Minesweeper as a Constraint Satisfaction Problem

by Chris Studholme

Introduction To Minesweeper

Minesweeper is a simple one−player computer game commonly found on machines with popular operating systems such as Linux or Microsoft Windows. The game consists of a 2 dimensional rectangular playing field (or board) where some known number of mines have been hidden. Initially, all of the squares on the board are "covered up" and no information is given to indicate the location of the mines. The player's job is to either deduce or guess which board squares are clear of mines and step there to reveal a number. If successful, this number will indicate the number of mines to be found in the squares adjacent to the square with the number. Obviously, the first move of the game must be a guess because no information has been provided. Since the board is a rectangular grid, each interior square has exactly 8 neighbouring squares, edge squares have 5 neighbours, and corner squares have 3 neighbours. Therefore, the number found under any given square will be in the range of 0 to 8 (inclusive). Game play continues until the player has uncovered (or "stepped" on) each and every square that does not hide a mine, while successfully avoiding all of the mines. If the player can do this, they are considered to have won the game. However, if at any point the player attempts to uncover a square that contains a mine, the game immediately ends and the player is said to have lost.

Minesweeper is NP−Complete

Minesweeper may seem like a simple computer game to pass the time with, but it recently became a hot topic in computational complexity theory with the proof by Richard Kaye that the minesweeper consistency problem is in a complexity class of problems known as NP−complete problems. NP−complete problems have two characteristic features: they are computable by a non−deterministic turing machine in polynomial time, and every other NP−complete problem can be reduced in deterministic polynomial time to the particular NP−complete problem in question. The former characteristic makes NP−complete problems difficult to solve computationally, at least in practice. Theoretically, NP problems may be as easy to solve as problems solvable in polynomial time by a deterministic turing machine, but no one has managed to prove this, if it is even possible. The latter characteristic of NP−complete problems means that if a way is ever found to compute one NP−complete problem in polynomial time on a deterministic turing machine, all NP−complete problems would then be solvable in polynomial time on a deterministic machine. There are many well−known and important problems that are NP−complete. The problem of satisfiability of boolean formula and the "traveling salesman" problem are two such problems.

According to Kaye, "the general minesweeper problems is: Given a rectangular grid partially marked with numbers and/or mines, some squares being left blank, to determine if there is

some pattern of mines in the blank squares that give rise to the numbers seen." Obviously, instances of the minesweeper problem have an answer of yes or no, as most problems studied in computational complexity have. The subject of this paper is, however, not to solve instances of the general minesweeper problem, but to develop an algorithm to play the minesweeper game just as a human player might do (and hopefully better). Throughout the paper I will be assuming that every instance of the game I am given to play is consistent (ie. winnable).

Unlike Kaye's general minesweeper problem, traditional games of minesweeper cannot be solved using logic alone. There will be many moments during a game where a guess must be made to determine the next move. Most notably, unless some sort of hint is given at the beginning of the game, the very first move is essentially a guess. The game playing algorithm described in this paper will make every attempt to deduce the location of mines and clear squares at each step during the game, but in the case where a guess has to be made, an estimate of the probability of each square containing a mine will be calculated. Using these probabilities, the move that has the greatest chance of success will be attempted. More on this later. First I'll describe the game playing software.

Programmer's Minesweeper (PGMS)

For this project I make use of a free software project called Programmer's Minesweeper (PGMS) developed by John D. Ramsdell and released in source form under the GNU General Public License (GPL). This application is written in Java and makes it very easy for a programmer to write a "strategy" for paying minesweeper. All the programmer has to do is simply implement an interface called "Strategy" that has a single method called "play()" and is passed a "Map" object as a parameter to be used to interact with the minesweeper board. PGMS comes with two built−in strategies called Single Point Strategy and Equation Strategy. From SinglePointStrategy.java:

> The Single Point Strategy makes a decision based on information available from a single probed point in the mine map.
>
> The strategy looks at a probed point. If the number of mines near the point equals the number of marks near the point, the strategy infers that near points whose status is unknown do not contain mines. Similarly, if the number of mines near the point equals the number of marks near the point plus the number of unknowns near, the strategy infers that the near points whose status is unknown contain mines.

From EqnStrategy.java:

> The Equation Strategy makes a decision based on a set of equations.

The source for Equation Strategy then goes on to suggest that a programmer wishing to implement their own strategy should not "cheat" by reading Equation Strategy's source code. I have dutifully obeyed this for the most part; however, as described below, I did need to make a small change to both built−in strategies to implement my idea for hinted games. My only indication of how Equation Strategy is from the name and brief description above. It

looks as though it attempts to do something similar to what my strategy does, but perhaps it is not as sophisticated.

Starting a Game of Minesweeper

One thing that I discovered while doing this project is how important it is to get off to a good start when playing minesweeper. Because of this, I have given considerable thought to what strategy would be best when starting a game and will describe my conclusions here before describing the general strategy for playing the game.

At the very beginning of the game, the player is presented with a board on which every square is covered. The only information the user has is the size of the board and the number of mines hidden. From these dimensions, the mine density, *d*, can be computed (approximately 20% for an expert game). Since no information as to the distribution of the mines is given, every choice of first move is equally likely to result in the game ending because a mine was found. Many people probably consider this fact of minesweeper to be a little "unfair"; however, it seems that what many people don't know is that with the standard version of minesweeper commonly found on personal computers, it is not possible to lose the game on the first move. This fact even came as a shock to me when a friend told me about it just a few weeks before completing this paper. It seems that standard implementation of the game will swap a mine with a clear square behind the scenes if that mine happens to be in the square that the player first chooses. This obscure rule is even, apparently, unknown to the author of PGMS as I had to implement it in that software myself.

