Pathfinding Evacuation Route

BINUS UNIVERSITY

BINUS INTERNATIONAL

Algorithm Design Analysis

Final Project

Student Information:

Student	**1**	**2**
Surname	**Rahyang**	**Jonathan**
Given Name	**Fadhillah Haidar Rahyang**	**Kevin Jonathan**
Student ID	**2702337211**	**2702342823**

Course Code : COMP6049001 **Course Name** : Algorithm Design Analysis

Class : L3BC **Lecturer** : Andreas Kurniawan, S.Kom., MTI.

Type of Assignment : Final Project Report

Table of Contents

I. Introduction

Society has faced multiple challenges, expected and unexpected alike. Both types of challenges sometimes pose a threat to people's lives, especially unexpected ones. One of the biggest threats that society, up until today, has been facing are natural disasters. Natural disasters can be defined as a combination of natural hazards and vulnerabilities that endanger vulnerable communities that are incapable of withstanding the adversities arising from them (Blaikie et al., 2014). Some of these natural events include,

floods, an overflowing of large water in urban areas requiring tall sturdy buildings,
earthquakes, a sudden violent shaking of the ground requiring large flat areas,
hurricanes, a storm with violent winds requiring also sturdy buildings,
and many more.

When these disasters do happen, it takes many lives and causes close families to become bereaved. To prevent this from happening, society made certain areas safe for the surrounding people to evacuate to. These points are called muster points and there are several muster points provided around large buildings that occupy many people. This ensures that everyone can be safe.

II. Problems

Despite the countermeasures created to prevent casualties, one big factor that prevents it from being 100% reliable is predictability. During the disaster, clear pathways and evacuators are present to guide the people into the closest available muster point. The question is, are the pathways completely safe? Due to the unpredictability of natural phenomena, rubles and building debris may have a chance to hinder these pathways, preventing people from going to safer areas, trapping them. Worst case scenario, all clear pathways have been blocked and the rescue requires air support such as a helicopter to rescue them.

III. Proposed Solution

One of the safer solutions involves finding a new route that the citizens can travel through to get to the muster point. To obtain this route, pathfinding algorithms can be utilised to trace necessary ground to conclude a safe path. One of the big advantages of this is that it happens live hence, whenever the current route is blocked again the algorithm retraces the area to scout another alternative for survivors to go through.

IV. Measurement

Several key measurements are required to conclude the algorithm's efficiency in solving the problem. Our presented problem involves saving as many human lives as possible during a natural disaster, where obstacles or debris are generated and the algorithm needs to trace updated paths to guide these survivors. With this, certain factors need to be made consistent to obtain results as accurate as possible. This includes the duration of the simulation, obstacle spawn order, and the map or grid. The factors that will be calculated are the amount of memory taken from each algorithm, the number of iterations, the time taken of each iteration, and the number of escaped survivors.

Several python modules are necessary to measure the total memory allocation and the execution time of an algorithm, which includes:

- Time: A python module that provides several functions to format time values into strings (Python 3, n.d.). This is necessary to keep track of execution time of a snippet of a code.
- Tracemalloc: A debug tool that traces memory blocks allocated by Python (Python 3, n.d.). This is necessary to measure the amount of memory used by each algorithm.

Lastly, to ensure consistency of results, all measurements are computed on the same device (Acer-Swift 3 SF314-42) with the following specifications:

- Operating System: Windows 11
- Processor: Ryzen 5 4500U @2.38GHz
- RAM: 8.00GB LPDR4

V. Theories of Algorithm

The space and time complexity table is as follows:

Algorithm	Space Complexity	Time Complexity
Dijkstra	O(V)	O(V log V)
A-star	O(V)	O(V log V)
A-star refactored	Reduced, but still O(V)	O(V log V)

a. Dijkstra

A famous path-finding algorithm that explores nodes of a given graph in an attempt to find the shortest or the lowest-cost path from the start to end node (Javaid, 2013). To do this, first it selects the start node and places it in a list to mark it as an already visited node. After that, it will locate every non-visited neighbouring node from that start node and place those in the visited list. This process will continuously repeat for every newly explored node until the end node is found. At the end of the algorithm after the end node is discovered, it will trace back to the start node, concluding the shortest path. If this algorithm was implemented in a non-weighted 50 by 50 grid, the visualised exploration would look like a circular spread.

In theory, if an adjacency list is used, the time complexity of Dijkstra's algorithm is O(V log V) where V is the number of nodes in the grid. The space complexity is O(V) regardless as the algorithm is bound to explore all nodes for the worst case scenario.

```
def dijkstra_algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}

    g_score = {node: float("inf") for row in grid for node in row}
    g_score[start] = 0

    open_set_hash = {start}
```

Figure 5.1. Snippet of Dijkstra's algorithm.

Figure 5.1 shows a code snippet of dijkstra's algorithm. This will be used to calculate the time and space complexity of the algorithm. Starting with time complexity,

count = 0 is O(1)

open_set = PriorityQueue() is O(1)

open_set.put((0, count, start)) is O(log V)

came_from = {} is O(1)

g_score = {node: float("inf") for row in grid for node in row} is O(V)

g_score[start] = 0 is O(1)

Overall time complexity is O(V log V), same with the theory.

In the case of space complexity, the worst case scenario is if all the nodes are explored which would be O(V). Initializing all nodes to infinity is also O(V). Overall space complexity is O(V), same with the theory.

b. A*

Another popular path-finding algorithm that is considered the best algorithm in most scenarios is the A* algorithm. Similar to Dijkstra's algorithm, it explores neighbouring nodes in a set graph to find the lowest-cost path. However, it is better than Dijkstra's Algorithm because it also calculates the euclidean distance of an explored node to the end node. This is possible due to its heuristic function. Dijkstra's algorithm makes optimal choices in a localised environment, whereas A-star uses its heuristic function to get the actual and estimated cost from start to end nodes. This allows the A-star algorithm to observe the problem in a wider area.

