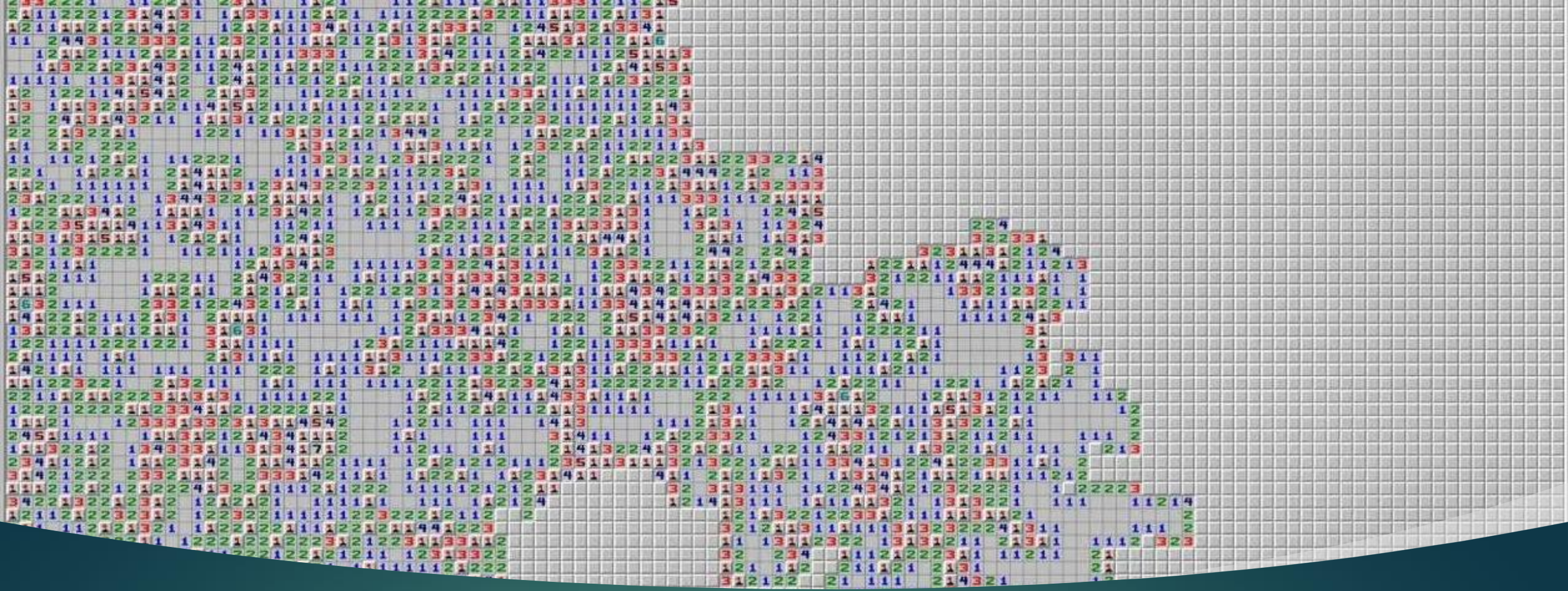


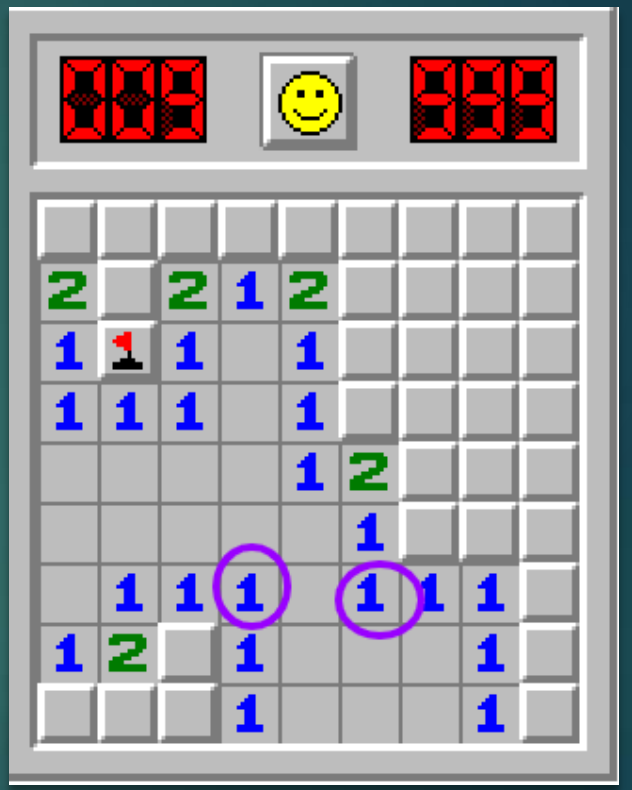
Knowledge Representation and Learning Project

