Implementation of Breadth First Search Algorithm for the Will They Survive Game

Ishak Palentino Napitupulu - 13524022
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: ishaknapitupulucxr2@gmail.com , 13524022@std.stei.itb.ac.id

***Abstract*— This paper presents an implementation of breadth-first search (BFS) for the grid-based survival game Will They Survive. The program models a disaster layout as a two-dimensional matrix containing citizens, empty cells, a safe point, a fire source, and burned cells. Each citizen is treated as a starting state, while the safe point is the goal state. BFS is used to find the shortest valid path from every citizen to the safe point by expanding four-directional neighboring cells and storing parent pointers for path reconstruction. After all paths are computed, the program runs a time-step simulation: citizens move along their planned paths while fire spreads from its initial source and may burn cells or citizens. The final output reports how many citizens survive and how many are burned. The implementation demonstrates how graph traversal can be used as a decision-making mechanism in a simple game simulation.**

Keywords— breadth-first search; graph traversal; grid map; shortest path; evacuation simulation; C programming

I. Introduction

Pathfinding is an important problem in games that require characters to move through a map. In Will They Survive, the map is represented as a grid containing citizens, open cells, a safe zone, and fire. The main question is simple but algorithmically meaningful: can each citizen reach safety before the simulation ends, and what path should each citizen follow?

The objective of this project is to implement a shortest-path strategy for a survival scenario. The game does not generate a complex open world; instead, it receives a layout from user input. This layout is transformed into a graph implicitly, where each walkable grid cell is a vertex and movement between adjacent cells is an edge. The program then uses BFS to determine the shortest route from a citizen to the safe cell.

Example map:
W 0 S
0 0 0
F 0 0

Fig. 1. Example Will They Survive layout with a citizen, safe point, and fire source.

The program combines this pathfinding result with a time-based simulation. Citizens move step by step along the BFS path. At the same time, fire expands from the fire source and converts affected cells into burned cells. A citizen that reaches the safe point is counted as safe, while a citizen that remains outside safety and is reached by fire may be counted as burned.

The main algorithm used in the implementation is breadth-first search. BFS is chosen because every valid move in the grid has the same cost: moving up, right, down, or left requires one step. Under this uniform-cost condition, BFS guarantees that the first time the safe point is reached, the discovered route has the minimum number of moves from the citizen to the safe point.

II. Theoretical Foundation

The first component of the model is the grid map. If the map has R rows and C columns, every cell can be represented by a coordinate pair (r,c). The set of vertices is:

$$V = \{(r,c) \mid 0 <= r < R \text{ and } 0 <= c < C\}$$

Not every vertex can be traversed. In the implementation, BFS may move into cells containing 0, W, or S. The cell 0 represents an empty path, W represents a citizen position, and S represents the safe point. Burned cells X and other non-walkable symbols are excluded from BFS expansion.

The grid graph uses four-directional movement. From a cell (r,c), the program considers moving to the upper, right, lower, and left neighboring cells. These movement offsets are stored in the dx and dy arrays inside findShortestPath.

Four-neighbor model:
(r-1,c)
(r,c-1) (r,c) (r,c+1)
(r+1,c)

Fig. 2. Four-directional grid movement used by BFS.

The neighboring relation can be written as the following edge definition.

```
Cell classes used by the graph model:
0 : walkable empty cell
W : citizen start state
S : safe goal state
F : fire source
X : burned cell, excluded from BFS
expansion
```

E = {((r,c),(r+dr,c+dc)) | (dr,dc) in {(-1,0),(0,1),(1,0),(0,-1)}}

A neighbor is valid only if it is inside the map boundary, is walkable, and has not been visited before.

The second component is BFS itself. BFS uses a queue to explore cells in increasing distance from the starting point. The queue initially contains only the citizen position. Each time a cell is removed from the front of the queue, all valid unvisited neighbors are inserted at the rear of the queue.

The BFS invariant is that cells closer to the start are processed before cells farther from the start. Therefore, when the algorithm first reaches S, the path stored through parent pointers is a shortest path measured by number of grid moves.

queue[rear++] = start, then repeatedly current = queue[front++]

This queue discipline is the main difference between BFS and depth-first search. A stack-based search may reach the safe point quickly in some cases, but it does not guarantee the shortest route in an unweighted grid. BFS provides the guarantee needed for a fair evacuation path.

