Implementation of Backtracking Algorithm in Minesweeper

Nayotama Pradipta - 13520089
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13520089@std.stei.itb.ac.id

***Abstract*— Minesweeper is a single player puzzle in which the player's objective is to solve all cells in a rectangular board without detonating the mines scattered throughout the board. Minesweeper was first released in 1989 together with Microsoft Entertainment Pack. Most people believed that Minesweeper involves guessing, while it may be true, this game can also be solved strategically. Anyone can solve this puzzle with the right strategy, including computers using a certain algorithm. That said, this paper discusses and dissects every aspect of the backtracking algorithm to solve the minesweeper puzzle.**

Keywords—Minesweeper, Backtrack, Algorithm, Efficiency, Depth-First Search

I. Introduction

Back in the 1980s, peripheral device in the form of mouse was not generally used with computers. A lot of people were not familiar with the clickable device. Minesweeper was then created by Microsoft to help people adapt with the left and right click of a mouse. Minesweeper itself is a board puzzle game consisting of cells that may or may not contain "mines", hence the name. These mines will cause the game to end whenever clicked, so players must only left-click the numbered cells and right-click the mines. The player will win if he/she manage to locate all numbered cells and avoid all mines. There are three difficulties/levels of minesweeper, specifically as follows:

1. Beginner: 10 mines randomly scattered in a 9 by 9 board
2. Intermediate: 40 mines randomly scattered in a 16 by 16 board
3. Expert: 99 mines randomly scattered in a 16 by 30 board

When a player runs the game, the program will initialize a board with plain cells. The game starts when the player clicks on one of the cells. The game should always let the first clicked cell to be a numbered cell to prevent an early "Game Over". The strategy of the game lies on the value of the numbered cells. The value of a numbered cell lies between 1 to 8 as this represents the number of mines that are adjacent to that cell. For instance, a cell with the number "8" means it is surrounded by mines. If a cell is blank, then the program will display all adjacent cells until the perimeter of the cells are numbered cells.

Figure 1. An example of a numbered cell. The flagged cells surrounding it are mines

A player can right-click on a cell that is suspected to be a mine. A flag will appear on that cell, as seen in figure 1. Players can only drop flags as much as there are mines in the board. As an example, there are only 10 flags that a player can use in a beginner level since there are only 10 mines in the board. In addition to that, there is a timer in the game to indicate how much time has passed since the player's first move.

Players can use simple math (additions) to finish the game, but there are also some cases where the player must completely guess by luck. The chances of having to guess is higher in harder levels since there are more cells, leading to higher probabilities of inadequate information to correctly identify a cell. Despite that, players can calculate the probability of each cell being a numbered or a mine cell, meaning that it does not involve wild guesses.

Minesweeper may be an old game and has a limited number of dedicated players, but it certainly is a fun and challenging puzzle that everyone should give a try. In fact, a lot of people know Minesweeper but does not really know how to play it. This paper is created with the hope of familiarizing everyone with Minesweeper as well as algorithm strategies.

A computer program can be made to solve the Minesweeper puzzle with a high win percentage. The program may lose whenever the information is not enough. A simple brute force algorithm will surely do the trick, but it is memory and time consuming. A good alternative would be the backtracking algorithm since it does not generate unnecessary possible steps. This paper shows a thorough implementation and analysis of the backtracking algorithm to solve a minesweeper puzzle.

II. THEORY

A. Backtracking Algorithm

Backtracking is an algorithm that solves a constraint satisfaction problem in a recursive manner. A constraint satisfaction problem is any problem that has clear constraints or rules. The goal in a CSP is to find the set of values that satisfy a condition/constraint. Backtracking is an improvement of exhaustive search since it only explores choices that will lead to the solution. Backtracking was first introduced in 1950 by an American mathematician D.H. Lehmer. There are several common properties in a backtracking algorithm:

1. Solution Space: All possible solution of the problem, written as a set of vectors with *n*-tuple
2. Generator Function: A function that generates a value x_k that is a component of the solution vector
3. Bounding Function: A boolean function that will return true whenever a set of value is leading to a solution and not violating constraints

