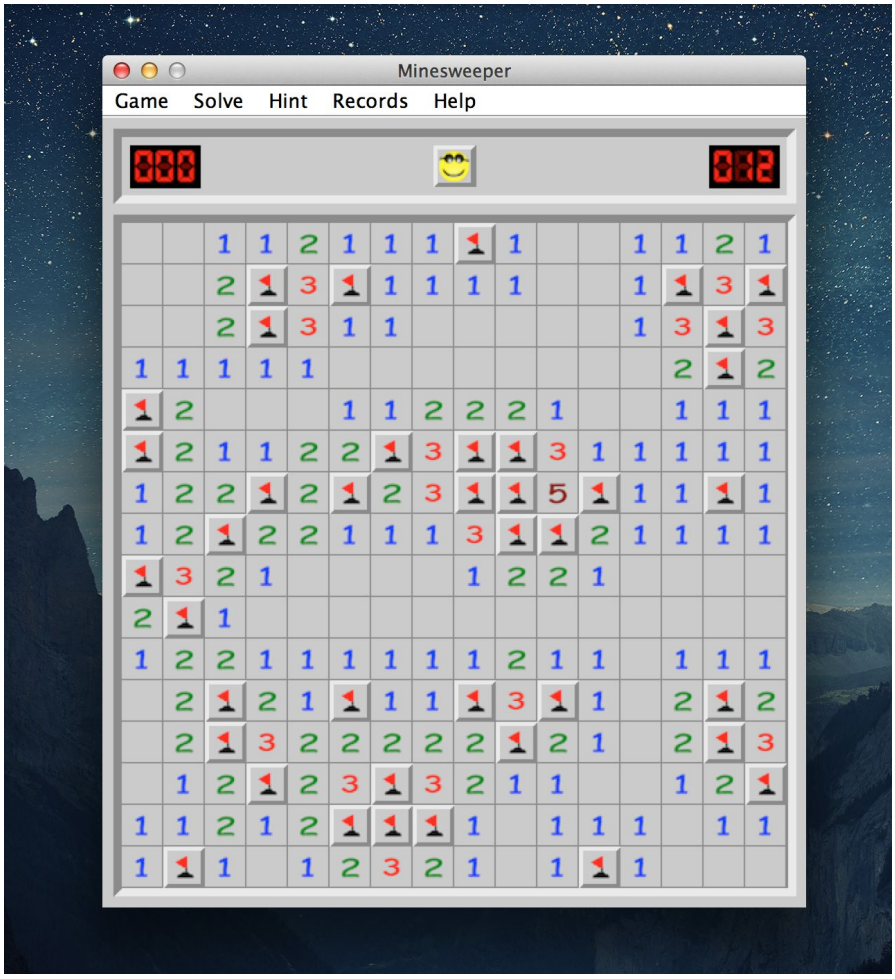
CS221: Introduction to Artificial Intelligence

Minesweeper AI Agent

Alex Engel and Fiona Meyer-Teruel



Introduction

Minesweeper is a game most people in our generation are familiar with. Because many of our peers have developed very specific and individualized methods for solving the game, we decided designing an artificially intelligent Minesweeper agent would provide a useful metric to measure our own performance against. This agent will take any size Minesweeper board (up to 25*25 with 80 mines) as input and work its way through the board, attempting to mark all mines with flags. Minesweeper is an NP-complete game as far as we know which means that there is

no optimal algorithm that can always compute a solution in polynomial time. Thus, our AI agent will do the best we can by transforming the game of Minesweeper into a constraint satisfaction problem (CSP) and solve this problem using backtracking search with forward checking alongside other heuristics in order to find some solution in polynomial time.

Infrastructure

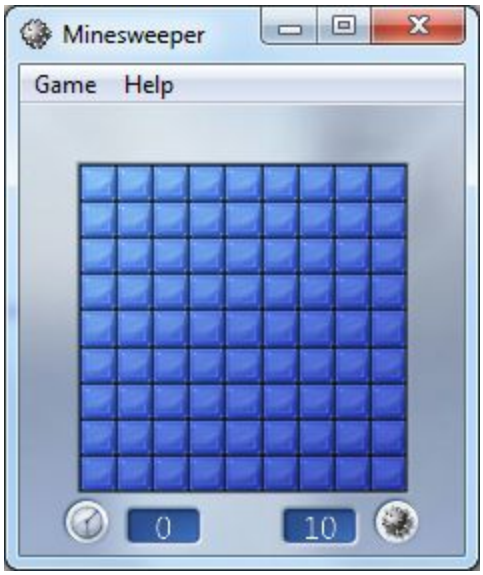


Fig. 1

A beginner's Minesweeper board is shown above in Fig. 1. To create our model, we first assign each tile a variable name X_{ij} that denotes the location of the tile on the grid where *i* is the column and *j* is the row. Each tile can either contain a mine or contain nothing. Thus, we will assign the domain of each X_{ij} to be $X_{ij} \in \{0,1\}$ where 0 represents an empty tile and 1 represents a tile containing a mine.