In theory, the time and space complexity of the A* algorithm is O(V log V) and O(V) respectively as it also explores nodes of the graph. However, with the help of the heuristic function, the time taken to search the best path is drastically decreased when compared to Dijkstra's algorithm. In return, the algorithm requires more memory to be used for the fast speed.

```
def a_star_algorithm(draw, grid, start, end):

    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}

    g_score = {node: float("inf") for row in grid for node in row}
    g_score[start] = 0

    f_score = {node: float("inf") for row in grid for node in row}
    f_score[start] = h(start.get_position(), end.get_position())

    open_set_hash = {start}
```

Figure 5.2. Snippet of the A-star algorithm.

Figure 5.2 shows a snippet of the A-star algorithm. This will be used to calculate the time and space complexity of the algorithm. Starting with the time complexity,

count = 0 is O(1)

open_set = PriorityQueue() is O(1)

open_set.put((0, count, start)) is O(log V)

came_from = {} is O(1)

g_score = {node: float(“inf”) for row in grid for node in row} is O(V)

g_score[start] = 0 is O(1)

f_score = {node: float(“inf”) for row in grid for node in row} is O(V)

f_score[start] = h(start.get_position(), end.get_position()) is O(1)

The h represents the heuristic function which utilises the euclidean distance, calculated in O(1) time. Overall time complexity is O(V log V), same with the theory.

In the case of space complexity, the worst case scenario for this algorithm is also O(V) where all nodes are explored. The initialization of each node to infinity is also O(V). Overall space complexity of this algorithm is also O(V), and is the same with the theory.

VI. Flow of Development

The objective is to deduce the algorithm that is better suited for our problem. Once decided, a refactored version of that algorithm will be made to further enhance its efficiency in solving our problem. Two algorithms will be used and compared during the simulation, which are, dijkstra and A*. In theory, the A* algorithm will perform better in every situation, but conducting tests is necessary for best practices.

Starting with the code, a total of 7 files are created to simulate a natural disaster with pathfinding algorithms providing paths for survivors to go from one point to another. The algorithms.py file contains all algorithms, including the refactored algorithm, that will be used for simulation. The constants.py file consists of variables that determine window application size and colour using RGB. The node.py file serves as the building blocks for the grid, which will be used as the map for the simulation. The survivor.py file is used to simulate the people escaping. Two main files are created namely main.py and main_refactored.py. The main.py file is created for the non-refactored algorithms as it is interchangeable. The main_refactored.py file, however, is exclusively created for the refactored algorithm.

Utilizing pygame modules allows for better visualisation as well as user-control for ease of manipulation. Certain keybinds on the keyboard will be used to use features such as saving a map, clearing a map, and loading a saved map. Each node is placed in a 50 by 50 grid consisting of squares. Different highlighted coloured nodes help differentiate node description to assist in visualising efficiency. The orange-coloured node represents the start node, the cyan-coloured node represents the end node, and the black-coloured nodes represent buildings and debris.

Since natural disasters happen at random and the damages occurred by these events are also random, naturally using the random module is a suitable idea to simulate the incoming debris. However, having different ordered obstacles appearing at every simulation instance could cause inconsistent results. Hence, the seventh file, obstacle_generator.py, is created. It still uses the random module, but this time, that randomly generated ordered obstacle will be used consistently throughout the simulation by storing it in a list.

VII. Results

To ensure consistency in analyzing the performance of the algorithms, several factors are made constant. This includes the grid size for the map, set to a 50 by 50 area, the duration of simulation, limited to only 3 minutes, the predetermined buildings on the grid, and the predetermined debris appearance. Figure 7.1 illustrates the map used for the simulation. The black nodes represent buildings, the orange node represent the start node, and the cyan node is the end node.

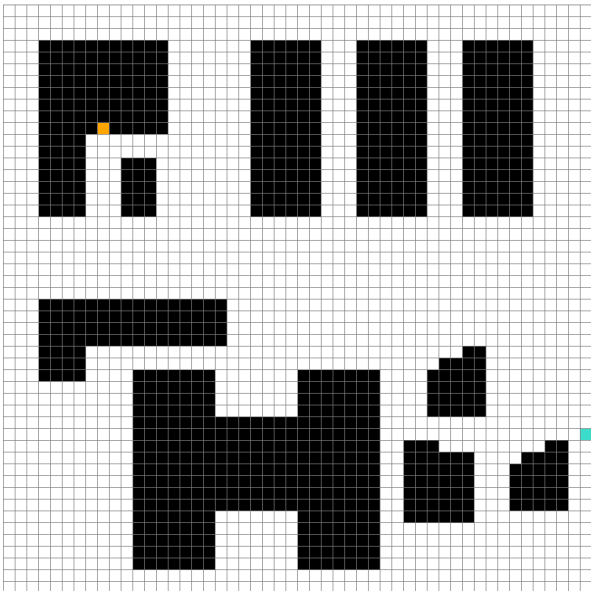

Figure 7.1. Map for simulation

The simulation lasts for 3 minutes and is run twice for each algorithm. Those values are used to calculate the average. Two factors are measured, the execution time as well as the peak memory of each algorithm. Each row is a successive iteration of an algorithm in finding the shortest path. An iteration is only generated if an obstacle is blocking the existing path.

a. Dijkstra's Algorithm

Execution Time / s			Peak Memory Usage / kB		
Simulation 1	Simulation 2	Average	Simulation 1	Simulation 2	Average
53.5315	53.3679	53.4497	253.23	253.23	253.23
51.6669	50.6207	51.1438	246.68	246.59	246.635
50.5253	50.6285	50.5769	246.38	246.38	246.38

Survivors: 1 (for each simulation)

For Dijkstra's algorithm, a total of 3 successive iterations was concluded. The execution time of all iterations is above 50 seconds with a gradual decrease as less nodes are required to be explored when more obstacles occupy the empty nodes. All the peak memory recorded for each iteration is approximately 250 kB. At the end of the simulation, a total of 1 survivor managed to escape the disaster.