MINESWEEPER

Alvise Dei Rossi #2004250

Generalities of the game

- **Objective of the game**: mines are hidden in a nxn grid. The player must discover all the mines by opening all the cells not containing them. At each iteration a cell is selected: if the cell contains a mine the game is lost, otherwise the number of mines in the surrounding cells is shown, which can be used to decide the next cell to be revealed. The game is won when all mines have been correctly discovered and all non-mine squares probed.
- **Some basic rules/observations**:
  - **Every cell either contains a mine, or a number between 0 and 8 (inclusive).** So in total a cell has 10 possible states.
  - **The first click is ALWAYS a safe click**. The game can't be immediately lost. **In many simplified versions** (which we will see at first), **the first click is always on a 0-cell**, allowing for an initial safe exploration.
  - **We can choose ourselves the dimension of the grid and the number of mines**. The more the density of mines in the grid, the harder to solve the game usually.
  - **The win cannot be guaranteed**, sometimes the best the player can do is to try to figure out which is the safest cell. In some cases it's down to pure luck, as multiple configurations are equally likely.



Problem encoding

- After choosing the size of the grid and the number of mines, the board is set-up: the position of the mines is randomly chosen (with the exception of the first cell to uncover, since the first-click is always safe). **The problem is represented as a (size×size) NumPy matrix, where elements in positions (i,j) (0 ≤ i,j ≤ size-1) are the number of adjacent mines or the number 9, which stands for the presence of a mine.** The board can be printed out anytime to follow the game: as a convention when it's printed out mines are represented by the letter M, un-probed cells are shown as a X, flagged mines (which are unprobed) are shown as F. For example an 8x8 grid with 10 mines:

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['0' '0' '0' '1' '2' 'M' 'M' '1']
1   ['0' '0' '0' '1' 'M' '3' '2' '1']
2   ['0' '0' '0' '1' '1' '1' '0' '0']
3   ['0' '0' '0' '0' '0' '0' '0' '0']
4   ['0' '1' '1' '1' '1' '1' '1' '0']
5   ['0' '1' 'M' '1' '2' 'M' '2' '0']
6   ['2' '3' '3' '2' '3' 'M' '3' '1']
7   ['M' 'M' '2' 'M' '2' '1' '2' 'M']
```

- **We use variables x_{ij} to indicate if in position (i,j) there's a mine** (x_{ij} = True) or not (x_{ij} = False). **These variables are mapped into integers for pysat:**

```
Position (0, 0), integer: 1
Position (0, 1), integer: 2
Position (0, 2), integer: 3
Position (0, 3), integer: 4
Position (0, 4), integer: 5
Position (0, 5), integer: 6
Position (0, 6), integer: 7
Position (0, 7), integer: 8
Position (1, 0), integer: 9
Position (1, 1), integer: 10
Position (1, 2), integer: 11
Position (1, 3), integer: 12
```

- **Initially the only knowledge the agent could have is a global equality cardinality constraint: In the entire grid there are n mines.** However, when the grid is large, this constraint produces a large number of clauses and/or auxiliary variables and slows down the SAT calls by a lot. It's better to not include it in the knowledge base as it is of little importance for most of the game, with a notable exception mentioned later, when it should be considered.
- **Every time a new cell (i,j) is uncovered, if it's not a mine, new knowledge is added:**

x_{ij} = False since there's no mine, and an equality cardinality constraint relative to the information of the cell: if the cell is showing that k adjacent cells contain mines, we encode that among the variables x_{ij} relative to its neighbors, k must be True. One way to do so is to express the cardinality constraint as a combination of at least k and not at most k, in CNF:

$$\bigwedge_{I \subseteq neighbors(x_{ij}),|I|=n-k+1} \bigvee_{x_{i'j'} \in I} x_{i'j'} \wedge \bigwedge_{I \subseteq neighbors(x_{ij}),|I|=k+1} \bigvee_{x_{i'j'} \in I} \neg x_{i'j'}$$

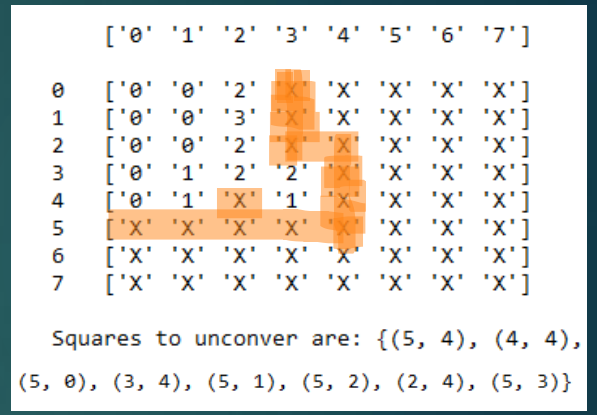
Both this way and more efficient cardinality encodings (Sinz) available in pysat are tested/used in SAT.

Legend for printed grids:

- **0-8** safe cell, giving information about the number of adjacent mines
- **X** unprobed/undecided cell
- **M** mine (in revealed board or lost game)
- **F** flagged mine (unprobed)

Details of the functioning of the logical agent

- We could at every step consider all the unknown cells, but **to scale up better, it's reasonable to consider at every step only the relevant cells**, i.e. the cells adjacent to the areas of the grid already explored (in orange in the picture). **An exception to this arises sometimes when global cardinality constraints allow in the endgame to infer that some (irrelevant so far) cells can't possibly contain mines**. To allow for this, all cells are checked when the remaining undecided cells are few (e.g. 5-10). Note that in this case the number of clauses introduced is acceptable.
- **For every relevant cell we can check if**, given what the agent knows so far, **it's a logical consequence that it is not a mine** (by checking if the conjunction of the knowledge base and the literal x_{ij} is unsatisfiable) **or a mine** (by checking if the conjunction of the knowledge base and the negated literal $-x_{ij}$ is unsatisfiable). **Essentially we proceed by sequential satisfiability checks**.
- **The agent then proceeds to probe all cells which are logically not mines.**



In the example picture: all relevant squares (in orange) are checked using SAT:

The agent, given the equality cardinality constraints of the uncovered squares and the assumptions of them not being mines **concludes that logically 8 of the relevant cells can't possibly be mines:**

- (5,0) and (5,1) trivially can't be mines since in (4,0) there's a 0.

-All adjacent squares to the 1 in (4,3) are not mines (i.e. (5,4),(4,4),(3,4),(5,3),(5,2)), with the exception of (4,2) which MUST be a mine to satisfy the condition imposed by the cell showing 1 in (3,1).

- (2,4) can't be a mine either because for sure (4,2) is a mine and (2,3) too (along with (1,3) to satisfy the condition of 2 in (2,2)), so the condition imposed by the 2 in (3,3) is satisfied by them already.

A more complete example using only SAT

Let's show an **example of the agent solving a small 6x6 Minesweeper game with 5 mines** (without getting stuck).

We **assume the first click is a 0** to allow for early exploration (otherwise it gets stuck immediately).

