Search Methods in AI

Minesweeper Part - 1 + 2

Submitted by
Abhinav Lodha (U20220003)
Subham Jalan (U20220082)

September 07, 2024

Contents

1. Problem Statement

Minesweeper is a classic logic puzzle game played on personal computers. The game consists of a grid of clickable tiles, with hidden “mines” scattered throughout the board. The objective is to clear the board without detonating any mines, with help from clues about the number of neighboring mines in each field. For this assignment, we were inspired from this beloved game and tried to create a search space problem from this game by adding some constraints.

Objective:

Starting from a random non-mine cell, the task is to path that visits every non-mine cell exactly once, avoiding all mines.

Key Assumptions:

1. The starting cell is guaranteed to be a non-mine cell.
2. All non-mine cells are reachable from the starting cell.
3. Additional constraint which only allows movement to adjacent cells from visited mines (the ones with number), including diagonally adjacent ones, with a distance of 1 per move.
4. If mines are placed such that they partition the grid, a random move is taken. (There is always some state to go to until you win.)

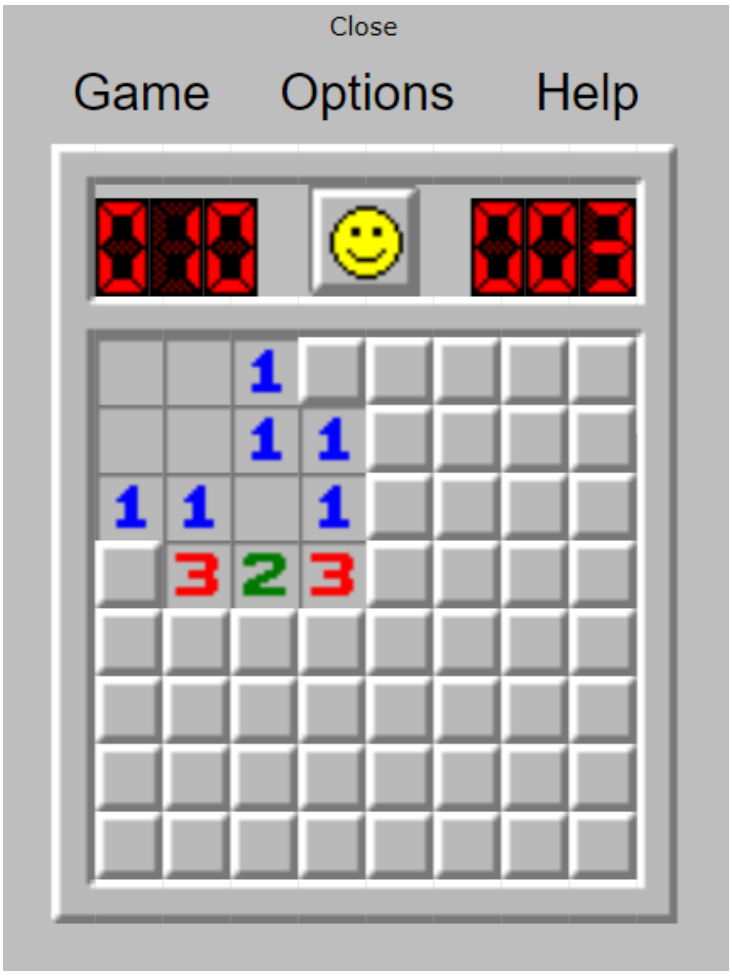

Figure 1: The original Minesweeper

2. Domain Representation

The standard algorithm for solving Minesweeper uses a heuristic approach to predict the location of mines. This algorithm relies on the numbers displayed on the visited cells to infer the positions of hidden mines. By analyzing these numbers, the algorithm can determine which neighboring cells are likely to contain mines and which are safe to click. The key is to carefully consider the relationship between the numbers and the surrounding unvisited cells to make accurate predictions and avoid triggering a mine. This heuristic will be used in our informed search algorithms. However, for BFS (Breadth-First Search) and DFS (Depth-First Search), we will not apply this heuristic, classifying both algorithms as uninformed searches.