Now that we have variables, we can create a factor graph. The factor graph will take the same shape as the Minesweeper grid and each variable will have a factor connecting it to each neighbor as shown below in Fig. 2.

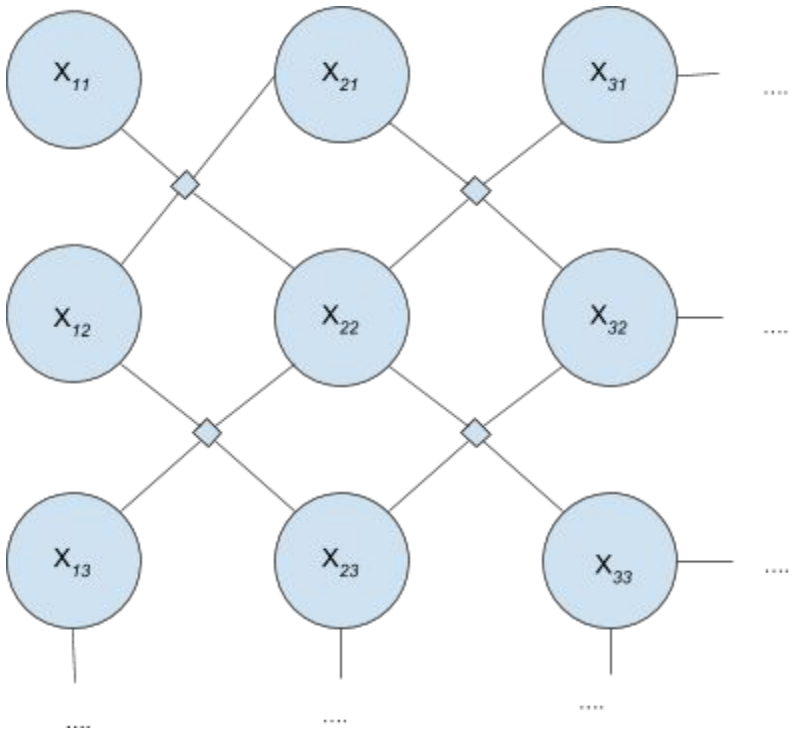


Fig. 2

Let's say our AI agent begins by examining X_{11} and finds that its value is 0. Upon discovering this, a number will be revealed under the tile that states the number of mines that that tile is touching, for this example, let's say 2. This allows us to place a constraint on our set of variables X as follows $f_1(X) = [X_{21}+X_{22}+X_{12} = 2]$. Now, say we have a constraint found later that says $f_i(X) = [X_{21}+X_{22}= 1]$. We can combine these two constraints and get a unary constraint that says $f_{i+1}(X)=[X_{12}=1]$ and we can now assign a value to X_{12}.

Approach

- **Assigning values to variables:** Now that we have a proposed model for our agent, we can take a look at the algorithm that the agent will follow to find a solution. First, we will describe the part of the algorithm that determines the values of tiles and then we will describe the part of the algorithm that dictates the order in which to proceed through the tiles. As mentioned above, our algorithm will always attempt to reduce constraints to their most trivial before moving on to the next step. At some point, all the remaining

constraints will be non-trivial and so we move on to a different method of completing partial assignments.

The key to this method is to find constraints that share a common variable and then have backtracking search (see below) find all solutions to these constraints. After we find all the possible solutions for this constraint set (each variable assigned 0 or 1), we iterate over these and check for any commonalities. For example, if all solutions say that a certain variable is a mine, then we know that variable must contain a mine and we can assign it a value of 1. On the other hand, if all solutions say that some tile is not a mine, we know that this variable can be assigned 0 and is safe to click.

This will work for many cases, but at other times, it will be necessary for our AI agent to make an educated guess. To do this, we refer back to the solution set mentioned in the paragraph above. In times when guessing is necessary, we want to click on the tile that is most likely to have a value of 0. To figure this out, we simply find the variable that has the most instances of 0 in the solution set and this will be the variable in the constraint set that is most likely to produce a favorable result. However, there may be times when no variable is optimal and a probing a somewhat random tile on the board might be more prudent. Whether or not this is the case depends on how many unclicked tiles and how many mines are left.