Before implementing the standard rule for first moves into PGMS, I had already implemented a mechanism by which the game could provide the player with a hint for the first move to get the game started. This hint, I thought, should be a location on the board where mine density is lowest. It seemed to me that during a game, navigating through areas of relatively higher mine density is more challenging that navigating areas of low density so I thought that giving a player an initial region of low density could be considered to be "not giving up too much". Because of these variations on how a game is to begin, my standard tests described later in the paper have all been done in triplicate ("can lose on first move", "cannot lose on first move", and "hinted game").

Despite these variations on how a game begins (and ignoring the "hinted game" for the moment), there is still a question of which square on the board is the best one to attempt to uncover first. Both of my opponents, SinglePointStrategy and EquationStrategy, start each game by randomly choosing a square on the board. As I discovered, randomly choosing a starting position is a good idea if one believes that some starting positions are better than others, but one has no idea which ones are better. I wanted to figure out which starting position is best so when I first started writing my game playing strategy I searched the Internet looking for minesweeper hints. The few pages I found all indicated that the best strategy was to start somewhere away from edges and corners (in the middle). This seemed reasonable to me at first because it seemed to give the player the most options (directions) in which to play.

Since I assume the mines are distributed randomly and uniformly, I see no reason to randomly choose a starting square. If starting in the middle of the board is the best choice, I decided I

would always start at the center square. Upon completing my strategy, I tested it against EquationStrategy and was disappointed to discover that while EquationStrategy could win 20% of the expert games it played, my strategy would only win 17% of the games it played. After convincing myself that my game playing strategy was sound and bug free, and not wanting to simply implement a random choice for starting the game as my opponents had done, I decided to try to come up with a theoretical answer to the question of which square is the best to start at.

My revelation was that simply finding a square without a mine is not the strategy to have. If, after the first step, the player has uncovered a square containing a number between 1 and 8 (inclusive), the only information the player can deduce from this number if whether it is a good idea for the second move (again, a guess) to be a square adjacent to the square just uncovered or a square further away. The only way a player can ensure that their second move is not a guess is to hope that the first move uncovers a square containing the number 0. If the first square is found to contain a 0, then all the squares adjacent to that square are guaranteed to be clear of mines. They can be immediately uncovered and some of them may also be 0. Even if they are not, having several adjacent squares uncovered may allow the player to deduce more information about the location of mines or squares lacking mines.

Let us consider the statistics. As mentioned above, if the mine density is *d*, the probability of a particular square being clear is:

$$Prob\{clear\} = 1 - d$$

For squares that are not in a corner or on an edge of the board, the probability that that square is not a mine and does not have any adjacent mines is:

$$Prob\{zero\} = (1-d)^9$$

Similarly, for an edge square that is not in a corner:

$$Prob\{zero\} = (1-d)^6$$

and for a corner square:

$$Prob\{zero\} = (1-d)^4$$

For an expert game of minesweeper, *d=0.20*, and therefore, the probability of a zero in the middle of the board is *0.134*, the probability of a zero on an edge is *0.262*, and the probability of a zero in a corner is *0.410*. Thus, it would appear that if finding zeros is the goal, the best place to look for them is in the corners. With this theory in hand, I modified my strategy to always start at a particular corner. With just this one change, my win ratio shot up to 1% or 2% higher than EquationStrategy. Now, I'm ready to describe the general strategy.

Description of CSPStrategy

My strategy is implemented by a class called CSPStrategy, along with several other "helper" classes. I'll describe CSPStrategy as a sequence of steps.

Step 1:

All board positions can be thought of as boolean variables. They either have a value of 0 (no mine) or a value of 1 (mine). Each time a square is successfully probed, a number is revealed. This number gives rise to a constraint on the values of all of the neighbouring variables (squares). This constraint simply states that the sum of the neighbouring variables (as many as 8 of them) is equal to the number revealed by probing the square. If the value of any of the variables is already known, the constraint can be simplified in the obvious way. If all of the neighbouring variables' values are known, the constraint is simplified to an empty constraint and can be thrown away. In addition to this, there are two degenerate cases. If the constant of the constraint is equal to either 0 or the number of variables, the value of all of the variables can be immediately deduced (either all 0 or all 1, respectively) and the constraint can be reduced to an empty constraint and thrown away. CSPStrategy maintains the set of constraints dynamically by adding and removing constraints as needed. The constraint set is never recomputed from scratch.

Step 2:

Given a set of non−trivial constraints, further simplification may be possible by noting that one constraint's variables may be a subset of another constraint's variables. For example, given $a+b+c+d=2$ and $b+c=1$, the former can be simplified to $a+d=1$ and the latter left as $b+c=1$. As a result of this simplification, some constraints may become trivial. If this happens, CSPStrategy will return to step 1 above. Note that during a typical game of minesweeper, the majority of the plays made in the game are a result of trivial constraints that are found in steps 1 and 2.

Step 3:

Given a set of non−trivial constraints that have been simplified as much as possible in steps 1 and 2, the constraints can now be divided into coupled subsets where two constraints are coupled if they have a variable in common. Using a backtracking algorithm, each of these coupled subsets can then be solved to find all possible solutions. I'll describe this backtracking algorithm later. The individual solutions to the set of constraints do not need to be stored explicitly. Instead, solutions are first grouped according to the number of mines that each particular solution requires. Then, for each variable, a tally of the number of solutions requiring a mine to be in the square represented by that variable is stored. To clarify this tallying process, note that these tallies are stored in a two dimensional array where one dimension is indexed by the number of mines the entire solution requires, while the other dimension is indexed by the variable (board position). The tallies are divided up by the number of mines each solution requires because the total number of mines remaining to be found is known in advance and can be used in some cases to throw out infeasible solutions. For example, if solutions are found for two subsets of constraints and in both solution sets, solutions requiring 2, 3 and 4 mines are found, but it is known that there are only 5 mines remaining on the board, all of the solutions requiring 4 mines can be eliminated.

