Roneil Rumburg (id: roneil), Steven Longoria (id: sxl1092), Miguel Francisco (id: miguelf)

Final Project Report: Singularity Chess

Abstract:

For our final project we created a simulation for a game called Singularity Chess and examined different approaches to writing agents to play the game. Singularity Chess is similar to chess in that it uses the same pieces and uses a board with black and white squares, but the board layout is very different and leads to some complex behavior with regard to how pieces move (see the picture to the right).

After making our simulation for the game, which includes functions for getting the valid moves for either color given a specific board layout, we went on to develop various agents to play the game, including a minimax agent, greedy agent, Monte Carlo agent, and random agent (which we used as our control to test the various other agents). Our minimax agent attempts to maximize the minimum payoff for the given agent as all minimax agents do. The greedy agent maximizes your minimum payoff for the next move (as oppose to the minimax agent which can look multiple moves ahead). The Monte Carlo agent randomly selects a subset of 10 moves from the list of valid moves for the next turn and selects the best of these moves.

Upon running the various agents, it became apparent that some agents played Singularity Chess better than others. While the random agent was our control (all of our other agents were able to outplay it without much effort) it became apparent that our Monte Carlo agent played better than our greedy agent, while our minimax agent outplayed our Monte Carlo agent. Our results will be expounded upon in the 'Experiments' section of our documentation

Introduction:

The game we chose to model, Singularity Chess, is a chess variant in which the number of rows in each column of the board changes based on the current row. The row in the center also has some very odd behavior; it changes the direction of pieces that are moving through it based on the quadrant of the board they came from. While computationally modeling chess is a somewhat solved problem, Singularity Chess, as far as we could tell, has never been computationally modeled before, and because we all enjoy playing it we thought it would be a great game to work with for our project.

What we specifically set out to do was develop agents based on techniques used for regular chess agents that could play Singularity Chess and compare them to one another to find the most optimal of the ones we made. This problem is interesting because while the general game simulation is similar to chess the pieces follow very different rules, and if techniques applied to Chess in the past can be accurately applied to Singularity Chess we'll know that techniques used in regular chess can likely be applied to other chess variants and that other chess techniques (i.e. complex evaluation functions) that we haven't used likely could also be applied to Singularity Chess.

The main challenges we anticipated and ended up having to deal with when developing these agents were performance issues; because simulating chess is a fairly complex task, and simulating Singularity Chess is even more so, we spent a fair amount of time dealing with performance issues to make our agents reasonably efficient, but even after this our minimax agent is still too slow for general use (the other two are quick enough to be used in human games). Developing an effective evaluation function also seemed challenging, but we were able to use a very simple evaluation function in our final solution that still produced fairly accurate agent behavior in agents that used it.

Prior work has mainly been done for regular chess. No Singularity Chess agents or simulations existed before we built this project, so all of what we've done is new in a sense but it did borrow heavily from regular chess.

Approach:

We approached this problem by first building the infrastructure necessary to write agents, including the game simulation, the ability to get the set of valid moves for a given color (taking into account check status) and the ability to branch and create new simulations from the current state (for agents that use their own simulations internally). Then we wrote our baseline agent, the RandomAgent, which picks a random move from the valid moves for its color. After this, we wrote three more intelligent agents, GreedyAgent, MonteCarloAgent, and MinimaxAgent, which all used the same evaluation function to determine the relative value of different game states. Each of which will be described in depth below.

The evaluation function we wrote was very simple, but was more than enough to allow the intelligent agents to win against the random agent. Our evaluation function assigns values to various pieces and then assigns a value of 3 to putting the opponent in check making it equivalent to capturing a bishop or knight.

The random agent makes the most sense as our control given that it does not attempt to play the game well, but instead simulates random moves of an unintelligent player. In essence this corresponds to an agent which does not have a policy for how to play chess, and the fact that this agent does not attempt to play the game well makes it easier for us to gauge how well our intelligent agents are doing with respect to "unintelligence".