- **Backtracking search:** This backtracking search works on small segments of the board at a time. To find these sets, we choose the most constrained variable and find all variables that are dependent on it. If the board has more than 20 unprobed tiles, we solve this set. Otherwise if there are less than 20 tiles on the board, our implementation finds solutions for all remaining variables. To do this, we start with the first variable and try assigning a value of 0 and 1 to it. If this assigned value does not violate any of our constraints, we recurse on this option to assign the next variable a value. As it does this, it looks at the constraints and checks to ensure that the current assignment of the variables does not violate any constraints we have. If they do not and we have successfully assigned every variable a value, we know that we have found a possible solution for that subset of variables on the board. If at any time we violate any constraints, we abandon that partial assignment of variables and backtrack to the last valid decision we made. We continue to do this until we find all valid solutions to that subset of variables.

- **Evaluating Solutions:** Now that we have all possible solutions to this subset of dependent tiles, we can analyze what we have in an attempt to draw conclusions. First of all, we know that if a certain variable has been assigned a value of 1 and is a mine in all solutions, then it must be a mine in the final solution of the board. The same goes for if all the solutions say that the variable must be safe. Otherwise, we add up the number of times that each variable appears as a mine in our set of solutions and divide this by the number of solutions we have, which gives us the probability that that variable is a mine. Now, if we have to guess, we can make an educated guess about the variable that is least likely to have a mine.
- **Guessing:** There are games of minesweeper where guessing is inevitable. However, we can adjust our guessing strategy to help us have the best likelihood of succeeding. Through testing, we often found that randomly guessing a corner square was a good strategy. Also, guessing an unconstrained variable ending up giving us better results than choosing a constrained variable.

Using the Model and Algorithm to Solve a 3x3 Minesweeper Board

Now we will show how to use our model and algorithm to solve a small board. Let's say we have a small 3x3 board containing two mines depicted in the factor graph below.

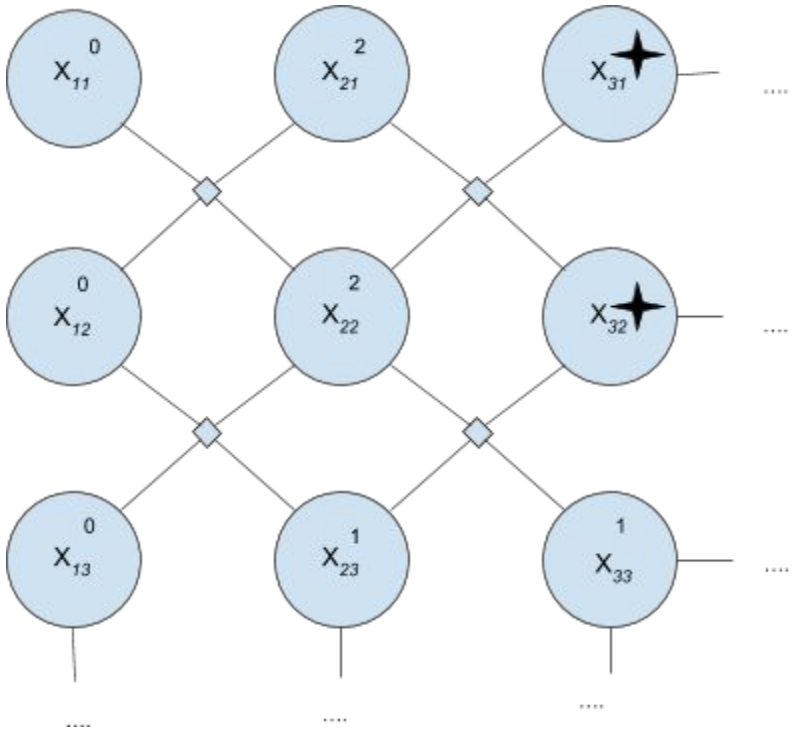


Fig. 3

You can see that there are mines in X_{31} and X_{32}. The small numbers in every other variable represent the number of mines that the specific variable is touching. The agent playing the game (human or AI) cannot see these mines or numbers when playing the game but they will help us to visualize our example. In the following example, we will denote these numbers as $f(X_{ij})$

We'll start out by probing X_{11}. We find that it does not contain a mine so we can assign it a value (X_{11}=0) and that it is not touching any other mines so $f(X_{11})$=0. Thus, we can get constraint [X_{21}+X_{22}+X_{12}=0] and consequently we can assign values to the variables as follows X_{21}=0, X_{22}=0, and X_{12}=0.