The third component is the visited matrix. The program stores visited[r][c] to prevent a cell from being inserted into the queue more than once. This avoids repeated work and prevents the algorithm from looping indefinitely on cyclic maps.

visited[nx][ny] = 1 when a valid neighbor is inserted into the queue

Because each cell is marked when it is enqueued, every cell enters the queue at most once. This property is what gives BFS linear time complexity relative to the number of cells and edges in the implicit graph.

The fourth component is path reconstruction. BFS does not only need to know whether S is reachable; the simulation also needs the exact sequence of coordinates that the citizen should follow. For this reason, the program stores the predecessor of each visited cell in a parent matrix.

parent[nx][ny] = current

After S is found, getPath walks backward from the safe point to the starting point using the parent matrix. The reversed sequence is then stored in the citizen path arrays dx and dy so that the simulation can move the citizen one step at a time.

The fifth component is the fire spread model. The program searches the layout for the initial fire source F. During simulation, a range value starts at zero and increases by 0.5 after each time step. When the range is an integer, cells inside the square centered at the fire source are converted into burned cells, except for S and F.

range_0 = 0, range_{t+1} = range_t + 0.5

This model is simple but sufficient to create pressure in the game. A path that is shortest in the static grid may still fail if fire reaches the citizen before the citizen reaches safety. The current implementation computes the BFS path before simulation, then evaluates survival during the time-step process.

burned area at integer range k: fireX-k <= r <= fireX+k and fireY-k <= c <= fireY+k

The sixth component is termination. The simulation continues while at least one W remains on the map. Once all citizens have either reached S or been removed by fire, the loop ends and the program counts safe and burned citizens.

The complexity of BFS for one citizen is O(R*C) in the worst case, because every cell may be visited once. The memory complexity is also O(R*C), mainly for the queue, visited matrix, and parent matrix.

T_BFS(one citizen) = O(R*C), S_BFS(one citizen) = O(R*C)

If there are K citizens, the program runs BFS independently for each citizen. Therefore, the pathfinding cost becomes:

T_BFS(all citizens) = O(K*R*C)

The simulation phase scans the grid repeatedly while citizens move and fire spreads. If the simulation lasts T time

steps, the dominant cost is bounded by O(T*R*C), because the implementation repeatedly checks cells and prints the whole layout.

The theoretical model therefore separates the algorithm into two parts: BFS for planning and time-step simulation for evaluation. BFS answers the shortest-path question, while the simulation answers the survival question under spreading fire.

BFS result: shortest path from each W to S in the static grid.

Simulation result: whether each planned path succeeds after fire expansion.

This separation makes the program easy to understand. The search logic is handled in findShortestPath and getPath, while the changing game state is handled in simulate.

The main limitation of this model is that fire is not considered during BFS planning. The path is shortest with respect to the initial map, not necessarily safest with respect to future fire positions.

Even with this limitation, the implementation is appropriate for demonstrating the use of BFS in a grid-based game. It shows how an abstract graph algorithm can be connected directly to visible movement and game outcomes.

III. IMPLEMENTATION

The program is implemented in C inside program.c. It includes stdio.h for input and output, windows.h for Sleep, and math.h for fmod. The constant MAX is set to 100, so the maximum supported map size is 100 by 100 cells.

The Layout structure stores the number of rows, the number of columns, the map elements, and the visited matrix. The map is stored as element[MAX][MAX], while visited[MAX][MAX] is used by BFS to mark cells that have already been explored.

```
typedef struct {
    int rows;
    int cols;
    char element[MAX][MAX];
    int visited[MAX][MAX];
} Layout;
```

Program flow:
read dimensions -> read map -> run BFS for each W -> simulate movement and fire -> print survivor counts

Fig. 3. Overall execution flow of program.c.

The Warga structure stores the movement path and current position of a citizen. The arrays dx and dy are used as coordinate sequences, not merely direction offsets. After BFS finishes, dx[i] and dy[i] contain the row and column of the citizen at simulation step i.

The Point structure is a small helper structure containing x and y coordinates. It is used for queue elements, parent entries, and temporary coordinate values inside BFS and path reconstruction.

Important symbols:
W: citizen
S: safe point
F: fire source
0: empty cell
X: burned cell

Fig. 4. Map symbols interpreted by the implementation.