Step 4:

Once all of the solutions to all of the subsets of constraints have been found, the solutions can be analyzed to see if there are any cases where a square is known either to be a mine or to be clear (the variable is either 1 or 0 in all solutions, respectively). If such an instance is found, mines can be marked and/or squares probed with certainty of success. Note that marking mines that the solution set indicates are there with certainty does not provide any new information and therefore a guess may still be required in step 5 or later; however, if there are any clear squares implied by the solution set, they can be probed, the new constraints can be added to the constraint set, and the algorithm can immediately return to step 1 for a new round of simplifying the, now expanded, set of constraints.

Step 5:

Figure 1: example of "craps shoot" situation.

In step 4 it may be discovered that a coupled set of constraints requires a guess to be made and that there is no possibility of new information ever making the guess easier or eliminating it entirely. I have named this situation a "craps shoot" because in the majority of cases where this situation arises, the guess is a 50/50 guess. This situation arises when the following two conditions are met: all solutions to a coupled set of constraints require exactly the same number of mines and none of the variables found in the coupled set of constraints have neighbours that are either unknown or are variables in some other constraint set (some constraint that is not a part of our particular coupled set). This situation arises reasonably often in games of minesweeper and I feel they deserve special attention because they are moments of pure chance. That is, they are points in the game where there is no possibility of using strategy or other heuristics to improve the situation. Furthermore, since the situation will never get any better as the game progresses, the "coin toss" should be done earlier rather than later so that in the event of failure, a new game can be started as quickly as possible. Because of this latter reason, step 5 is actually done immediately after the solutions to the coupled subsets of constraints are found in step 4 (and before any certain mines or clear squares are dealt with).

Step 6:

In the event that there are no squares to probe with certainty of success and any craps shoots were successful, a guess has to be made as to where to probe next. Note that an attempt to guess the location of a mine (no matter how good the odds) is never made because marking the location of a mine never provides any new information to work with. To estimate the probability that a particular square is clear of a mine, I first assume that all of the solutions to a particular coupled subset of equations found in step 4 are equally likely. Then, the probability that a particular square is clear is calculated by simply taking the number of solutions where that variable has a value of 0 and dividing it by the total number of solutions found for that particular coupled set of constraints. The variable with the highest probability of being clear is considered the "best guess" among the constrained squares for the next move. Further discussion of this method of estimated probabilities and some empirical testing will be presented later on.

Now, before this "best guess" is used, since both the total number of mines remaining to be found and the expected number of mines each coupled subset should have are known, an estimate of the probability of finding a mine (or not finding a mine) among the board positions that are currently unknown (do not appear as variables in any constraints) can be estimated. In some cases, the probability of successfully probing one of these unconstrained squares is higher than the probability of successfully probing the "best guess" determined above. If this is the case, the algorithm proceeds to step 7; otherwise, the "best guess" square determined above is probed and, if successful, the new constraint is added to the constraint set and the CSPStrategy returns to step 1.

Step 7:

If it is decided that none of the constrained squares are good choices of places to probe, some (perhaps random) unconstrained square must be probed in the hope that the number under it helps the situation. It should be noted that this situation is very similar to the situation the player is in when a (unhinted) game is first begun (that is, the beginning of a game is a special case of this situation). In the case of the beginning of a game, it was decided above that the best strategy was to look for 0's in the corners of the board. After much thought (and no explanation) I have decided that looking for 0's in the corners of the board is still a good strategy even if the game is half finished. Therefore, when a unconstrained square must be guessed, CSPStrategy first checks if there are any unconstrained corner squares remaining and if there are some, a random one is chosen. If there are none, a random unconstrained, non−corner, edge square is the next best choice. If none of these are available, then an interior unconstrained square needs to be chosen.

In this latter situation I reasoned that it would be best to guess an interior square that has the greatest chance of aiding the existing problem (the existing set of constraints). To do this, an attempt is made to guess a square that, if successful, will yield a constraint that is coupled to the existing set of constraints in some way. This idea lead me to postulate whether it is best to maximize the overlap between the new constraint and the existing set, or to minimize the overlap in an attempt to increase the set of constrained board squares by the maximum amount. I did an empirical test and found that there is very little difference between these two strategies and that if there is a difference at all, the strategy where the overlap between the new constraint and the existing ones is maximized is better. Furthermore, I found that seeking to maximize the number of constrained board positions can have an adverse effect on the performance of CSPStrategy since the time it takes to solve a coupled set of constraints is roughly exponential in the difference: *number of variables − number of constraints*. With this empirical evidence in hand, when CSPStrategy has run out of corner and edge squares to guess, it will guess a square that provides a new constraint which has the largest number of variables in common with the existing constraints. Whatever guess is made, if it is successful, the new constraint is added to the constraint set and the algorithm returns to step 1.

That concludes the description of CSPStrategy's game playing algorithm. In the next section I will describe the backtracking algorithm in more detail. After that, if any of the details of the algorithm are still a little hazy, I suggest the reader consult the source code for CSPStrategy.