Now let's probe X_{12}. Similarly to above, we find that f(X_{12})=0. This gives us the constraint [X_{11}+X_{13}+X_{21}+X_{22}+X_{23}=0] and we can assign values X_{13}=0,X_{23}=0.

Now we probe X_{21} and find that $f(X_{21})$=2, which gives us the constraint [X_{11}+X_{12}+X_{22}+X_{31}+X_{32}=2]. Since we know that X_{11}+X_{12}+X_{22}=0, we can trivialize the constraint to [X_{31}+X_{32}=2]. Since the domain of a variable is [0,1] we know that X_{31}= 1 and X_{32}=1. Thus we have found our two mines and the board is solved.

Literature Review

Many people have created Minesweeper AI agents. Because this is an NP-complete problem, there is no perfect strategy.

The forums Stack Overflow and Reddit provided a lot of insight into the multitude of approaches individuals have used to solve the board (StackOverflow 2009, Reddit 2012). The biggest problem in Minesweeper comes when the player must guess tiles because there is not enough information uncovered to determine 'safe' moves. Many of the algorithms therefore revolve around developing robust probability models that can increase performance beyond the average player's intuition (StackOverflow 2009, Reddit 2012).

The most robust model we found belonged to Bai Li, who developed an incredibly comprehensive algorithm that utilizes many of the same techniques we developed (Li 2012). He first uses the 'straightforward' algorithm, which finds all the obvious mines and safe locations where the probability of each case is 1. Then, he employs the 'Tank Solver' algorithm, which uses a multisquare approach to determine other

mine and safe locations based on neighboring fields. Finally, his method determines all possible mine locations to complete the game and uses the probabilities generated from each of these options to guess at the locations of the final mines. His algorithm was very successful, solving the advanced minesweeper board 50% of the time (Li 2012). We used Bai Li's algorithm as our oracle, since it managed to solve the minesweeper board incredibly effectively, but we decided not to utilize his 'Tank Solver' algorithm because of its dependence on brute force solving rather than methods we learned in this course.

Error Analysis

Because Minesweeper is an NP-complete problem, we knew we would be unable to solve every game with 100% accuracy.

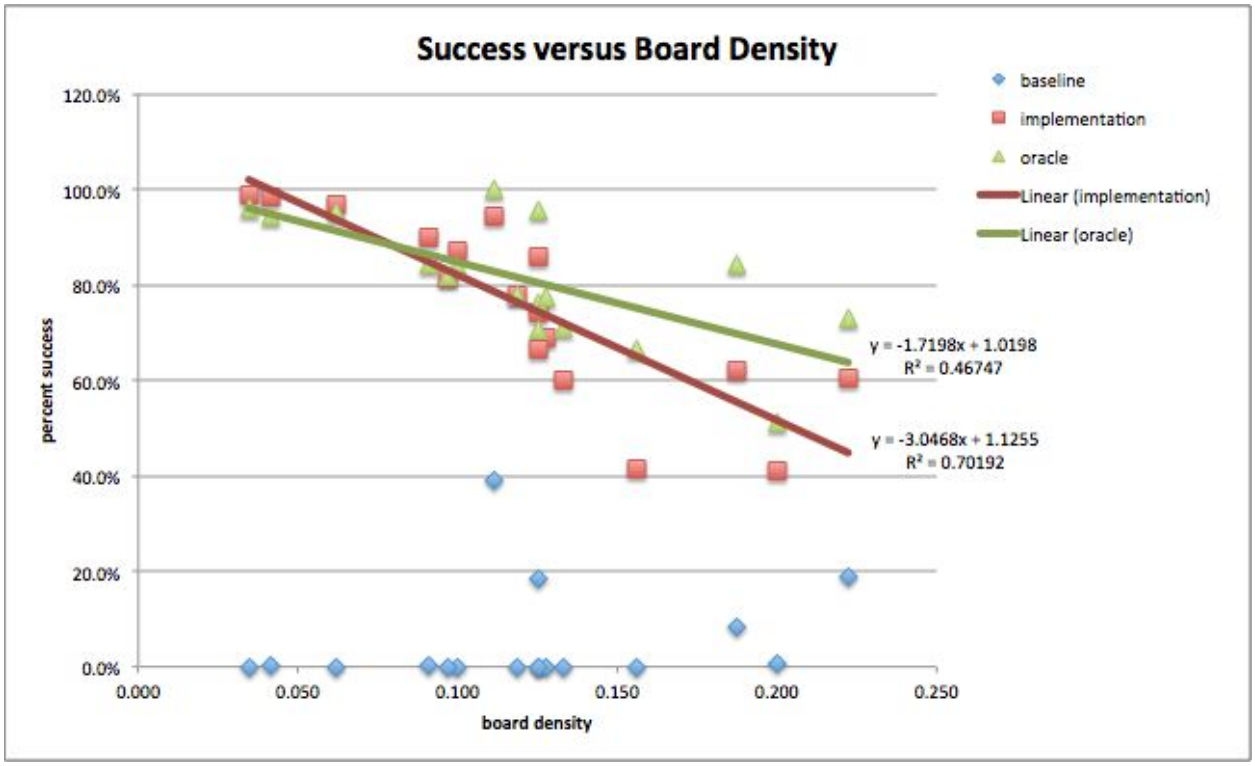
To create a set of metrics to measure our Minesweeper AI's performance, we compared against a baseline and an oracle.