2.1. Initialization Constants

- **BOARD_SIZE**: Number of rows or columns in the board. Number of cells will be BOARD_SIZE * BOARD_SIZE.
- **NO_OF_MINES**: Number of mines the board contains. Always less than the number of cells in the board.

2.2. Conventions

Conventions for the state of the board-

Description	Symbol	COLOR
Unvisited Safe Cell	U	Dark Gray
digit 0-8 (Visited)	$0 - 8$	Light Gray
Mine	M	Red
Marked Mine	T	Yellow
Available Moves	$*$	

Table 1: Conventions

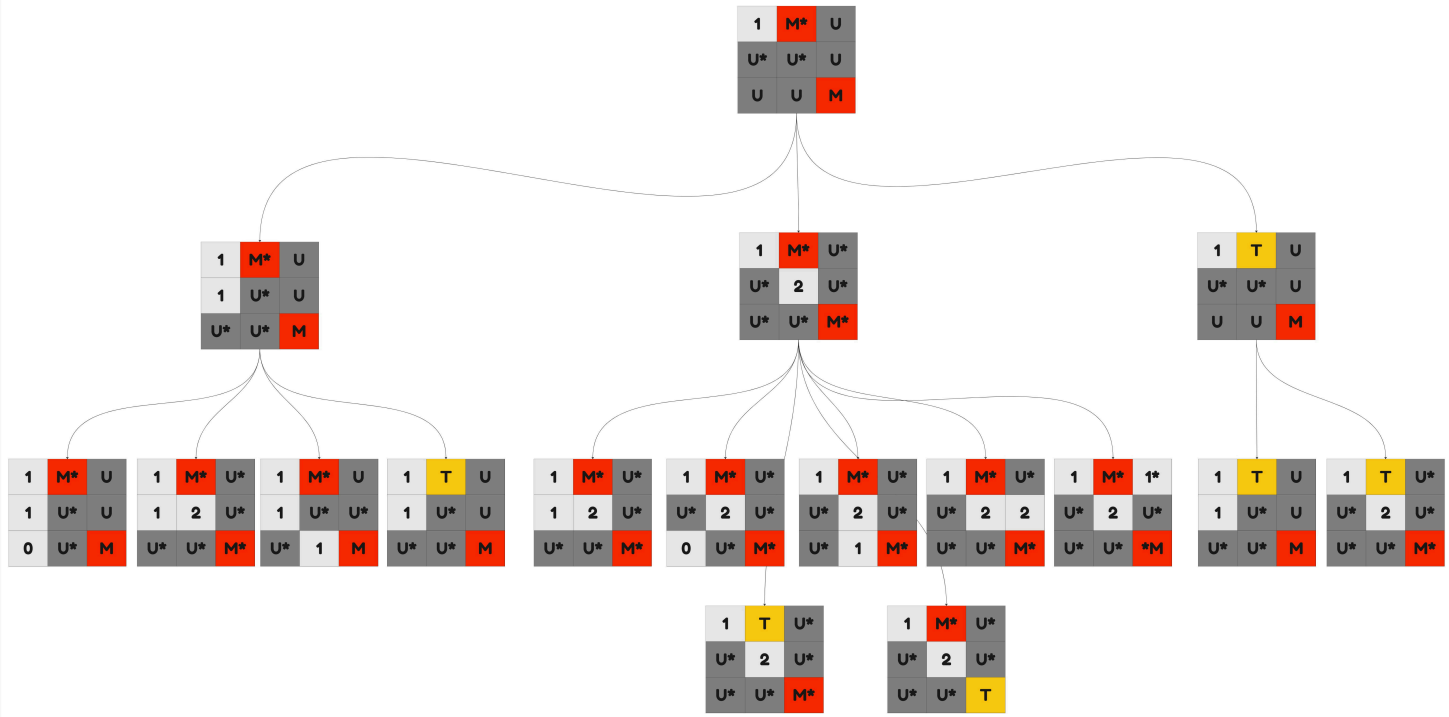

Figure 2: Our 3x3 Minesweeper State Flow Chart

3. MoveGen

3.1. Definition

Our MoveGen function is straightforward. It takes the current state of the board, represented as a matrix where each value symbolizes a particular cell on the board. The function outputs a set of potential cells that can be visited in the next turn (cells marked with $*$ in *Figure 2*). This set may include cells that contain mines! If we visit a mine, it becomes marked (cells marked with T in *Figure 2*). A potential cell is an unvisited cell (cells marked with U or M in *Figure 2*).

