PA 3 - Chess

Julien Blanchet | CS 276 F20 (Artificial Intelligence) | Prof. Li.

October 9, 2020

Description

The objective in this assignment was to create a python AI for playing chess. My implementation closely follows the standard structures for minimax and alpha-beta pruning as described in our textbook. There are a few elements of my design worth mentioning:

1. The heuristic function is passed into the AI class object. This allows for easy testing and comparison of heuristics.
2. Rather than repeating code in the minimizing and maximizing code paths, I combined the code paths and chose the `comparison` and initial `best_score` values based of the `is_maximizing` test.
3. To increase the efficiency of alpha-beta pruning, I implemented a simple move reordering - capture moves are moved to the front of the order, followed by the rest of the moves.
4. To reduce duplicate code, I had the minimax implementation reuse the alpha-beta code, but with pruning disabled.
5. For the iterative deepening task, I used threading to run an input listener and the AI move-chooser on background threads. This allows me to abort the search when the user types and return the best move found at that point.
6. I setup `test_chess.py` to be compatible with `pytest`. As I coded the AI, I created tests to validate expected behavior. This allows me to easily check for regressions by running `pytest` later on (which runs the full suite of tests).
7. I used the online FEN creator. to create specific test configurations. This helped me isolate and test various behaviors such as end-game checkmate scenarios.
8. I used the standard "material heruistic" for most of my tests, but I also implemented a composite heuristic that promotes states that lead to increased move options. The idea here was to promote development of the board. The effectiveness of this approach is discussed in the evaluation section.
9. For my implementation of iterative deepening, I used threading to allow the user to cut off the deepening process at any point and have the AI return the best move found so far. Iterative deeping is the best AI to play against for this reason - it'll match the pace of the game that you prefer.

Evaluation

Thankfully, the algorithms work! I coded a series of tests to validate expected functionality, which I will describe individually:

`test_onemovecheckmate()`

```
. . . . . . . k    . . . R . . . k
. . . . . . p p    . . . . . . p p
. . . R . p . .    . . . . . p . .
. . . . . . . .    . . . . . . . .
. . B . . . . .    . . B . . . . .
. . . . . . . .    . . . . . . . .
. K . . . . . .    . K . . . . . .
. . . . . q . .    . . . . . q . .
  (1) d6d8           (2) checkmate
```

White's turn to move. This validates that the AI will take a checkmate when it can. The black queen on the bottom row (which the white bishop could take) is a red herring.

`test_twomovecheckmate()`

```
. . . . . . . k    . . . . . . . k    . . . . . . . k    . . . . R . . k
. . . . . p p p    . . . . . p p p    . . . . . p p p    . . . . . p p p
. . . . . . p p    . . . . . . p p    . . . . . . p .    . . . . . . p .
. . . . . . . .    . . . . . . . .    . . . . . . . p    . . . . . . . p
. . . . . . . .    . . . . . . . .    . . . . . . . .    . . . . . . . .
. . . . . . . R    . . . . R . . .    . . . . R . . .    . . . . . . . .
. K . . . . . .    . K . . . . . .    . K . . . . . .    . K . . . . . .
. . . . . . . n    . . . . . . . n    . . . . . . . n    . . . . . . . n
  (1) h3e3           (2) h6h5           (3) e3e8           (4) checkmate
```

This is similar to the last test, but one move away. With $depth \geq 2$, white chooses the checkmate. When tested with $depth = 1$ (`test_twomovecheckmate_onedepth`), white instead takes the black night at h1.

`test_chooses_promotion`

```
. . . . . . . .    . . . . . . . Q    . . . . . . . Q
. k . . . . . P    . k . . . . . .    . k . . . . . .
. . . . . . . .    . . . . . . . .    . . . . . . . .
. . . . p . . .    . . . . p . . .    . . . . p . . .
p . p p . . . .    p . p p . . . .    p . . p . . . .
. p . . . . . .    . p . . . . . .    . p p . . . . .
. . . . . . . .    . . . . . . . .    . . . . . . . .
. R . K . . . .    . R . K . . . .    . R . K . . . .
   (1w) h7h8q         (1b) c4c3          (2a) h8g7
```

This tests if the AI can prioritize promoting a piece and play defensively. If the

white rook moves to start capturing black's pawns, it will release a black pawn to get promoted. The white AI ends up promoting the queen and leaves the rook in place to prevent promotions of black's pawns. Black advances the pawn c4c3. From then on the white AI keeps the black AI in check and the game eventually ends in a draw.

`test_alphabeta_vs_minimax_starting`

This test compares the performance of Minimax search with and without alpha-beta pruning. The test asks the AIs to compute a move from the starting chess position - a good performance test, since there are many pieces to consider and many moves possible from this initial position. The results, as shown in the following table, clearly show the a performance advantage for the alpha-beta AI. Additionally, we can verify that alpha beta always returns the same move as minimax for the same depth, but expands fewer nodes while doing so. Lastly, as expected, both AIs examine all 21 possible moves for depth = 1.