b. A-star Algorithm

Execution Time (seconds)			Peak Memory Usage (KB)		
Simulation 1	Simulation 2	Average	Simulation 1	Simulation 2	Average
3.495	2.873	3.184	286.98	287.08	287.03
3.514	2.9441	3.22905	278.34	278.34	278.34
3.541	2.947	3.244	278.2	278.2	278.2
3.695	3.001	3.348	278.12	278.12	278.12
3.6182	3.0132	3.3157	278.03	278.03	278.03
3.774	3.0738	3.4239	277.94	273.99	275.965
3.7157	3.0403	3.378	277.86	273.91	275.885
4.39	3.573	3.9815	273.81	273.81	273.81
4.541	3.719	4.13	273.77	273.77	273.77
7.736	6.2552	6.9956	284.75	284.75	284.75
7.653	6.2961	6.97455	281.26	281.26	281.26
7.644	6.3121	6.97805	281.25	281.25	281.25
8.181	6.6779	7.42945	284.6	284.6	284.6
8.1988	6.8394	7.5191	284.62	284.62	284.62
8.6445	6.9719	7.8082	284.61	284.61	284.61
7.5382	6.319	6.9286	281.11	281.11	281.11
9.05	7.351	8.2005	281.02	281.02	281.02
9.207	7.5847	8.39585	281.04	281.04	281.04
16.1973	13.2024	14.69985	281.08	285.98	283.53
10.3199	8.654	9.48695	284.77	284.83	284.8

Survivors: 10 (for each simulation)

In the case of the A-star algorithm, it has 20 successive iterations. The execution time initially starts from 3 seconds and generally increases to 9 seconds. There are certain iterations where the execution time is higher than the expected

trends and the reason for that is solely from the unluckiness of the obstacle formation. The peak memory is approximately 280 kB. A total of 10 survivors escaped in these simulations

c. Refactored A-star Algorithm

Execution Time (seconds)			Peak Memory Usage (KB)		
Simulation 1	Simulation 2	Average	Simulation 1	Simulation 2	Average
3.065	2.794	2.9295	42.3	42.17	42.235
3.3772	2.815	3.0961	27.84	27.84	27.84
2.912	2.7122	2.8121	31.6	31.6	31.6
2.9472	2.6968	2.822	31.34	27.34	29.34
2.904	2.6734	2.7887	26.97	26.97	26.97
3.2891	3.0357	3.1624	31.77	31.77	31.77
3.301	3.0002	3.1506	31.69	31.69	31.69
2.176	2.0352	2.1056	25.37	25.37	25.37
3.3982	3.138	3.2681	27.31	27.31	27.31
2.653	2.4638	2.5584	29.54	29.54	29.54
3.358	3.1774	3.2677	27.06	27.06	27.06
3.3032	3.0122	3.1577	30.44	30.44	30.44
3.045	2.807	2.926	29.47	29.47	29.47
4.0142	3.806	3.9101	27.86	27.86	27.86
5.9784	5.5699	5.77415	27.33	27.33	27.33
1.821	1.8612	1.8411	12.34	12.34	12.34
7.23	7.9044	7.5672	45.88	45.88	45.88
3.4382	3.165	3.3016	13.93	13.93	13.93
7.9013	6.9452	7.42325	50.07	50.07	50.07
0.586	0.529	0.5575	6.17	6.17	6.17
7.8249	7.1283	7.4766	50.05	50.05	50.05
6.213	5.6814	5.9472	27.83	27.83	27.83
6.8994	6.3683	6.63385	45.81	45.81	45.81
8.1912	7.5009	7.84605	45.62	45.62	45.62
7.1421	6.6123	6.8772	45.48	45.48	45.48
10.225	8.7761	9.50055	52.59	53.29	52.94
7.7359	7.2511	7.4935	45.55	45.55	45.55
5.5972	5.0318	5.3145	24.55	24.55	24.55
6.6043	6.2372	6.42075	27.59	27.59	27.59

6.447	5.7241	6.08555	26.52	26.37	26.445

Survivors: 15 (for each simulation)

Since the A-star algorithm managed to rescue more survivors, more continuous path updates, and much faster searching, it has been chosen to be the algorithm to be improved to solve our problem. A total of 30 successive iterations were done with this algorithm. Some execution times are done quickly as the start node for the A-star algorithm is dynamically changed according to the survivor's latest position when an obstacle is blocking the path. For the peak memory usage, it has been drastically decreased with the highest record being 50.07 kB.

d. Comparison Graphs

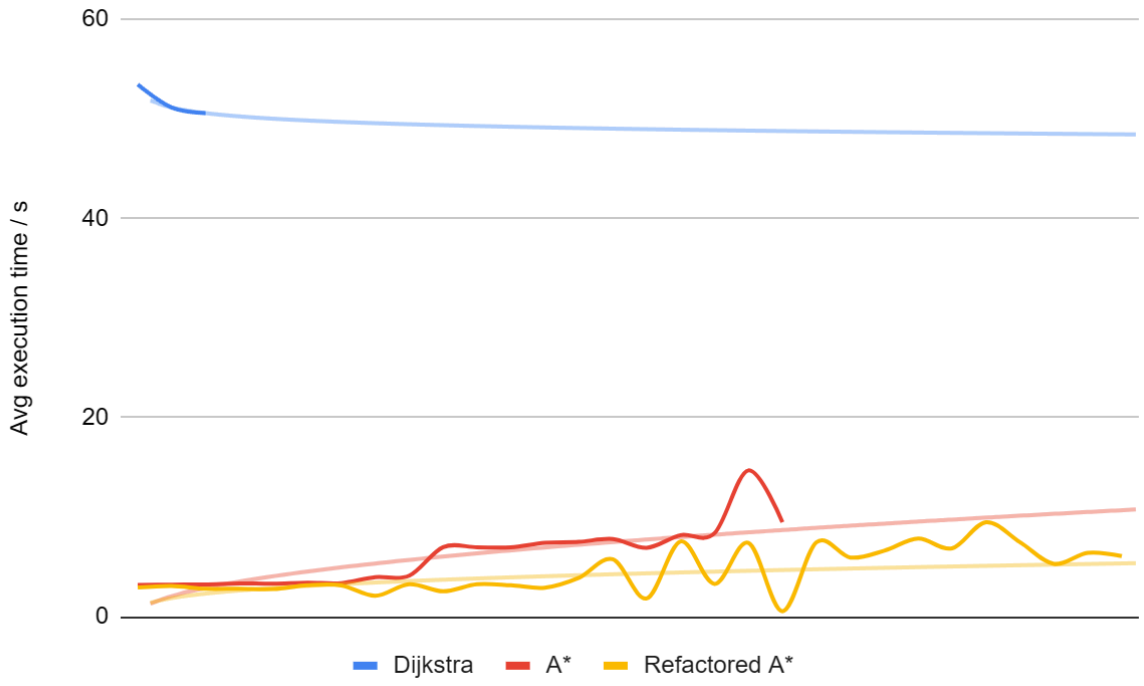

Figure 7.2. Average execution time for each algorithm.