Description of the Backtracking Algorithm

The backtracking algorithm used in CSPStrategy is quite simple but tailored for this application. The goal when solving the constraint satisfaction problems that arise in CSPStrategy is not to simply find a solution that satisfies the constraints, but to instead find all of the solutions that satisfy the constraints. The algorithm works as follows:

Variables are assigned values one by one in a particular order. Any variable that is known (from one of its constraints) to be restricted to a particular value is chosen first and assigned that value. Otherwise, variables will be chosen in order of their level of constraint with variables that appear in a large number of constraints being chosen before those that appear in fewer constraints.

Each variable is tested with a value of 0, and then later 1. After each test assignment to a variable, all of the constraints which have that variable in them are checked for possible violation. If a constraint is found to be unsatisfied by an assignment, the other possible assignment is tested. If that assignment also doesn't work, backtracking occurs immediately. After each assignment, the algorithm recurses to find another variable to assign. When all variables have been assigned and all constraints are satisfied by the assignment, a solution has been found and is tallied before backtracking to find the next solution.

It should be noted here that no attempt is made to intelligently jump back several variables or to track nogoods that arise when dead ends are reached. The backtracking algorithm used here was sufficient for the purposes of this project; however, there is a lot of room for improvement.

Empirical Verification of Probabilities

As noted above, I was able to do some empirical testing of my estimates of probabilities. I have some doubts regarding my assumption that all solutions to a particular coupled set of constraints are equally likely. In particular, it should be noted that CSPStrategy will throw out some solutions to a coupled set if it can be proved that the solution is not possible due to restrictions on the number of mines remaining. This leads me to consider the possibility that

Estimated Probability of Success	***Number of Successes***	***Number of Observations***	***Actual Success Rate***
34%	72	215	33.5%
50%	51117	102296	50.0%
58% to 63%	271	474	57.2%
67%	5646	8591	65.7%
70% to 79%	1269	1732	73.3%
80% to 89%	1862	2306	80.7%
90% to 97%	198	216	91.7%

Table 1: Observed success rates for craps shoots.

solutions that require different numbers of mines may have non−equal probabilities of occurring.

CSPStrategy outputs a line to a log file every time it is forced to make a guess. Here is an example:

```
GUESS: 83% educated ... good.
```

Since the line indicates the type of guess, the estimated probability of success and the outcome (success or failure), it is possible to empirically measure whether the probabilities are being accurately calculated or not.

Table 1 shows the results of observing the craps shoots (step 5) that occur during the game. This is the only guess that I am confident the probabilities are being accurately calculated for, and the results appear to back me up on that. It should also be noted from table 1 that the vast majority of these guesses are 50/50; hence the name "craps shoot".

Table 2 shows the results for an "educated" guess. This is a guess made in step 6 above and its probability calculation depends on the assumption that all solutions to a particular set of coupled constraints are equally likely. From the results it seems a though these probabilities may be being overestimated. I suspect that more complicated statistics than what I have considered may need to be applied.

The final table, table 3, shows the results for guesses made in the unconstrained region of the board. These are guesses made in step 7 of the algorithm described above and the majority of them are actually guesses made in one of the corners of the board. The results here seem quite

Estimated Probability of Success	***Number of Successes***	***Number of Observations***	***Actual Success Rate***
50%	8535	17006	50.2%
52% to 65%	1544	2586	59.7%
66%	21510	32274	66.6%
67% to 74%	3827	5465	70.0%
75%	12145	17039	71.3%
76% to 79%	3204	4156	77.1%
80% to 84%	82253	111689	73.6%
85% to 89%	75462	91848	82.2%
90% to 94%	55368	62680	88.3%
95% to 99%	11831	12600	93.9%

Table 2: Observed success rates for guesses among the constrained variables (educated guess).

Estimated Probability of Success	*Number of Successes*	*Number of Observations*	*Actual Success Rate*
16% to 48%	56	128	43.8%
50% to 59%	7743	14307	54.1%
60% to 69%	43851	65453	67.0%
70% to 74%	33973	46361	73.3%
75% to 78%	33361	43663	76.4%
79%	103223	129794	79.5%
80%	347816	433651	80.2%
81% to 84%	125695	153516	81.9%
85% to 89%	69973	81889	85.4%
90% to 99%	55847	63460	88.0%

Table 3: Observed success rates for guesses in the unconstrained regions of the board.

reasonable; expect perhaps for the highest probability category. The probability of success here is calculated as:

$$Prob\{success\} = 1 - expected\ number\ of\ mines / number\ of\ unconstrained\ squares$$

where the *number of unconstrained squares* is known with certainty, while the *expected number of mines* is the total number of mines remaining minus the number of mines the constrained squares are expected to require. Errors in the estimate of this probability must be due to an imprecise calculation of the number of mines a particular set of coupled constraints requires, and that calculation currently depends on the assumption that all solutions to the constraints are equally likely. Therefore, it is that assumption that needs attention if this work is to be continued.

Basic Performance Measurements

PGMS has a useful feature where the user can have any number of games played quickly and consecutively without a user interface slowing things down. After the games have been played, the total number of wins is output. For my tests, however, I wanted more than just the mean number of wins. I wanted some indication of the standard error of the mean so I could determine if differences between the various strategies was significant. Therefore, I altered PGMS to allow the playing of any number of sets of any number of games. From the results, the mean, variance, and standard error of the mean could be calculated. For all of the tests described below, I ran 100 sets of 100 games (10,000 games total) to get each data point.

I had three strategies to test:

- CSPStrategy,
- EquationStrategy, and
- SinglePointStrategy;

three levels of play:

- beginner (8x8 with 10 mines),
- intermediate (15x13 with 40 mines), and
- expert (30x16 with 99 mines);

and three game modes:

- standard rules (cannot lose on first step),
- hard rules (standard PGMS, can lose on first step), and
- hinted game (least dense region is given).