The isSimulateEnd function checks whether the simulation should stop. It scans the whole layout and returns false if at least one W still exists. If no W remains, the simulation is considered complete.

```
typedef struct {
    int dx[MAX], dy[MAX];
    int x, y;
    int isSafe;
} Warga;
```

```
if (inside_map && walkable &&
!visited[nx][ny]) {
    visited[nx][ny] = 1;
    parent[nx][ny] = current;
    queue[rear++] = (Point){nx, ny};
}
```

The findShortestPath function is the core BFS routine. It initializes the parent matrix with (-1,-1), marks the starting cell as visited, inserts the start point into the queue, and then processes the queue until either S is found or no more cells can be explored.

Pseudocode:

```
while front < rear:
    current = queue[front++]
    if element[current.x][current.y] == S:
reconstruct path
    for each of four directions: enqueue
valid unvisited neighbor
```

```
When a valid neighbor is discovered, the
program stores:
 parent[nx][ny] = current; queue[rear++] =
 (nx,ny); visited[nx][ny] = 1
```

The getPath function reconstructs the answer after BFS reaches S. It starts from the safe point and repeatedly follows parent links until it returns to the citizen starting cell. Because this traversal produces the path backward, the function reverses the order before writing it into the Warga arrays.

```
while (!(end.x == start.x && end.y ==
start.y)) {
    path[length++] = end;
    end = parent[end.x][end.y];
}
path[length++] = start;
```

The implementation flow for path planning can be summarized as follows. First, main reads the grid. Second, it scans for every cell containing W. Third, before each BFS call, it resets the visited matrix to zero. Fourth, it calls findShortestPath. Fifth, the resulting coordinate sequence is stored in the corresponding Warga entry.

The simulate function runs the dynamic part of the game. It first locates S and F. Then, while at least one citizen remains on the map, it moves citizens according to their stored BFS paths, updates safe status when a citizen reaches S, expands fire according to the current range, prints the layout, and waits one second using Sleep(1000).

```
 while (!isSimulateEnd(layout)) {
     move citizens using BFS path[step];
     spread fire when range is an integer;
     display the updated layout;
     step += 1;
 }
```

Citizen movement uses the current simulation step as an index into the stored path arrays.

```
warga[k].x = warga[k].dx[step]; warga[k].y
= warga[k].dy[step]
```

If the new citizen coordinate equals the safe coordinate, the citizen is marked as safe and is not written back as W on the map. Otherwise, the map cell at the new coordinate is updated to W.

```
if warga[k].x == safeX and warga[k].y ==
safeY: warga[k].isSafe = 1
```

The fire spread logic is implemented after citizen movement. The program checks fmod(range,1.0) to decide whether the fire should expand on the current step. Cells inside the current fire square are converted to X unless the cell is S or F.

```
Final counting logic:
if (warga[i].isSafe == 1) safe += 1;
else dead += 1;
printf("jumlah warga yang selamat = %d",
safe);
```

Each displayed simulation frame shows the current state of the grid after movement and possible fire expansion. This gives the user a direct view of whether the BFS-planned path still allows a citizen to survive under the fire model.

output per frame = current layout matrix followed by an empty line

At the end, main counts citizens whose isSafe value is 1 as survivors. Citizens that do not have isSafe equal to 1 are counted as burned. The program then prints the number of safe citizens and burned citizens.

```
Sample final output:
jumlah warga yang selamat = 1
jumlah warga yang terbakar = 0
```

Fig. 5. Example final result produced by the BFS evacuation simulation.

IV. Result and Discussion

A small local test was executed using a 3 by 3 layout. The initial map was W 0 S on the first row, 0 0 0 on the second row, and F 0 0 on the third row. In this case, BFS finds the path (0,0) -> (0,1) -> (0,2). The safe point is reached in two moves.

The displayed simulation confirms this behavior. First, the citizen remains at the starting coordinate for the first indexed path state. Then the citizen moves to the middle cell of the first row. Finally, the citizen reaches S, while the fire expands around F and burns nearby cells in the lower part of the grid.

For this test case, the program prints one survivor and zero burned citizens. This result is consistent with the BFS path length and the fire spread timing: the citizen reaches the safe point before the fire expansion reaches the top-row route.

The result illustrates why BFS is suitable for the pathfinding portion of the game. Since every step has equal cost, the shortest path is the path with the fewest moves. BFS discovers that path without needing a heuristic or weighted cost function.

