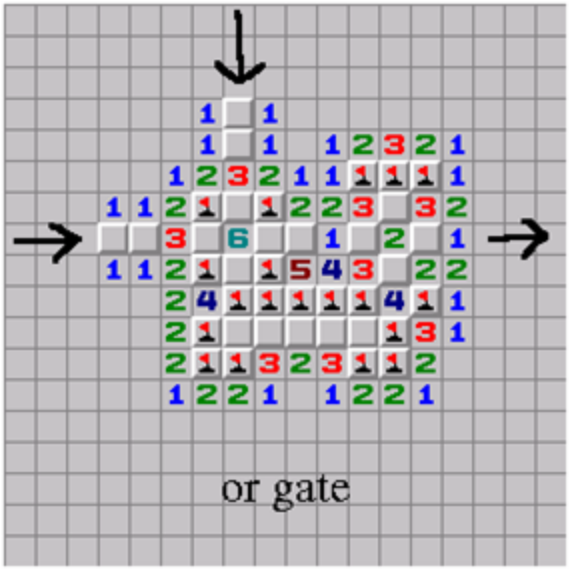
Max Minesweeper

By: Max Kofford

Introduction

Games and puzzles have long interested humans as a way to pass the time. One such puzzle that interested me from an early age was Minesweeper. Minesweeper is a timed puzzle game in which the player tries to uncover a board of squares by revealing them one at a time. The player loses the game if he uncovers any square that is marked as a "mine". How the player can avoid uncovering a mine is to look at the already uncovered squares. Squares that are uncovered show how many mines are directly touching that square including horizontal, vertical and both diagonal directions. A covered square's value can be discovered by figuring out the exact locations of mines using the shown numbers. Through careful calculation and an occasional guess a player can completely uncover the minesweeper board and win the game.

If a player can win this game of logic, can a computer also win it? This question was asked and answered in a paper written by Richard Kaye called "Minesweeper is NP-complete" for the Mathematical Intelligencer. In this paper he proposed the argument that solving one group of face up squares to find the mines near them for minesweeper is a NP-complete problem and therefore can be solved by a computer. He first argued that it was in NP by proposing that a computer could check a given solution (a set of squares near the uncovered squares) and verify that every uncovered square has exactly enough mines to satisfy it. This is a polynomial time algorithm because you would just place the given solution of mines into current board (would take the number of mines time) then scan through every face up square and verify that it has the correct number of mines near it (at most would be the number of squares on the board * 8 nearby squares for each). With minesweeper a part of NP, how he then argued that it was NP-complete was show a way to convert Boolean circuits into a minesweeper grid. If the newly created minesweeper grid was solved then the Boolean circuit satisfiability problem would also be solved. Because the Boolean circuit satisfiability problem is a member of NP-complete this would also make Minesweeper NP-complete.



An example of a boolean or gate converted into a minesweeper grid. http://sed.free.fr/complex/mines.html

The Project

The project that I did with regards to minesweeper was to convert the problem into a Boolean equation so that it could be solved using a solver such as MiniSat or a theorem solver such as Z3. In order to complete the project I actually divided it up into 4 main steps:

1. Make a minesweeper GUI and its underlying rules.
2. Make/encode a solver.
3. Try to create a second solver.
4. Add in some better guessing code.

1. Making Minesweeper

For this part I think I wanted to create my own minesweeper. I think it would have been possible to try to link my solver up to an already existing minesweeper but that would cause a lot more problems than it would solve. I would either have to find some source code for minesweeper, hack my way in to use one that's already been compiled or to somehow use image recognition to guess which values are on which squares. All of these methods seemed like they would be harder and less fun then creating my own so I created my own.

The minesweeper GUI i created.

It was relatively straight forward just creating a 2d array of buttons for the GUI and encoding the rules for minesweeper when a button was clicked. The most annoying parts was to make the array of buttons malleable for bigger sized minesweeper boards and to create the case where an empty square is clicked on which causes all the empty squares connected to it to also become visible.