I obtained one data point for each combination of these test conditions so I have a total of 27 data points. The results are presented as three graphs (one for each game mode) in figures 2, 3, and 4. The standard error of the mean for all data points was 0.5 or less and has not been plotted. All differences seen are significant and reproducible.

Clearly, CSPStrategy is superior; however, I have to suspect that most of the difference in outcome is due to the difference in how each strategy starts a game. From the hinted game results (where a starting position was given) it can be seen that the difference between CSPStrategy and EquationStrategy is much smaller.

While making modifications to PGMS to provide statistics on the games played, I also modified it to measure the amount of CPU time consumed to play each game and output statistics on CPU use. All tests were run on an Apple Powerbook G3 (Pismo) 400MHz with 128MB of RAM. The JDK was obtained from the blackdown.org group and describes itself as:

```
Classic VM (build Linux_JDK_1.2.2_FCS, native threads, nojit)
```

Although PGMS was altered to measure the CPU time used by each strategy, and not the wall clock time, attempts were made to ensure that the machine had a light load or no load while the tests were being run. The results of these tests for the "standard rules" case are shown in figure 5.

From the little peek I took at the source code for EquationStrategy it seemed that this strategy is a less complicated strategy than CSPStrategy; however, it appears that EquationStrategy's implementation is quite poor from an efficiency point of view. To be fair to the author of that

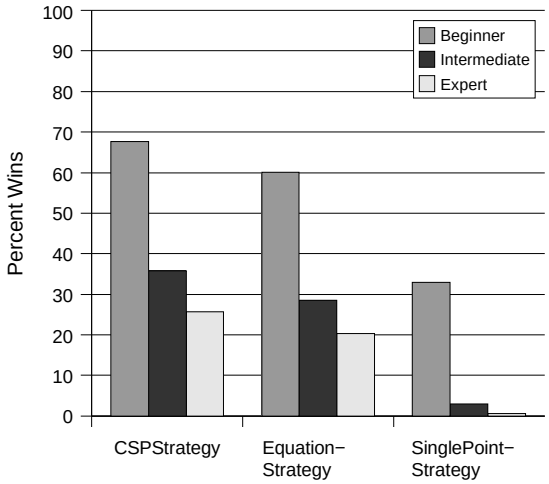

Figure 2: Win ratio for hard rules.

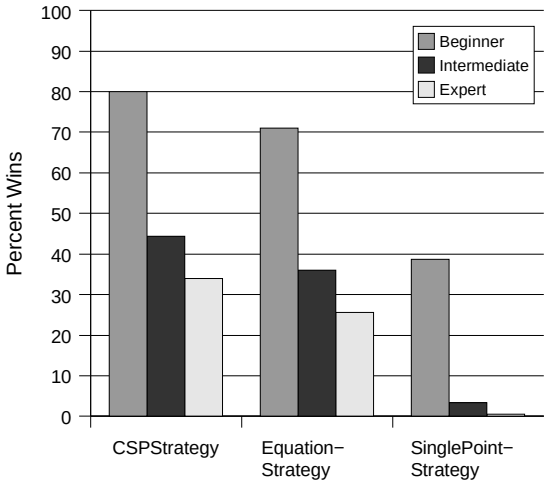

Figure 3: Win ratio for standard rules.

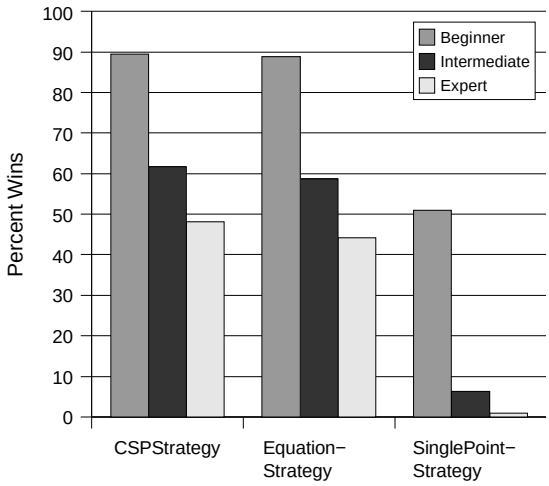

Figure 4: Win ratio for hinted games.

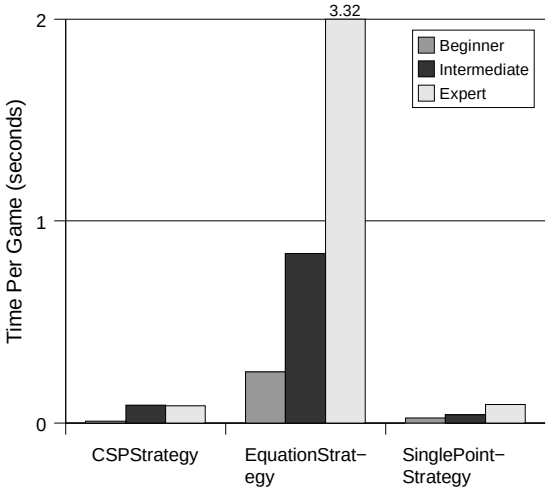

Figure 5: Average CPU time per game.

code, he was probably not very concerned with efficient use of processing power, but instead, he was probably more concerned with the win ratio that could be achieved. With CSPStrategy, solving the constraints for all solutions has time complexity that is roughly exponential in the number of variables, and therefore, even though the average case run time is quite good, it is possible for CSPStrategy to get stuck working on solving a large set of constraints that it has no hope of ever completing. This makes CSPStrategy somewhat inconsistent in its CPU use and I believe this is the reason why the CPU time required for the intermediate games (shown in figure 5) is nearly identical to the CPU time required for the expert games. Further analysis of the time complexity of CSPStrategy is given later.

