HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Artificial Intelligence

Minesweeper

Final Project

March 30, 2017

Jón Steinn Elíasson

jonse07@ru.is

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

Minesweeper is a single player logic puzzle game popularized by windows but originally from the 1960s. [1] The game is played on a grid of cells where some cells contain a bomb. Originally all cells are closed and the goal is to open all that do not contain a bomb.

Definition 1.1. A *Minesweeper board* $\mathcal{B} = (B, b)$ is a pair of $n \times k$ grid of cells,

$$B = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, k\} \subset \mathbb{N}^2$$

where n and k are the number of cells horizontally and vertically respectively and a function $b : B \to \{0, 1\}$ that maps each cell to the number of bombs it contains.

Example 1.1. The Minesweeper board in figure 1 shows the grid and the function b of each cell. It is defined by $\mathcal{B} = (B, b)$ where

$$B = \{(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)\}$$

and

$$b(x, y) = \begin{cases} 1 \text{ if } (x, y) \in \{(1,2), (3,2)\} \\ 0 \text{ otherwise} \end{cases}$$

1	0	0	0
2	1	0	1
	1	2	3

Figure 1: Minesweeper board example

Definition 1.2. For a cell $(x, y) \in B$ in a Minesweeper board $\mathcal{B} = (B, b)$, the set of surrounding cells,

$$\text{adj}(x, y) = \{(x + i, x + j) : (i \neq 0 \vee j \neq 0) \wedge i, j \in \{-1, 0, 1\} \wedge (x + i, x + j) \in B\},$$

is called the *adjacent cells*.

Example 1.2. Table 1 shows the adjacent cells for all cells in the Minesweeper board in figure 1.

Cell	Adjacent cells
$(1,1)$	$\{(1,2), (2,1), (2,2)\}$
$(2,1)$	$\{(1,1), (3,1), (1,2), (2,2), (3,2)\}$
$(3,1)$	$\{(2,1), (2,2), (3,2)\}$
$(1,2)$	$\{(1,1), (2,1), (2,2)\}$
$(2,2)$	$\{(1,1), (2,1), (3,1), (1,2), (3,2)\}$
$(3,2)$	$\{(2,1), (3,1), (2,2)\}$

Table 1: Adjacent cells

Definition 1.3. Given a Minesweeper board $\mathcal{B} = (B, b)$, let $E = \{(x, y) : (x, y) \in B \wedge b(x, y) = 0\}$, that is the set of cells with no bomb. For any cell $(x, y) \in E$, we say that the *adjacency number* of (x, y) is $\sum_{c \in \text{adj}(x,y)} b(c)$.

Example 1.3. Table 2 shows all adjacency numbers for the Minesweeper board from figure 1.

Cell	Adjacency number
$(1, 1)$	1
$(2, 1)$	2
$(3, 1)$	1
$(2, 2)$	2

Table 2: Adjacency numbers

When playing the game, if a cell does not contain a bomb it shows the adjacency number when opened. Figure 2 shows an example of a board where all cells have been opened, marking bombs with X. The game is lost by opening a cell with a bomb.

0	2	X	X
0	2	X	3
1	3	2	2
X	2	X	1

Figure 2: Complete 4×4 minesweeper board

Although the game is simple, it can be hard to win. At some point you might have to sort to guessing, either if the player can not conclude anything from the given table or there is nothing to conclude (regardless of the player). Solving minesweeper has in fact been shown to be NP-complete [2].

The main objective of this assignment is to create an agent that plays minesweeper successfully. This is a constraint satisfaction problem where the potential variables are the closed cells, their domain $\{0, 1\}$ and the constraints are the requirements formed by the adjacency numbers. There is also an element of guessing when nothing can be concluded from the constraints.

2 Motivation

The project is chosen from a list of suggested topics. There were many interesting topics suggested but I chose Minesweeper mainly because I used to play it a lot when I was younger. Overall, I think it's a very interesting game where with simplistic rules but difficult strategies which I think is a good characteristics for a game.