B. Minesweeper

Minesweeper boards are made up of cells that either contains blank, a mine, or a number between 1 to 8. Minesweeper can be categorized as a CSP (Constraint Satisfaction Problem) since any cells in a board can be represented as a variable with a domain of 0 and 1. 0 indicates that the cell is not a mine, otherwise 1. When a numbered cell is analyzed, it becomes the constraint for the number of mines adjacent to its neighboring cells. A challenging problem in a Minesweeper board is when the information known is insufficient to make a conclusion. Guessing in minesweeper should not be done randomly as it will not guarantee the best move. Whenever a guess should be done, corners should always be the first choice since it has the highest chance of being mine-free. However, not all guesses can be strategically chosen. There are some scenarios where the chance of a mine is 50-50 in a cell, so the only way of moving forward is to choose either one of the cells. Whenever this scenario occurs, it is best to directly choose one of the cells to avoid unnecessary computation.

C. Backtracking Algorithm in Minesweeper

Backtracking is one of the possible core algorithms to solve a Minesweeper puzzle since it fits the criteria of a Constraint Satisfaction Problem. Backtracking algorithm is used in Minesweeper to recursively find all possible solutions of the cells that satisfy the constraint (Number of mines adjacent to cells must be equal).

There are two possible ways to which backtracking can be used in Minesweeper depending on the input:

1. Program/Player start with an entire covered board
2. Program/Player start with an entire uncovered board, each cell has a value from 0-8 indicating the total amount of mine surrounding it

The first case is the original Minesweeper game, and it is a lot more complex than the second one. The first case can be solved with Depth First Search that uses backtracking algorithm during the search. The second case can be solved with regular backtracking, and it eliminates the chance of guessing since all the information are enough to lead to the solution. This paper includes the second case to highlight the core of backtracking algorithm.

The general algorithm for the second is as follow:

1. Create two matrixes, one to store the resultant board that will be updated in each recursion, and the other one to store the visited cells when traversing in the resultant cell.
2. If all the cells have been visited and satisfies the constraint, then return true
3. If all the cells have been visited but it does not satisfy the constraint, then return false
4. If false, then find an unvisited cell and mark that cell as visited, for instance (a,b)
5. If it is possible to assign a mine to that (a,b), then decrement the number of mines in adjacent cell and do step 2-5 again (recursion)
6. If the process of recursion on (a,b) returns true, then return true, else return false

D. Depth First Search in Minesweeper

Another algorithm that can be used in Minesweeper is Depth-First Search. DFS is an approach that also implements the backtracking algorithm. The backtracking in DFS is applied in the leaf nodes. DFS should also be implemented to problems that can be transformed in an explicit tree. In a minesweeper puzzle, the root of the tree is the first cell clicked by the player. From there, the search search continues to one the direction until it is impossible to identify which cells are mine-free. At this point, DFS uses backtracking to another direction, and this will be recursively done until the game ends. A good explanation of the creation of State space Tree using DFS algorithm can be seen in the figure below:

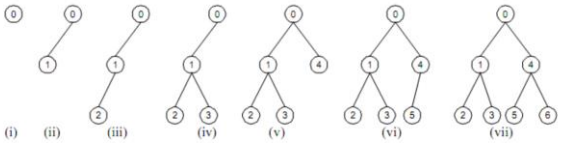


Figure 2. State Space Tree Creation with DFS

III. BACKTRACKING ALGORITHM IMPLEMENTATION

In the case where there are no covered cells in the beginning, the implementation of the backtracking algorithm can be seen below:

```
#define MAXM 100
#define MAXN 100
```

```
int dx[9] = { -1, 0, 1, -1, 0, 1,
-1, 0, 1 };
int dy[9] = { 0, 0, 0, -1, -1, -1,
1, 1, 1 };

bool isValid(int x, int y)
{
    return (x >= 0 && y >= 0 && x
< N && y < M);
}
bool canAssignMine(int
arr[MAXN][MAXM], int x, int y)
{
    if (!isValid(x, y))
        return false;
    for (int i = 0; i < 9; i++) {
        if (isValid(x + dx[i], y +
dy[i])
            && (arr[x + dx[i]][y +
dy[i]] == 0))
            return false;
    }
    for (int i = 0; i < 9; i++) {
        if (isValid(x + dx[i], y +
dy[i]))
            arr[x + dx[i]][y +
dy[i]] -= 1;
    }
    return true;
}
bool findUnvisited(bool
visited[MAXN][MAXM],
                   int& x, int& y)
{
    for (x = 0; x < N; x++)
        for (y = 0; y < M; y++)
            if (!visited[x][y])
                return (true);
    return (false);
}
bool isVisitedandSatisfied(int
arr[MAXN][MAXM],
            bool
visited[MAXN][MAXM])
{
    bool done = true;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M;
j++) {
            done = done &&
(arr[i][j] == 0) && visited[i][j];
        }
    }
    return (done);
}
bool
SolveBackTrackMinesweeper(bool
grid[MAXN][MAXM], int
arr[MAXN][MAXM], bool
visited[MAXN][MAXM])
{
    int x, y;
    if (isVisitedandSatisfied(arr,
visited))
        return true;
    if (!findUnvisited(visited, x,
y))
        return false;
    visited[x][y] = true;
    if (canAssignMine(arr, x, y))
{
        grid[x][y] = true;
        if
(SolveBackTrackMinesweeper(grid,
arr, visited))
            return true;
        grid[x][y] = false;
        for (int i = 0; i < 9;
i++) {
            if (isValid(x + dx[i],
y + dy[i]))
                arr[x + dx[i]][y +
dy[i]]++;
        }
    }
    if
(SolveBackTrackMinesweeper(grid,
arr, visited))
        return true;
    visited[x][y] = false;
    return false;
}
```

An input example would be an array below:

```
int arr[MAXN][MAXN] = {
     { 0, 0, 1, 1, 1, 0, 0, 2, 2},
     { 0, 0, 1, 1, 1, 0, 0, 2, 2},
     { 0, 0, 1, 1, 1, 0, 0, 1, 1},
     { 0, 0, 1, 2, 2, 1, 0, 0, 0},
     { 0, 1, 2, 3, 2, 1, 0, 0, 0},
     { 1, 2, 3, 3, 2, 1, 0, 0, 0},
     { 1, 2, 2, 2, 1, 1, 0, 1, 1},
     { 2, 2, 2, 1, 1, 1, 0, 1, 1},
     { 1, 1, 1, 1, 1, 1, 0, 1, 1}
};
```

Or better shown in the figure 3:

Figure 3. Example of a 9 by 9 solved Minesweeper

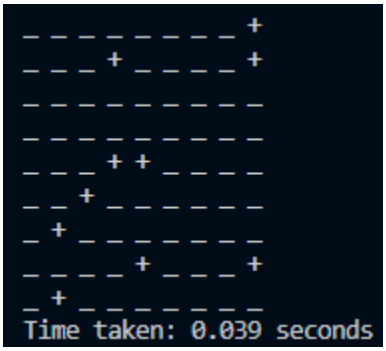


Figure 4. Output program, "+" means the cell contain mine

While the time taken for the program is less than one second, it certainly cannot be ignored. This algorithm takes 0.039 second for a 9 by 9 board, which means that it will be take longer for the intermediate and hard level. In fact, an experiment with a 16 by 16 minesweeper board with 40 mines took more than five minutes to solve.

The Implementation for the first case is much more complex than the second since not all blocks are uncovered. In a true Minesweeper game, even with a solvable board, a program will not be able to perfectly solve all puzzles. In this case, Depth First Search and Constraint Propagation Algorithm is used to create the solution that resembles human logic. Constraint Propagation Algorithm is similar to flagging the obvious mines cells. The purpose of Constraint Propagation is to eliminate all known cells and only focus on the unknown.

There are three possible conditions for each covered cells in a minesweeper board. The first condition is achieved when the value of the numbered cell is equal to the number of covered cells surrounding it. This literally means that all covered cells are mines. The second condition is when the value of the numbered cell is equal to surrounding flagged cell but there are still other covered cells. The other covered cells can be safely assumed as numbered cell. The last condition is when there are two or more covered cells that cannot be determined as the first or second condition. To solve the last condition, the backtracking method is going to be used to search all the possible solutions that are still true to the bounding function.

