# Minesweeper solving goal-based intelligent agent using BFS and DFS

Assignment 2 – Artificial Intelligence, KNU Spring semester '20

Max Wiersma

2020/04/29

## 1. Background

Minesweeper is a classic game where the player is to find all squares on a grid that are not mines. It is played on a grid of squares which all start our being hidden. Clicking on a square either reveals the number of mines adjacent to it, or if the square is a mine, the game ends in a loss. If none of the adjacent squares are mines, those squares are recursively revealed too. The aim is to use the information about the number of mines next to the squares to logically deduce where the mines are. The game ends in a win when all non-mine squares have been revealed.

Our implementation* uses a square n x n grid and m mines, both given as parameters at the start of a game. In most modern versions, including ours, the first click never reveals a mine, in order to decrease the importance of luck.

*I would like to emphasize that all (100%) of the submitted code was written entirely by myself.

## 2. System components

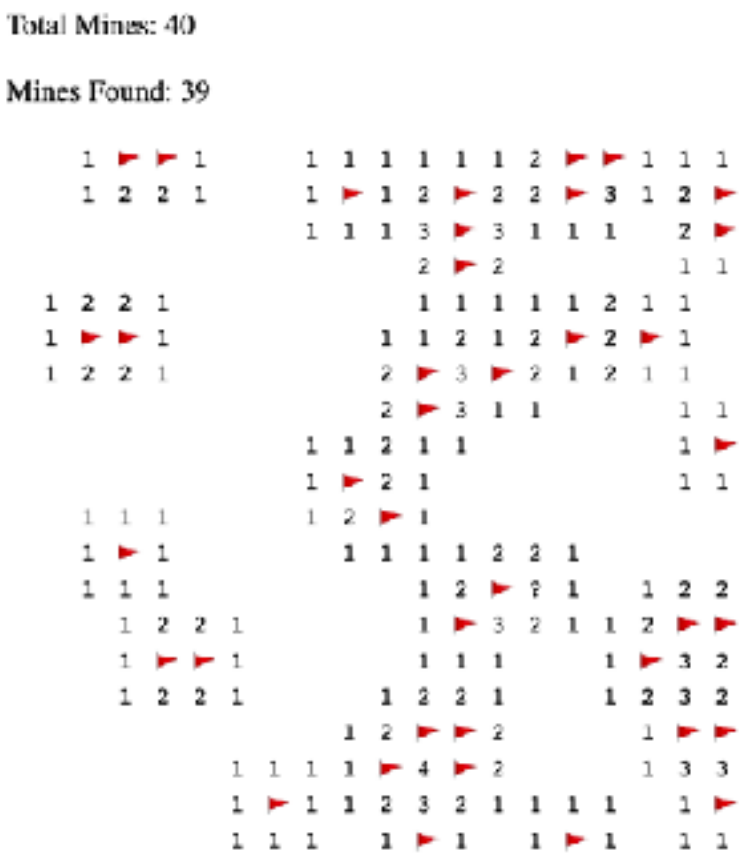
The system consists out of three parts: an implementation of the Minesweeper game, an intelligent agent and an optional web front-end to visualize the agent playing the game.

For the most part, the implementation of the game is straightforward. A 2-dimensional grid of nodes is initialized and among those m nodes are randomly chosen and set to be mines. The number of adjacent mines of the remaining nodes can then be easily calculated. As mentioned, if the player clicks on a square with no adjacent mines (hereafter "empty square"), its adjacent squares need to be revealed automatically too. If this reveals other empty squares, their adjacent squares need to be revealed too. This is implemented recursively leading to a Depth-First Search. As soon as an empty square is found, its adjacent squares are checked.

Pseudocode for DFS function to open empty squares

```
def dfs(empty_node):
for n in empty_node.adjacent_nodes() {
        if n.is_empty_node {
                n.is_revealed = True
                dfs(n)
        }
        else {
                n.is_revealed = True
        }
}
```

The web front-end visualizes the intelligent agent playing the game. Every time the agent clicks a square, the new state of the board is asynchronously sent to the client, which then immediately visualizes it.




Web view screenshot

The last component of the system is the intelligent agent. It is described in more detail in the next paragraphs.

# 3. Intelligent Agent

The intelligent agent used is a goal-based agent whose goal is to end up in the state where all non-mine squares have been revealed, winning the game. Just like a normal player would, it uses logical deduction to reason about which square to click, and interacts with the game by virtually clicking on a given square.