Our baseline solver was a single-space Minesweeper AI agent that would select mine locations based on a single location's number. It cycles through a set of uncovered tiles and if it can determine a neighboring safe tile or mine location from the value of the uncovered tile, it will uncover the safe tiles or flag the mine location and add these points to its queue of single locations. This algorithm mimics basic human behavior in a game of minesweeper, but also fails to determine solutions once the board passes the beginning levels of the game.

Our oracle was Bai Li's effective algorithm, since it performs better than most human players and we could not find a more effective solver in our research. Since Minesweeper is NP-complete, we knew no matter how good a solver would be, it would be unable to solve the grid with 100% accuracy. Unfortunately, Bai Li's algorithm did not run on our machines and when looking over the source code, we found some bugs and hacks he had implemented that suggested the algorithm did not always correctly predict wins and losses. We implemented his algorithm in python to compare our results to his benchmark.

As can be seen from the table in Appendix A and the graphs in Appendix B, our algorithm was very competitive when compared to Bai Li's algorithm, and even performed better on boards with fewer mines. Our main discrepancy is that we chose to guess a tile without running through every possible option when faced with a large board to reduce runtime. Our algorithm was approximately 5 times faster than Bai Li's for many of the more challenging boards.

The graphs in Appendix B show a variety of methods we used to determine a relationship between the game layout and the algorithm success, specifically the relationship between the algorithm and the board size, number of mines, or board density (the number of mines/total tiles). We found the best approximator was the success versus the board density, which seemed negatively correlated.



While our results did not always prove as robust as our oracle's results, our algorithm lost accuracy when it increased its speed, by reducing the number of tiles it considered when calculating options. Meanwhile, Bai Li has a very aggressive brute force method that takes much longer to run than our own.

Our project code can be found at https://github.com/fionamt/minesweeper.

Bibliography

"How to Write Your Own Minesweeper AI. Very Comprehensive Has Many Pictures!" *Reddit*. 2013. Web. 9 Dec. 2015. <https://www.reddit.com/r/programming/comments/15c4e1/how_to_write_your_own_minesweeper_ai_very/>.

