April 9, 2017
COSC 322 - Artificial Intelligence
Prof. Yong Gao
Team 6

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

COSC 322 AI Project - Final Report: Game of the Amazons

Project Background

The goal of this project is to develop a Java based AI for the Game of Amazons, and to compete in matches against the AI of other teams. It is intended that we implement techniques from our AI studies such as heuristic evaluation functions and maximizing utility to design a smart AI.

Software Architecture

In the table below there is an abbreviated list of the classes created to represent the AI designed to play the Game of Amazons.

Class	Type	Description
AmazonBoard	N/A	This class represents the game board for the Game of Amazons. The structure uses a 12x12 array to hold individual tiles. Outer row is initially filled with arrows to reduce complexity
AmazonBoardCalculator	N/A	This class contains all functionality for calculating valid moves and the strengths of squares on the current board.
AmazonSquare	N/A	Class defines a single square of the board for Game of Amazons. Holds data of its current strengths and distances, position on board, its occupying queen or arrow, and if it's captured or not
AmazonEvaluator	Abstract	Evaluators implement this class - must override the evaluateBoard function
AmazonIterations	N/A	Keeps track of iterations for initial minimax evaluator
AmazonMove	N/A	This class defines a move that can be made in the game of amazons - initial, final, and arrow shot
MinimaxEvaluator	Extends AmazonEvalutor	Initial Minimax evaluator using alpha-beta pruning.
BestMobilityEvaluator	Extends AmazonEvalutor	Class that evaluates the board based on the best difference between max mobility and current mobility
InvalidMoveException	Extends Exception	Exception class thrown when an invalid move is made by opposing player (or caught when evaluator tries to return invalid move)
MaxMobilityEvaluator	Extends AmazonEvalutor	Class that evaluates the board based on maximum mobility.
NodeMinimaxEvaluator	Extends AmazonEvalutor	Class that contains final implementation of evaluator which uses minimax and alpha-beta pruning
NodeMinimaxLiteEvaluator	Extends AmazonEvalutor	Same class as above, but is slightly more efficient, as it stores moves to execute/undo, rather than cloning the game board.

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

RandomEvaluator	Extends AmazonEvalutor	Evaluator class that plays valid random moves
AmazonPlayer	Abstract	Players implement this class - designed such that instantiating a player, starts a game. Contains the references to UI and board, and handles messaging to the game server
AmazonNode	N/A	Class represents a node in the game's search tree
AmazonAIPlayer	Extends AmazonPlayer	Class that plays the game. Joins game server, handles game messages, and is instantiated with chosen evaluators to determine it's 'AI'
AmazonConstants	N/A	Data class that holds game rule and configuration values
AmazonGameClient	Extends GameClient	Extension of GameClient class that connects to game server and packages up the players move into accepted format for game client for sending
AmazonBoardUI	N/A	Class that creates visual display of game board. Contains heat map of distance functionality, and timerUI and handles repaints
AmazonUI	N/A	Class that provides frame for visual representation of the game

Overall Design

This project was composed of a few major categories of problems to solve – game logic, server communication, scoring methods and search strategy.

Game logic

Representation of the game state was an important problem that would guide our entire project. The initial implementation was just a simple 10x10 2D array that would hold an integer corresponding to the piece that was in the square (queen, arrow or open). We expanded upon this by storing a representation of the square in the array, instead of just the piece value. Within our square structure, we would hold the co-ordinates, piece type, and all the metrics that we would use to calculate the positional strength of the square. Most of these calculations are in exponential time, so we don't want to be repeating them for each evaluation. This allowed for significant flexibility when implementing our scoring calculations, and minimized the headaches when dealing with direct array manipulation.

Determination of acceptable moves is the key component to making this game work. Some of the groups had issues with accidently deleting Amazons off the board by overwriting the piece type in the square. Our system had multiple redundancies to prevent this type of situation from occurring. From each Queen, it would generate a list of possible moves based on iterative expansion in the 8 directions of movement, and halting in that direction once an unavailable square is found. When a spot was chosen in that list, it ran the algorithm again to generate a list of possible arrow shots from that position. We could use these lists to determine the validity of a sent or received move.

Server communication

The allowable communication to the server was minimal. There were messages to code for players joining a room, game start and color assignment, player movement, and user logout. When our game was launched, we would join a room and wait for a message to signal another

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

player had joined the same room. The server would assign colors, and if we were black, we would start our evaluation. When our timer expired, we would send our move to the server, and wait for the opponent to complete their turn.