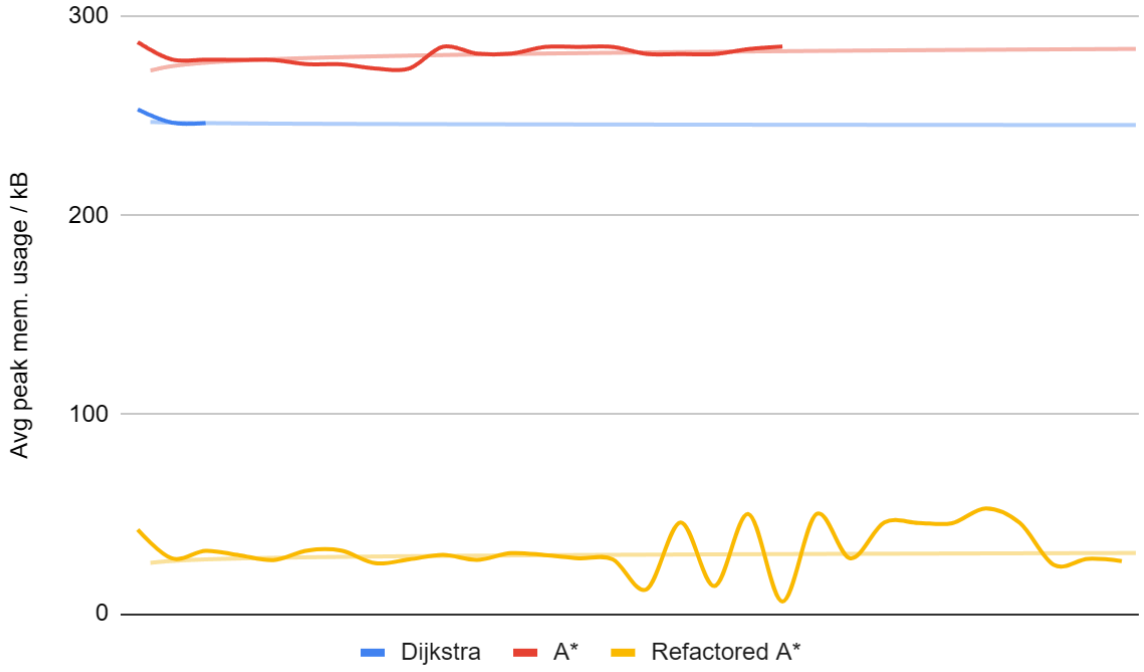

Figure 7.3. Average memory usage for each algorithm

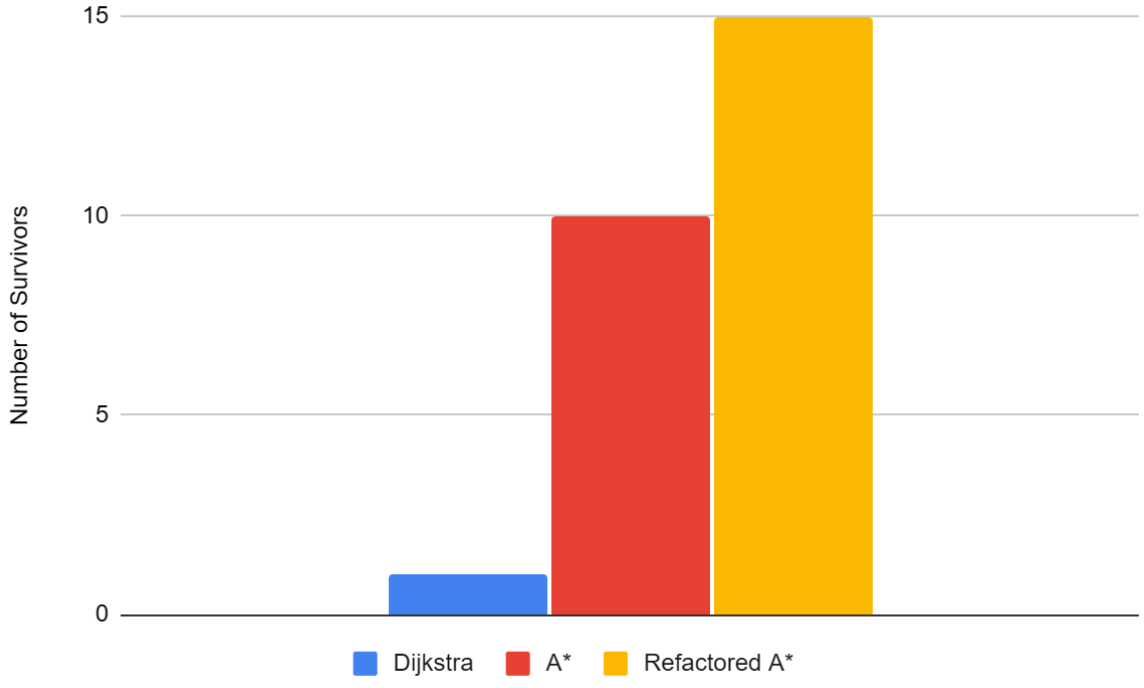

Figure 7.4. Survivor count for each algorithm in the simulation.

