Amazons AI Report

April 10, 2015

Mathew Levasseur

Stuart King

Maddie Cunning

Erin van den Brink

Jordan Romero-Porter

Introduction

The lifecycle of this project has seen numerous revisions to our original plan as the program was being developed. The original goal was to implement a modified version of Monte Carlo Tree Search (MCTS) with Upper Confidence Bounding (UCB), but this proved to be too weak in the opening stages of the game. Instead we applied the Amazong[1] heuristic with an α-β minimax tree search with much better results. Because the heuristic evaluation is computationally expensive it wasn't feasible to do a search depth greater than 2-ply, and while this proved to be fine in the opening and early-middle game phases it consequently had a much weaker late game. Finally, we adopted a combination of the two searches: Amazong + minimax for the first half of the game and MCTS for the remainder. By incorporating both algorithms for their respective strongest game phases, the program's performance was significantly enhanced and scored 14-1 (7-0 match wins) in the tournament.

General Program Design

The program Is broken into separate components: the GUI, the internal game board (including evaluation heuristic), the AI (search algorithms only), and the main server connection.

Graphical User Interface

A modified chess GUI[2] that displays a 10x10 board with white on the bottom, black on top, labeled y-axis from 0-9 (for ease of use when reading coordinates sent from the server), x-axis labeled a-j, black and white chess queens, and colors "arrow-shot" squares red. Includes a function to read an array of coordinates and update the board.

Gameboard

A 2-D array of Gamepiece objects. All of the different aspects of the board are represented as objects and all extend Gamepiece; Blank squares, White and Blackqueens, and Arrows. Gamepiece stores the numerical coordinates of the piece and has functions to return its position in either string or numerical format. The choice was made to make the board an object because it allowed for easy evaluations that would require scanning the entire board over and over again otherwise.

Blank:

- Stores how many Blank neighbours surround it.
- Maintains the minimum queen moves (queen distance) and king moves (king distance) for black and white.
- Returns several local heuristic scores to be combined for a global score.

Queens:

- Maintains its position on the board and contains a move function to update itself.
- Returns a heuristic value measuring how mobile/cut-off this queen is.

Arrow:

- An object only to follow the convention of the other pieces.
- Maintains its location on the board and a value to programmatically discern if the object at a given coordinate is an arrow.

Gameboard:

- Uses doMove and undoMove to play/unplay moves. Moves can be in string format, numerical coordinate format or in encoded integer format (see Search section for a detailed explanation).
- Checks a given move and arrow shot to see if they're legal.
- Keeps a history of all moves played so far in a stack.
- Algorithms to calculate minimum queen and king distance for each player to each square on the board.
- Calculator functions for several heuristic metrics.
- Evaluation function that consolidates all heuristics and returns a score for the current position.
- Several testing print methods to visually display different heuristic metrics on the board.

AI

An AI object is instantiated with its own instance of Gameboard and assigned a player value (1 for white, 2 for black). In this way a player can choose to play against the computer, make the AI verse itself, or compete as intended on the server. When it's the AI's turn it checks to see how far the game has progressed. If 50 or more moves have been played then MCTS is run, otherwise α-β is used. The AI has several features that didn't get implemented in time for the final release that stand as areas for future improvement. These include:

- A threaded think function that looks ahead and constructs an ordered game tree via Iterative Deepening during the opponent's turn, and passes the tree to the current search function on interrupt (should be interrupted after opponent moves).
- A 30 second cut-off timer for α-β to prevent time-outs in certain positions in the opening.
- Multi-threading to take advantage of processing power and facilitate deeper searches.
- Implementing NegaScout[3] (or Principle Variation Search) to improve α-β.
- Adding an opening book.
- Remembering "killer moves" (Killer Move heuristic - moves that cause an α-β cut-off) to search previously known good moves first.

Main program

Using Dr. Gao's sample implementation of an Amazons program[4], the main class is responsible for initially connecting to the game client, launching the GUI, parsing AI moves and handling game messages. If our program enters a room first it instantiates an AI object as a white player and begins searching for its first move, otherwise it waits for the opponent's move, instantiates a black AI object, plays the move on AI's board and runs a search. The GUI is updated after every move played.

Heuristic Algorithms and Functions

For a full description of the Amazong heuristic evaluation please read [1]. This section discusses the algorithmic techniques used to find the various metrics used in the heuristic, as well as the formulae used to dynamically weight the separate portions of the heuristic. Remember that the board is stored as an array of Gamepiece objects, and the queen locations are stored in an array as well. The algorithms exploit these facts to reduce board scanning to at most once during computation.

T1 - Min. Queen Distance (generateQueenMoves())

The metric used to score the board based on which player can reach a square the fastest using typical chess-queen type moves. In case of a tie, the player whose turn it currently is receives a small score in their favour (±0.12).

The algorithm begins by scanning in all directions from each queen's position and assigning each blank's respective queen-color distance to a 1.