```
First-click: (2, 2)

UNCOVERED BOARD CONFIG

    ['0' '1' '2' '3' '4' '5']

0   ['M' '1' '0' '1' 'M' '1']
1   ['1' '1' '0' '2' '2' '2']
2   ['0' '0' '0' '1' 'M' '1']
3   ['1' '1' '1' '1' '2' '2']
4   ['1' 'M' '1' '0' '1' 'M']
5   ['1' '1' '1' '0' '1' '1']
```

First click (2,2) is a 0-cell. The board is generated. (The agent has no knowledge of it)

```
    ['0' '1' '2' '3' '4' '5']

0   ['X' 'X' 'X' 'X' 'X' 'X']
1   ['X' 'X' 'X' 'X' 'X' 'X']
2   ['X' 'X' '0' 'X' 'X' 'X']
3   ['X' 'X' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X']

Squares to unconver are: {(1, 2), (3, 2),
(1, 3), (3, 3), (3, 1), (2, 1), (2, 3), (1, 1)}
```

System infers that all 8 the neighbor cells to the first-click are not mines and can be uncovered (probed).

```
    ['0' '1' '2' '3' '4' '5']

0   ['X' 'X' 'X' 'X' 'X' 'X']
1   ['X' '1' '0' '2' 'X' 'X']
2   ['X' '0' '0' '1' 'X' 'X']
3   ['X' '1' '1' '1' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X']

Squares to unconver are: {(0, 1), (3, 0),
(2, 0), (1, 0), (3, 4), (0, 2), (0, 3)}
```

Trivially all cells adjacent to 0-cells are safe to probe ((0,1),(0,2),(0,3),(1,0),(2,0),(3,0). The reasoning for (3,4) is more complex: given the presence of a 2-cell in (1,3) and a 1-cell in (2,3) and the fact that, as said, (0,2) and (0,3) are not mines, there must be a mine in (0,4) and one in either (1,4) or (2,4) to satisfy the 2-cell. Which means, since the 1-cell in (2,3) would already be satisfied by one of those, that (3,4) can't contain a mine.

```
    ['0' '1' '2' '3' '4' '5']

0   ['F' '1' '0' '1' 'F' 'X']
1   ['1' '1' '0' '2' 'X' 'X']
2   ['0' '0' '0' '1' 'X' 'X']
3   ['1' '1' '1' '1' '2' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X']

Squares to unconver are: {(4, 4), (1, 4), (4, 3), (4, 2), (4, 0)}
```

Given the new information, the system begins to flag as mines squares that has to be mines ((0,0) and (0,4)). It also concludes that (1,4) is safe, as the 1-cell in (0,3) already is adjacent to a mine, which implies (due to the presence of the 2-cell in (1,3) that (2,4) is a mine too, hence (4,4),(4,3),(4,2) are safe. Leaving (4,1) to be for sure a mine to satisfy the 1-cell in (3,2), hence (4,0) is safe.