AI	Nodes Examined	Computation Time	Move Chosen
Minimax (d=1)	21	0.008s	g1h3
AlphaBeta (d=1)	21	0.008s	g1h3
Minimax (d=2)	421	0.14s	g1h3
AlphaBeta (d=2)	60	0.05s	g1h3
Minimax (d=3)	9323	7.21s	e2e3
AlphaBeta (d=3)	1771	1.44s	e2e3
AlphaBeta (d=4)	1495	0.79s	g1h3
AlphaBeta (d=5)	21060	7.66s	g2g3
AlphaBeta (d=6)	208037	97.69s	g1h3

A similar test, `test_alphabeta_vs_minimax_2movecheckmate`, finds an even larger difference in performance when a checkmate is possible within two moves - in this case, Minimax (d=4) examined 11397 nodes and took 3.35 seconds while AlphaBeta (d=4) examined only 247 nodes and took only 0.11 seconds.

`test_move_reordering`

Test move-reordering examines the effect of reordering moves so that captures are examined first. Using a midgame board configuration, we can see that re-ordering the moves dramatically reduces the number of nodes examined and slices the computation time by 2/3. Note that the move reordering does come at a cost of more computation time per node (but this is offset by the decrease in number of nodes). |AI|Nodes Examined|Computation Time| |-|-|-| |Alpha-Beta (d=4, with move reordering)|3005|1.31s| |AlphaBeta (d=4, without move reordering)|18619|4.64s|

`test_heuristic_change`

This test examines the impact of changing the heuristic. Using a board at

the staring position, the standard `material` heuristic is compared to a composite heuristic which also takes into account the number of moves available to the player (and encourages positions with more available moves). From the results, we can see that the computation of possible moves for all leaf nodes leads combined with a larger number of visited nodes leads to significantly longer computation time for the more advanced heuristic. However, a more development oriented move (moving a pawn forward) is ultimately selected. Going forward, I'd try to develop heuristics that are still insightful and valid, but more computationally light.

AI	Nodes Examined	Computation Time	Move Selected
AlphaBeta (d=4, material heuristic)	1495	0.81s	g1h3
AlphaBeta (d=4, composite heuristic)	5763	5.27s	e2e3

Iterative Deepening Test

I tested my iterative deepning impementation by playing against it. The code does select different moves when given time to search through larger depths. The following is example output which shows the move selection changing when depth is increased from 1 to 2:

```
`r n b q k b n r
p p p p . p p p
. . . . p . . .
. . . . . . . .
. . . P . . . .
. . . . P . . .
P P P . . P P P
R N B Q K B N R
---------------
a b c d e f g h

Black to move

Starting Iterative Move Search
Press any key + enter to stop move searching...
Move found (d=1): f8b4
Now searching at d=2
        Reminder: Press any key + enter to stop move searching...
Move found (d=2): d8g5
Now searching at d=3
        Reminder: Press any key + enter to stop move searching...
Move found (d=3): d8g5
Now searching at d=4
        Reminder: Press any key + enter to stop move searching...
```

```
Move found (d=4): d8g5
Now searching at d=5
        Reminder: Press any key + enter to stop move searching...
e
Iterative Alpha Beta Done
        Final depth: 4
        Final move: d8g5
r n b . k b n r
p p p p . p p p
. . . . p . . .
. . . . . . q .
. . . P . . . .
. . . . P . . .
P P P . . P P P
R N B Q K B N R
---------------
a b c d e f g h`
```

Discussion Questions

- Minimax and Cutoff Test
  - `test_alphabeta_vs_minimax_starting` examines the speed of minimax at various depths. We can see that the number of nodes explored increases quickly with the depth of the search. In all tests with the non-pruning minimax AI, the maximum depth of the search was equal to the cutoff depth of the AI.
- Evaluation Function
  - I found that using the material value heurstic did lead to intelligent play. As discussed in `test_onemovecheckmate`, `test_twomovecheckmate`, and `test_chooses_promotion`, the AI makes reasonable decisions when put in these situations and always takes a checkmate when one is available within the search cutoff depth. Increased depth does generally lead to more intelligent moves - for example, the insufficient depth in the `test_twomovecheckmate_onedepth` test cases the AI to miss an easy checkmate.
- Alpha Beta Pruning
  - As discussed in the above test sections `test_alphabeta_vs_minimax_starting`, enabling alpha-beta pruning yield much faster search times, especially in cases where a good move (like a checkmate) is available. The pruning seems to increase the searchable depth by about 2.
  - As discussed in the `test_move_reordering`, implementing move reordering significantly speeds up the search.
  - In all cases, an alpha beta search yielded the same move of a minimax search at the same depth.
- Iterative Deepening

- My iterative deeping implementation leads to a very usable chess interface. I used the command-line interface for making the moves, but it also works with the gui version. As discussed in testing section, the iterative process does lead to different and better moves as the search depth increases.

Sources Consulted

- https://www.youtube.com/watch?v=l-hh51ncgDI
- How to represent an infinite number in python
- Chess piece values
- Heuristic Ideas