0/1	0/1	B	0/1	0/1
0	0/1	1/1	0/1	0
1/1	0	1/1	0	1/1
0	1/0	1/1	1/0	0
1/0	1/0	W	1/0	1/0

The algorithm then scans the board looking for every blank square that currently has a 0 step count for a given player and stores the tiles into an array. While keeping track of how many iterations we've done, for each tile in the array we make queen-like moves and, if we find a tile that has a step count equal to our iteration count, we set the current tile's step count to (iterations + 1) and remove it from the array. We repeat this process, incrementing the iteration

count each time, until we do a run where we either haven't updated any tiles (meaning the remaining tiles are completely blocked off) or there are no tiles left in the list.
Through numerous timing and memory tests, this method proved to have a 35%-60% increase in speed and memory efficiency than a "forward first" approach (making steps from each queen, then steps from each tile with a 1-step count, 2-step count, etc. until all tiles are accounted for) at all phases of the game.

T2 - Min. King Distance (generateKingMoves())

A metric that scores in the exact same way that the T1 metric does, except this metric looks at the minimum chess-king like moves it takes for each player to reach a square. This algorithm operates in exactly the same way as the T1 algorithm as well, with the exception that instead of having to check the full length of a direction the algorithm only has to check one square away in each direction from the queen/current blank square. In order to speed this check up, testing was done by hand to see the most common directions that a square finds a neighbour it can reference to update its own step-count. Generally, in order from most common to least common directions the algorithm searches bottom-right, bottom, right, bottom-left, top-right, left, top, top-right.

c1 and c2 - Local Area Scores

Heuristic metrics that rely on Min. Queen Distance and Min. King Distance scores respectively. They reward moves in earlier phases of the game that replace cclear local disadvantages by small disadvantages and small advantages by clear advantages.

ω - Board Partitioning Factor

ω is a measurement of how divided the board is into privately owned territories. It follows that this is also an estimation of how close the game is to the "filling phase", or when both players have their own territories and begin to fill them in. It calculates this by summing up all the squares that a white and a black queen can both reach. If only one color can reach a given square, it adds a value of 0 for that particular tile. The closer to 0 **ω** is, the closer to the filling phase the game is.

T - The Dynamic Sum of T1, T2, c1 and c2

As the game progress, the various metrics become more or less representative of the actual evaluation of the board. Because of this, static weights can't be applied. Instead, we needed to come up with four functions f(**ω**) that are partitions of 1 and sum together to equal 1. [1] did not explicitly define these functions aside from the constraints mentioned, so we had to find a family of equations that modelled these behaviours.

The functions to weight T1, c1, c2 and T2 respectively:

$$f_1 = 2^{(-\frac{1}{45e}\omega)}$$
$$f_2 = 0.18(1 - 2^{(-\frac{1}{45e}\omega)})$$
$$f_3 = 0.22(1 - 2^{(-\frac{1}{45e}\omega)})$$
$$f_4 = 0.6(1 - 2^{(-\frac{1}{45e}\omega)})$$

Queen α - Individual Queen Mobility Score

Each queen is awarded an internal "mobility" score based on how many squares it can move to in one turn. Each blank square is in charge of keeping track of how many empty neighbours are surrounding it; the more empty neighbours a square has, the more it's worth to the queen being evaluated. Similarly, the closer a blank square is to the queen, the more it's worth. However, a square is only considered in the evaluation of a particular queen if the opposing player can also reach that square (not necessarily in one step, it just has to be possible to reach). This results in **α** not only being a measurement of mobility but also of how cut off a particular queen is from the opposing player's queens (the closer **α** is to 0, the less mobile or more separated the queen).

[1] contains an error in the calculation of this score, the revised function is as follows:

$$\alpha_a \ = \ \Sigma\, 2^{-(d_k(a,b)\,-\,1)} N(b) \ \forall\, b \in [\ d_q(a,b) = 1,\ D_q^{3-j}(b) < 127]$$

where

- a is the square the queen is on
- d_q(a,b) is the Min. Queen Distance from a to b
- d_k(a,b) is the Min. King Distance for THIS queen from a to b
- N(b) is the empty neighbour count of square b
- j is the player (1 for white, 2 for black)
- D_q is the Min. Queen Distance for the opposing player

m - Mobility Penalty Metric

m is a metric that dynamically penalizes the player based on how segregated their own queens are at different stages of the game. Essentially, the earlier in the game a player's queens becomes completely cut off from the rest of the board, the more severe the penalty. Being another area in the Amazong heuristic that was not explicitly defined, it required us to come up with a mathematical function f(ω,α) that satisfied four given constraints:

1. $f(0, y) = 0$
2. $\frac{\delta f}{\delta x}(x, y) \geq 0$
3. $\frac{\delta f}{\delta y}(x, y) \leq 0$
4. $2f(x, 5) \ < \ f(x, 0)$

The function we came up with was the following:

$$f(x, y) = ax^b e^{(-cy)}$$

where a, b and c are all constants that can be modified to fine-tune the evaluation. We chose a = 1.5, b = 0.5, c = 0.2

The Evaluation

The evaluation comes together with T + m. A board evaluation is only run at leaf or terminating nodes during a tree search. Because of our object-oriented design and various algorithmic improvements to common graph-search methods, the search is capable of evaluating over 650,000 leaf nodes between 24-27 seconds at a depth of 2-ply at the beginning of the game.

