opponent do a random pick.
  4. Choose better choices if I can avoid my opponent's checkmate situation.

- **Implementation & Performance**

  I printed some status parameters in my output which might helped me notice some feedbacks and status of my program:

  Distinction: the subtraction of my best and worst action values.
  Confidence: the optimal value for my best choice.
  Piece count: piece count on each board.
  Max Piece count: piece count for the max player.
  Max check: checkmate count in each board for max player.
  Min check: checkmate count in each board for min player.
  Last step: the last step the opponent taken.
  Depth/Status: the total step taken/max or min player's turn.

  The last line shows the next board number you'll take over.



We can reach depth 11 in a not-bad time, and time for depth 12 is also acceptable:

Cut-off depth	11	10	9	8	7
Avg. time to take first max step(ns)	12,120,124,163	3,212,343,165	1,136,283,787	369,672,468	294,637,289
Avg. time to take first min step(ns)	6,277,037,559	1,807,585,826	510,988,831	248,566,100	122,960,479

Future work

There are several system-level enhancements which might help achieve ever better performance:

- Parallel calculating

  We can easily divide a minimax searching task into sub-problems, that means we can fully use the powerful multi-thread ability on our multi-core devices.

- Time borrowing

  We can get use of the opponent's thinking time to do some pre-calculation.

- Result reusing

  we can pre-store some calculated utility values and reuse them in some future calculating.

- Vertical heuristic valuation

  we can search deeper (or even reach the real utility if possible) based on some heuristic functions to dig more valuable branches.