A GUI for Minesweeper is also fairly simple and although not a requirement, I wanted to have something visual. It does add a lot to a project like this. I also thought it might be good idea to do a large programming assignment on CSP since we covered it a lot but did not have a large assignment for it.

3 The solution

3.1 An overview of the approach

The board generation is done by the GUI controller and the only information passed between it and the agent is the next move of the agent and the resulting adjacency number if not a bomb. The board is completely random and can have a bomb at the first cell the agent chooses.

Figure 3 shows a simplified version of how the agent works. The red arrows are information being passed while the blue are state transitions, when there are no pending moves.

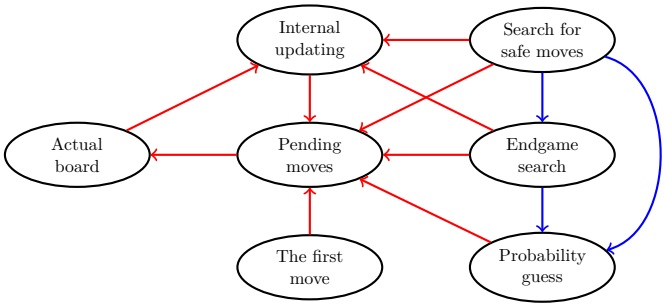


Figure 3: Agent overview

Until a game is lost or won, the agent is asked for his next move and the resulting adjacency number sent back. The agent has a collection of pending moves which is initially contains the opening move. If any moves are pending, some move is sent to the GUI controller and removed from the pending moves.

The agent keeps track of his own board, from his perspective. At any time the agent receives an adjacency number or finds a bomb, he updates his internal perspective of the board and adds, removes or changes constraints. This is repeated for any bomb that he does find when updating. All safe cells he finds are added to the pending moves.

If the collection of pending moves becomes empty, we split the constraint into groups, disjoint of variables and search each for guarantees. Any bombs found are passed to internal updating while safe cells are added to pending moves. If no safe cells are found we check if we are in the endgame and if so add an additional constraint and try again.

If no safe cells have been found at this point, we resort to guessing. This is done by calculating the probability of each variable containing a bomb and picking the least likely but we also consider the unknown cells that are not considered variables.

3.2 Board generation

There are 4 different types of board generation, set by two boolean flags, all of whom are randomly generated. The first flag prevents bomb being surrounded, shown by the shaded cells in figure 4.

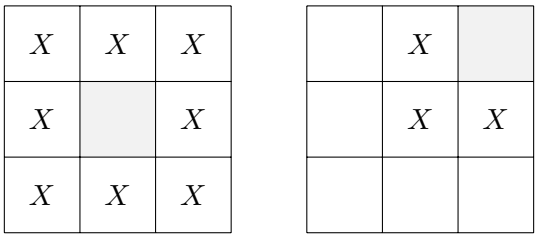


Figure 4: Generation with no surrounded bombs

The second flag recursively splits the generation into 2 or 4 parts (horizontally, vertically or both) to spread bombs more evenly. This is only done if the size of the part (or the whole board initially) is larger than a set max width/height. This is demonstrated in figure 5 where 3 bombs are on each side, horizontally.

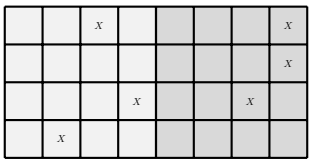


Figure 5: Generation with bomb spreading

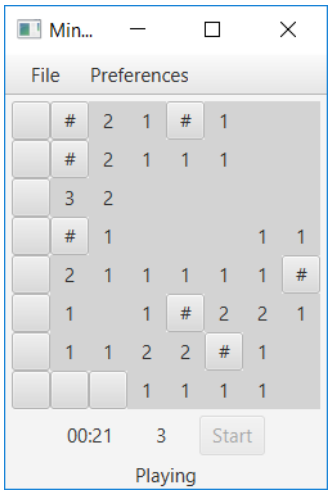
3.3 GUI

The GUI does not support changes of the generation mechanics and the final version has both flags as false so the board generated is completely random. There are 3 fixed level settings in the GUI shown in table 3, modeled after the initial Microsoft version.