Its environment consists of the grid of squares and the values they represent. It is partially observable, again just like a real player - before clicking a square, it does not know whether it will reveal a mine or how many adjacent mines it will have. Since the location of the mines is selected randomly at the start of the game, the environment is stochastic. However, once established it does not dynamically change, hence being static. The agent is the only actor in the environment and time only progresses when it takes an action (of clicking a square), making it a single-agent, sequential environment. In summary, the environment is:

- Partially Observable
- Stochastic
- Sequential
- Static
- Discrete
- Single-agent

There are multiple ways to evaluate the performance of the agent. They can mainly be divided into time-based and success-based metrics. The former category can contain metrics such as the time in seconds until the game is finished, the number of node evaluations made, or the amount of CPU time used. The latter could be the average number or percentage of mines revealed - the stochastic element of Minesweeper means it is not possible to create an algorithm that always wins - or the percentage of games won. In the statistical analysis that was performed, “average time until the game is finished” and “percentage of games won” were the two measures used.

The actuators and sensors of the agent would theoretically be respectively a computer input device to select the squares and a camera to read the computer display and translate it into the state of the game. In our case these are simply virtually simulated by feeding them directly as data structures. This makes no difference in the operation of the agent.

| | |
|---|---|
| Performance | Success-based, Time-based |
| Environment | Grid of squares |
| Actuators | Clicking (Input device) |
| Sensors | Reading the board sate (Camera) |

Intelligent Agent PEAS

## 4. Agent Algorithms

Every turn, the agent needs to decide which square to click on. Since the goal is to end up in a state where all non-mine squares have been clicked, it makes sure to click on a square that is not a mine. Thus, an algorithm that decides whether a square is a mine or not needs to be implemented.

First (after the first square has been clicked on), the algorithm creates a list of nodes that have adjacent mines. Then, for each node, starting at the beginning of the list, it checks whether it can deduce the locations of the adjacent mine(s). Since the number of adjacent mines is given by the game, if the number of adjacent unrevealed nodes equals the number of adjacent mines, all of them must be mines. These nodes are then flagged as “definitely mine”, represented by a flag symbol in the web client. If then for a different node the number of adjacent flags, i.e. confirmed mines, is equal to the number of adjacent mines, this means that all of its adjacent mines have already been found and any other adjacent unrevealed nodes must not be mines and can thus safely be clicked on.

```
def algorithm():
queue = get_revealed_nodes()
while queue {
        n = queue.pop()
        evaluate_adjacent_unrevealed_nodes(n)
        safe_nodes = get_safe_nodes()
        if safe_nodes {
          click(safe_nodes[0])
          check_game_won()
          queue.insert(get_newly_revealed_nodes))
        }
}
```

Pseudocode representation of main algorithm

The algorithm finishes when either there are no more new revealed nodes to evaluate or when the game has been won by revealing all non-mine nodes.

The manner in which the “queue.insert” function is implemented decides whether the algorithm does a Depth-First Search or a Breadth-First Search. If the newly revealed nodes are added to the front of the queue it becomes a LIFO queue (really a stack) meaning the next iteration of the algorithm will evaluate such a newly revealed node, creating a Depth-First search. If they are added to the end of the queue, it is a standard FIFO queue, with nodes that had already been revealed at an earlier point being evaluated first, creating a Breadth-First search.