From these basic tests, it was also possible to get an idea of why a typical game of minesweeper is won or lost. Using data from CSPStrategy playing 10,000 games at the expert level with standard rules, the pie chart in figure 6 was computed.

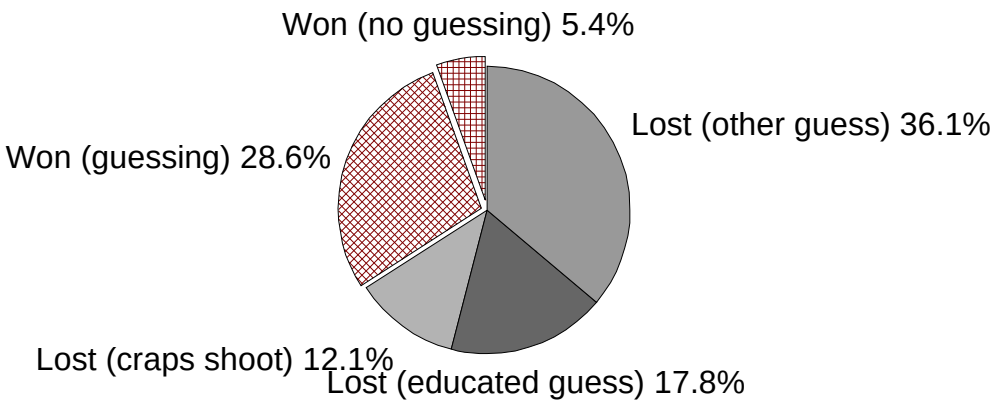

Figure 6: Distribution of reasons for losing a game (at expert level) of minesweeper.

Advanced Minesweeper Analysis

To perform some more advanced tests on CSPStrategy, I modified PGMS to allow games to be played on a board of arbitrary size with an arbitrary number of mines. The first advanced test I did was to determine how win ratio varies with mine density for the three standard board sizes. Figure 7 illustrates the results after 10,000 games for each board size (using standard rules). As can be seen, the data points were fitted to a logistic curve; however, the fit is not quite perfect.

Another test that was performed was to see how board size affects the win ratio when mine density is kept fixed. For these tests I fixed the mine density at *0.2* and tested a variety of square boards. Figure 8 shows percent wins as a function of board size. Clearly, a larger board presents a harder game even if mine density is kept constant.

I also used the board size tests to explore the computational complexity of CSPStrategy. Figure 9 presents those results on a semi−log graph with a trend line that represents an exponential dependance on linear dimension. My intuition was that CSPStrategy would require time that is exponential in the linear dimension (as opposed to exponential in the board area) because the variables that appear in a coupled set of constraints to solve often appear to lie in a roughly straight vertical or horizontal line. However, from figure 9 it appears that CSPStrategy may be sub−exponential in the linear dimension. The major outlier in figure 9 (the 10x10 case) was the result of an unusually large set of constraints having to be solved. This set wasn't the largest I've seen, but it had the largest number of solutions with just over 6 million.

The last set of tests I did was to determine how the shape of the board affects the difficulty of the game. I ran a series of tests on boards with 900 squares and 180 mines (and using the standard rules). The results are shown in figure 10. It seems that long, skinny boards are difficult the play on and that a player should prefer to play on a board that is as close to square as possible.

At the end of each game CSPStrategy played, it reported the size of the largest coupled set of constraints it had to solve and the number of solutions found. I considered the size of a