Name	Width	Height	Bombs
Small	8	8	10
Medium	16	16	40
Large	24	24	99

Table 3: Different levels

Playing as a human is supported although mostly for fun. A screen shot of the GUI can be seen in figure 6, which is of the small level setting. The start button is enabled if you set the player as `Computer` and will play the game as the agent. The labels next to it are the bombs remaining and a timer.



Esc quit
F1 new game
F2 start (computer only)
1 small board
2 medium board
3 large board

Figure 6: The GUI

3.4 Opening

The opening move is a random non-edge cell. The reason for avoiding edges is that any adjacency number other than 0 will result in worse probabilities in adjacent cells than another random guess outside them while a adjacency number of 1 in a non-edge cell will result in the adjacent cells being more likely than a random guess outside them.

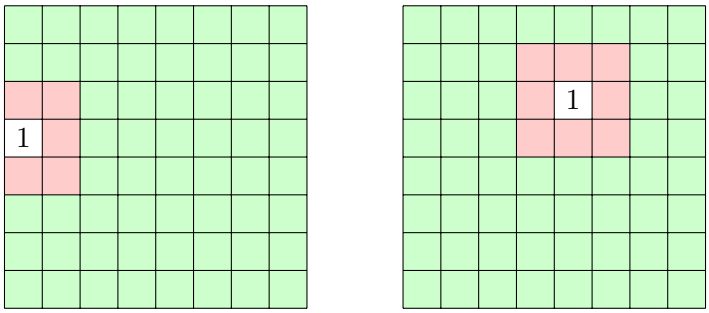


Figure 7: Opening move

In figure 7 we can see two scenarios, both a 8×8 board with 10 bombs, one where the opening cell is on

an edge and one where it isn't. In the first case, we have

$$P(\text{red}) = 1/5 = 0.2 \text{ and } P(\text{green}) = 9/(8 \cdot 8 - 6) = 9/58 \approx 0.155,$$

so we would guess at random in the green ones in this case. In the latter, we have

$$P(\text{red}) = 1/8 = 0.125 \text{ and } P(\text{green}) = 9/(8 \cdot 8 - 9) = 9/55 \approx 0.164,$$

and we would choose a red cell at random. Finding an adjacency number of 0 at start would be best but if we get an adjacency number of 1, we prefer to do that outside the edges since that produces the least likely cells to contain bombs. Any other adjacency number in the opening move results in guesses in 'the green cells', regardless of being on an edge or not.

3.5 Data structures

The key data structure is the one that keeps track of constraints. It is a map from the position of the adjacency number to the pair of updated adjacency number and unknown adjacent cells. It is updated dynamically so no unnecessary constraint are kept (except for possible duplicates which are filtered elsewhere) and each is updated when new information is discovered. The agent also stores all his findings about the board in a 2D array.

Another important data structure is the constraint groups. It is a map from a set of constraints to their variables. The constraints do keep their variables internally but the reason for having them as values in a map, is ease of use for the constraint framework. When created, it starts with an empty set and goes through all the constraints and if none of the variables belong to any of pre-existing groups, a new one is created. If the variables are shared with some pre-existing groups they removed from the map, merged into a new set into which the current constraint is added which is then added back to the map. This makes sure that the constraint groups are completely independent.

Suppose we were making truth tables to search for satisfiability in n boolean variables given some constraints, then there would be 2^n possible assignments. If we split those into k independent groups (of equal size) we have $2^{(n/k)} + 2^{(n/k)} + \ldots + 2^{(n/k)} = k \cdot 2^{(n/k)}$ possible assignments so this reduces the problem quite a bit. Taking a numeric example, suppose $n = 15$ and $k = 3$, then $2^n = 32768$ while $k \cdot 2^{(n/k)} = 96$.