2. Making the first Solver (first and second method).

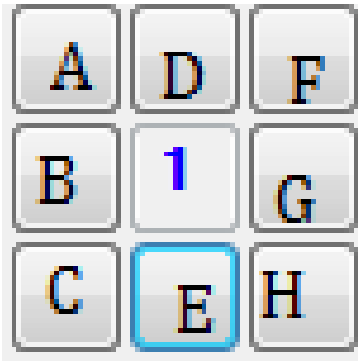
The general idea that I came up with behind this solver was that it would follow these steps:

- If at any time the input to the solver cannot find a solution then take a guess.
- The solver would generate a giant set of Boolean constraints that would constrain where mines could exist depending on the current visible set of numbers.
- The Boolean constraints would be inserted into Z3 and checked to see if a solution was found.
- If a solution was found then save that list of assignments for the variables and add a counter case that would prevent that exact solution and input it again into Z3 and repeat.
- If no solution was found then check all of the lists of assignments.
- If any variable in the lists never changed then that variable is true for the minesweeper field.
- If there was no variables that never changed then return that the solver cannot find a solution
- Else return all the variables that never changed among the list of assignments.

The Boolean constraints were generated by finding every square that was uncovered, was also a number, and had at least 1 covered square next to it. All the covered squares found this way were added as the variables for the Boolean constraint. The constraints were every possible combination of mines for the nearby squares that would satisfy the uncovered number. An example:

There would be a maximum of 1 mine in these 8 squares near the number 1. Therefore the mine could be in any of these 8 squares. The Boolean equation generated for this number would be:

(A ^ !B ^ !C ^ !D ^ !E ^ !F ^ !G ^ !H) V
(!A ^ B ^ !C ^ !D ^ !E ^ !F ^ !G ^ !H) V
(!A ^ !B ^ C ^ !D ^ !E ^ !F ^ !G ^ !H) V
(!A ^ !B ^ !C ^ D ^ !E ^ !F ^ !G ^ !H) V
(!A ^ !B ^ !C ^ !D ^ E ^ !F ^ !G ^ !H) V
(!A ^ !B ^ !C ^ !D ^ !E ^ F ^ !G ^ !H) V
(!A ^ !B ^ !C ^ !D ^ !E ^ !F ^ G ^ !H) V
(!A ^ !B ^ !C ^ !D ^ !E ^ !F ^ !G ^ H) V



The generated equation is a combination of every possible "n choose k" value for the empty squares near the number. The n is the set of empty squares near the number. The k is the number minus the number of nearby uncovered mines. This is the biggest cause of the equation to "blowup" because for the number 1 it only had 8 possible values but for the number 2 it had 28 possible values and the number 3 had 56 possible combinations. These combinations would occur for every number found that had empty squares nearby which could result in very large equations that would take several seconds sometimes for some of the bigger grids I generated. I initially considered finding and using an outside combinatoric generator but all the combinatoric code I could find for c# would generate one random n choose k instead of every possible combination so I ended up writing my own. As a base I decided to use the example provided at stackoverflow.com as an answer of a c++ implementation that would generate all possible combinations for n choose k.

The counter case that was generated when a solution came back was adding a case into the Boolean equation where that exact assignment of variables could not be found again. An example of this was: If the solution returned was A , !B , !C , !D , !E , !F, !G , !H then the counter equation that was added was : **!**(A ^ !B ^ !C ^ !D ^ !E ^ !F ^ !G ^ !H).

This would generate every possible solution to the current uncovered grid and therefore ended up being pretty slow. Especially since I had the combinatorics problem with possible positions for the mines there ended up being a huge blowup for the number of solutions that were generated. In order to try to reduce the number of solutions I decided to try and stop generating solutions early by detecting when variables hadn't changed in a while so that it could break early. This method I called method 2. The problem with that was that sometimes it would break too early and return some values that were incorrect because it had stopped too early

3. Making the second solver.

At this point in time the minesweeper solver was pretty slow and often failed due to guessing more often than it needed. I decided to try and create a second solver using a suggestion from my teacher

Zvonimir Rakamaric. He suggested that instead of generating all possible solutions that I should instead use the there exists functionality. For this solver I used these general steps:

- If at any time the input to the solver cannot find a solution then take a guess.
- The solver would generate a giant set of Boolean constraints that would constrain where there could exist a mine depending on the currently visible set of numbers.
- For every variable two solutions would be run:
    - The first solution would add a exists with the variable assigned to true
    - The second solution would add a exists with the variable assigned to false.
    - If the first solution returned unsat then there was a mine there.
    - If the second solution returned unsat then there isn't a mine there.
- After all solutions are generated then return all the cases where there was a unsat as the list of known values.
- If no unsat solutions were found then return that the solver cannot find a solution.