```
    ['0' '1' '2' '3' '4' '5']

0   ['F' '1' '0' '1' 'F' 'X']
1   ['1' '1' '0' '2' '2' 'X']
2   ['0' '0' '0' '1' 'F' 'X']
3   ['1' '1' '1' '1' '2' 'X']
4   ['1' 'F' '1' '0' '1' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X']

Squares to unconver are: {(5, 4), (5, 5), (1, 5),
(0, 5), (5, 0), (5, 1), (2, 5), (5, 2), (5, 3)}
```

Easy to see, thanks to the two newly flagged mines in (4,1) and (2,4) and to new cell probed that (5,0),(5,1),(5,2),(5,3),(5,4),(0,5),(1,5),(2,5) are safe. To satisfy the 2-cell in (3,4) a mine must be in either (3,5) or (4,5), hence given that there's a 1-cell in (4,4) then (5,5) must be safe too.

```
    ['0' '1' '2' '3' '4' '5']

0   ['F' '1' '0' '1' 'F' '1']
1   ['1' '1' '0' '2' '2' '2']
2   ['0' '0' '0' '1' 'F' '1']
3   ['1' '1' '1' '1' '2' 'X']
4   ['1' 'F' '1' '0' '1' 'X']
5   ['1' '1' '1' '0' '1' '1']

Squares to unconver are: {(3, 5)}
```

Thanks to the new information provided (the 1-cells in (5,4) and/or in (5,5)), we can spot the final 5th mine, which must be in (4,5). Hence the last safe cell (3,5) can be probed, and the game is won.

Performance and limitations

- **The approach can scale up quite well, as long as satisfiability can find safe squares.** When the first click provides initial exploration (i.e. it's a 0-cell), a classic beginner mode minesweeper game (8x8 with 10 mines) is solved almost instantly, a classic intermediate game (16x16 grid with 40 mines) can be solved in about 1-3 seconds in most cases and an equivalent to the classic expert game (22x22 grid with 99 mines) is solved in about 10 seconds. (Super-human level in all modalities). All of this when the agent doesn't get stuck.
- Efficient cardinality constraints built in in pysat are only marginally better on average than the naïve cardinality constraints, if at all.
- **So far, with the framework described, the system can get stuck. This happens in situations when nothing can be said about relevant squares**: we don't know for sure if they're mines or not. In this case **the system can't progress further without modification. This problem arises especially early on** when the first-click isn't allowing an initial exploration (i.e. when it's not a 0-cell) **or in the late endgame**, when only few mines have to be spotted yet, but more than one configuration is possible. An example of this:

```
UNCOVERED BOARD CONFIG

    ['0' '1' '2' '3']

0   ['1' '3' 'M' '3']
1   ['M' '3' 'M' 'M']
2   ['2' '3' '2' '2']
3   ['M' '1' '0' '0']
```

```
    ['0' '1' '2' '3']

0   ['X' 'X' 'X' 'X']
1   ['X' '3' 'F' 'F']
2   ['X' '3' '2' '2']
3   ['X' '1' '0' '0']