A VisitedCells set is maintained, and the MoveGen function iterates through that set, checking all valid neighbors (unvisited or mine). If valid, those cells are added to the output set.

3.2. Pseudo Code

Here is the pseudo code of MoveGen function for our modified minesweeper.

```
def MoveGen(board:List[List[Cell]]) -> set():
  output = set()
  for (i,j) in VisitedCells: # can be infered or stored from current board state.
    for x in [-1,0,1]:
      for y in [-1,0,1]:
        if InBoard((i+x,j+y)):
          if (board[i+x][j+y] == "Unvisited" or board[i+x][j+y] == "Mine"):
            output.add((i+x,j+y))
  return output
```

Listing 1: Pseudo code of MoveGen function

4. GoalTest

4.1. Definition

Our GoalTest function takes the current state of the board, represented as a matrix where each value symbolizes a particular cell on the board. It outputs a boolean value (*True* or *False*) to indicate whether the current state is the goal state. True signifies that the goal state has been reached.

The goal state is considered reached if all the mine-free cells have been visited. To check if the board has reached the goal state, we count the visited cells (excluding mines) and compare this count to the total number of safe cells on the board (mine-free cells). If they are equal, we return True.

4.2. Pseudocode

Here is the pseudo code of GoalTest function for our modified minesweeper.

```
def GoalTest(board:List[List[Cell]]) -> bool:
  # VisitedCells can be infered or stored from current board state.
  if len(VisitedCells) == [BOARD_SIZE*BOARD_SIZE - NO_OF_MINES]:
    return True
  else:
    return False
```

Listing 2: Pseudo code of GoalTest function

5. Breadth First Search

We have implemented Breadth First Search in our algorithm.

5.1. Properties of Search

1. Search begins at (0, 0).
2. A cell may be visited only from its neighboring cells. In case no moves are available, a random move is taken.
3. The search is non-deterministic.
4. Queue is maintained to search in a breadth-first manner.
5. A cell is added to the queue only if it is a safe cell.
6. It is guaranteed that the search will visit all the safe cells. However, the number of mines hit may vary across different instances of the search.
7. A cell that is a mine may be visited. However, its neighboring cells are not added to the queue and a mine counter is increased to mark naiveness of the algorithm.

5.2. Pseudo Code

Here is the pseudo code of Breadth First Search function for our modified minesweeper.

```
def BreadthFirstSearch(board:List[List[Cell]],startMove:tuple):
  queue = [startMove]
  visited = [[False]*BoardSize for i in range(BoardSize)]
  visited[startMove] = True
  while queue:
    currentMove = queue.deque()
    if safe and unvisited:
      visit(currentMove)
      correctVisits +=1
      potentialMoves = MoveGen()
      queue.enqueue(potentialMoves)
      for move in potentialMoves:
        if move is unvisited:
          visit(Move)
    if mine:
      markMine(currentMove)

  return
```

Listing 3: Pseudo code of Breadth First Search function

6. Depth First Search

We have implemented Depth First Search in our algorithm.

6.1. Properties of Search

1. Search begins at (0, 0).
2. A cell may be visited only from its neighboring cells. In case no moves are available, a random move is taken.
3. The search is non-deterministic.
4. Stack is maintained to search in a depth-first manner.
5. A cell is added to the stack only if it is a safe cell.
6. It is guaranteed that the search will visit all the safe cells. However, the number of mines hit may vary across different instances of the search.
7. A cell that is a mine may be visited. However, its neighboring cells are not added to the stack and a mine counter is increased to mark naiveness of the algorithm.