There are several methods/functions that should be created to implement the minesweeper solver. These functions include:

1. A function that returns true whenever the program uncovers a mine cell
2. A backtracking function
   - Bounding function (Constraint)
3. Remove Known Cell function (Constrain Propagation)

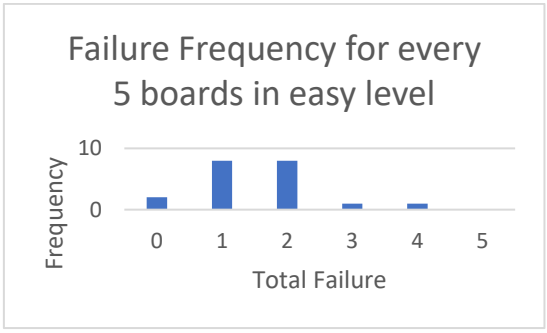
The pseudocode for the core algorithm:

```
function search(self, cell) → void
Variable Declaration
  leftovers, sss: dictionary
  squares: list of dictionary keys
  mines_left, squares_left: integer
  solutions : array of solution
Algorithm
  leftovers ← {}
  squares ← list of leftovers.keys
  mines_left ← self.num_mines -
self.flagged_num_mines
  squares_left ← length(squares)
  solution ← []
  m traversal[moves]
    if m.constraints then
      const traversal [m.constraints]
        if const not in leftovers then
          leftovers[const] ← 1
        else
          leftovers[const] ←
leftovers[const] + 1
  if mines_left < squares_left then
    backtrack([], solutions,
self.board)
  if solutions then
    sss ← {}
    i traversal [0…length(solutions)-1]
      j traversal
[0…length(solutions[i])-1]
        currSquare ← squares[j]
        if currSquare not in sss then
          sss[currSquare] ←
solutions[i][j]
        else
          sss[currSquare] ←
sss[currSquare] + solutions[i][j]
    addSafeSquare ← false
    square,count traversal [sss.items]
      if count = 0 then
        addSafeSquare ← true

  self.squareToProbe.append(square)
    if not addSafeSquare then
      randomSol ← randomize(0,
length(solutions) - 1)
      arr ← solutions[randomSol]
      square, value traversal
[zip(squares, arr)]
        if value = 0 then

  self.squareToProbe.append(square)
  else
    squares_left ←
list(set(self.board_coord -
self.marked_squares)
```

```
     random_square ← randomize(0,
len(squares_left - 1)
     next_square ←
squares_left[random_square]

   self.squareToProbe.append(next_square
)
   self.start()
   ➔
function backtrack(input arr, solutions
: array, board: Minesweeperboard) →
List of solutions
Variable Declaration
   Valid : boolean
Algorithm
   if (length(arr) < squares_left) then
      ➔
   else if (sum(arr) > mines_left) then
      ➔
   else
      choice traversal [0,1]
        arr.append(choice)
        if sum(arr) = mines_left and
len(arr) = squares_left then
          valid ←
checkSolutionValidity(board, arr)
          if valid then
            solutions.append(arr)
        backtrack (arr, solutions, board)
     ➔ solutions
```

With the help of constraint propagation, the program works far faster than the first program (C++). Implementation of the backtracking algorithm is efficient as only valid solutions are calculated. In addition to the implementation, the success rate of the program is recorded in three different difficulties.

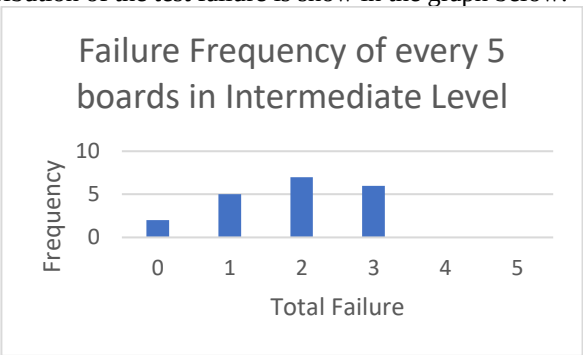
For the easy difficulty, the test was conducted 100 times in groups of 5. An easy level has a 9 by 9 board and 10 mines. The distribution of test failure is shown in the graph below:



Graph 1. Failure Frequency for every 5 boards in easy level

From the graph, it can be concluded that there are around one to two failure in every 5 puzzle for an easy game. Statistically speaking, the program has a 69% success rate for an easy minesweeper level.