Our intelligent agents all make sense for the given task given that Chess is a two player zero-sum game, and greedy algorithms (algorithms which attempt to maximize the players net gain), Monte Carlo simulators (ours selects a subset of 10 moves from the set of all valid moves and chooses the best of these for the next move), and Minimax algorithms (which maximize the minimum gain for a player) are all algorithms which operate well in two player zero-sum games for historical reasons. Algorithms of this nature (in particular minimax agents) have been applied to Chess in the past, so it was reasonable of us to presume that they would operate well in Singularity Chess. Despite this, our intelligent methods maintain some strengths and weakness between one another which will be expounded upon below.

Trade-offs between each agent are primarily based on effectiveness versus speed. Agents which were less effective, such as the random agent and greedy agent, ran near instantaneously. However, more intelligent agents, such as the minimax and Monte Carlo agents, ran exponentially slower. We decided to run the minimax agent and the Monte Carlo agent at depth of 1, meaning that they plan ahead by one move, assuming that the opponent is an agent of the same type. Because depth 1 games between minimax and Monte Carlo agents ran so slowly, upwards of 20 minutes per game, we decided that exponentiating runtime even further with depth 2 would be impractical.

Experiments:

For our experiments we made each of our agents play against all of the other agents to evaluate their performance, and we also compared the Random agent to itself as a control value to put the other values into context. See appendix 1 for the win/loss/stalemate percentages for each matchup and appendix 2 for our raw data from the program (including number of games in each matchup). For each of the set of percentages in appendix 1 we ran as many simulations as we reasonably could given the time that they took. Further, to reduce bias, we allowed each agent to play as white for 50% of the games simulated (white has a small advantage because they move first).

The outcomes for accuracy (performance in the simulation) were very interesting, and indeed what we expected. The random agent stalemated a majority of the time, and all of the intelligent agents were able to beat the random agent in the vast majority of games played. Then, based on the other matchups, it seemed that Greedy agent was the worst, Monte Carlo agent was the second best, and Minimax agent was the best.

The correlation between time complexity and accuracy was also very interesting. Let n be the number of possible moves that can be made. The Greedy agent's pick move was of O(n) time and O(n) space complexity and performed the worst of the three intelligent agents accuracy-wise. Our Monte Carlo agent is O(num-samples) time and O(n) space complexity, and in our case we sample 10 moves then sample 10 moves for the opponent after making those 10 moves, so it always does 100 samples. In chess, 100 is almost always going to be larger than n, so the Monte Carlo agent is doing more work than the Greedy agent. And finally, the Minimax agent is O(n^2) time and O(n) space complexity, because for each move the agent can make it simulates n moves for the opponent. So the agents' algorithmic complexity was directly correlated to accuracy, because the Minimax agent was the most complex and the most accurate, the Greedy agent was the least complex and the least accurate, and the Monte Carlo agent was in between. This makes sense, because the algorithms that do more work do so in an attempt to achieve greater accuracy.

Conclusion:

For our project we learned how to model a very abstract problem and build agents for it. Given the abstract nature of Singularity Chess, the fact that we were able to model it quite simply using the most common models for regular chess was quite intriguing. Given more time to fiddle with our results and findings, the next steps would be finding a way to shave down some of the time of the minimax agent (possibly with alpha-beta pruning for larger depths). Also, building a more complex evaluation function which took various other factors into account would be useful as well as developing other models for playing chess, such as agents that would play more aggressively or defensively than others. Despite this, we are quite pleased with our findings given the current models we were able to use.

