ADSA-4A-IBO1

Final Workshop

Workshop n°8

BLAZE Dominique – BENTCHIKOU Nour
06/12/2017

Summary:

This project written in Python and PyQt4 is just another minesweeper; but this one has a helper, and even better, an auto-solver (which plays all the game for you). Different approaches are possible to

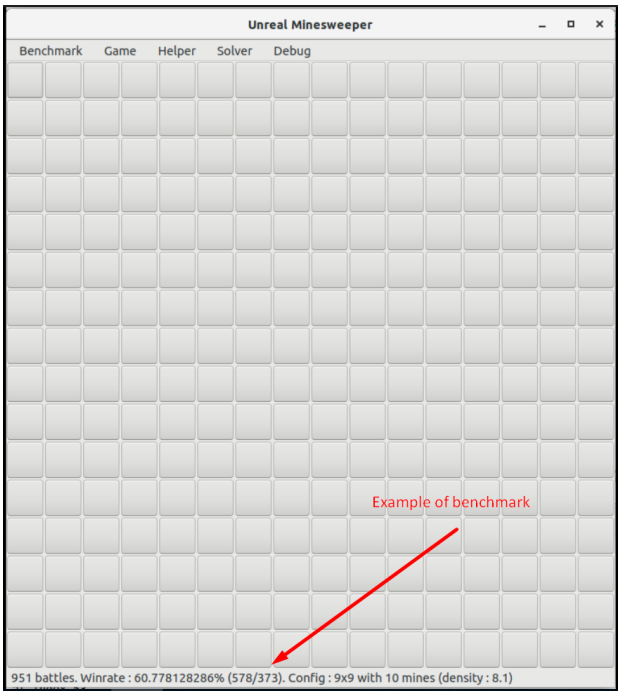
solve minesweeper solutions; we are using here 3 levels of solver: trivial known patterns, probabilistic approach, and a combinatory approach (using also probabilities).

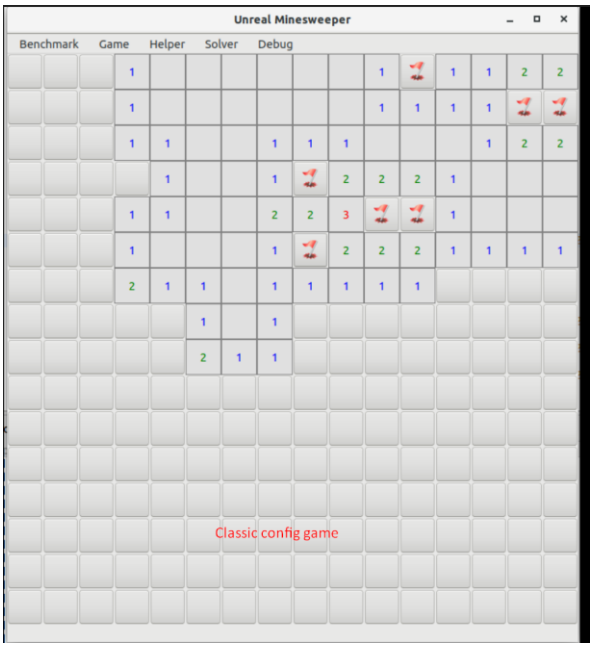
Software features:

We have been a bit exigent about our software features. There is a non-exhaustive list, with some screen shots:

- Classic minesweeper software: all rules are implemented (safe first round, auto-discover of empty area …)
- Three levels of helper are available in “Helper” menu
- An auto-solver in “Solver” menu is available with a step by step or solve all game option for each level (with eventually a delay to allow user to see steps)
- A benchmarking system in menu “Benchmark” to compute the win rate of each approach that depends on the number of mines density and/or the board dimensions.
- Debugging features (enable/disable the display of some exhaustive data in GUI)

The project follows a pattern close to the MVC, it allows us to run a game without view (in order to run benchmarks faster). We have divided benchmark speed by at least 100 with this trick (that’s logical …)





Theory

Generic solver algorithm

The solver algorithm is the same for each level. In fact, only the computed probabilities of each case depend on the level. For instance, with the first level, the computed probabilities are only 0 (safe) or 1 (mined). In the second level, each square has a computed probability, and in the last level, these probabilities are more accurate (and some squares with “medium probability” in level 2 are insured as safe or mined in the last level 3, because this last one considers all board’s constraints).