In the span of a 3 minute simulation, the execution time for dijkstra's algorithm is the longest with an approximate time of 50 seconds. The trend decreases

due to insufficient iterations however, theoretically it does decrease because the algorithm is already exploring most of the nodes and having these nodes decreased, less nodes are required to be explored. A* and its refactored version has way lower execution times with the refactored being slightly better. As in the case of memory usage, Dijkstra use less than A*, but the refactored one has been made to further improve memory usage. Lastly, the refactored A* algorithm managed to rescue the most survivors.

VIII. Discussion

a. Dijkstra & A-star Comparison

When comparing the results of Dijkstra's and A-star algorithm, there is an obvious difference in path-finding efficiency. In the span of 3 minutes, Dijkstra's algorithm managed to get 3 successive paths whereas the A-star algorithm managed to get 20 successive paths. This goes back to the theory of A-star algorithm where it utilises a heuristic function to calculate the total cost, allowing it to observe the solution in a bigger scope than Dijkstra's. Additionally, the A-star algorithm managed to guide 10 survivors towards the safe point, an additional 9 when compared to Dijkstra's algorithm.

However, the quick searching ability from the A-star algorithm comes with a price. The peak memory of each successive iteration is approximately 30 kB more than the successive iterations from Dijkstra's algorithm. This may seem small, but the map used for this simulation is 50 by 50. Theoretically, the peak memory would increase for a larger grid as the calculated approximate distance would also be further away and more nodes are required to be explored.

b. A-star & Refactored A-star Comparison

After careful analysis, we have concluded that the A-star is still the more suitable option when it comes to swift rescue. From there, we managed to further modify the code to enhance the efficiency of the algorithm based on our problem. Starting with the initial point of the algorithm, we figured that it is better if the starting point of the algorithm is based on the survivor's location when an obstacle is blocking the path. This would mean changing the position dynamically based on the

survivor position. Figure 8.1 shows the snippet that determines the position of the algorithm's starting point when searching for a new path. Figure 8.2 shows the refactored version of it

```
# Recalculate path if grid changes
if simulation_started and start and end and grid_changed:
    if original_start == None:
        original_start = start

    for row in grid:
        for node in row:
            node.update_neighbors(grid)
```

Figure 8.1. Initial start of algorithm before refactoring

```
# Recalculate path if grid changes
if simulation_started and start and end and grid_changed:
    if original_start == None:
        original_start = start
    if survivor:
        start = survivor.current_spot
    else:
        start = original_start

    for row in grid:
        for node in row:
            node.update_neighbors(grid)
```

Figure 8.2. Initial start of algorithm after refactoring

In the refactored version, it has been made so the start node for the algorithm is dynamically changed based on the survivor's current spot when an obstacle is blocking the path. This allows the algorithm to be executed in a shorter distance and in a quicker search. Additionally, the survivor is required to travel a lower distance each time the algorithm is run, allowing for more survivors to escape.

Next is applying a few modifications to the algorithm itself. This allows the algorithm to take less memory when executed. Figure 8.3 is a snippet of the original A-star algorithm. Figure 8.4 is the refactored version.

```
def a_star_algorithm(draw, grid, start, end):

    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}

    g_score = {node: float("inf") for row in grid for node in row}
    g_score[start] = 0

    f_score = {node: float("inf") for row in grid for node in row}
    f_score[start] = h(start.get_position(), end.get_position())

    open_set_hash = {start}
```

Figure 8.3. Original A-star algorithm

```
def refactored_a_star_algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}

    # Store g_score and f_score only for nodes in open_set
    g_score = {start: 0}
    f_score = {start: h(start.get_position(), end.get_position())}

    open_set_hash = {start}
```

Figure 8.4. Refactored A-star algorithm

The difference between the original A-star algorithm and the refactored version is the amount of nodes being initialised to a value. In the original version, the value of each node that exists on the grid is initialised to infinity. This is the tentative distance method of the algorithm. When a node is explored, a new value is set as it is guaranteed to be less than infinity.

In the refactored version, no nodes are initialised to infinity. This means that the algorithm is not required to go into every individual node and assign the value to infinity. This allows the algorithm to dynamically treat unknown nodes as infinity. The upfront cost for this part of the code when the program is launched is reduced from O(V) to O(1). This is especially efficient for maps on a bigger grid. However, the worst case scenario is still exploring all nodes, making the space complexity overall still O(V). Though, the difference is observable when calculating actual values of memory allocation.

IX. Conclusion and Recommendation

In conclusion, when conducting tests with Dijkstra's algorithm and the A-star algorithm, the A-star algorithm is deemed to be the more suitable algorithm for evacuations during natural disasters. The execution time is much smaller and more survivors are able to escape in the span of 3 minutes. Despite it requiring more space to run, the time taken for path guidance is prioritised in dire situations such as this. Then, a refactored version of the A-star algorithm by dynamically changing the algorithm's starting position and removing its initialization of tentative distances. This reduces both the time taken for execution as well as the memory used.

For recommendations, when testing these algorithms, it is better to run the simulation more than twice to get the average. Additionally, the testing phase is limited to only 1 map on 1 set area, a 50 by 50 grid. It is suggested, if time permits, to test these algorithms on multiple maps set on different grid areas, for instance, 10 by 10, 20 by 20, etc. With more conducted tests, the algorithms can further prove its efficiency when it comes to rescuing survivors during a natural disaster.

X. References

Blaikie, P., Cannon, T., Davis, I., & Wisner, B. (2014). *At risk: natural hazards, people's vulnerability and disasters*. Routledge.

Javaid, A. (2013). Understanding Dijkstra's algorithm. *Available at SSRN 2340905*.