This used the same Boolean constraint generator that I used in the first solver and I had already figured out how to run Z3 with all the stuff so it was much faster to implement. This solver ended up being much faster and less guesses too. It was probably due to only having to run the number of variables * 2 solutions instead of all possible valid solutions.

4. Guessing better.

After finishing the second solver I noticed that even though the solver was generating solutions for uncovered numbers it was not being very good when it came to guessing. My initial guessing approach was to just guess randomly. This wasn't a very good guessing approach because often you can use the information about nearby squares in order to guess better. The second guessing approach I implemented was to guess near already found locations. If a square was uncovered then possibly it would have a higher chance of getting a square without a mine. This guesser also wasn't very good so I added a third one that would calculate the best square nearby all found values. It would scan through all uncovered numbers with nearby covered squares and calculate: (the number – nearby mines)/nearby covered squares. Whichever number was the highest value for this calculation had the highest chance of guessing and getting a non-mine.

The Results

After getting everything to work I then presented it in class. After presenting I decided to run some tests and see just how good my stuff was because it failed too many times in class. These were the results:

For a easy difficulty minesweeper board tested on 50 different games with the guessing and time taken (in seconds) averaged across the games:

	Guess1 - random	Guess2 - neighbor	Guess3 –best neighbor
Method1 - checkall	Won – 24 Lost – 26 Win guess – 2.25 times Lost guess -2.46 times	Won – 25 Lost – 25 Win guess – 2.2 times Lost guess -2.32 times	Won – 25 Lost – 25 Win guess – 1.24 times Lost guess -1.96 times

	Time taken - .376	Time taken - .360	Time taken - .285
Method2 – breakearly	Won – 23 Lost – 27 Win guess – 2.47 times Lost guess -2.55 times Time taken - .351	Won – 23 Lost – 27 Win guess – 1.74 times Lost guess -1.81 times Time taken - .379	Won – 25 Lost – 25 Win guess – 1.72 times Lost guess -2.04 times Time taken - .321
Method3 - exists	Won – 34 Lost – 16 Win guess – 1.88 times Lost guess -3.43 times Time taken - .463	Won – 36 Lost – 14 Win guess – 1.02 times Lost guess -1 times Time taken - .488	Won – 35 Lost – 15 Win guess – 1 times Lost guess -1 times Time taken - .498

This would seem to indicate that the method 3 is by far the best and that the better guessing algorithm increases the win ratio only slightly. Also, the amount of time taken does not take into account the delay with GUI updates. For testing I disabled the GUI to make it run faster which would account for much of the delay (when testing with GUI enabled the time taken was increased by around 7 to 10 seconds each).
The best guesser with the best method for the 3 game modes:

	Easy	Medium	Hard
Guesser 3 Method 3	Won – 35 Lost – 15 Win guess – 1 times Lost guess -1 times Time taken - .498	Won – 26 Lost – 24 Win guess – 1 times Lost guess -1 times Time taken - .454	Won – 10 Lost – 40 Win guess – 1.2 times Lost guess -1 times Time taken - .554

Conclusion

I think my solver does solve the minesweeper problem. For any given arrangement of displayed squares my solver can find a solution. The first two methods sometimes give incorrect solutions but the third solver always gives the correct solution. The only times that the third solver loses is on a guess. This would indicate to me that I need to improve my guesser even further because that is the problem whenever the third method fails. The time for solving the hardest mode for minesweeper is still far above the speed a human can solve it. According to the minesweeper Wikipedia the best verified recorded time for a human on the hard difficulty was 32 seconds. My minesweeper solver on the hard difficulty even with the GUI enabled usually solves it in less than 10 seconds. If minesweeper was a problem that needed to be solved quickly, my program could do it.

Resources / sources:

Kaye, Richard (2000). "Minesweeper is NP-complete". *Mathematical Intelligencer*
http://www.cs.montana.edu/courses/spring2004/spring2004/current/513/resources/minesweeper.pdf

c++ n choose k implementation:
Matthieu N -

http://stackoverflow.com/questions/5095407/n-choose-k-implementation

Minesweeper world record -http://www.minesweeper.info/worldranking.html