3.6 Internal updating

Only unknown cells adjacent to a cell with a known adjacency number are considered variables. Any adjacency number is reduced for each known bomb we know about adjacent to it so the perspective of the board is different for the agent to how the GUI draws it. In figure 8, we see the same board with the same cells opened, where the one on the left is how the agent sees it while the right one is how the GUI shows it.

X		
1	3	
		X

X		
2	5	
		X

Figure 8: Agent's perspective compared to ours

Definition 3.1. An adjacency constraint is *trivial* if the adjacency number is 0 or equal to the number of variables.

Example 3.1. Figure 9 shows both cases of trivial constraints. The adjacency number is shown on a a/b format where a is what the GUI would display while b is what the agent would have stored internally. In the

left one, we have

$$\sum_{1 \leq i \leq 6} x_i = 0 \Rightarrow x_i = 0 \text{ for } i = 1, 2, \ldots, 6.$$

In the right one, we have

$$x_1 + x_2 = 2 \Rightarrow x_1 = x_2 = 1.$$

x_1	x_2	x_3
x_4	$2/0$	X
X	x_5	x_6

$3/2$	x_1	
x_2	X	

Figure 9: Trivial constraints

Whenever the agent receives an adjacency number, he updates internally. We remove the cell containing the adjacency number from any constraint containing it. The new adjacency number forms a new constraint if not trivial. The adjacent cells that are unknown must add up to the number we just received, subtracted by the known bombs adjacent to it.

If the constraint is trivial or any updated constraint become trivial, all unknown adjacent cells are either added to pending moves or known bombs (depending on which case of trivial constraint).

When we learn the position of a new bomb (including those learned from the trivial constraint) we also must update any constraint containing its cell. They are removed from the constraint and the adjacency number is reduced by one. If they become trivial they go through the same process as we mentioned earlier.

3.7 Search

If we have no pending moves, we try to conduct one logically. We begin by splitting the constraint into groups of disjoint constraints in variables. This also take cares of duplicates of any constraint by using a set.

2	2	2	2
x_1	x_2	x_3	x_4
2	2	2	2

Figure 10: Duplicate constraint

For each constraint groups, we create a model. These constraint are propagated and stored. Then we iterate over all variables and for each, we assume that it has a bomb. If that leads to a contradiction, we know it must be safe and add it to the pending moves. If the assumption didn't lead to a contradiction, we assume that it has no bomb and if that leads to a contradiction, we update internally. After each assumption, we remove the assumption and restore the model to the propagated state from the constructor.

The method of checking whether anything can be concluded about a cell is shown in listing 1. The conversion from our constraint group to a Choco model has been removed for simplification and replaced by a comment.

```java
private Model model;
private Map<Position, IntVar> varMap;
public MSModel(Set<ConstraintInfo> constraints, Set<Position> variables) {
  /* Convert our data structures to a Choco model and create a map between the two */
  this.model.getSolver().propagate();
}
public boolean hasBomb(Position position) {
  return containsContradiction(model.arithm(varMap.get(position), "=", 0));
}
public boolean hasNoBombs(Position position) {
  return containsContradiction(model.arithm(varMap.get(position), "=", 1));
}
private boolean containsContradiction(Constraint assumption) {
  model.getEnvironment().worldPush();
  model.post(assumption);
  Solution sol = model.getSolver().findSolution();
  model.getEnvironment().worldPop();
  model.unpost(assumption);
  model.getSolver().hardReset();
  return sol == null;
}
```

Listing 1: Searching with Choco

As an example, suppose we have the following constraint group (the variables are indexed by their position where $(1, 1)$ is the upper left corner),

$$\left\{x_{(1,1)} + x_{(2,1)} = 1, x_{(1,1)} + x_{(2,1)} + x_{(3,1)} = 2\right\},$$

seen in figure 11. Note that the figure is from the agent's perspective.