A feature we would have liked to implement would be some sort of generic message system. This would allow us to customize messages and create an automated framework that would automatically play the game without any user input. The games would play until completion, then send a message to reset the board and begin another game. The limited message in use by the server made this difficult, but we did experiment with storing messages in a specifically crafted move. The move consisted of 6 int values corresponding to the co-ordinates of the initial, final and arrow shot position. Using bit operations, we could convert a string into these int values, and have it sent through the send move message. The encoding and decoding proved to be a hassle, and we decided to focus more on improving our AI aspects.

Scoring methods

Evaluation of the current state of the board would calculate a numerical score based on the strength of positions. Our methods will be described in detail later, but each uses some form of calculating the distance between the player and the square. Open and available squares that are closer will give a higher score to the player. Maximizing this would indicate that the player controls a large amount of the board, while the opponent controls a small amount of the board.

To overcome some of the challenges we had with using a single calculator over the course of the entire game, we altered our scoring method to consider the phases of the game. We separated the game into start, mid and end phases. The start phase was the initial moves of the game where there are no defined regions of control. When enough squares had been blocked, the mid phase would consist of filling the board and creating captured zones. The end phase was the last few moves that would decide the captured zones, and the trivial filling of those zones afterwards. In our AmazonConstants file, we could specify after how many turns would pass before the game would proceed to the next phase.

Search strategy

Our search strategy was primarily alpha-beta pruning with iterative deepening. Using the scoring methods, we could evaluate the positional strength for an instance of the board multiple turns in advance. Our implementation was effective and consistently produced the best move based on our scoring methods. We don’t think that much could be done to improve the efficiency of our search strategy. Alternate methods were researched, such as a Monte Carlo tree search, but did not prove fruitful. This will be expanded upon in the additional research section.

Software Features

Heat map

In developing our game board data structure and the initial algorithms for calculating metrics, we found it useful to have a visual representation of the current state of the board (Fig. 5). It would also show an output for the metrics, using a decimal or hex value to indicate the relative value of the square.

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

```
Piece types:   Strength:      Mobility:      White Q Dis:   Black Q Dis:   White K Dis:   Black K Dis:
XXXXXXXXXXXX   000000000000   000000000000   XXXXXXXXXXXX   XXXXXXXXXXXX   XXXXXXXXXXXX   XXXXXXXXXXXX
X   *      X   046402345640   089852368960   X222X887877X   X111X111111X   X444XDCBBBBX   X321X123333X
X   XXXX   X   069600006860   0BEB00009A90   X122XXXX776X   X211XXXX212X   X333XXXXAAAX   X321XXXX222X
X   *O XX  X   069700300640   0BED72200960   X212XX1XX67X   X111XX2XX11X   X322XX1XX9AX   X221XX2XX11X
X      X *XX   057544303000   09CA74303500   X221111X5XXX   X111112X1XXX   X221111X8XXX   X111112X1XXX
X *XOXXXX XX   030000000400   055050000300   X1XXXXXXX4XX   X1XXXXXXX1XX   X1XXXXXXX7XX   X1XXXXXXX1XX
XXOX XX   XX   000030024400   005040013400   XXXX1XX443XX   XXXX8XX221XX   XXXX1XX876XX   XXXXEXX322XX
X    XXXXX X   046640000040   08AA60000050   X1111XXXXX2X   X7888XXXXX2X   X1112XXXXX5X   XDDDDXXXXX3X
X   XX  X  X   069700540550   0BFD00760770   X212XX12X22X   X878XX55X33X   X222XX22X44X   XDCCXX88X44X
X   X   X  X   069705760760   0BFD06880980   X212X111X22X   X887X655X43X   X333X111X34X   XDCBX987X55X
X     O    X   046554045540   08AA86768760   X11111X1111X   X77766X4443X   X44321X1234X   XDCBA9X7666X
XXXXXXXXXXXX   000000000000   000000000000   XXXXXXXXXXXX   XXXXXXXXXXXX   XXXXXXXXXXXX   XXXXXXXXXXXX
```

Fig. 1 Console output of our game board data structure

However, when developing our evaluators and scoring calculators, we found it difficult to follow the moves and debug the logic behind them with the text output. To solve this, we modified our UI to color each square on a gradient of green to red, corresponding to the relative value of whichever metric we chose (Fig. 6). This provided us better analysis capabilities, as well as a pleasing visual effect.