Python 3. (n.d.). *time — Time access and conversions*. Python documentation. Retrieved January 14, 2025, from https://docs.python.org/3/library/time.html

Python 3. (n.d.). *tracemalloc — Trace memory allocations*. Python documentation. Retrieved January 8, 2025, from https://docs.python.org/3/library/tracemalloc.html

XI. Links

Github link to the code:
https://github.com/KevinJ0nathan/Pathfinding-Evacuation-Routes

Canva link to the presentation:
https://www.canva.com/design/DAGbndrpRw4/LCh0BD9fBpZe0ui9GMUZWg/edit?utm_content=DAGbndrpRw4&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

XII. Screenshots & User Manual

When conducting algorithm testing, there are several points to consider. First is the algorithm selection, then the controls of the visualisation, and lastly node representation. When it comes to the non-refactored algorithm selection, manual changes are required to be made in the main.py file. Figure 9.1 shows the part of a snippet code necessary to change the algorithms to either dijsktra_algorithm or a_star_algorithm. In figure 9.2, select main_refactored.py to execute the refactored algorithm, no changes are needed.

```
# Recalculate path if grid changes
if simulation_started and start and end and grid_changed:
    if original_start == None:
        original_start = start

    for row in grid:
        for node in row:
            node.update_neighbors(grid)

    start_time = time.time()
    tracemalloc.start()

    # Change algorithm function to either dijkstra_algorithm or a_star_algorithm
    path = dijkstra_algorithm(lambda: draw(window, grid, ROWS, width), grid, start, end)
    if path is None:
        no_path_exist = True
        print(str(survivors) + "survivors escaped")
    else:
        no_path_exist = False
```

Figure 9.1. Part of code to manually change algorithms.

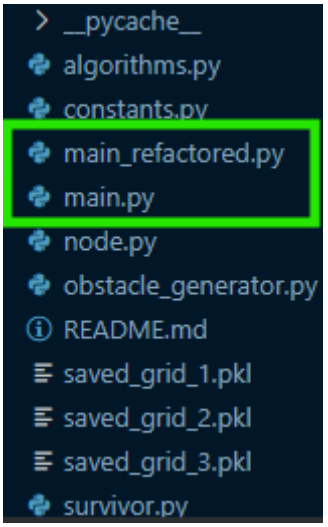

Figure 9.2. Files of program.

Next is the controls. Using pygame allows for mouse and keyboard inputs. The controls are as follows,

- Mouse Button 1 (left click): Adding Objects
- Mouse Button 3 (right click): Removing objects
- Spacebar: Starting the simulation
- C key: Clearing all nodes
- P key: Saving a grid for future use
- Number 1 to 9 keys: Loading the saved grids

Lastly, the node colour representation. Different colours are displayed on nodes to distinguish each node description. The colour representation are as follows,

- Red: Already explored nodes
- Green: Nodes being explored
- Cyan: End node
- Orange: Start node
- White: Empty node
- Black: Nodes with obstacle
- Purple: Nodes used for pathway

Some screenshots of the application were taken. Figures 9.3, 9.4, 9.5, and 9.6 illustrate some appearances of the application.

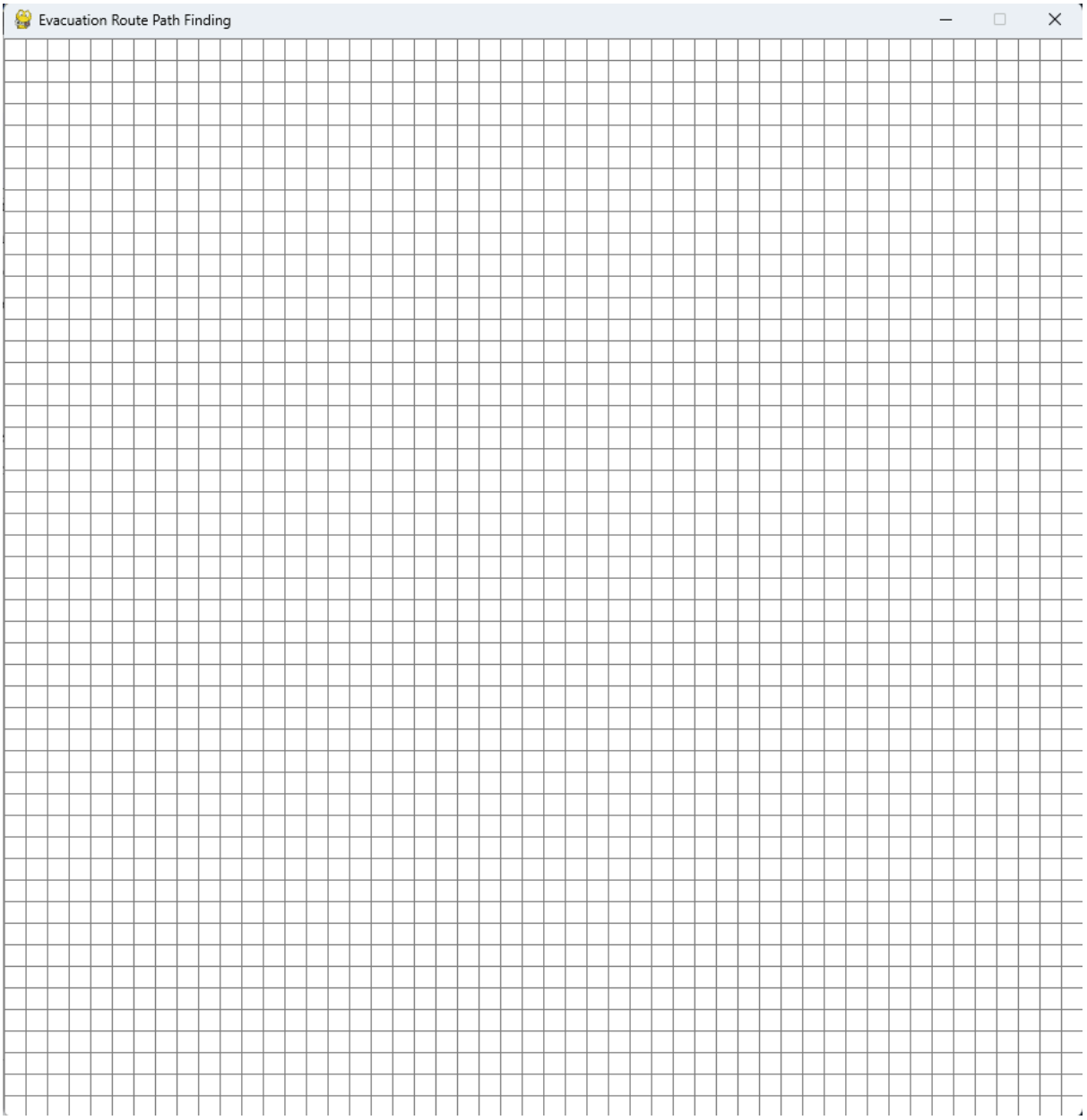

Figure 9.3. Application view after opening.