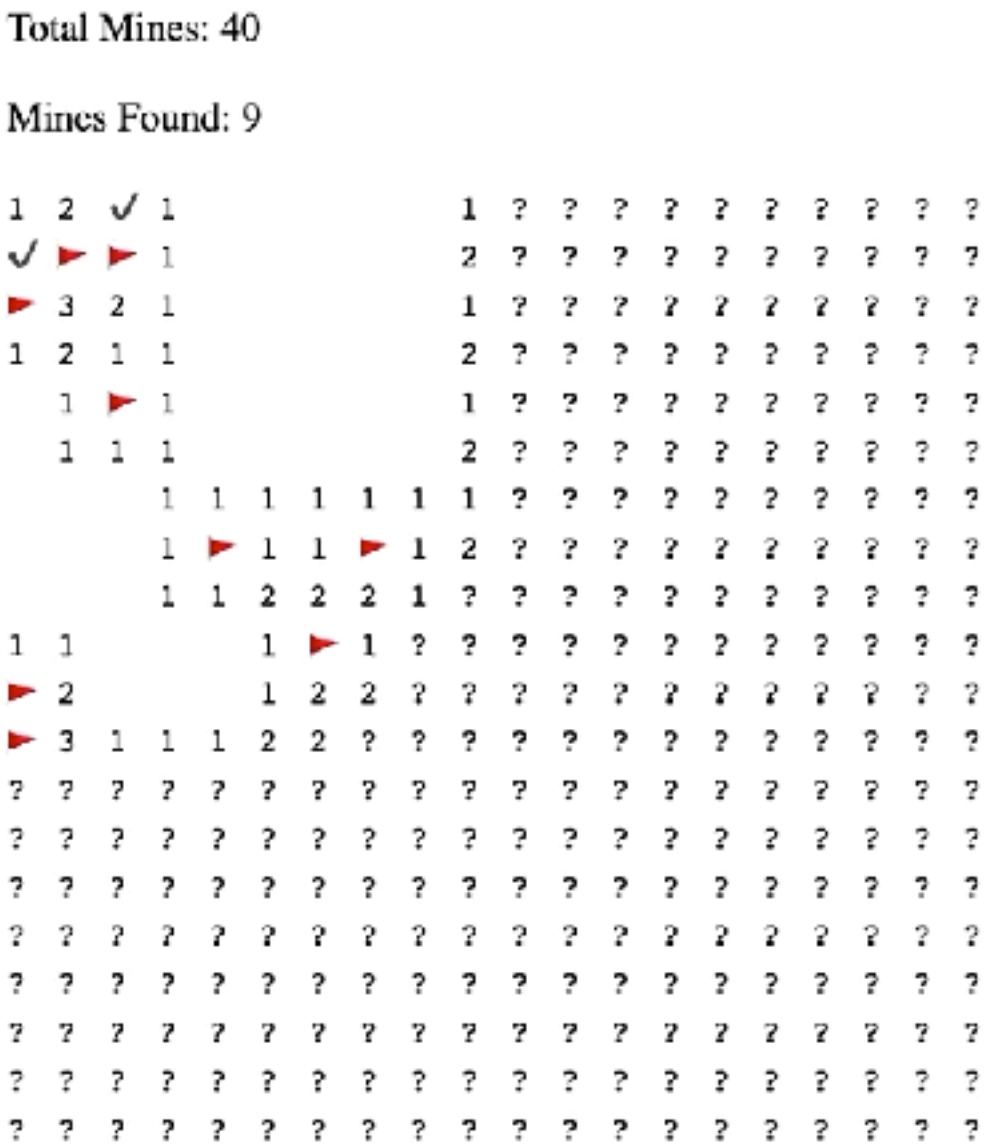
| Algorithm | Action |
|---|---|
| BFS | Adds newly revealed nodes to the back of the queue |
| DFS | Adds newly revealed nodes to the front of the queue (stack) |

## 5. Run environment

The Minesweeper implementation and the intelligent agent were fully written in Python 3.7. Only core libraries that are part of Python were used. However, due to the integration with the web client, it may be necessary to install the libraries ‘Flask’ and ‘Flask-SocketIO’. The backend of the web client is built on these two libraries, while the frontend uses basic HTML, CSS and Javascript, requiring no additional frameworks.

The statistics were gathered using the core ‘timeit’ library and processed using numpy, pandas and seaborn.

## 6. Video of intelligent agent (BFS Version) in action as visualized by web client




## 7. Statistical Analysis

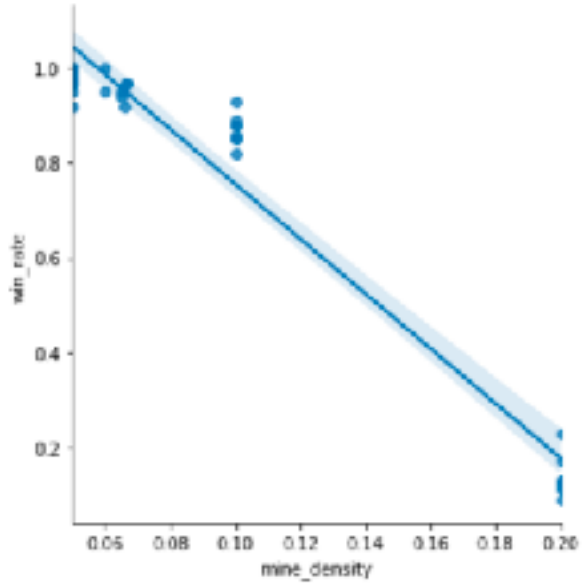
Using the Python timeit library, the intelligent agent was run under a variety of settings. Under each combination of settings, 100 runs were performed. The performance metrics looked at were win rate (% of games successfully completed) and the time in seconds that it took until the end of a game.

| | |
|---|---|
| Grid size (1-side) | 10, 20, 30, 40 |
| Number of mines | Grid size / [20, 15, 10, 5] |
| Evaluation algorithm | BFS, DFS |

## 8. Performance Evaluation

By far the biggest factor deciding the win rate of the algorithm was the number of mines. This is here expressed as “mine density”, being simply the percentage of squares that were mines. As can be seen, regardless of the other settings, there is a very clear linear downwards trend, with more mines meaning a far lower win rate. At a mine density of 0.06,

the algorithm has an almost 100% win rate, whereas at a mine density of 0.2 this goes down to 5%~25% depending on the other settings.




The variables affecting the execution time were less clear - here it was more a combination of both grid size and mine density that had a multiplicative effect.