6.2. Pseudo Code

Here is the pseudo code of Depth First Search function for our modified minesweeper.

```
def DepthFirstSearch(board:List[List[Cell]],startMove:tuple):
  stack = [startMove]
  visited = [[False]*BoardSize for i in range(BoardSize)]
  visited[startMove] = True
  while stack:
    currentMove = stack.pop()
    if safe and unvisited:
      visit(currentMove)
      correctVisits +=1
      potentialMoves = MoveGen()
      stack.push(potentialMoves)
      for move in potentialMoves:
        if move is unvisited:
          visit(Move)
    if mine:
      markMine(currentMove)

  return
```

Listing 4: Pseudo code of Depth First Search function

7. Best First Search

We have implemented Best First Search in our algorithm.

7.1. Heuristic

At every state, two possible **types** of moves are possible.

1. **Smart Move** - In case a move is possible which is guaranteed to be safe, we visit that cell.

   To compute smart moves, we are using principles of set theory. Every time we visit a cell, we store the number of neighboring mines and the neighboring cells as a **Evidence**. An evidence stores information gathered from visiting a single cell. Every time a cell is visited, we add a new evidence to our set of evidences, and do a scan to find if new inferences can made. A new inference can mark a cell as a certain safe cell or a certain mine.

   Example - If we visit a certain cell - (1, 0) and get the mine count as 3, we store the following evidences -

   `e1 = Evidence(cells=[(0, 0), (2, 0), (1, 1), (0, 1), (2, 1)], mineCount = 2)`

   `e2 = Evidence(cells=[(1, 0)], mineCount = 0)`

   Next, if we visit (0, 0) and get the following evidence -

   `e3 = Evidence(cells=[(0, 1), (1, 0), (1, 1)], mineCount = 2)`

   Then we can infer using `e2` and `e3` that -

   `e4 = Evidence(cells = [(0, 1), (1, 1)], mineCount = 2)`

   Using `e4`, we can conclude that the cells in `e4` are indeed mines (since number of cells = mineCount) and we can use this information to infer that the rest of the cells in `e1 - e4` (set difference) are all safes.

   This gives us the following conditions for an inference -
   1. Number of cells in an evidence = mineCount
   2. Cells in an evidence is a subset of cells of another evidence.
   3. Number of cells in an evidence is positive but mineCount is 0.

   These inferences are done using an $O(n^2)$ operation of scanning all evidences against each other, every time a cell is visited.

2. **Naive Move** - In case a Smart Move is not possible, a naive random move is taken (the a is chosen from all the unvisited cells on the board).

7.2. Properties of Search

1. Search begins at (0, 0).
2. A cell may be visited either smartly (using evidences) or randomly (when no smart move is available).
3. The search is non-deterministic.
4. A cell that is a mine may be visited (this only happens when a naive move is taken). A mine counter is increased to mark a failure in the search algorithm.

7.3. Pseudocode

```
class Evidence:
    def __init__(self, cells: set, mines: int):
        self.cells = cells
        self.mines = mines
    def __len__(self):
        return len(self.cells)

def smartMove(board):
  evidences = set()
  for i in range():
    for j in range():
      minesCount = 0
      evidence = set()
      if isVisited(board[i][j]):
        for neighbour in neighbours:
          if isValid(neighbour):
            evidence.add(neighbour)
          elif neighbour == Mine:
            minesCount +=1
      evidences.add(Evidence(evidence,neighbour.value - minesCount))
  InferenceMade = True
  while InferenceMade:
    InferenceMade = False
    for evidence1 in evidences:
      for evidence2 in evidences:
        if evidence1 != evidence2 and evidence1.isSubset(evidence2):
            evidence2.cells = evidence2.cells - evidence1.cells
            evidence2.mines = evidence2.mines - evidence1.mines
            InferenceMade = True
    for evidence in evidences:
      if evidence.mines == 0:
          safeMoves.add(evidence.cells)
          evidences.remove(evidence)
      elif evidence.mines == len(evidence.cells):
          minesDetected.add(evidence.cells)
  if safeMoves:
    return safeMoves.pop()
  return None
```