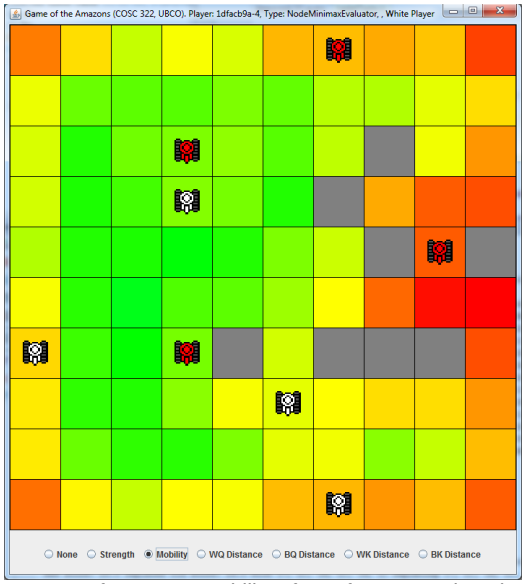

Fig. 2 Heat map for square mobility after a few moves into the game. Green is highest value, and red is lowest value.

Threading/Parallel Evaluators

In creating our timer system for moves, we decided to run the Evaluator in its own thread. This would allow us to control evaluation by starting and stopping the thread at the appropriate time. When starting, we would load the current game board into the Evaluator, and start the turn timer. The Evaluator would run its search algorithm, storing the best move as it is discovered. Once the 30 seconds were up, we would stop the thread and return the best move found by the Evaluator during that time.

We expanded on this to allow for evaluators to be run in parallel without significant coding overhead. With our modular approach to the evaluators and scoring calculators, we could simply start multiple threads with different evaluators and compare results. So long as the scoring calculator was consistent between each Evaluator, we could easily compare the sum total of scores in the sequence of moves that the Evaluator had determined to be optimal. If this were

combined with our idea for automating game play, we could do a comprehensive analysis of performance for each evaluator with different scoring calculators.

Graphics

This was more of a fun addition, rather than anything functional. Instead of Amazons shooting arrows, we decided that tanks blowing holes in the ground might make the game a bit more exciting (Fig. 7). Using free resources from OpenGameArt, we created patterns for rock and lava that would adjust to adjacent squares and create a seamless texture.

Fig. 3 Game art for 'Tanks of the Amazon'

AI Technical Aspects

A simple breakdown of how our AI works is the following:

Evaluators determine optimal moves using **score calculators** derived from various **metrics**

Metrics

For every square on the board, we run several calculations to assign values that are considered when deciding the attack and defensive potential of the square.

Moves

$$N = the\ number\ of\ available\ moves\ from\ a\ particular\ position\ via\ a\ Queen\ move$$

A very simple calculation to decide whether the square is considered desirable. A square that is nearly trapped will have less moves than one that is open. This is best used to determine potential escape routes from a square.

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

Queen's Distance (Fig. 4)

$$D_{queen} = the\ minimum\ number\ of\ moves\ it\ would\ take\ a\ player's\ queen\ to\ reach\ a\ particular\ square$$

This replicates the movement of the Amazons, but in the early game, this is not particularly useful. Until there are sufficient numbers of arrows blocking off squares, the value for this will typically be equal for both players. Once several squares are blocked off, this value will give the best estimation of positional strength.

King's Distance (Fig. 5)

$$D_{king} = the\ minimum\ number\ of\ moves\ it\ would\ take\ a\ player's\ queen\ to\ reach\ a\ particular\ square, if\ the\ queen\ were\ only\ able\ to\ move\ to\ an\ adjacent\ square\ each\ turn\ (like\ a\ king\ in\ chess)$$

A restriction on Queen's distance can make it much more effective in the early game. When few squares are blocked, King's Distance can give a better indication of positional strength. In the beginning phases, an optimal setup for King's distance would be each queen sitting half the distance from the center of the board to the outside corners. This would give strong access to each of the 4 quadrants of the board. However, in the late game, King's Distances will be very high due to corridors forming. It would be possible to have King's Distances of over 20 when the Queen's Distance was ~3 to 4; this would distort the positional strength of squares.