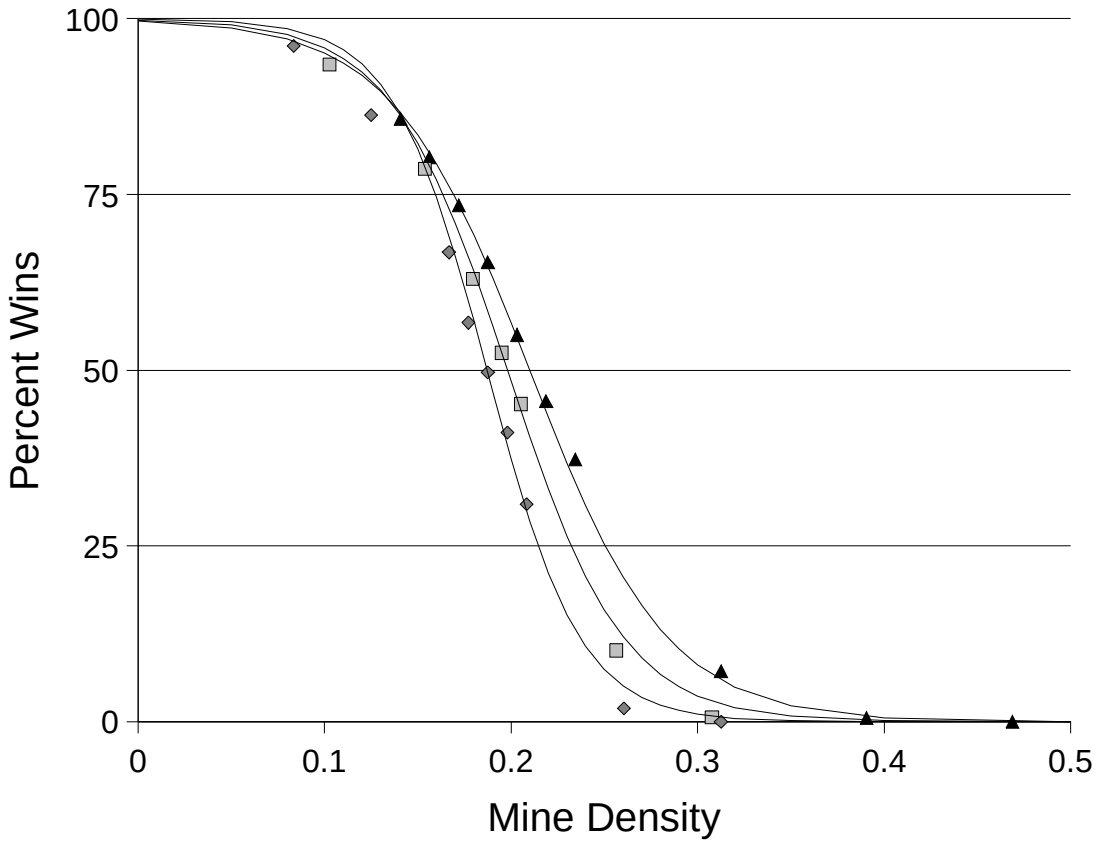

Figure 7: Win ratio as a function of mine density for the three standard difficulty levels: triangles for the beginner level, squares for the intermediate level, and diamonds for the expert level. Data points are shown here fitted to logistic curves.

coupled set of constraints to be the number of variables minus the number of constraints (ie. the number of degrees of freedom). In 600,000 games, the largest problem solved had 69 variables, 30 constraints (39 degrees of freedom), and 1,620,880 solutions. The set with the largest number of variables and constraints had 100 variables, 67 constraints (33 degrees of freedom), and 9,240 solutions. The set with the largest number solutions was mentioned above and had 46 variables, 16 constraints (30 degrees of freedom), and 6,060,240 solutions. The vast majority (more than 99%) of the sets of constraints solved had very few solutions (less than 1,000); even the relatively large sets (despite the figures presented here).

Conclusions / Discussion

By considering minesweeper to be a constraint satisfaction problem, an efficient algorithm for playing the game was developed. This algorithm makes use of CSP techniques for both finding an appropriate next move and for calculating the probability of success when a guess has to be made. No matter how good a player is, guesses will inevitably be required in most games.

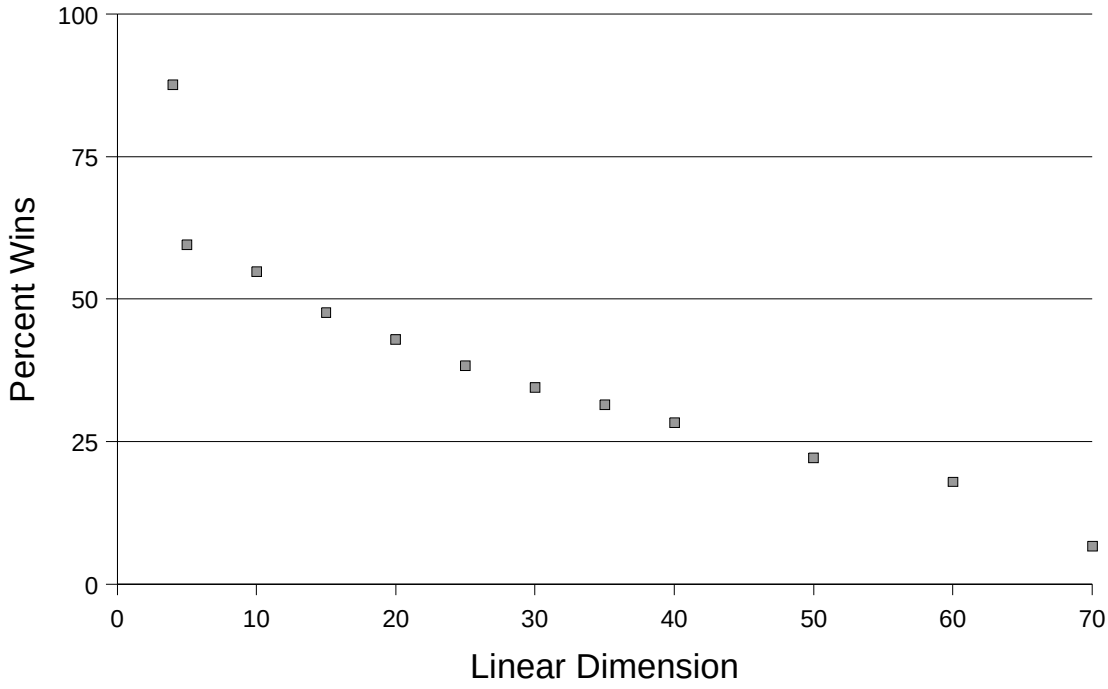

Figure 8: Win ratio as a function of linear dimension for games on a square board with a mine density of *0.2*.