Search Methods

To save memory and computation time, rather than storing a copy of the board in each node of the tree, only an encoded integer representation of the move, the score, parent node pointer and children (successor) list are stored. This way, nodes are lightweight and very fast to generate and repurpose for garbage collection.

Encoding Moves

When moves are being calculated there are 6 coordinate values generated: the initial x-position, initial y-position, destination x-position, destination y-position, arrow x-position and arrow y-position. Since the values are between 0 and 9 inclusive, the maximum bit value is 1001. With 6 values there are a total of 24 bits. By reversing the previously mentioned order of coordinates, we can shift an integer by 4 bits and OR the coordinates in one at a time, reversing the process later when we need to decode the move.

Example (3, 0, 3, 6, 6, 6) (Queen d1 to d7, shoot to g7):

Bit representation of non-encoded move: (0011, 0000, 0011, 0110, 0110, 0110)

Move after first shift: 0000 0000 0000 0000 0000 0000 0000 0110
(0011, 0000, 0011, 0110, 0110)

Second shift: 0000 0000 0000 0000 0000 0000 0110 0110
(0011, 0000, 0011, 0110)

Sixth shift: 0000 0000 0110 0110 0110 0011 0000 0011

α-β Minimax Search

Because of the enormous branching factor in Amazons it is inefficient to generate an entire moveset for each level of the tree. By only generating the nodes that the algorithm explicitly requires for its search we save enormous amounts of time and memory, especially in the earlier stages of the game. A normal α-β search algorithm generates a list of successors for a given state, and then for each successor runs α-β. Our algorithm instead generates and runs α-β one move at a time. If at any moment $\alpha > \beta$ at the current depth, successor generation stops immediately and we move up the tree.

To traverse the tree, when a new legal move is generated the AI plays it on its own private board and saves the move and blank square replaced by the arrow shot on a stack. When we move back up the tree the AI pops the move off the top of the stack and calls undoMove(), replacing the arrow shot with the respective blank square and reversing the queen move.

Monte Carlo Tree Search

When the program switches to using MCTS to evaluate the position it first generates a 1-dimensional integer array to represent the board. A smaller, more lightweight board was needed because the algorithm requires examining many more positions than α-β Minimax. Because MCTS doesn't require heuristic evaluations at terminating nodes our object-designed board wasn't necessary since there was no need to store metrics such as MQD, MKD, α, empty neighbours, etc. An integer representation was good enough.

The live version of the search generates the first list of all possible moves, selects one at random and simulates a game all the way to termination. During the simulation all legal moves are generated for the current position and one is randomly played. Due to this generation of all legal moves the algorithm was only capable of achieving ~32,000 simulation in 30 seconds on a half-filled board. Also, heap size had to be measured and the algorithm cut short if the heap grew too large.

An improved version was under development that improved the simulation count to >800,000 in 30 seconds with no heap overflow problems but, due to a bug that randomly caused null pointer exceptions to be thrown, this version was unable to be used during the tournament. The improvements made were converting ArrayLists into Node arrays, and randomly generating a move instead of generating all moves and randomly selecting one. That is, a random queen would be selected, then a random possible direction to move in, a random step size to take, and then a random direction and step size for an arrow shot.

Strengths and Weaknesses

Our program excelled in the opening and middle phases of the game, aggressively trapping opposing queens while trying to maximize its own queens' mobility. The evaluation faltered however when determining how much to punish a player for a cut off queen. If an opponent's queen was completely cut off from the board, the heuristic would score heavily in favour of itself without accounting for the fact that the opponent had n-number of squares permanently under their control. Also, the computer would start to play incorrectly shortly before the filling phase of the game because it would hold off enclosing the last enemy queen for as long as possible, since doing so also meant enclosing its own queens. This resulted in many instances where the computer could have won in very few moves but chose to keep its own pieces "mobile".

Sources

1. An Evaluation Function for the Game of Amazons:
www.math-info.univ-paris5.fr/~bouzy/ProjetUE3/Amazones-EF.pdf.gz

2. Chess GUI:
http://stackoverflow.com/questions/21142686/making-a-robust-resizable-swing-chess-gui

3. NegaScout (PVS):
http://en.wikipedia.org/wiki/Principal_variation_search

4. Dr. Gao’s Simple Player:
https://people.ok.ubc.ca/yonggao/teaching/cosc322/project/gameclient/Amazon.java