STUCKED
```

```
for model in MS.enum_models():
    print(model)
```

```
[-1, -2, -3, -4, 5, -6, 7, 8, 9, -10, -11, -12, -13, -14, -15, -16]
[-1, -2, 3, -4, 5, -6, 7, 8, -9, -10, -11, -12, 13, -14, -15, -16]
[-1, -2, 3, 4, 5, -6, 7, 8, -9, -10, -11, -12, 13, -14, -15, -16]
[-1, -2, -3, 4, 5, -6, 7, 8, 9, -10, -11, -12, -13, -14, -15, -16]
[-1, 2, -3, 4, 5, -6, 7, 8, -9, -10, -11, -12, 13, -14, -15, -16]
[-1, 2, -3, -4, 5, -6, 7, 8, -9, -10, -11, -12, 13, -14, -15, -16]
[1, -2, -3, -4, 5, -6, 7, 8, -9, -10, -11, -12, 13, -14, -15, -16]
[1, -2, -3, 4, 5, -6, 7, 8, -9, -10, -11, -12, 13, -14, -15, -16]
```

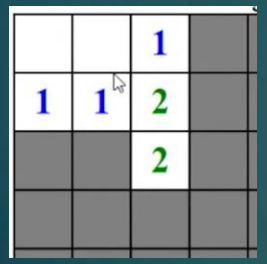
Agent gets stuck, no 100% safe logical action can be taken but there are 8 possible models. We can reason probabilistically: for example, out of the 8 models, only 2 has a positive literal 1 (i.e. there's a mine in position (0,0)). Hence it's probably a good choice to probe (as it is the case, this time, as shown in the uncovered board config).

- To try to solve this, for any unchecked cell (i,j) **we can count the number of models where it is safe ($-x_{i,j}$), and choose the cell with the highest number of safe models.** This approach tries to maximize the probability to win.
- Unfortunately, **#SAT is computationally intensive and gets prohibitive as the grid's size is increased.** This means that the approach to exactly compute it for every cell is feasible only with small boards or with very few mines left (in the endgame).

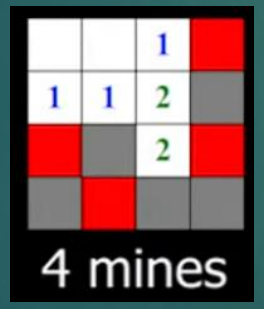
Model counting and reduced problems

- **We can address the computational problem by considering a reduced problem every time the agent gets stuck**, which has an approximate solution equal to the optimal one for the complete problem. **A separate solver is instantiated which uses only the variables relative to the reduced problem and the related cardinality constraints.**
- In fact, **we don't need to count the models for all the board, but only for the part of the board already explored and the adjacent cells. The amount of models for only the relevant cells will** (almost) **always be limited and It's possible to enumerate them exactly.**
- For every model of the smaller problem, **we can assume that all other mines which are not part of the models of the smaller problem are scattered uniformly in the rest of the (unconsidered) grid**. Hence, **for every model of the smaller problem, we can define the number of models for the complete game as the number of possible combinations in which the remaining mines are scattered across the rest of the grid** (I.e. |C| choose |RM|, where |C| is the number of unconsidered cells by the smaller problem and |RM| is the number of remaining mines to distribute, calculated as the difference between the number of mines in the original minesweeper game and the number of mines explained by the model of the smaller problem).

For example: say we have a 16x16 grid with 30 mines, but we get immediately stuck in the corner:



This configuration doesn't have an unique solution



A possible model is this one, mines in red

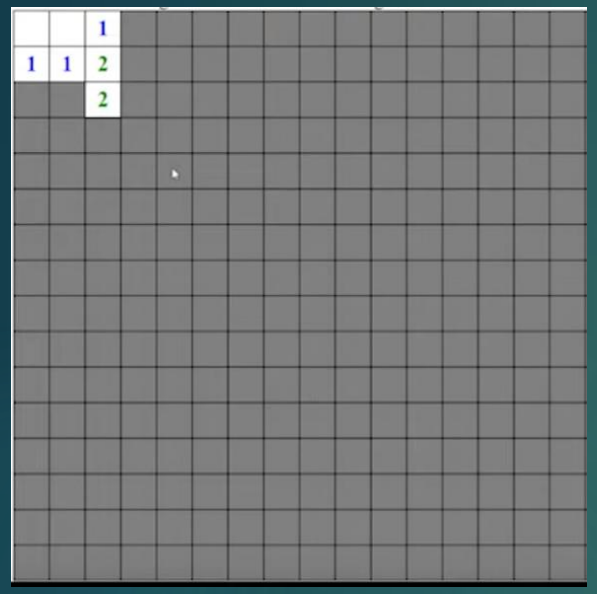
We know that for this model of the reduced problem, we have to distribute all the remaining 26 mines (30-4) in the unconsidered 241 cells of the rest of the grid (16x16-4x4-1, where 4x4-1 is the number of cells of the smaller problem considered; the cell in the lower left is not relevant). The number of possible combinations is: $\binom{241}{26} \approx 5.2 * 10^{34}$

Hence we can expect about 5.2*10^34 models of the complete game where that part of the grid has the configuration proposed.

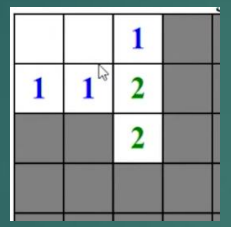
- We can then sum up the estimated number of models for the complete game for each possible configuration and choose to probe the cell which has the lowest number of models where it's a mine. (example next slide).

Probabilistic reasoning and model counting – a complete example
Reduction to simpler problems

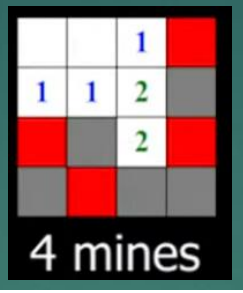
Say we have a 16x16 grid with 30 mines. We get stuck at the beginning in the corner with a configuration which has no unique solution.



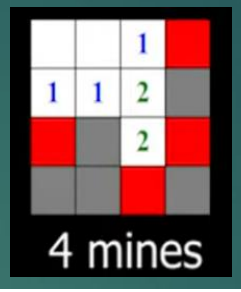
Complete problem



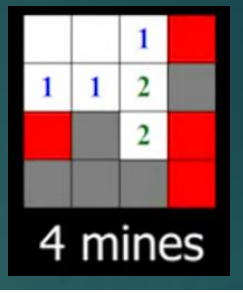
Reduced problem



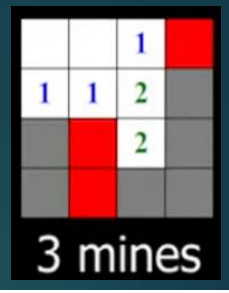
$\binom{241}{26} \approx 5.2 * 10^{34} models$



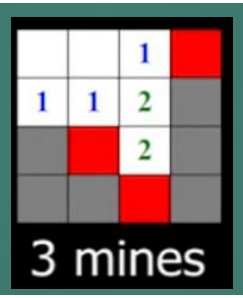