1	2	X

Figure 11: Constraint example

Assuming $x_{(1,1)} = 1$ or $x_{(1,1)} = 0$, then $x_{(2,1)} = 0 \wedge x_{(3,1)} = 1$ and $x_{(2,1)} = 1 \wedge x_{(3,1)} = 1$ satisfy the constraint respectively and we don't know if $x_{(1,1)}$ is 0 or 1. Now suppose that $x_{(3,1)} = 0$, then both $x_{(1,1)} = x_{(2,1)} = 1$ to satisfy the latter constraint which leads to a contradiction in the first so $x_{(3,1)}$ must contain a bomb.

3.8 Guessing

If we do not find any moves we must resort to guessing. We can still do better than random. Suppose we have the constraints shown in figure 12.

$x_{(1,1)}$	$x_{(2,1)}$	$x_{(3,1)}$
$x_{(1,2)}$	2	$x_{(3,2)}$
$x_{(1,3)}$	1	$x_{(3,3)}$
$x_{(1,4)}$	$x_{(2,4)}$	$x_{(3,4)}$

Figure 12: Board probabilities

If there are 2 bombs in the top row, there must be exactly 1 bomb in the bottom row, a total of $\binom{3}{2} \cdot \binom{3}{1}$ possibilities. If there is 1 bomb in the top row, there must be no bombs in the bottom row and one bomb in the middle rows, a total of $\binom{3}{1} \cdot \binom{4}{1}$ possibilities. There can't be more than 2 in the top row or less than 1 so we have covered all possibilities. The total number of possible solutions is therefore

$$\binom{3}{2} \cdot \binom{3}{1} + \binom{3}{1} \cdot \binom{4}{1} = 3 \cdot 3 + 3 \cdot 4 = 9 + 12 = 21.$$

How many times did each variable have a bomb? For the variables in the bottom row, they have a bomb one time for each of the $\binom{3}{2}$ cases of the top row having two. The middle row variables have a bomb one time for each of the $\binom{3}{1}$ cases of the top row containing 1. All of these variables therefore contain a bomb 3 times in total. The top row variables contain a bomb 2 out of 3 times for each variable in the bottom row, a total of 6 and one time for each of the middle row variables, a total of 4, yielding a grand total of 10. Now we can compute the probability for all variables,

$$10/21 \approx 0.476190476 \text{ and } 3/21 \approx 0.142857143$$

for the top row variables and everything else, respectively. As we can see, we are much better off choosing from a row, other than the top one.

The constraint framework provides us with the means to get all solutions to a constraint group so we can just count the occurrences of each variable being a bomb in a given set of solution and divide it with the number of solutions. We do that and store in a probability map.

We do however need to take into account those unknown cells that aren't considered variables. They can have a better chance of not containing a bomb. We do that by taking the sum of the minimum number of bombs in each solution, add that to the number of bombs we know of and subtract from the initial number of bombs. This is then divided by the number of unknown cells that are not variables which gives us the worst case probability of choosing outside the variables (since we took the minimum amount of bombs in each solution). If this probability is lower than the lowest one in the map, we choose at random from the non-variables. Otherwise we pick the one in the map with the lowest probability of containing a bomb. If many exists, we pick one at random.

3.9 End game

A situation can arise where you can use the number of bombs remaining to conduct information about a cell. These are rare but they do happen. The constraint for that is, that the sum of all unknown cells is equal to the remaining bombs. If we would apply this at start, we would include all the cells and wouldn't split any of the constraint groups, which would have a major impact on our search.

This constraint can be added in the later stage of the game when the remaining cells positions are not that many. We do so at the point of 15 unknown cells left but we do not use it unless we don't find any by our normal method. If we change to end game mode for the agent, it checks between searching with constraint groups and guessing, and if something is found, we can avoid guessing.

4 Results

4.1 Statistics

No goal in terms of time was set but his solver works really fast. Any playable level available in the GUI plays instantly. Playing 3000 games, 1000 of each size, takes about 5-6 seconds. Playing 100 games on a 50×50 board with 400 bombs takes about 2.5 seconds. A single 150×150 game with 3800 takes about 2.5 seconds to win. Timing of wins for various levels is shown in table 4.