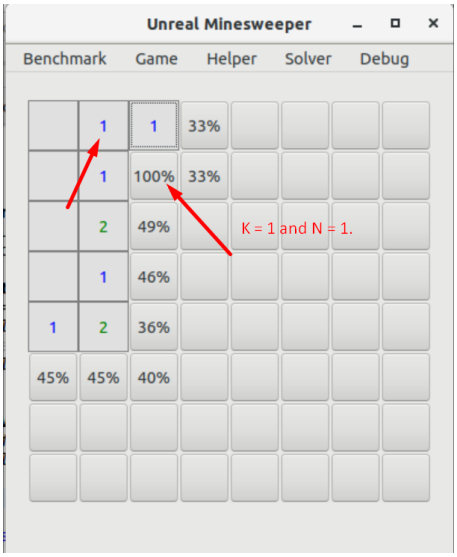
So, we have the same algorithm for each level (with eventually a light variant for level 3). The rules are simple: Always resolve trivial patterns when they exist. Otherwise, click on the square with the smallest mine probability. However, if this probability is bigger than the “average board probability” (which is dynamically computed), a random unknown square is triggered (here we can use a variant in level 3).

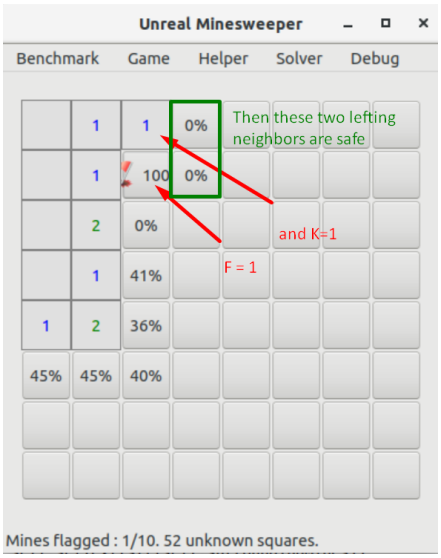
This is a simplified flowchart:

Level 1: trivial patterns

We will be brief about this level; these patterns are well-known, and we have mainly two trivial cases. Consider K the number displayed in a case: we have N unknown neighbors (between 1 and 8), and F the number of flagged neighbors.

- If K=N+F, the N neighbors are mined (F can be null)
- If F=K, the N neighbors are safe (more trivial)





You can see results of this level in the benchmark part at the end of this report.

Level 2: Probabilistic approach

First, consider that each square is indexed by an id, and not by coords [x,y]. So, it's possible to use our algorithm with another kind of grid (for instance, hexagonal grid, or in a 3D minesweeper). Indeed, each "numbered square" matchs to a "constraint set".

Here is an example with a 5x5 grid. Squares are indexed as shown below:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Let's say that the square number 6 is revealed, and contains the number « 1 ». The probability for each neighbor box to be mined is 1/8 (12,5%) – so by extension K/N.

0 12,5%	1 12,5%	2 12,5%	3	4
5 12,5%	**1**	7 12,5%	8	9
10 12,5%	11 12,5%	12 12,5%	13	14
15	16	17	18	19
20	21	22	23	24

(Example 1)

Let's assume now, that a second box is revealed.

Considering only the set {11, 12, 13, 16, 18, 21, 22, 23} (so N = 8) and knowing that K=3, probability for each box to contain a mine is K/N = 3/8 = 37.5%

0	1	2	3	4
5	6	7	8	9
10	11 37.5%	12 37.5%	13 37,5%	14
15	16 37,5%	**3**	18 37,5%	19
20	21 37,5%	22 37,5%	23 37,5%	24

We will now take into acount the existing pairing of these two information. In that case we will calculate an average of probabilities for the boxes in common. It means that the probability of the set {11, 12} is : $\frac{\frac{1}{8}+\frac{3}{8}}{2} = \frac{1}{4} = 0{,}4$

0 12,5%	1 12,5%	2 12,5%	3	4
5 12,5%	**1**	7 12,5%	8	9
10 12,5%	11 40%	12 40%	13 37,5%	14
15	16 37,5%	**3**	18 37,5%	19
20	21 37,5%	22 37,5%	23 37,5%	24

Level 3: Constraint propagation approach

We use here, the same model as seen above, except that the set is not defined by the average probability of boxes. Let's explain thanks to example 1; the only constraint of the set {11, 12, 13, 16, 18, 21, 22, 23} is « this set contains 1 mine and 0 flag ».