Listing 5: Pseudo code of helper classes and functions

Here is the pseudo code of Best First Search function for our modified minesweeper.

```
def BestFirstSearch(board:List[List[Cell]],startMove:tuple):
  while correctVisits < totalVisits:
      if smartMoveAvailable():
        move = smartMove()
      else:
        # no inference can be made
        move = isValid(startMove) or naiveMove()
      if isValid(move): # safe and unvisisted
        queue = [move]
        move = queue.deque()
        visit(move)
        correctVisits+=1
      if move.adjMineCount == 0:
        # explore all neighbours if no mines around
        for neighbour in move.neighbours:
          if isValid(neighbour):
            visit(neighbour)
            correctVisits += 1
            if move.adjMineCount == 0:
              queue.enqueue(neighbour)
      if move is mine:
        markMine(move)
      else:
        # invalid move
        return

  return
```

Listing 6: Pseudo code of Best First Search function

8. Comparison

All the three algorithms are run 100 times on a 6 x 6 board with 5 mines. In every run, the location of mines may be different. The starting position is fixed to (0, 0) for all the three algorithms. Following are the results of the algorithms (averaged over 100 runs).

Search	Time per Search (ms)	Number of Cells Visited
Breadth First Search	0.733	35.54
Depth First Search	0.731	35.77
Best First Search	1.47	29.38

Table 2: Grid Size: 6 x 6, Number of Mines: 8

Search	Time per Search (ms)	Number of Cells Visited
Breadth First Search	50.322	399.91
Depth First Search	50.722	399.89
Best First Search	68.370	360.17

Table 3: Grid Size: 20 x 20, Number of Mines: 40

8.1. Inferences

1. The lower value of Number of Cells Visited for Best-First Search shows that taking optimal moves indeed results in the search picking cells that are less-likely to be mines and faster winning of game. However, doing a naive BFS/DFS results in hitting mines (because the information of mines in neighboring cells is not being used) and an overall higher number of cells being visited before all safe cells are visited.
2. BFS and DFS take almost equal amounts of time for execution. However, the time taken by Best First Search is higher because of scanning of evidences upon every move.
3. The average number of cells visited is close to $x = (\text{grid size})^2 - \text{number of mines}$ for Best First Search. However, it is slightly higher than x because there are cases when Naive Moves are taken, resulting in visiting of mines.
4. The average number of cells visited is very close to $x = (\text{grid size})^2$ for BFS and DFS because these visit almost every cell (including mines) before all safe cells are visited. However, the true value is slightly lower than x because there are cases when the game may win early without visiting all cells (because of the placement of mines in such a manner).

9. Fun

We made custom UI to visualize the game!