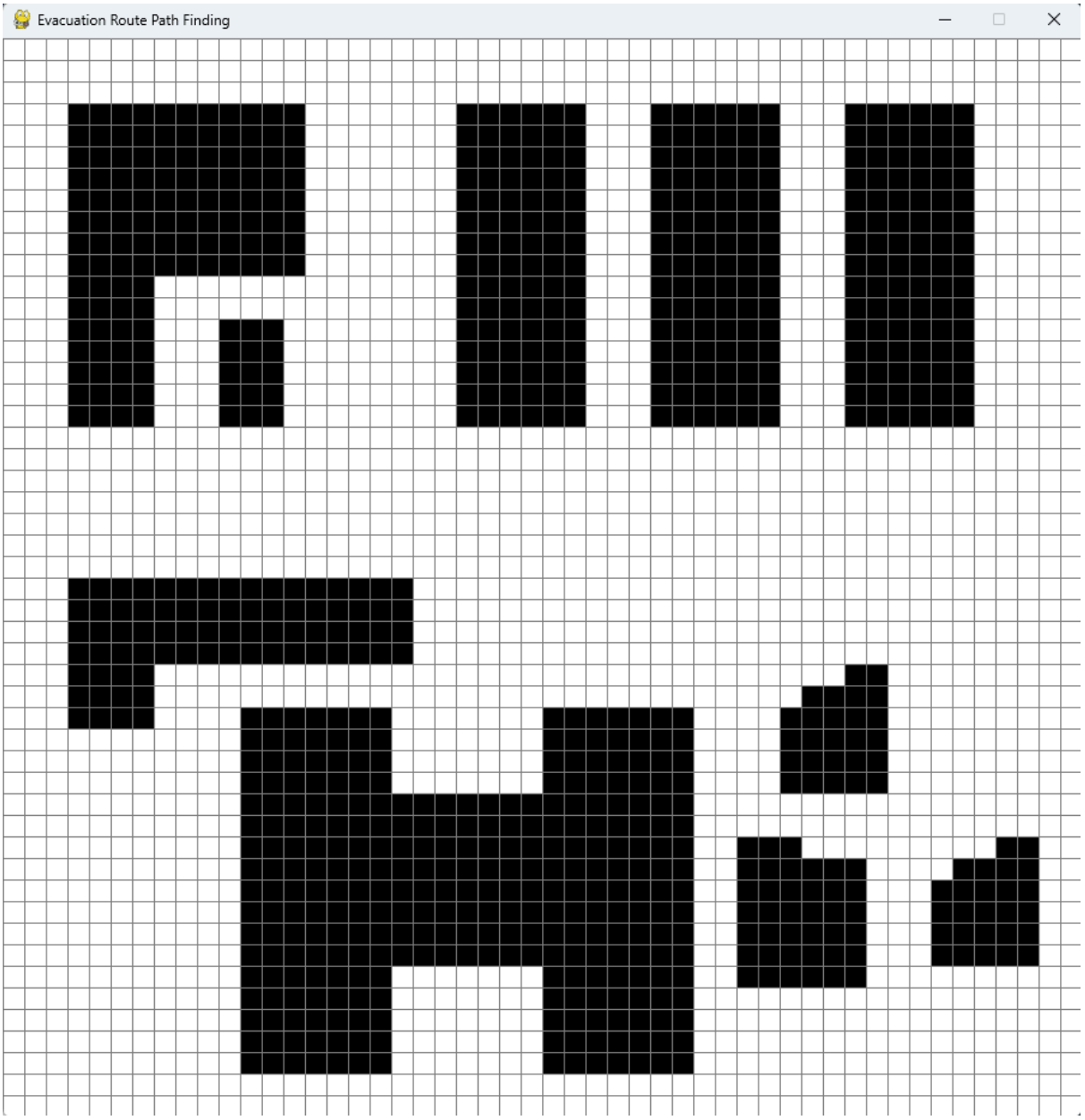

Figure 9.4. Using an existing map for the simulation.

