G. Biswas
10/15/2014

CS 360: Fall 2014
Programming Assignment 2 – Constraint Satisfaction

Due: Sunday, November 9, 2014 @ midnight.

Background:

The game of Minesweeper requires a player to determine the location of "mines" hidden randomly throughout a two dimensional grid i.e. "minefield". Each of the grid locations is initially covered by a tile. The player may open a tile, flag a tile as a mine location, or set a tile as a question mark. Clues describing the number of adjacent mines to a tile are displayed when the player opens a tile. A player "wins" by opening all of the non-mine tiles. A player loses when they open a tile containing a mine.

Minesweeper is easiest to learn by playing. When you uncover a square that doesn't have a mine under it, the game displays the number of mined squares adjacent to the current square (a number from 0 to 8). If this number is zero, the game must perform the uncover operation on all squares adjacent to it. 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 neighboring squares, edge squares have 5 neighbors, and corner squares have 3 neighbors. The figure above shows a partial game state with a number of tiles uncovered, and some still covered. The red marks on a square indicates that the player has correctly identified a bomb on that location. A number on a square indicates the number of neighboring squares that have bombs. No number on a square implies that none of its neighbors has a bomb.

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[1].

The Assignment:

For this assignment, you will 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) (http://www.ccs.neu.edu/home/ramsdell/pgms/). 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 (1) Single Point Strategy and (2) Equation Strategy. 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.

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.

There is still a question of which square on the board is the best one to attempt to uncover as the first move. Both the SinglePointStrategy and EquationStrategy start a game by randomly choosing a square on the board. Therefore, there is a small chance that these strategies lose the game in the very first move, but in general, a random choice of a starting position is a good idea if one has no idea which positions are better than others. Often one tends to start at a position close to the center of the square. (Why is this a better move probabilistically than choosing one of the edge squares?)

Also note that simply finding a square without a mine is not the best strategy to use, though it certainly is a good survival strategy. If, after the first step, a 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. In other words, even after the first move, the player cannot entirely play deterministically, so some element of guesswork may still be involved. Your goal is to use CSP methods to make the best de-

[1] Kaye, R. (2000). Minesweeper is NP-complete. The Mathematical Intelligencer, 22(2), 9-15.

cisions. And your approach will be evaluated by how well your approach compares to the EquationStrategy.

Hints about Approach:

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 neighboring variables (squares). This constraint simply states that the sum of the neighboring 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 neighboring 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. Develop a method that 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, your methods should 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*[2], each of these coupled subsets can then be solved to find all possible solutions. 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.

In **further steps**, develop your own approaches, where you make decisions on positions of mines and safe locations that you can probe further. Remember there will be cases where the coupled constraints will still require you to make guesses (because of lack of complete information), but then try and determine if some moves are less risky than others (i.e., the probability of finding a mine is lower than other possible locations). If it turns out that none of the constrained locations

[2] You have to develop the backtracking algorithm

are good choices, then a truly random decision has to be made on the next move. Again, use the interior versus corner square heuristic if that makes sense to you.

Experimental Strategy

Compare your minesweeper program against the EquationStrategy method by playing the beginning level game (8 × 8 square with 10 mines) a number of times so you can document: (1) the win ratios of the two approaches; and (2) average CPU time per game. Also compute the win to guess ratio ($\frac{\#wins/\#games\ played}{\#guesses/\#games\ played}$) – of course, *#games played* can be factored out. Similarly, the loss to guess ration may also be computed. Analyze the results, and discuss them.

For extra credit, you can extend the board size (say 10 × 10, 12 × 12, up to 15 × 15. Assume 15% of the board is covered by mines. What happens if you assume that a greater percentage of the board is covered by mines? (Keep it to < 20%).

What to turn in

The written assignment and programming assignment each will be graded for a maximum of 40 and 60 points, respectively For the final submission, you will use the online submission process to turn in your code and a README file. All code should be working and sufficiently well documented. You will be graded in part based on this documentation.

The README file should include:

1. A description of your approach and any special points that you think we should know about,
2. The pseudo code for your CSP algorithm.
3. Details on how to run your code

The written assignment should describe in greater detail

1. The structure of the program
2. A justification of the approach.
3. Experimental setup, discussion of results, and conclusions.