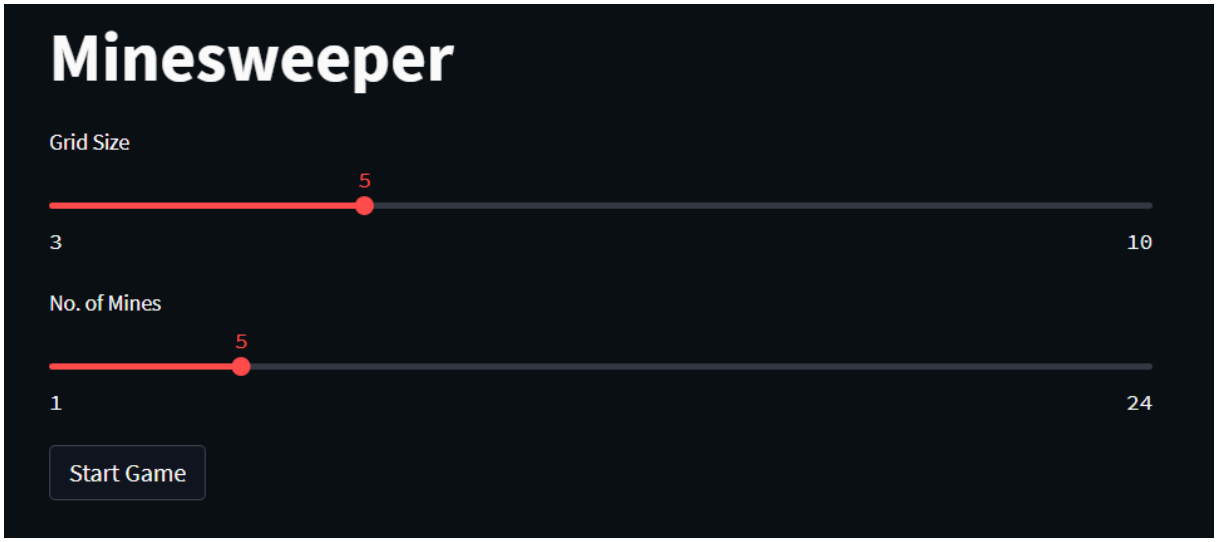

Figure 3: Configuration of the Game

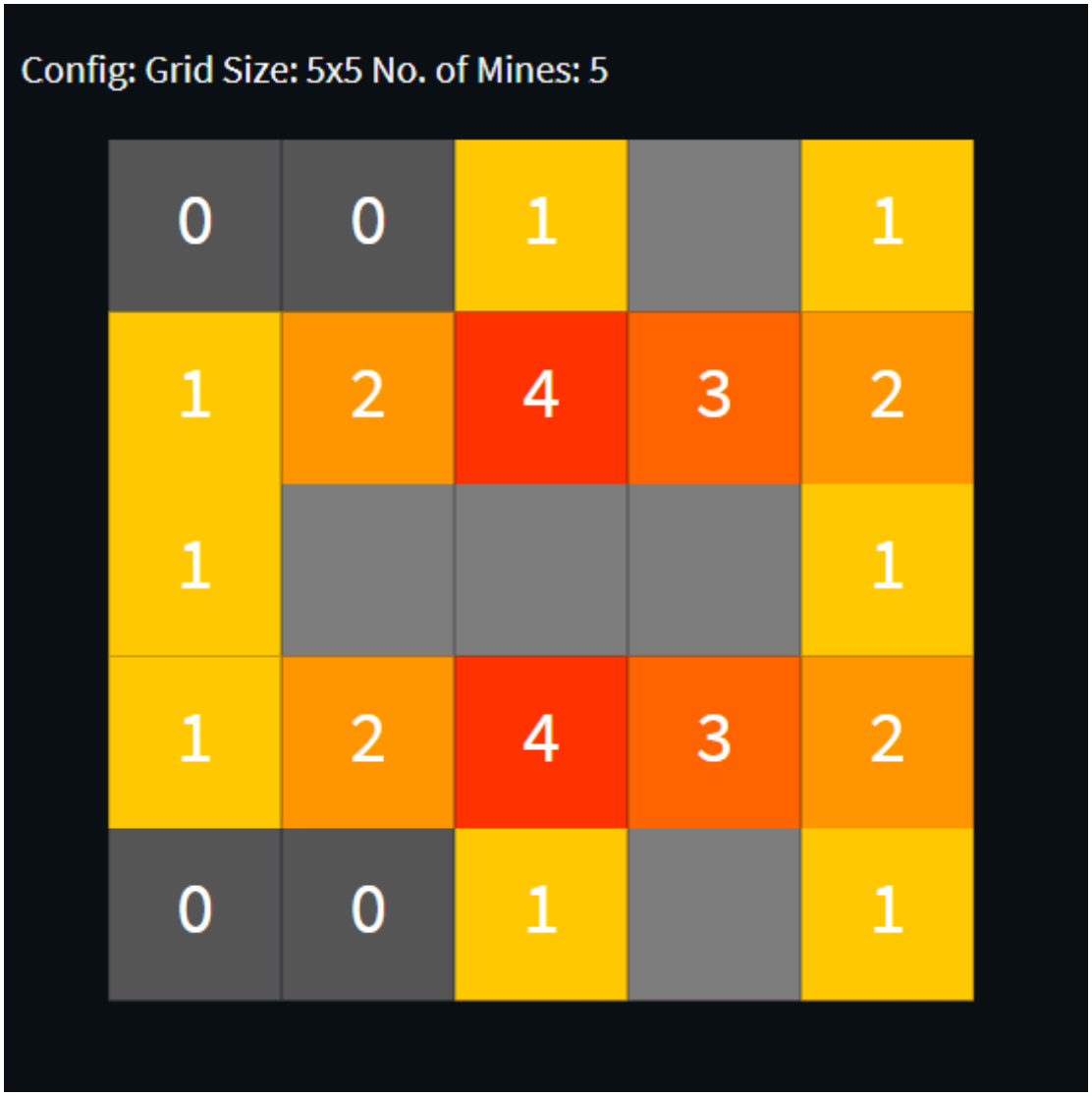