We enumarate the number of configuration possible, and for each box we will keep the number of configuration where the square was be mined .

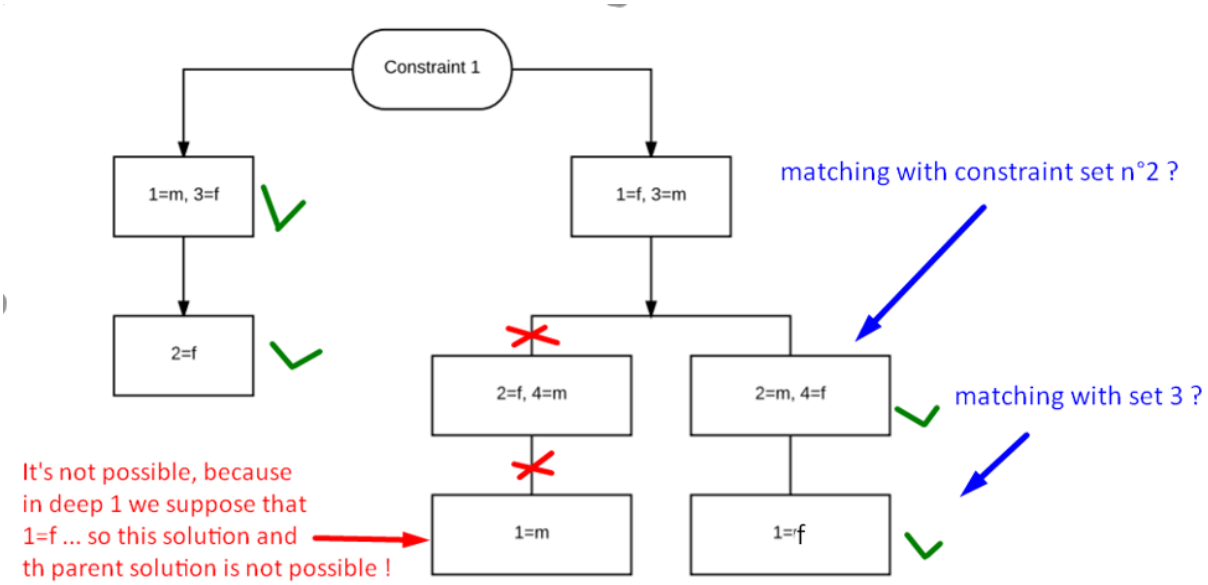
Each configuration implies new constraints in the sets that have boxes in common. We test recursively all the possibilities. As a schema often speaks better than words, here is an example with two sets with short constraint (so the size of our tree is reasonable).

Consider this set of index: {1, 2, 3, 4} and these 3 constraints:

{1, 3} -> 1 (means the set {1, 3} contains one mine)
{2, 3, 4} -> 2
{1, 2} -> 3

By convention, 'f' means the square is free (safe), and 'm' means the case is mined.

We can build with this a tree of possibilities (this method is called "Backtracking" in the world of CSP - Constraint Satisfaction Programming).



Here, we have only two possible solutions (indeed, the third constraint had eliminated an edge).
Now, we can associate index -> nb_mined_solution/nb_possible_solution
Here it's :

1 -> 1/2 (means the square 1 is mined one time for 2 possible configuration – matching with all constraints)
2 -> 1/2
3 -> 1/2
4 -> 0/1

So, we have a mine probability of 50% on square n° 1, 2 and 3 but we know that square 4 is safe! (because there is no possible solution where 4 is mined …)

The deep of the three is the number of coupled sets … and children of a parent are all possible configuration with the given parent configuration. This work is programming in python with the usage of customized iterables (with yield keyword). So, the iterables which browse all constraints are real-time updated.