$\binom{241}{26} \approx 5.2 * 10^{34} models$



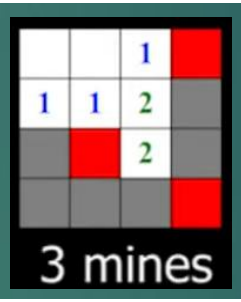
$\binom{241}{26} \approx 5.2 * 10^{34} models$



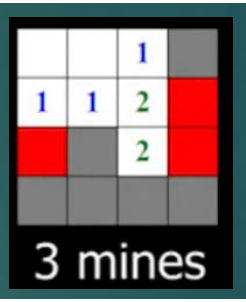
$\binom{241}{27} \approx 4.2 * 10^{35} models$



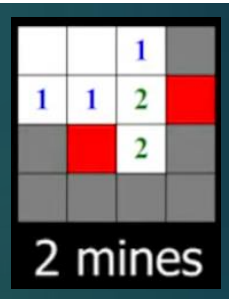
$\binom{241}{27} \approx 4.2 * 10^{35} models$



$\binom{241}{27} \approx 4.2 * 10^{35} models$



$\binom{241}{27} \approx 4.2 * 10^{35} models$



$\binom{241}{28} \approx 3.2 * 10^{36} models$

The 8 possible configurations that satisfy the reduced problem, along with the number of possible models of the complete problem that contains them. We can obtain these by simple model enumeration for reduced problems, they're almost always few (<40).

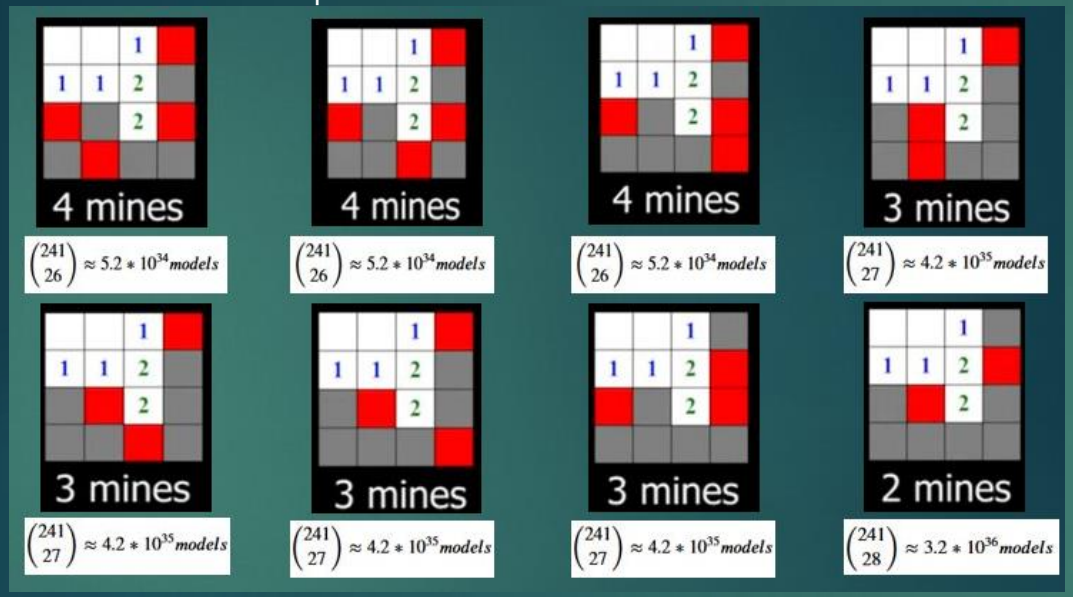
- Notice how simpler solutions (fewer mines used to explain the information seen) results in more models for the complete problem.
- **We can now compute the number of total models as a sum of all the combinations for all the models of the reduced problem:**

$$models \approx 5.2 * 10^{34} * 3 + 4.2 * 10^{35} * 4 + 3.2 * 10^{36} \approx 5.0 * 10^{36}$$

- **Probabilities to be a mine can now be computed for every relevant cell by simply summing up the number of models where a cell is a mine in the configurations above and divided by the estimated total number of models.**

Probabilistic reasoning and model counting – a complete example
Probabilities calculation

We can now calculate for all the 8 relevant cells, what's the approximate probability that they're mines. The indexes of the positions start from 0.



$$P((0,3) = mine) = (5.2 * 10^{34} * 3 + 4.2 * 10^{35} * 3)/(5.0 * 10^{36}) \approx 0.281$$
$$P((1,3) = mine) = (4.2 * 10^{35} + 3.2 * 10^{36})/(5.0 * 10^{36}) \approx 0.719$$
$$P((2,3) = mine) = (5.2 * 10^{34} * 3 + 4.2 * 10^{35})/(5.0 * 10^{36}) \approx 0.115$$
$$P((3,3) = mine) = (5.2 * 10^{34} + 4.2 * 10^{35})/(5.0 * 10^{36}) \approx 0.094$$
$$P((3,2) = mine) = (5.2 * 10^{34} + 4.2 * 10^{35})/(5.0 * 10^{36}) \approx 0.094$$
$$P((3,1) = mine) = (5.2 * 10^{34} + 4.2 * 10^{35})/(5.0 * 10^{36}) \approx 0.094$$
$$P((2,1) = mine) = (4.2 * 10^{35} * 3 + 3.2 * 10^{36})/(5.0 * 10^{36}) \approx 0.885$$
$$P((2,0) = mine) = (5.2 * 10^{34} * 3 + 4.2 * 10^{35})/(5.0 * 10^{36}) \approx 0.115$$

$$models \approx 5.2 * 10^{34} * 3 + 4.2 * 10^{35} * 4 + 3.2 * 10^{36} \approx 5.0 * 10^{36}$$

According to this approach, the best choice is to probe one of the cells between (3,3),(3,2),(3,1).

Two observations:

- **One last step forward which can be taken to improve this approach is to estimate in a similar way the probability to find a mine by probing a cell which is not adjacent to the explored area.** This is easy to do and quite useful in the early game in the cases when the first click is on a cell with a large number (3,4,5,6,7,8). It should be said, gathering information close to the explored area instead of probing elsewhere, should usually be preferred as it increases the possibility to derive logical consequences. So probing new areas should be at least a bit penalized in the calculations. As the area explored of the grid expands, actions taken casually drop to 0.
- **The approach can still get slow if there are a lot of different possible configurations.** This is unlikely and tend to happen only if we have different unconnected regions of the grid explored (which should be discouraged by the penalization mentioned above) or large grids (expert mode can get slow). A solution (not explored in this project) could be to consider separately unconnected explored regions.

An example with probabilistic reasoning

To show an example which would get stuck with only SAT we consider an 8x8 grid with 10 mines (classic beginner mode). Most moves are logical consequences, but when the agent gets stuck, probabilistic reasoning allows to probe the least dangerous cell.

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
1   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
2   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
3   ['X' 'X' 'X' '3' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
6   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
7   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
```

```
Probabilities:
{'random': 0.15727,
 (2, 2): 0.375,
 (2, 3): 0.375,
 (2, 4): 0.375,
 (3, 2): 0.375,
 (3, 4): 0.375,
 (4, 2): 0.375,
 (4, 3): 0.375,
 (4, 4): 0.375}
```