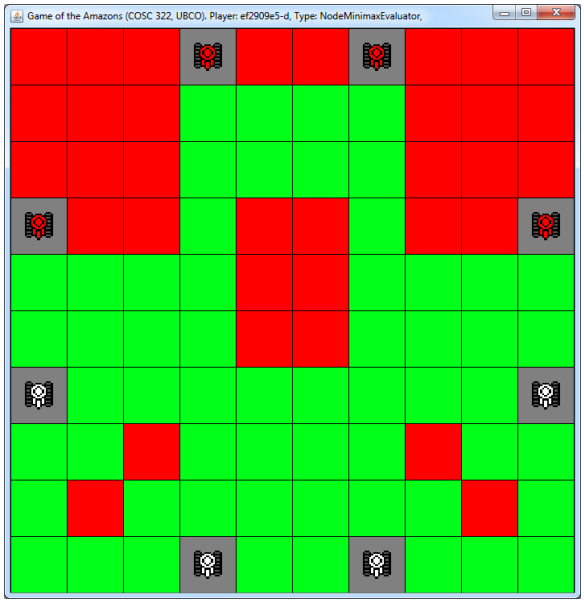

Fig. 4 Heat map of D_{queen} for the white player's starting move. Green is lowest value, and red is highest value.

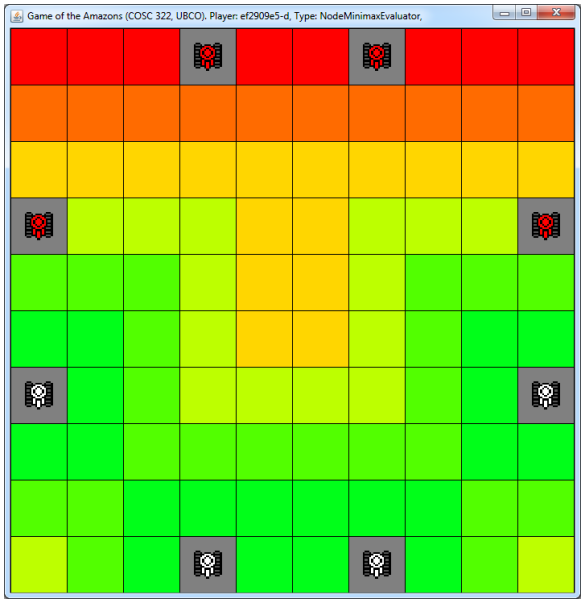

Fig. 5 Heat map of D_{king} for the white player's starting move. Green is lowest value, and red is highest value.

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

Strength (Fig. 6)

$$S = the\ number\ of\ open, adjacent\ squares\ to\ the\ square\ being\ calculated$$

This is a very local calculation of how desirable a square is.

There is limited usefulness to this metric on its own, as there would be no distinction between a completely open board and a board with only squares adjacent to the square being calculated being open.

Mobility (Fig. 7)

$$M = \sum S * 2^{-D_{king}}, for\ all\ moves\ in\ N$$

Mobility combines the metrics for moves and strength to give a better indication of how desirable a square is. A desirable square will be relatively open, but still have strong escape routes.

This expands on *S* by considering positions beyond those that are adjacent. A low mobility will indicate that you are close to being surrounded, while a high mobility indicates many decent escape routes.

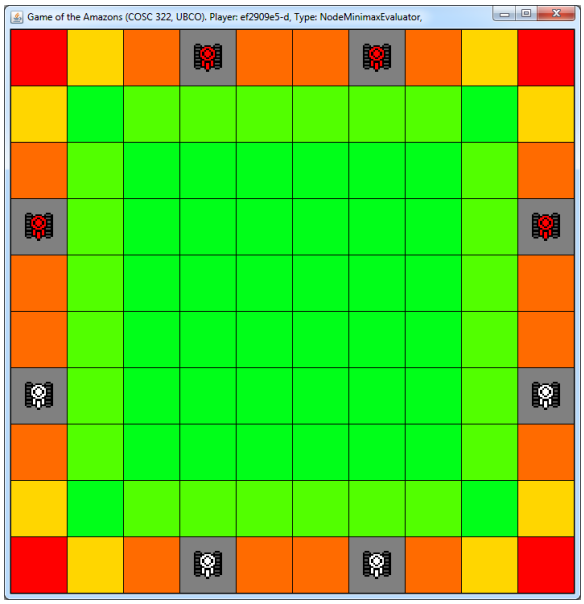

Fig. 6 Heat map of ***S*** for the white player's starting move. Green is highest value, and red is lowest value.