Width	Height	Cells	Bombs	Time (s)
8	8	64	10	0
16	16	256	40	0
24	24	576	99	0
50	50	2500	500	0
150	150	22500	3800	2.5
500	500	250000	40000	336

Table 4: Timing of a winning play

A goal of a 50% win ratio was set but with no regards to which size. The following tables include statistics for 1000 games of each level, available in the GUI, and one custom game only playable in tests.

Games played	1000
Games won	846
Games lost	154
Games lost on first move	87
Win ratio	84.6%
Win ratio excluding first move losses	92.66%

Table 5: Statistics for the small map

Games played	1000
Games won	546
Games lost	454
Games lost on first move	149
Win ratio	54.6%
Win ratio excluding first move losses	64.16%

Table 6: Statistics for the medium map

Games played	1000
Games won	430
Games lost	570
Games lost on first move	188
Win ratio	43%
Win ratio excluding first move losses	52.96%

Table 7: Statistics for the large map

Games played	100
Games won	41
Games lost	59
Games lost on first move	19
Win ratio	41%
Win ratio excluding first move losses	50.61%

Table 8: Statistics for a 50×50 map with 400 bombs

From the simulations in tables 5, 6 and 7, the total number of times the end game saved us from guessing was also counted.

Small	Medium	Large
0	18	23

Table 9: End game constraint saves from guessing in 1000 games

4.2 Comparison

In the article *Minesweeper as a Constraint Satisfaction Problem* by Chris Studholme, he lists the his win ratio in a graph. The are about 68%, 35% and 25% in ascending order of difficulty. His hardest level is different from mine so if we disregard that, my solution has about 15% better win ratio in the smallest game and 19% in the medium one.

5 Future work

5.1 Improve probability model

As the probability is calculated now, we just take all possible solutions and count the occurrences of each variable containing a bomb in total. A group of variables belonging to a constraint group can have different number of bombs. If we would take into account, the probability of a constraint group containing some number of bombs in total (for all possible number of bombs) and use conditional probabilities we should get better results.

5.2 Constraint simplification

The constraints $x_1 + x_2 + x_3 + x_4 = 2$ and $x_1 + x_2 = 1$ can be simplified into $x_1 + x_2 = 1$ and $x_3 + x_4 = 1$. It would improve the agent if we were to handle these simplifications when splitting the constraint groups since they could potentially split some groups further.

5.3 Explore more constraints

A rather fun property of minesweeper is that the sum of adjacency numbers for any board is equal to the sum of adjacency numbers in its complement, that is where the bombs and empty cells have been swapped.

1	1	1	0	0
1	X	1	1	1
2	3	3	2	X
2	X	X	2	1
X	3	2	1	0

X	X	X	X	X
X	8	X	X	X
X	X	X	X	5
X	6	7	X	X
2	X	X	X	X

Figure 13: A minesweeper theorem on complements

In the example in figure 13, we have $3 \cdot 0 + 9 \cdot 1 + 5 \cdot 2 + 3 \cdot 3 = 28 = 2 + 5 + 6 + 7 + 8$. It would be fun to explore if this theorem (or others) could help solve the endgame.

6 Conclusion

Constraint satisfaction problem are usually hard to solve and often belong to high complexity problems in theory. We can however use various methods to solve them efficiently. This project is an example of that, with minesweeper being NP-complete.

After many days of tireless work, I'm pretty happy with the results. All my goals have been met and I have a nice little GUI to demonstrate what has been done. I had a lot of fun working on this and it was very satisfying to see it all come together (the data structures, the constraints, the probability, the GUI, etc). I would definitely recommend this as a programming assignment in the course.

A Material used

Table 10 lists everything that was used in the project.