The first-click is no longer forced to be a 0, in this case it's a 3. Choosing cells close to the 3 is way more dangerous than choosing randomly.

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
1   ['X' 'X' 'X' 'X' '1' 'X' 'X' 'X']
2   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
3   ['X' 'X' 'X' '3' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
6   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
7   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
```

```
Probabilities:
{'random': 0.1675,
 (0, 3): 0.06667,
 (0, 4): 0.06667,
 (0, 5): 0.06667,
 (1, 3): 0.06667,
 (1, 5): 0.06667,
 (2, 2): 0.4,
 (2, 3): 0.3,
 (2, 4): 0.3,
 (2, 5): 0.06667,
 (3, 2): 0.4,
 (3, 4): 0.4,
 (4, 2): 0.4,
 (4, 3): 0.4,
 (4, 4): 0.4}

Squares to unconver are: {(0, 4)}
```

The cell chosen randomly is a 1-cell. Still no 100% safe action available: the agent assess correctly that the safest cells are those near the 1 and not adjacent to the 3.

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
1   ['X' 'X' 'X' 'X' '1' 'X' 'X' 'X']
2   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
3   ['X' 'X' 'X' '3' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
6   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
7   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']

Squares to unconver are: {(2, 5), (2, 3), (2, 4)}
```

The new information (1-cell in (0,4)) combined with the old, is enough to conclude by SAT that logically the cells (2,5),(2,3) and (2,4) are safe, and we can inspect them.

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['X' 'X' 'X' 'X' '1' 'X' 'X' 'X']
1   ['X' 'X' 'X' 'X' '1' 'X' 'X' 'X']
2   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
3   ['X' 'X' 'X' '3' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
6   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
7   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
```

```
Probabilities:
{'random': 0.12006,
 (0, 3): 0.38875,
 (0, 5): 0.38875,
 (1, 2): 0.0755,
 (1, 3): 0.01185,
 (1, 5): 0.21065,
 (1, 6): 0.67062,
 (2, 2): 0.5877,
 (2, 6): 0.67062,
 (3, 2): 0.5877,
 (3, 4): 0.73726,
 (3, 5): 0.04025,
 (3, 6): 0.67062,
 (4, 2): 0.36245,
 (4, 3): 0.36245,
 (4, 4): 0.36245}