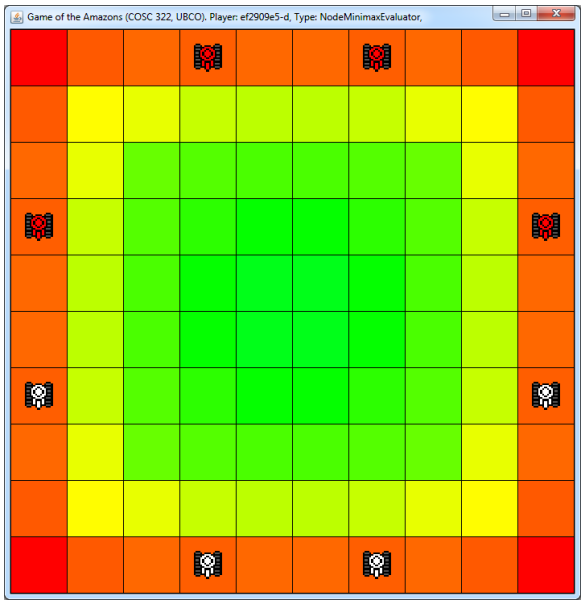

Fig. 7 Heat map of ***M*** for the white player's starting move. Green is highest value, and red is lowest value.

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

Score Calculators

Using these metrics, we created several calculators to utilize them. All the calculators will return an integer array containing the scores for white and black. This allows more flexibility to the evaluators when doing a comparative analysis of the scores.

Movement Score

$$Score = \sum_{\substack{Squares\ occupied \\ by\ the\ Player's \\ Amazons}} N$$

The only advantage to this calculator is speed. At most, it will need to examine 4 * 3 * 9 = 72 squares, since there are 4 queens and a max of 9 possible moves for movement in the vertical, horizontal and diagonal directions. Other calculators will end up examining each square multiple times since there is exponential growth when calculating distances. It gives no thought to capturing areas, so it could be thought as more of a brute force attack, rather than strategic.

Terrain Score

$$Score = \sum_{Empty\ squares} \Delta\left(D_{Player}, D_{Enemy}\right), where\ D\ can\ be\ D_{Queen}\ or\ D_{King}\ and$$

$$\Delta(P,E) = \begin{cases} 0 & if\ P = E = \infty \\ K & if\ P = E < \infty \\ 1 & if\ P < E \\ -1 & if\ P > E \end{cases}$$

In simple terms, gain a point if you closer to a square than your opponent, and lose a point if you are further. If the distance is equal, the value *K* estimates the advantage of player who moves first. This estimates how many squares are under the 'control' of a player. Since they are closer a square, it should be easier for them to prevent the opponent from reaching it.

Relative Terrain Score

$$Score = \sum_{Empty\ squares} \Delta\left(D_{Player}, D_{Enemy}\right), where\ D\ can\ be\ D_{Queen}\ or\ D_{King}\ and$$

$$\Delta(P,E) = \begin{cases} 0 & if\ P = E = \infty \\ K & if\ P = E < \infty \\ J & P < \infty, E = \infty \\ -J & E < \infty, P = \infty \\ E - P & if\ P < E \\ P - E & if\ P > E \end{cases}$$

Similar to the terrain score, but this will give points based on the relative difference in moves to a particular square (i.e. if your D_{queen} is 2 and the opponents is 5, you get 5 – 2 = 3 points). Since it takes the difference, it gives a better estimation of relative control than the Terrain score. This would give incentive for the player to barricade areas to reduce the opponent's score. If the square was captured, the value J would allow you to

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

change the weighting. A captured square should give preference over a contested square, but we found that too high of a value causes the AI to act hyper-defense, and immediately start to enclose small areas at the risk of being enclosed by the opponent. We found the best results when this value was between 3 and 5.

Mobility Score

$$Score = \sum_{\substack{Squares\ occupied \\ by\ the\ Player's \\ Amazons}} M$$

This works somewhat opposite to the Relative Terrain score, as it would give more incentive to trap the opponent, rather than secure territory for itself. This is helpful in the early game, since it is unlikely you will be able to enclose any significant territory. In the late game, this score is not useful, as moves with a distance above a couple squares become increasingly uncommon.