Figure 4: 5x5 Minesweeper with 5 mines

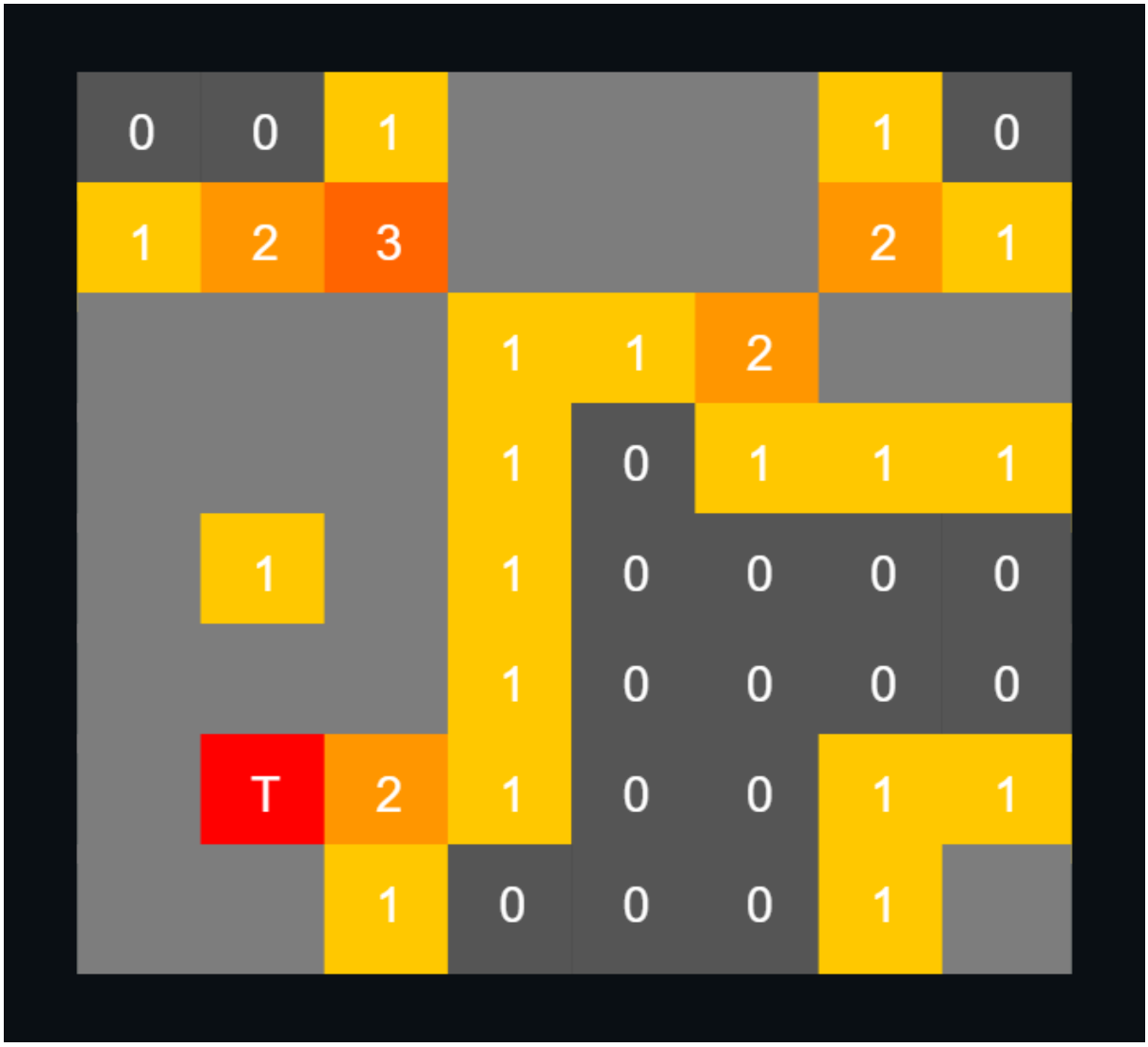

Figure 5: 8x8 Minesweeper with 15 mines