Squares to unconver are: {(1, 3)}
```

Unfortunately new information doesn't entail new logical safe actions. The agent must reason probabilistically again, concluding that (1,3) is by far the safest cell. Notice how it's possible to reason how restrictive (hence few) the models would have to be for a mine to be there.

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['X' 'X' 'X' 'X' '1' 'X' 'X' 'X']
1   ['X' 'X' 'X' 'X' '1' 'X' 'X' 'X']
2   ['X' 'X' 'X' '2' '1' '3' 'X' 'X']
3   ['X' 'X' 'X' '3' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
6   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
7   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
```

```
Probabilities:
{'random': 0.11526,
 (0, 2): 0.20018,
 (0, 3): 0.79517,
 (0, 5): 0.07618,
 (1, 2): 0.14283,
 (1, 5): 0.12865,
 (1, 6): 0.66667,
 (2, 2): 0.86182,
 (2, 6): 0.66667,
 (3, 2): 0.20018,
 (3, 4): 0.79517,
 (3, 5): 0.07618,
 (3, 6): 0.66667,
 (4, 2): 0.38094,
 (4, 3): 0.38094,
 (4, 4): 0.38094}

Squares to unconver are: {(0, 5)}
```

Once again, no 100% safe action, the system must reason probabilistically. The square (0,5) is chosen. Once again, after seeing the result it's possible to reason about it and see that indeed few possible configuration imply that a mine is there.

```
    ['0' '1' '2' '3' '4' '5' '6' '7']

0   ['X' 'X' 'X' 'X' '1' '0' 'X' 'X']
1   ['X' 'X' 'X' '2' '1' 'X' 'X' 'X']
2   ['X' 'X' 'X' '2' '1' '3' 'X' 'X']
3   ['X' 'X' 'X' '3' 'X' 'X' 'X' 'X']
4   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
5   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
6   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']
7   ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X']

Squares to unconver are: {(1, 5), (1, 6), (0, 6)}
```

Finally, a 0-cell is spotted and safe actions are available by SAT. From this point on the agent was able to win the game only using SAT. (after 12 more reasoning iterations)

Grid size	N° mines	Mines density	Probabilistic reasoning enabled	Imposed first-click 0-cell	Win %	Mean time per game	Median time per game
4x4	5	0.3125	No	No	8%	0.002s	0.001s
4x4	5	0.3125	No	Yes	55%	0.007s	0.006s
4x4	**5**	**0.3125**	**Yes**	**No**	**52%**	**0.148s**	**0.014s**
4x4	5	0.3125	Yes	Yes	77%	0.014s	0.008s
6x6	6	0.1667	No	No	16%	0.006s	0.002s
6x6	6	0.1667	No	Yes	73%	0.032s	0.024s
6x6	**6**	**0.1667**	**Yes**	**No**	**72%**	**0.032s**	**0.024s**
6x6	6	0.1667	Yes	Yes	91%	0.027s	0.021s
8x8 (beginner)	10	0.1563	No	No	22%	0.020s	0.003s
8x8 (beginner)	10	0.1563	No	Yes	69%	0.137s	0.060s
8x8 (beginner)	**10**	**0.1563**	**Yes**	**No**	**82%**	**0.356s**	**0.078s**
8x8 (beginner)	10	0.1563	Yes	Yes	85%	0.151s	0.066s
16x16 (intermediate)	40	0.1563	No	No	19%	0.327s	0.008s
16x16 (intermediate)	40	0.1563	No	Yes	68%	1.056s	1.127s
16x16 (intermediate)	**40**	**0.1563**	**Yes**	**No**	**64%**	**0.950s**	**1.118s**
16x16 (intermediate)	40	0.1563	Yes	Yes	89%	1.468s	1.197s
22x22 (expert-like)	99	0.2045	No	No	7%	1.8s	0.01s
22x22 (expert-like)	99	0.2045	No	Yes	18%	6s	6s
22x22 (expert-like)	**99**	**0.2045**	**Yes**	**No**	**31%**	**6s**	**6s**
22x22 (expert-like)	99	0.2045	Yes	Yes	52%	38s **	8s

** This skewed mean is due to an outlier game, where most likely too many configurations for the reduced problems induced VERY slow performance (20 mins to solve).

Some performance metrics

100 Games were run per category. In bold the cases where we exploit both SAT and #SAT and there's no initial help in exploration.

Observations:

- **SAT-only based systems obtain good win % only when starting with some initial exploration (first-click is a 0-cell)**, otherwise they mostly get stuck.
- **Including #SAT of reduced problem and probabilistic reasoning ALWAYS improve substantially the win %** and usually don't slow down too much the process (with the exception of very large grids, expert-like)
- Win % is in general, regardless of the method used, related to both dimension of the grid and mines density in the grid (a 4x4 dense game is harder than a 8x8 game with half the density).
- **Win % obtained are close to those claimed by expert human players.**
- **Time to solve the games is always super-human.**