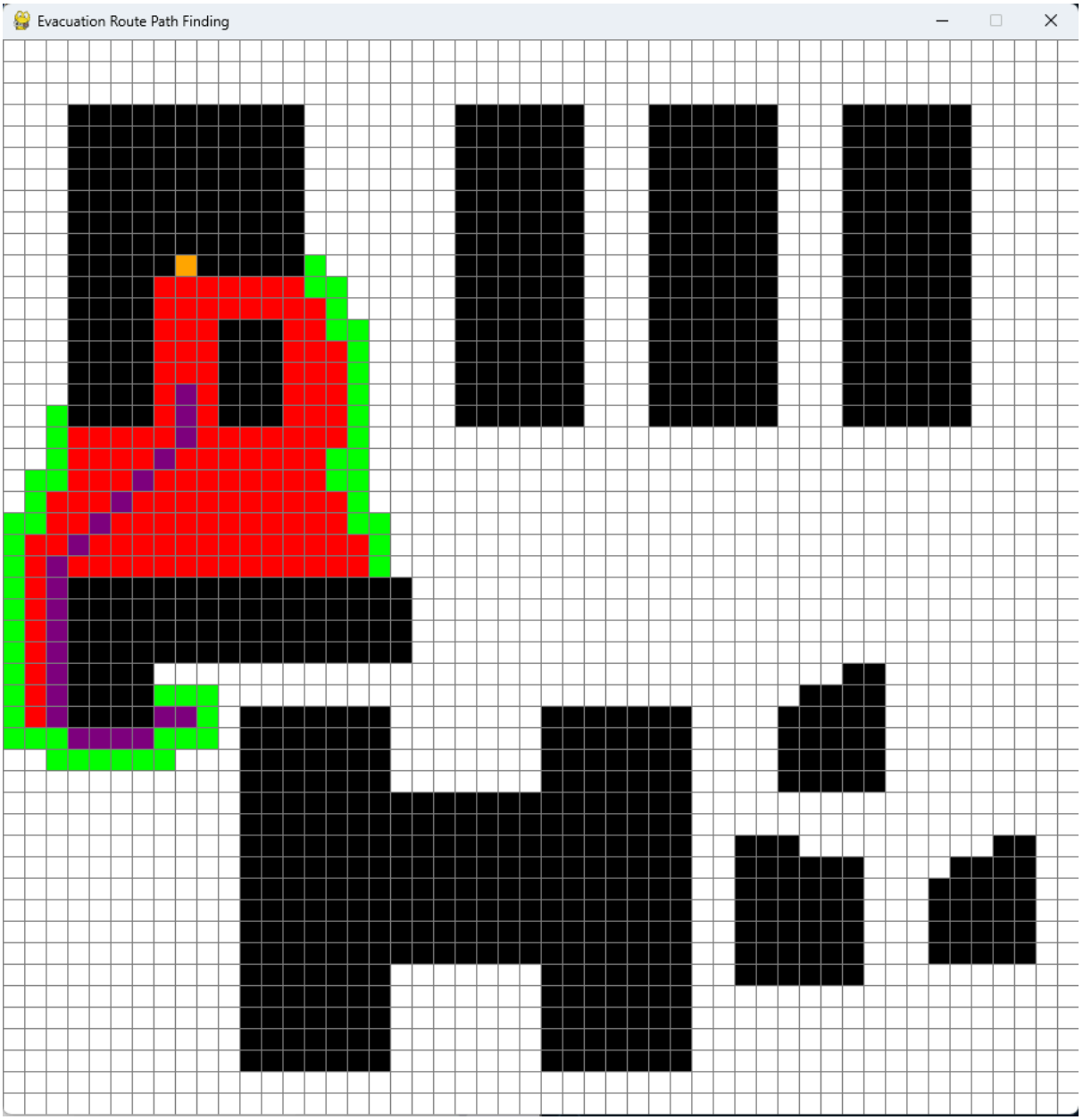

Figure 9.5. A-star algorithm in the process of tracing the shortest path.

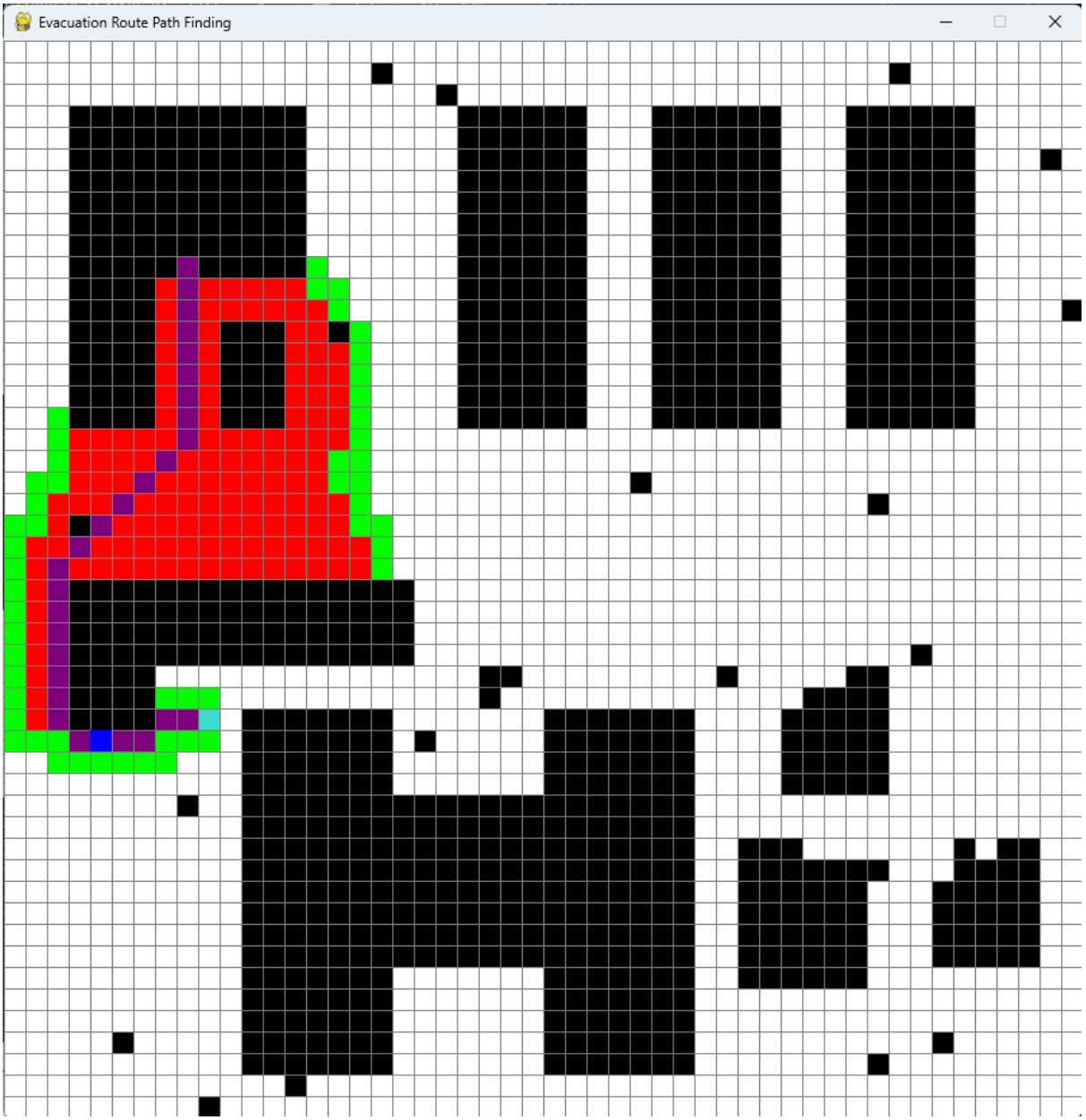

Figure 9.6. During simulation, predetermined obstacles spawn to simulate destruction.