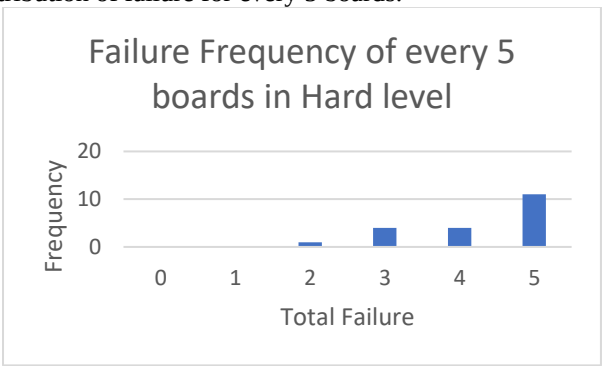
For the intermediate difficulty, the test was conducted with the same manner as easy difficulty. The only difference is that the intermediate level has a 16 by 16 board and 40 mines. The distribution of the test failure is show in the graph below:



Graph 2. Failure Frequency for every 5 boards in intermediate level

The graph for intermediate level shifts a little to the right, as most of the total failure is between 2 and 3 for every 5 puzzles. The success rate of the program is 63%.

The same experiment is applied for the hard difficulty with 16 by 30 board and 99 mines. The following graph depicts the distribution of failure for every 5 boards:



Graph 3. Failure Frequency for every 5 boards in hard level

Graph 3 shows that the majority of experiments result in failure, with 11 out of 20 being 100% failure. In total, the success rate of the program drops to just only 15%.

IV. Analysis Of the Solution

When the program receives a totally uncovered board, it can solve the puzzle with a 100% success rate. The backtracking algorithm is not the most efficient algorithm, this can be seen by the considerable amount of time needed to solve an intermediate puzzle. In fact, the time complexity of this algorithm is

$$O(2^{M*N} * M * N)$$

with M and N as the dimension of the board. The time complexity illustrates that the time taken becomes a lot larger for bigger boards. This is proven from the huge difference of time between easy level and intermediate level (From 0.039 second to more than 5 minutes).

The second program (Starts with covered board) is different to the first as there are chances of a failure. The efficiency of the algorithm is much higher thanks to the

constraint propagation. An obvious pattern can be seen from the graph 1, 2, and 3. As the size of the board increases, the success rate of the program deteriorates. This is because larger boards allow higher chances of inadequate information. The algorithm has a hard time in solving hard puzzles, with most of the tests being a failure. In addition, some of the tests took too long that the program must be terminated and executed again. While backtracking maybe a lot faster than brute force algorithm, it is not the most efficient. Further modifications of the program can be made to improve efficiency. Modifications may include adding a method to calculate probabilities on adjacent cells that are unknown.

V. Conclusion

Minesweeper may look like a simple board game, but there are complex calculations that can be made to finish the puzzle. The recursive algorithm in the form backtracking is a great solution to this problem. The algorithm must be implemented together with constraint propagation and applied in Depth First Search to maximize the efficiency of the program. A program can perfectly solve an uncovered board but will not have the same precision in a covered board. No matter how much modifications made in a program, the success rate will never reach 100%, especially in larger boards. Implementation of the backtracking algorithm can also be applied to normal players, not just programs. Players will have a high chance of winning Minesweeper if they apply the backtracking algorithm correctly, but it can be time consuming.

Youtube Link

Further explanation of this paper is available at:
https://www.youtube.com/watch?v=GuUCRP6Fd3o

Acknowledgment

I offer my deepest gratitude to God and everyone who has supported me during the making of this paper. I would also wholeheartedly thank my lecturer Dr. Nur Ulfa Maulidevi, S.T., M.Sc. for all the knowledge and teachings given to me during my fourth semester in Informatics.

References

[1] Munir, Rinaldi. (2021). Algoritma Runut-balik (Backtracking). Available at: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf (Last accessed on 21 May 2022)

[2] Munir, Rinaldi. (2021). Breadth/Depth First Search (BFS/DFS). Available at: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf (Last accessed on 21 May 2022)

[3] Mehta, A. (2021). Minesweeper Solver. Available at: https://www.geeksforgeeks.org/minesweeper-solver/ (Last accessed on 21 May 2022)

[4] Levengood, C. (2020). Solving Minesweeper in Python as a Constraint Satisfaction Problem. Available at: https://lvngd.com/blog/solving-minesweeper-python-constraint-satisfaction-problem/ (Last accessed on 21 May 2022)

[5] Becerra, D. (2015). Algorithmic Approaches to Playing Minesweeper. Available at: https://dash.harvard.edu/bitstream/handle/1/14398552/BECERRA-SENIORTHESIS-2015.pdf (Last accessed on 21 May 2022)

Statement

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2022

Nayotama Pradipta 13520089