Software	Usage
Java	Programming language
JavaFX	GUI
IntelliJ	IDE
JUnit4	Testing
Choco solver	Constraint programming
GitHub	Source control
LaTeX	Report, slides
Overleaf	Report, slides
Minesweeper strategy	Test cases
Minesweeper as a CSP	Research
Minesweeper Online	Play until GUI was working

Table 10: Material used

B Testing

The code is thoroughly tested, with almost 100% coverage without the GUI which is not tested. The missing part from the `MSAgent` is a `try-catch` block for a Choco contradiction expectation that will never be thrown.

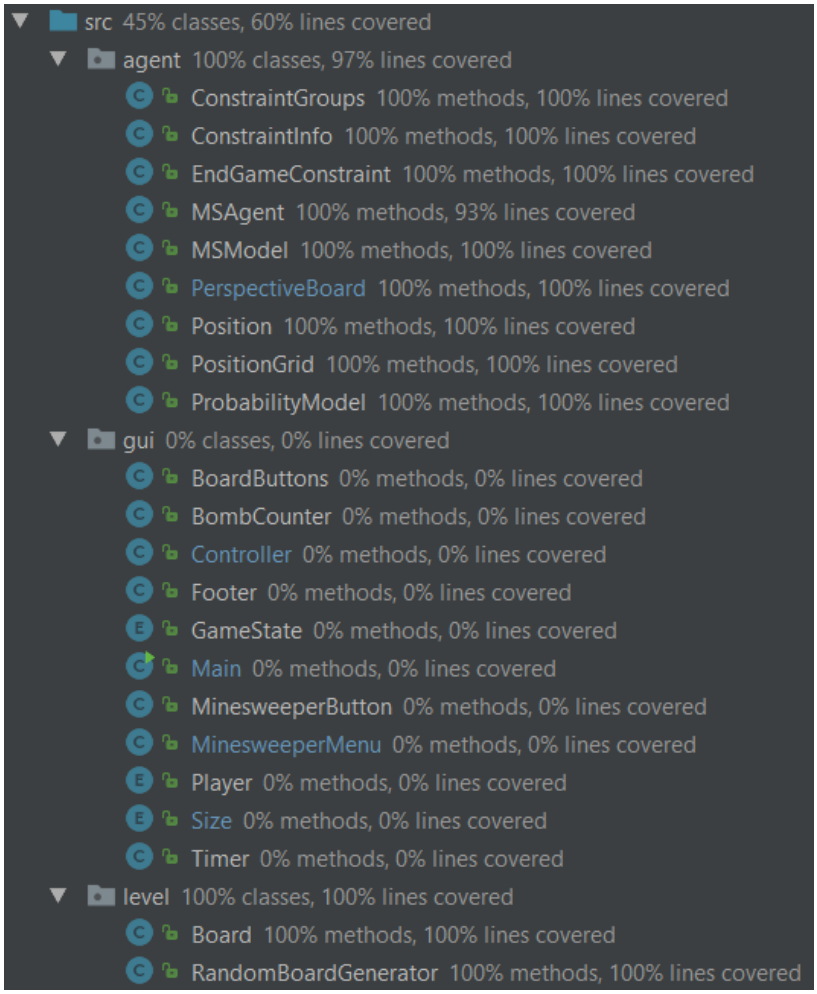


Figure 14: Code coverage

Highlights of tests include:

- Simulating several thousands of games, showing statistics.
- Comparing probabilities calculated by hand vs one with the constraint model.
- Testing wins on levels that don't need guessing by manually choosing first move.
- Internal updating and data structures.
- Contradicting assumption given specific game scenarios.
- Division of constraint groups.

C Source code

A GitHub repository can be found on my GitHub page and here is a direct download link. This includes the Choco library.

D Demo

Here is a video demo of the GUI and the solver in action on youtube.

References

[1] Minesweeper, Wikipedia, https://en.wikipedia.org/wiki/Minesweeper_(video_game), 6.3.2017.

[2] Kaye, Richard, *Minesweeper is NP-complete*, Mathematical Intelligencer, vol 22, number 2, pp9-15, 2000.