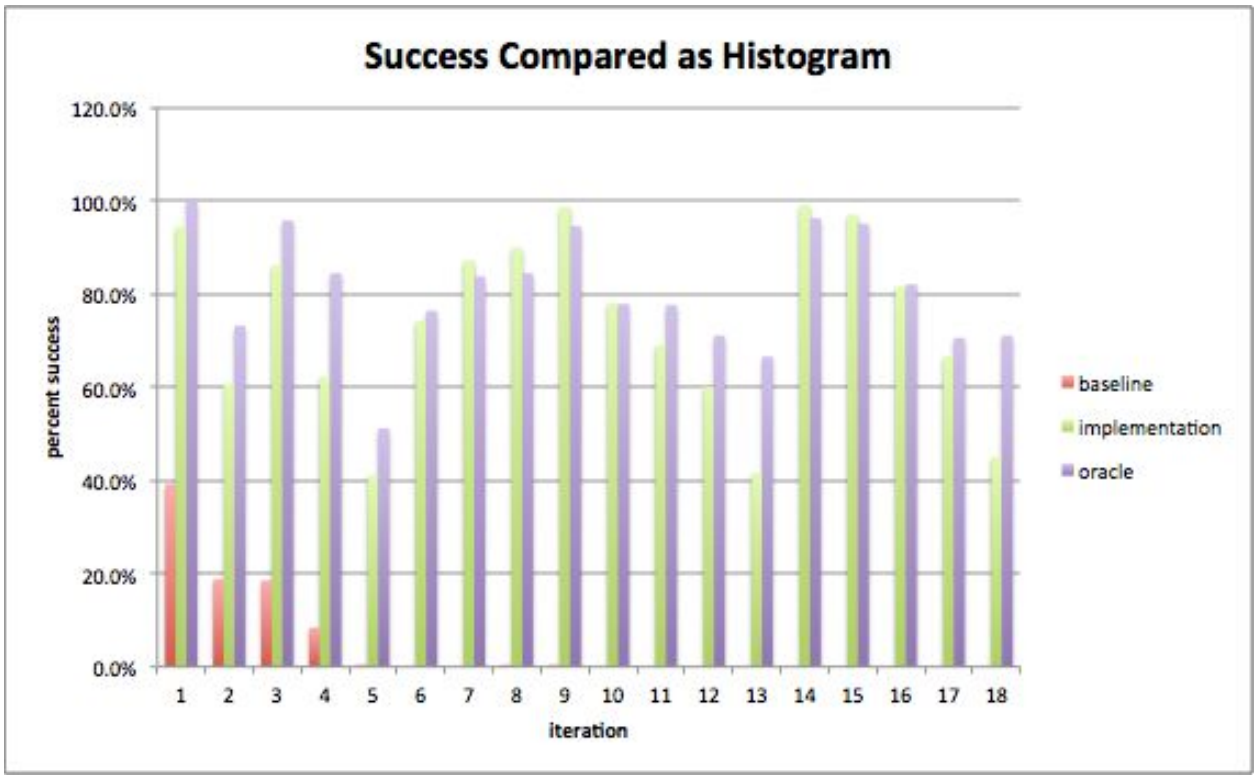
Li, Bai. "How to Write Your Own Minesweeper AI." *Lucky's Notes*. 23 Dec. 2012. Web. 9 Dec. 2015. <https://luckytoilet.wordpress.com/2012/12/23/2125/>.

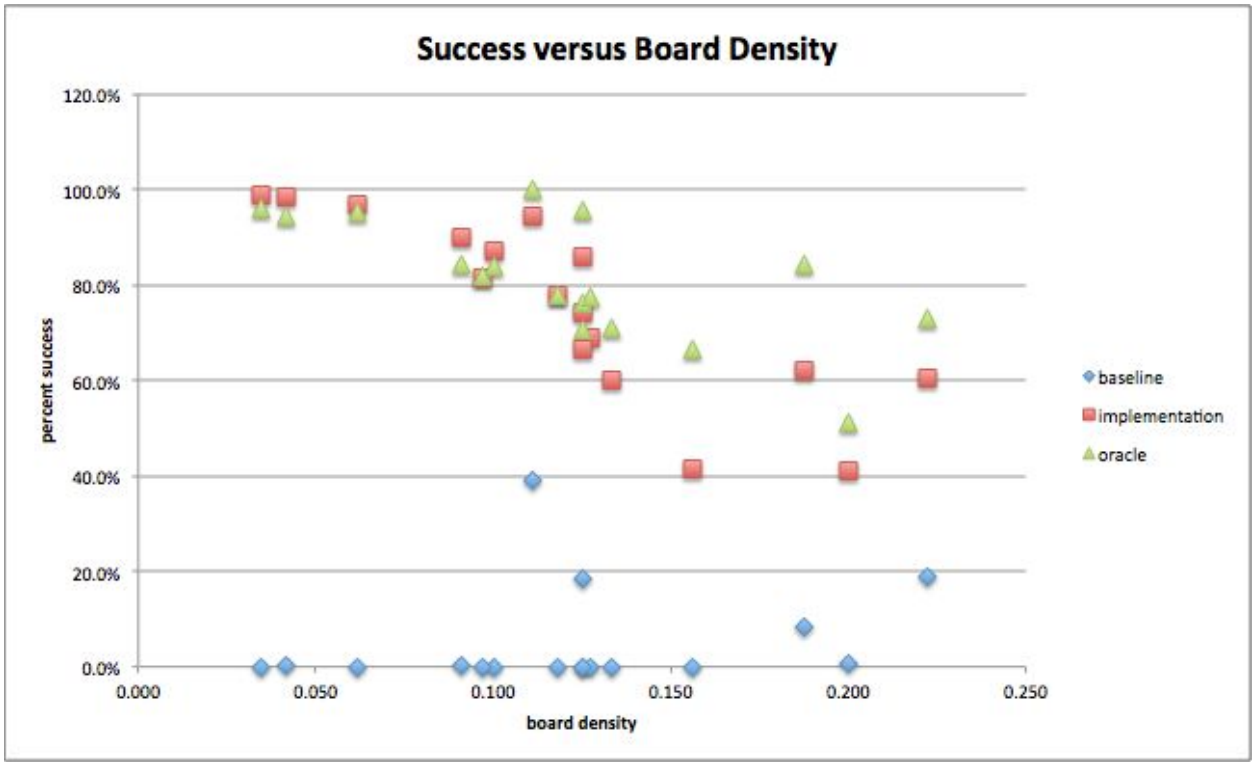
"Minesweeper Solving Algorithm." *Stack Overflow*. 2009. Web. 9 Dec. 2015. <http://stackoverflow.com/questions/1738128/minesweeper-solving-algorithm>.