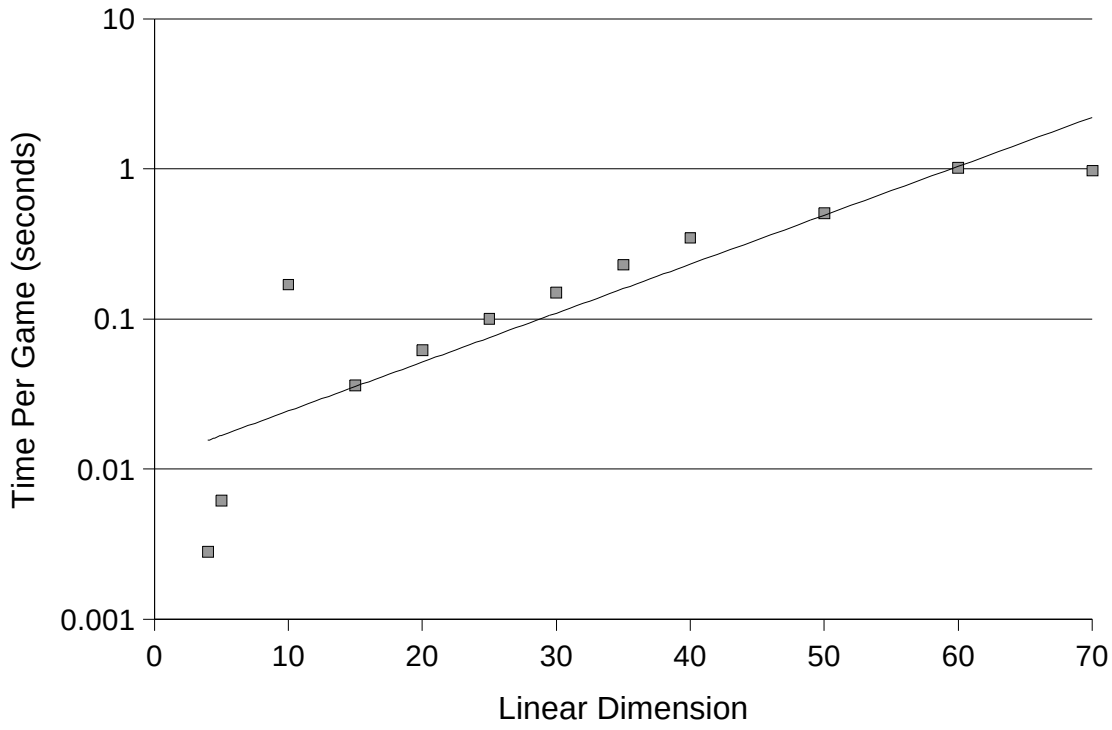

Figure 9: Average CPU time per game plotted on a semi−log graph and fitted to an exponential function.

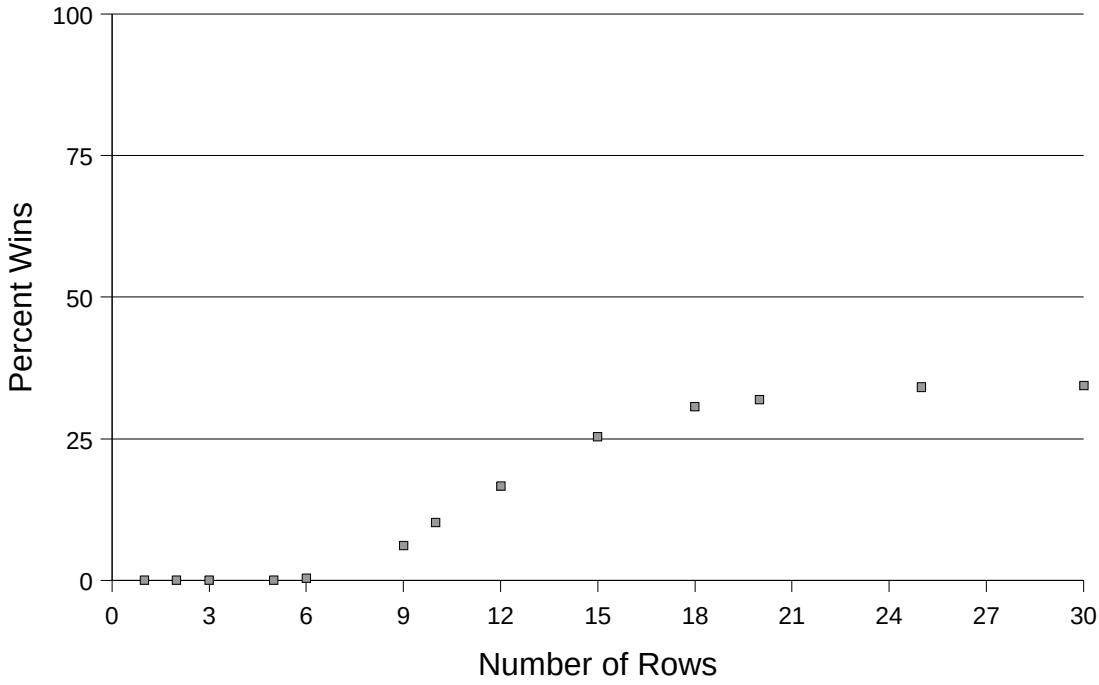

Figure 10: Win ratio as a function of board aspect ratio. All of these games were played on a board with 900 squares and 180 mines.

I see the following areas where CSPStrategy could be improved:

1. A better understanding of the likelihood of each solution found when solving coupled sets of constraints should improved the accuracy of the probability calculations when guessing is required.

2. Should the speed of the backtracking algorithm become a limiting factor, there is room for considerable improvements to the algorithm.

All sorts of additional tests and variations of the rules could also be tried.

Additional resources and source code for this project can be obtained at:

http://www.cs.utoronto.ca/~cvs/minesweeper/

References

Richard Kaye. Minesweeper is NP−Complete. *Mathematical Intelligencer* , volume 22 number 2, pages 9−15, 2000.

John D. Ramsdell. Programmer's Minesweeper (PGMS). http://www.ccs.neu.edu/home/ramsdell/pgms/