The implementation also shows a meaningful distinction between shortest path and safest path. The BFS route is optimal only under the static map interpretation. If fire spread is considered as a changing obstacle, a longer path could sometimes be safer than the shortest path. The current program evaluates survival after planning rather than integrating fire timing into the search state.

Another important observation is that the program runs BFS separately for each citizen. This keeps the code simple and makes each citizen independent. However, it also means that the program does not coordinate citizens, does not avoid collisions between citizens, and does not reserve cells for different movement times.

There are several implementation limitations that can be improved. The isSafe field should be initialized to zero for every citizen before simulation begins. The program should also handle the case where no path to S exists, because otherwise the path arrays may not contain a valid route. In addition, BFS could be extended to include time as part of the state, allowing the search to avoid cells that will burn before the citizen arrives.

Overall, the implementation successfully demonstrates the central idea: BFS can transform a grid into a shortest-path decision process, and the result can be used directly by a simple game simulation to answer the question Will They Survive?

V. Conclusion

This project implements BFS for the Will They Survive game using a two-dimensional grid map in C. Citizens are treated as starting points, the safe cell is treated as the goal, and walkable cells form an implicit unweighted graph.

The main algorithm is breadth-first search. By using a queue, a visited matrix, and a parent matrix, the program explores cells in increasing step distance and reconstructs the first path that reaches S. Because all movement costs are equal, this path is guaranteed to be one of the shortest paths from the citizen to safety.

The algorithm is appropriate because the game map is discrete, movement is limited to four directions, and the objective of each citizen is to reach the safe point with the fewest moves. The simulation then uses the BFS path to animate movement, apply fire spread, and determine the final survivor count.

Future development could improve the model by initializing all citizen state explicitly, handling unreachable safe points, supporting multiple safe cells, adding obstacles, and extending BFS into a time-aware search that avoids cells predicted to burn before arrival.

Appendix

Github:
https://github.com/Ishakpln/Makalah-2-STIMA-Will-They-Survive

Acknowledgment

First and foremost, the author wishes to offer his sincere gratitude to God Almighty for the blessings, perseverance, and guidance granted to him during the preparation and completion of this research paper.

The author would also like to convey his heartfelt appreciation to his beloved family for their unconditional love, constant prayers, and continuous support, which have remained an important source of motivation throughout his academic journey.

Special appreciation is addressed to Dr. Ir. Rila Mandala, M.Eng., Ph.D., along with all members of the teaching staff of the IF2211 Algorithm Strategies course at Institut Teknologi Bandung, for the valuable knowledge, direction, and assistance provided throughout the learning process.

Lastly, the author extends his gratitude to his fellow students in the Informatics Engineering program for their cooperation, encouragement, and companionship during the semester. Their support and contributions have been sincerely valued.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA, USA: MIT Press, 2009. Chapter 22 discusses elementary graph algorithms including breadth-first search. Diakses pada 19 Juni 2026

[2] R. Sedgewick and K. Wayne, Algorithms, 4th ed. Boston, MA, USA: Addison-Wesley, 2011. [Online]. Available: https://algs4.cs.princeton.edu/41graph/ Diakses pada 19 Juni 2026

[3] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, Algorithms. New York, NY, USA: McGraw-Hill, 2008. Chapter 3 covers graph decompositions and graph traversal. Diakses pada 19 Juni 2026

[4] R. Munir, Strategi Algoritma. Bandung, Indonesia: Program Studi Teknik Informatika, Institut Teknologi Bandung. Materi kuliah graf dan strategi pencarian digunakan sebagai landasan konseptual BFS. Diakses pada 19 Juni 2026

[5] GeeksforGeeks, "Breadth First Search or BFS for a Graph." [Online]. Available: https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/ Diakses pada 19 Juni 2026

[6] CP-Algorithms, "Breadth First Search." [Online]. Available: https://cp-algorithms.com/graph/breadth-first-search.html Diakses pada 19 Juni 2026

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, pp. 269-271, 1959. This reference provides broader context for shortest path problems. Diakses pada 19 Juni 2026

[8] Amit Patel, "Introduction to the A* Algorithm." [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html Diakses pada 19 Juni 2026

STATEMENT

In this statement, I declare that this paper I have written is my own work, not a duplication or translation of someone else's paper, and is not plagiarized.

Bandung, 19 Juni 2026

Ishak Palentino Napitupulu 13524022