Kd – Qd – Mob Score (King's Distance, Queen's Distance, Mobility Score)

This was an attempt to combine multiple metrics. Numerous iterations were tested using both terrain and relative terrain scoring for the King's and Queen's distances, and using relative and absolute mobility values (player mobility – opponent mobility, and just player mobility, respectively). There were difficulties in trying to weight each particular score, as they could greatly vary over the course of the game. Mobility would range from around 250 in the early game, to around 20 in the mid-late game. Queen's Distance stayed relatively consistent, averaging around 3 per square in the early game to around 5 in the late game. King's Distance was quite erratic in the mid-late game, as barricading could massively increase distance with a single move. Obviously, we wanted to maximize mobility, and minimize the Queen's and King's distances, but we were unable to find a weighting that would consistently produce good results.

Amazong Score

This was an attempt to implement an evaluation function created by Jens Lieberum. However, we did not get good results, as there were significant optimization problems with the weighting/normalization of the components that would need to be solved before the algorithm would be effective. A detailed breakdown of the algorithm is included in the Additional Research section.

Evaluators

The evaluators contained the actual AI of the project.

Random Evaluator, Best Mobility Evaluator, Max Mobility Evaluator

These evaluators were our early test evaluators when creating our server communication. They simply found a random move, or did a simple evaluation of mobility for only the current turn. There is no AI in these evaluators.

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

MinimaxEvaluator

The function starts by copying the state of the board at the beginning of the turn, and then performs an iterative deepening alpha-beta search until either the depth reaches 15 or the thread is told to wrap up. After each depth is finished, the function finds the move with the best possible score and sets it to bestCurrentMove.

Inside the alpha-beta function there are several conditions that allow the function to exit gracefully and return the last found best score. The alpha-beta function itself iterates through every possible move at the current state of the board, then performs the move before traversing to the next depth. Each move is kept track of, and after the max depth is reached, the move is undone.

At the max depth, the score is calculated per whether it was the opponents move or the current players move, with the prior having its sign reversed. To cut back on calculated moves, saving exponential computation time, the moves are arranged in descending order of mobility before calculating the follow up moves with alpha beta.

NodeMinimaxEvaluator

The NodeMinimax evaluator follows the same design as the regular Minimax Evaluator, but creates a node at each move that contains the move and current state of the board. It's heavier on memory but when the depth is increased it uses the nodes calculated in the last run through. This way, we can order the list of next moves based on their previous scores, which cuts off branches much quicker in the alpha beta function.

NodeMinimaxLiteEvaluator

This evaluator uses the same logic as NodeMinimaxEvaluator, but does not clone the game board for each node. It only keeps the move created the node, and the original board can be modified with executeMove or undoMove. It is theoretically more memory efficient, since each board will have a list of all 100 squares, but a move consists of only 3 squares. However, in testing, there was no significant improvement. It is likely that our AI was CPU limited, rather than memory limited, or it could be related to how the JVM allocates resources.

Additional Research Effort

Monte Carlo - Tree Search

We looked at implementing an additional game-tree search technique that combines the Monte Carlo method of randomized searching and tree search. Monte Carlo is a game independent algorithm that determines more advantageous moves by playing out whole games after a move is made and seeing its outcome. As more simulations are ran, the MC method becomes more effective.

In relation to the Game of Amazons, we would have needed to essentially implement an algorithm that accomplishes the following:

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

1. Initialize the tree search used in the evaluation to the root itself and then select a new node of the game tree, containing its evaluation, to be added later to the current search tree.
2. Then a random game (or not random for a stronger MC evaluation) is performed from the given selected node until the end of the game. This provides an evaluation for the node that is generally a win/loss/draw value or the score of the game (captured tiles)
3. Then, add the given node to the current search tree. Then keep on exploring more nodes

To maximize the results of the MCTS method, the score of a node needs to be averaged over the node's evaluations from the above algorithm and a portion of how many times it has been visited as a weighting.

For the most effective results, the Monte Carlo method would have been combined with our other evaluation function of node minimax search to make the algorithm more efficient and reduce the randomness of its results.

