Anthony Poerio
adp59@pitt.edu
University of Pittsburgh
CS1571 – Artificial Intelligence
Homework #02
Game of the Amazons – Report

Project Overview

For our second assignment in Dr. Hwa's AI class, our goal was to create an intelligent agent able to automatically play a chess variation called "**Game of the Amazons**".

Very broadly, **Game of the Amazons** is played by placing 4 queens on a chess board, and moving them in alternating turns. After each move, the newly placed queen has the additional task of shooting an '**arrow**' into some location on the board. The cell on the board in which the arrow lands becomes unusable thereafter.

Using these game mechanics, *each player's goal* is to **minimize** the amount of space in their opponent's queens can move. Once all queens are fully isolated (either by arrows or other queens), the game ends and we count the total amount of area in which each side *could* move. The player who has more free space at the end of the game wins.

Approach

To start tackling this problem, my initial approach was to divide the problems into a series of smaller problems, solve each, and compose the solutions while passing the necessary data along at each step, until I was able to obtain the 'optimal' move for each board position. (Note that each move is only 'optimal' from the point of view of algorithm's heuristic, as far forward as I could calculate it. A truly optimal move wouldn't always be practical to find, at least in a reasonable time period.)

Steps:

1. Read in the board position
2. Find locations of all queens during the next ply
3. Finding all possible moves each queen can make (including arrow shots)
4. Construct a search tree of depth 2-ply into the future (1 whole 'move')
5. Running a Minimax-AlphaBeta Search on that search tree, evaluating each board position using a heuristic function
6. Getting the optimal move (gain, based on whichever heuristic value function is defined)
7. Return the optimal move

In general, this approach was straightforward and it worked reasonably well. But the huge state space of a 10x10 board presented many practical problems, as will be discussed going forward.

Branching Factor

I quickly learned that because of the huge state space inherent in this problem, the branching factor from each node could be in the thousands, very quickly.

Even generating all possible nodes for 1-move on a 10x10 state space with 4 queens in a reasonable time-frame is **impractical** unless the algorithm itself was extraordinarily efficient, and unfortunately my algorithm didn't achieve that level optimality.

While testing and refining my algorithm, I coped with this by using smaller state spaces—boards of 5x5, 6x6, and 7x7, to ensure that everything worked properly and that AI performed reasonably well.

This is a reasonable fix for testing purposes, but as far as a production-level fix, it falls short. It simply doesn't address the problem of finding a way to generate all the moves more efficiently on a 10x10 board.

One early solution I tried was to only generate 1-ply ahead, and use a greedy algorithm, picking the best move that my player can make at any given time—without regard to how my opponent could counter. This actually worked reasonably well, as it cut the generation time in half, and it didn't require running a full alpha-beta search (I just used the min() or max() functions in python, depending on my heuristic). But I felt that it went against the spirit of the project's goals.

Still, what it comes down to is that we have really only have one option to deal with branching factor. We must generate fewer states overall, and therefore perform less computation. For my algorithm, the problem is that I create a full level of a search try at each step. And so the Minimax-AlphaBeta algorithm's runtime isn't what's holding me back. *It's enumerating the states that the Minimax needs to search through.* I think that a more intelligent and optimized algorithm could incorporate the actual creation of each state (or not) into the Alpha-Beta pruned Minimax algorithm. But because that would require re-architecting my whole code based, I decided this was not practical late in the project.

Time Limit

One of the most difficult aspects of this program is the inclusion of a time limit. Of course, given unlimited time, we could construct a search tree of any depth and essentially evaluate *all* of that state space. But practically, we need a cutoff that's reasonable given the problem at hand.

In this case, we need to make a decision in a matter of seconds so that a human can reasonably play a game with the AI without waiting half an eternity to evaluate *every possible eventuality* that the game could conceivably take.

To take the time limit into account, I decided to depth limit my search tree to depth=2. I also implemented a fully Alpha-Beta pruned Minimax search to find the best possible state of this tree.

Ultimately, though, even these measures might not be enough for the 10x10 board. The biggest problem really is at the beginning of the game, when there are just so many possibilities and such a huge branching factor.

I did not have time to implement this, but given more time, I would truncate the search tree to perhaps 1-level early in the game. At this point, looking further ahead isn't a huge strategic advantage because the board is so wide open, and so I think that the game-play quality wouldn't suffer much. And it might make the 10x10 board playable.

Interestingly, we have 2 competing ideas of optimality for path-finding problem. 1) Number of nodes traversed; and 2) Total cost of those nodes. Unicost, BFS, and the complete search algorithms that do not account for weights (with a heuristic) performed the best with respect to number of nodes, while those algorithms that used a heuristic (informed search) outperformed with regards to total lengths here. Because cost is more important to the path planning problem, I would suggest that we want to use an informed search for this problem – A* and ID-A* performed the best from the informed search group.

Heuristic

I tried two different heuristic functions—an agent that plays offensively, and another agent that plays defensively.

For both agents, the meat of the function is finding the number of available spaces within which the opponent can move. I leveraged the code for counting squares in the Board class to do this.

From there, I essentially had two options:

1. Maximize the space that my agent has available each turn (i.e. – self preservation, defensive tactics)
2. Minimize the space that the opponent's queens have available (i.e. – attacking, offensive tactics)

For the **defensive** agent, we are just using the raw-number for my color's queens returned by the count area-function and finding the MAX of that in our alpha-beta minimax search.

For the **offensive** player, I took the number of spaces that my opponents queens had to move within, negated it (i.e. – 10 → -10), and ***then*** found the max of those negative numbers. In this way, the lowest numbers would have the highest weight.

Both ideas worked reasonably well. Generally, I found the defensive player to perform better in practice. The offensive player made more 'silly' mistakes, on account of its being overly aggressive—it would frequently overextend its position and crowd its queens, making them easy targets to surround in just a couple of moves.

The defensive player was more challenging to play against, and even beat me once or twice on the small board (though that was early on and I didn't have a great feel for optimal strategies yet). Moreover the defensive players games were often much longer, and so if it is playing against other algorithmic players, I feel more comfortable letting *them* make silly mistakes, rather my AI. I guess **when we only have time to look one or two moves ahead, it's really more to your advantage to play it safe.** So in the end, I chose to submit he **defensive agent**.

Versus Humans

In order to test my agents, I played a series of games against them—both as black and as white. Representative transcripts of these games can be found in the folder **/transcripts/part2** within main source directory.

For each set of games, you can see the board size (organized by folder), and each .txt file has the side played by my AI agent identified in the filename.

I found that when I played very aggressively and sought to crowd my AI's queens and their positions I was able to beat it much more easily—no matter which agent was used.

If I gave the AI space to move within, the agents performed much better. Presumably this is because they had more states to evaluate and could therefore choose more optimal positions—but I think maybe that crowded the opponent's space is just a better strategy for the game overall.

Given more time, I would write some functions to help the algorithm play in a style based on what I found to be beneficial in my own playing, but that's more of a stretch goal for future improvement.

Overall, the AI performs qualitatively better on smaller boards, and also when it has the second move (black). Honestly, I think that's because it helps keep the AI out of trouble early on. When it plays as white, it seems to overextend itself too quickly and leave at least one queen out to be an easy target (and both agents are guilty of this).

Given more time—and thereby the opportunity to explore more of the possible state-space for each position—I believe these agents would both do much better. But given the time constraints for this project, I'm reasonably happy with how everything turned out.

The AI plays the game well enough to be competitive (though I almost always beat it), and playing the game against was actually fun. For an algorithm written in a class, I suppose I can't ask much more than that. So, all in all, I'm happy. Thanks for taking the time to read through.