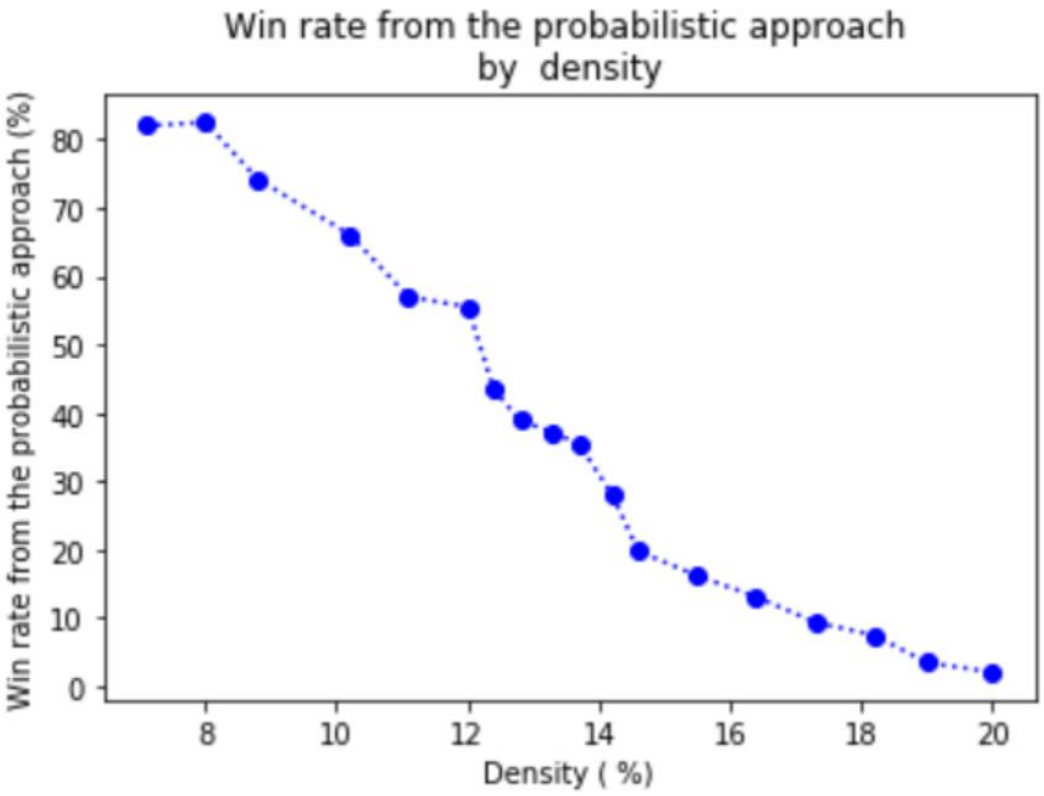
We are trying to optimize it by post-processing order of constraint sets by "importance order" (nearest sets or more coupled sets). But the complexity of this algorithm is still in O(2^n). However, it's possible to limit deep of tree; the post-process order is in this case a decisive factor of winrate.

Results

Thanks to our benchmarking system, we have compute the average winrate of our algorithms depending mine density or board length.

	Lvl 1	Lvl 2	Lvl 3
Beginner (9x9, 10 mines)	43.2%	57.8%	85.2%
Beginner (8x8, 10 mines)	27.3%	36.1%	73.8%
Intermediate (16x16, 40 mines)	9.5%	18.9%	65.2%
Expert (16x30, 99 mines)	0.0%	0.19%	27.9%

We have also build a special graph for level 2, computing winrate depending of the mine density (for a 15x15 board). We can see that this algorithm is rather good with low density, but it fastly become poor with a little more mine density …



Resources

https://www.info.ucl.ac.be/~pvr/minesweeper.pdf
http://www.cs.utoronto.ca/~cvs/minesweeper/minesweeper.pdf
http://kti.mff.cuni.cz/~bartak/constraints/propagation.html
http://www.cmears.id.au/articles/mines-solver.html
http://www.minesweeper.info/articles/MinesweeperStatisticalComputationalAnalysis.pdf
http://www.minesweeper.info/articles/InteractiveConstraintBasedApproachToMinesweeper.pdf
http://www.minesweeper.info/articles/MinesweeperOnGraphs(Golan).pdf
https://mrgris.com/projects/minesweepr/
https://github.com/mortii/Minesweeper
https://github.com/madewokherd/mines
https://delphi.developpez.com/defi/demineur/defieur/
https://delphi.developpez.com/defi/demineur/topic/
http://pyqt.sourceforge.net/Docs/PyQt4/classes.html

And more ...

Example of game entirely resolved by the solver (with CSP approach):