Evaluation Function for Game of Amazons by Jens Lieberum

Outlined in this paper is the general AI strategy that Jens Lieberum's Game of Amazons AI, 'Amazong' uses to play the game. It guided a significant portion of our direction and gave us many different heuristics to test results with.

Outlined in the paper is the use of territorial and positional evaluation and the mobility of individual amazons. The heuristics described are all based on the distances described as the Queen's distance and King's distance, which are defined below:

- $d_1(a,b)$ of two squares a and b is the minimal number of chess queen moves needed to go from a to b. When no path exists $d_1(a,b) = \infty$
- $d_2(a,b)$ of two squares a and b is the minimal number of chess king moves needed to go from a to b
-

Clearly, $d_1(a,b) < d_2(a,b)$. The distance of player j from square a is then given by:

- $D_i^j = \min\{d_i(a,b) \mid the\ square\ b\ is\ occupied\ by\ an\ amazon\ of\ player\ j\}$

The idea behind this concept is that $D_i^2(a) < D_i^2(a)$ indicates that player 1 has better access to the square *a* than player 2 does. D_i^j is used in the formula for the evaluation functions t_1 and t_2, both territory based evaluation functions.

$$t_i = \sum_{Empty\ squares} \Delta\left(D_i^1, D_i^2\right), \qquad where$$

$$\Delta(n,m) = \begin{cases} 0 & if\ n = m = \infty \\ K & if\ n = m < \infty \\ 1 & if\ n < m \\ -1 & if\ n > m \end{cases}$$

Both territorial evaluation functions perform quite well, especially right before the filling phase, however t_2 becomes less and less effective as the game progresses. But they both also suffer at the very start of the game because large positive differences have the same calculated score as

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

small positive differences, despite them providing evidence of a larger advantage. Due to these shortcomings, it is important to use other local evaluation functions at some points in the game that differentiate between smaller and larger advantageous positions, which they call c_1 and c_2. These are determined initially by statistical theory and game strategy but ultimately trial and error determines good choices for c_1 and c_2. The reason multiple local evaluation functions are needed is because at different stages of the game each functions' importance, and thus their weighting, are vastly different and changing.

From the formula of c_1, one can see that precise differences in distances will result in strength values for player 1 of {0.25, 0.5, 075, or 0}. These happen in cases such that $(D_1^1(a), D_2^1(a))$ are equal to {(2, 3), (1, 2), (1, 3), (n, n)} respectively. Unlike c_1, the formula for c_2 shows us that only large distance differences show a clear advantage for one player since it depends solely on the difference of $D_2^2(a) - D_2^1(a)$.

By combining t_i and c_i into one evaluation function, the evaluation function t is made to be defined as:

$$t = f_1(w)t_1 + f_2(w)c_1 + f_3(w)c_2 + f_4(w)t_2, where$$

$$w = \sum_{Empty\ squares\ a} 2^{-|D_i^1(a) - D_i^2(a)|}$$

$$And, (f_i)_i\ is\ a\ partition\ of\ 1\ which\ weights\ the\ strongest\ portion\ of\ t\ at\ specific\ points\ in\ the\ game$$

Because the strengths of each distance based evaluation function changes as the game progresses, a typical weighted sum function w_s, where static weights are defined for each evaluation function is not appropriate. For this reason, the weight function w is used to emphasize the relative strength of each evaluation function at the appropriate stage of the game.

The mobility of individual amazons is used to have the AI play offensively and try to enclose opposing amazons inside small regions at the beginning of the game, before the filling phase has started. The mobility, m, is used as a correction term for the sole evaluation function t, to cause the AI to play aggressively at the start of the game. This is needed because enclosing opposing amazons doesn't necessarily cause a significant change in t, especially in the beginning moves of the game. Without a noticeable effect on t, the AI would not attempt to reduce opposing amazons' territory by enclosing them, thus the mobility correction term m is required.