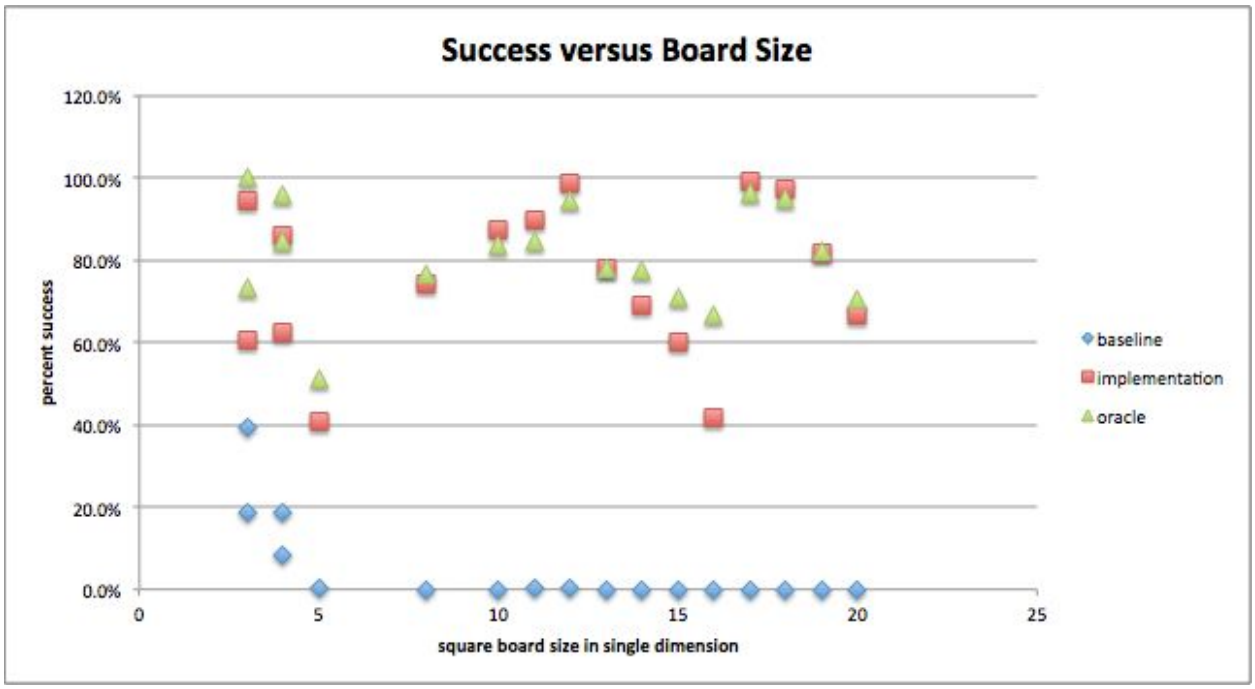
Appendix A

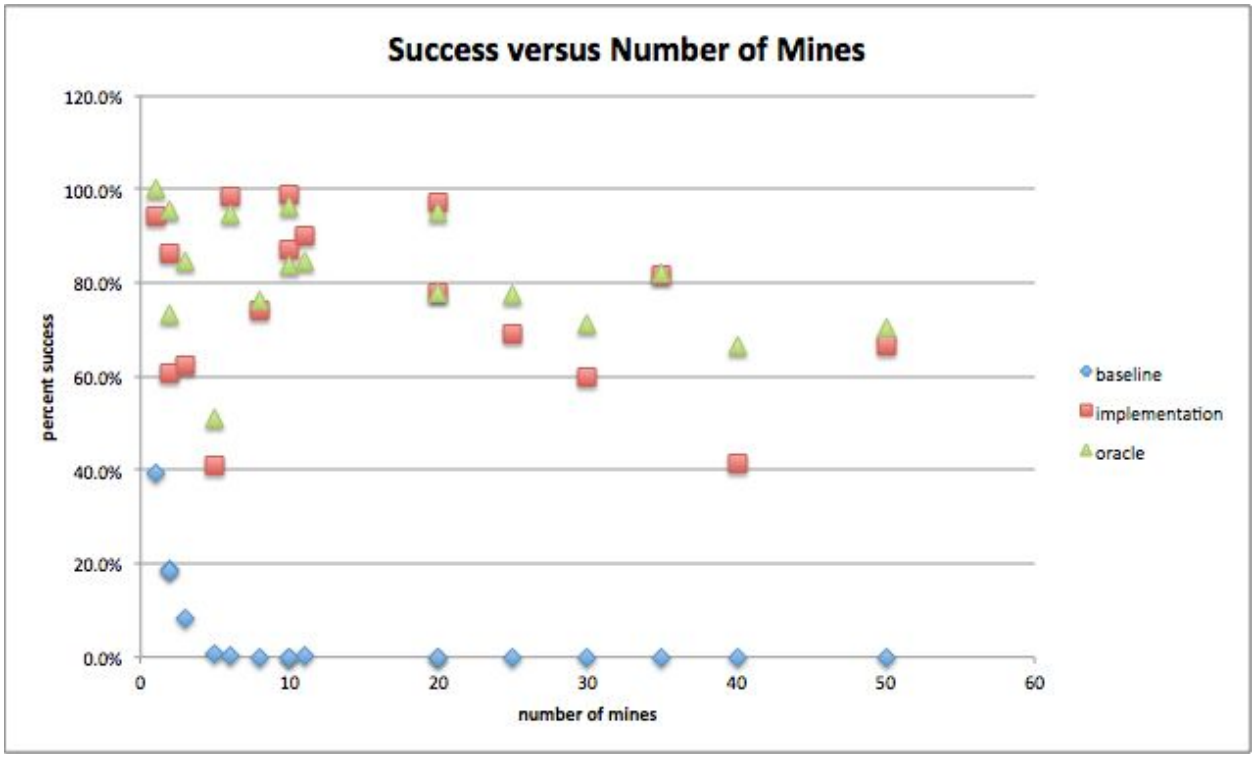
	Board size	Number of mines	Number of games	Board density	Number of games won			Percentages		
					Baseline	Imple-mentation	Oracle	Baseline	Imple-mentation	Oracle
Beginner	3	1	1000	0.111	392	944	1000	39.2%	94.4%	100.0%
	3	2	1000	0.222	188	606	731	18.8%	60.6%	73.1%
	4	2	1000	0.125	185	861	957	18.5%	86.1%	95.7%
	4	3	1000	0.188	84	622	844	8.4%	62.2%	84.4%
	5	5	1000	0.200	5	410	511	0.5%	41.0%	51.1%
	8	8	1000	0.125	0	741	763	0.0%	74.1%	76.3%
	10	10	1000	0.100	0	872	837	0.0%	87.2%	83.7%
	11	11	1000	0.091	1	899	844	0.1%	89.9%	84.4%
Intermediate	12	6	1000	0.042	2	986	945	0.2%	98.6%	94.5%
	13	20	500	0.118	0	390	389	0.0%	78.0%	77.8%
	14	25	500	0.128	0	345	388	0.0%	69.0%	77.6%
	15	30	500	0.133	0	300	355	0.0%	60.0%	71.0%
	16	40	200	0.156	0	83	133	0.0%	41.5%	66.5%
Advanced	17	10	500	0.035	0	495	481	0.0%	99.0%	96.2%
	18	20	200	0.062	0	194	190	0.0%	97.0%	95.0%
	19	35	200	0.097	0	163	164	0.0%	81.5%	82.0%
	20	50	200	0.125	0	133	141	0.0%	66.5%	70.5%
	20	60	100	0.150	0	45	71	0.0%	45.0%	71.0%

Appendix B


Success Compared as Histogram
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
percent success
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
iteration
baseline
implementation
oracle


Success versus Board Density
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
percent success
0.000 0.050 0.100 0.150 0.200 0.250
board density
baseline
implementation
oracle


Success versus Board Size
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
percent success
0
5
10
15
20
25
square board size in single dimension
baseline
implementation
oracle


Success versus Number of Mines
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
percent success
0
10
20
30
40
50
60
number of mines
baseline
implementation
oracle