References:

1. Singularity Chess Rules: http://abstractstrategygames.blogspot.com/2010/10/singularity-chess.html
2. Video Example of Singularity Chess: http://www.youtube.com/watch?v=C8xrGaAy98A
3. Chess Piece scoring Values (used in evaluation function): http://en.wikipedia.org/wiki/Chess_piece_relative_value
4. Minimax Chess: http://www.aihorizon.com/essays/chessai/intro_print.htm

Appendix 1: Win/Loss/Stalemate rates for experiments

Agent Types	Agent 1 win rate	Agent 2 win rate	Stalemate rate
RandomAgent vs. RandomAgent	23.2%	25.8%	51.0%
GreedyAgent vs. RandomAgent	44.2%	8.6%	47.2%
MonteCarloAgent vs. RandomAgent	78.5%	1.0%	20.5%
MinimaxAgent vs. RandomAgent	90.0%	1.5%	8.5%
MonteCarloAgent vs. GreedyAgent	67.25%	5.5%	27.25%
MinimaxAgent vs. MonteCarloAgent	78.0%	9.0%	13.0%
MinimaxAgent vs. GreedyAgent	82.0%	9.5%	8.5%

Appendix 2: Raw game win/loss data from Experiments

<u>RandomAgent vs. RandomAgent:</u>
Games: 500
Random agent 1 wins: 116
Random agent 2: 129
Stalemates: 255

<u>GreedyAgent vs. RandomAgent:</u>
Games: 500
Greedy agent wins: 221
Random agent wins: 43
Stalemates: 236

<u>MonteCarloAgent vs. RandomAgent:</u>
Games: 200
Monte Carlo agent wins: 157
Random agent wins: 2
Stalemates: 41

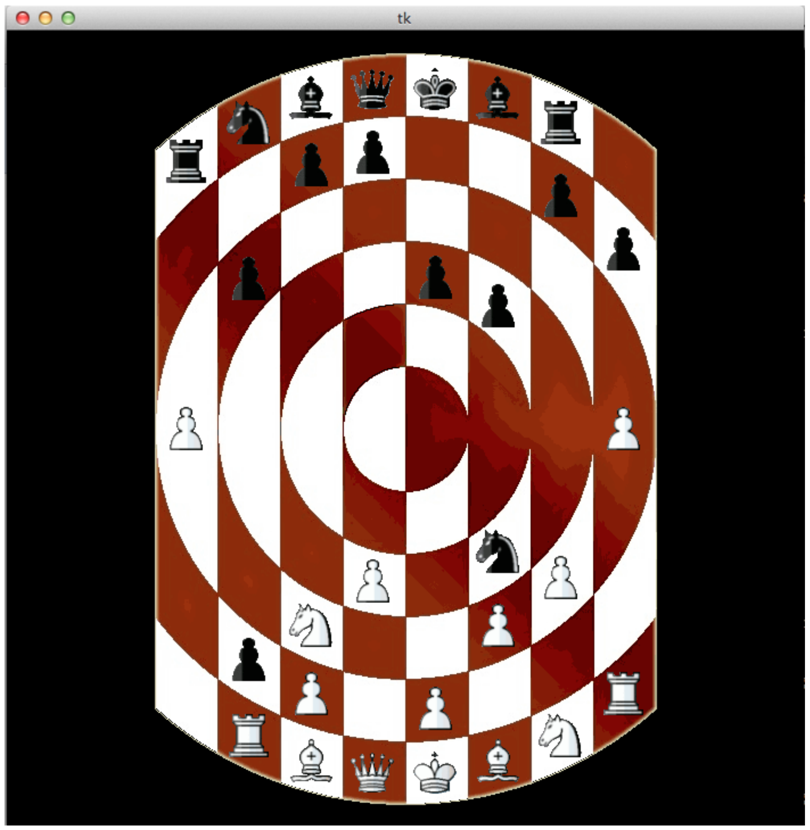
<u>MinimaxAgent vs. RandomAgent:</u>
Games: 200
Minimax agent wins: 180
Random agent wins: 3
Stalemates: 17

MonteCarloAgent vs. GreedyAgent:
Games: 400
Monte Carlo agent wins: 269
Greedy agent wins: 22
Stalemates: 109

MinimaxAgent vs. MonteCarloAgent:
Games: 200
Minimax agent wins: 156
Monte Carlo agent wins: 18
Stalemates: 26

MinimaxAgent vs. GreedyAgent:
Games: 200
Minimax agent wins: 164
Greedy agent wins: 19
Stalemates: 17

Appendix 3: Screenshot of GUI



Pieces in the GUI move as agents move them in a simulation.