The value of mobility is derived from considering the amount of open squares accessible by one chess King's move from square b, or what they define as $N(a)$. While traversing the game search tree, the value of $N(a)$ can be computed at each new node and used in calculating $\propto_A$ for an amazon *A* of player *j* on square *a*, where all squares b are chosen by $d_1(a, b) \leq 1$ and $D_1^{3-j}(b) < \infty$.

$$\propto_A = \sum_b 2^{-d_2(a,b)} N(b)$$

If $\propto_A = 0$, we can safely say that Amazon A is enclosed and has no moves available

Using the weighted function w from the territory calculations, we can now define the mobility correction term *m* as:

Jeff Soloshy
Drew Lindsay
Jimmy Huynh
Robin Myhr
Zifang Jiang

$$m = \sum_{B} f(w, \propto_B) - \sum_{A} f(w, \propto_A)$$

$$For\ a\ suitable\ function\ f\ \geq\ 0$$

The final choice of f for use in the evaluation function $t\ +\ m$, is the most difficult optimization problem of its conception and is determined by restricting f to known properties that don't change across tests, mainly $f(0, y)\ =\ 0$ and $(\frac{df}{dx})(\frac{x}{y})\ \geq\ 0$.

Team Contributions

Jeff	AI implementation and testing
Drew	Game logic, server communication, graphics
Robin	Reports, additional research, UI
Jimmy	Emotional support
Zifang	Additional emotional support

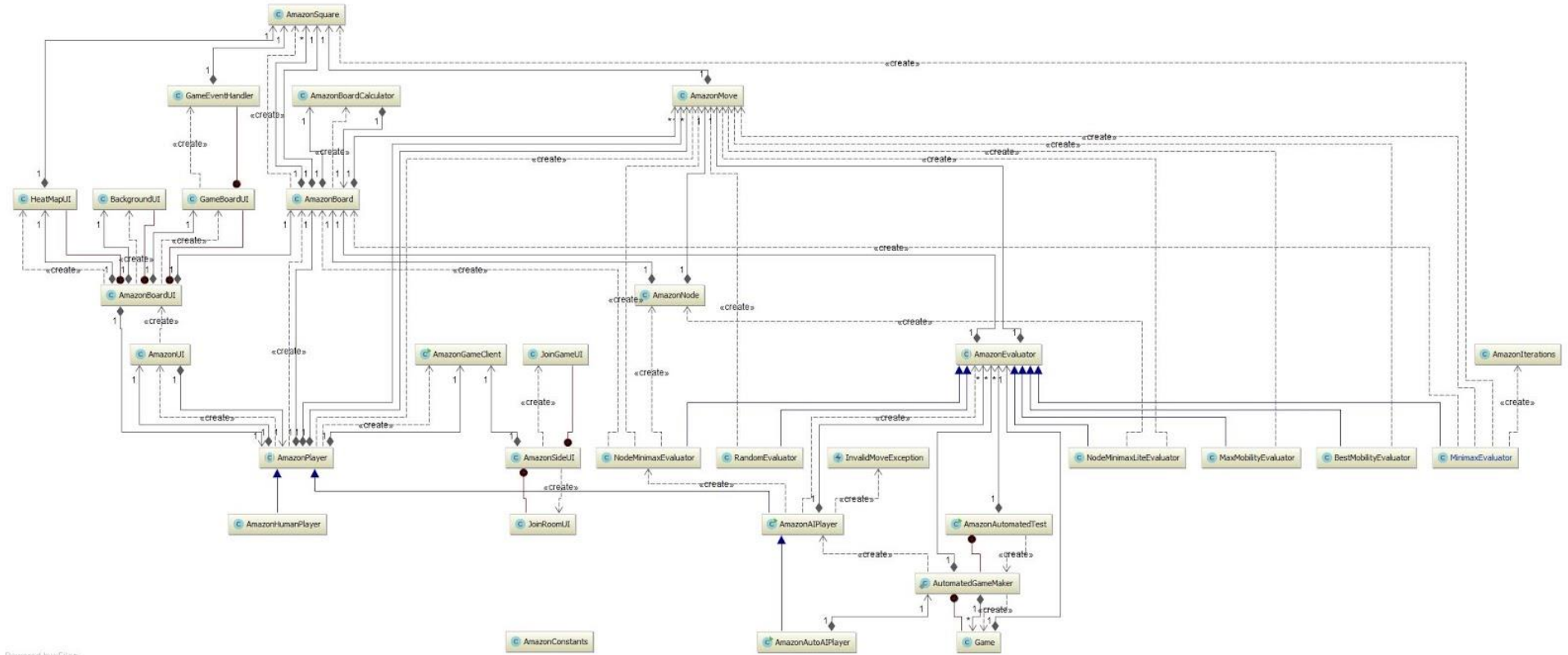

Fig. 10 UML diagram of the project