Finding Shortest Escape Route for Evacuation in a Building Using *Breadth-First-Search Algorithm*

Ahmad Fahmi Pratama - 13516139
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
afahmi3@yahoo.co.id

***Abstract*—In evacuation planning in a building, finding shortest to the nearest exit is an important aspect. But sometimes the evacuees inside the building are still confused of choosing a path / route that they should follow to find nearest emergency exit, because of the unfamiliarity with the building. They tend to go to the main entrance which sometimes far enough from their position, making it dangerous. The objective of this paper is to find shortest escape route in a building for evacuation, if there are any incidents such as fire or earthquake. This paper presents a shortest path-based algorithm which finds nearest exit in a building, using Breadth-First-Search algorithm. The floor plan is treated as a graph, with any unit step of a person is represented as its nodes. Then, the algorithm traverses the graph to find a shortest way to the nearest exit from the person's position. There may be multiple emergency exits that are reachable, the evacuee can then choose any exits available that he/she wants to go through.**

***Keywords*—Shortest path, graph traversal, Breadth-First-Search algorithm, evacuation planning**

I. Introduction

While designing a building layout, making effective and efficient evacuation planning is a must. It must be easy for the visitors in the building to get out of the building as quickly as possible whenever a disaster occurs using the safe way that the layout given. However, sometimes finding a safe way to escape or finding emergency exits in a building is not really easy if the building is quite complex and huge, or that many emergency exits are blocked with some obstacles caused by the disasters, trapping evacuees inside the building.

Disasters can occur at any time, also when visitors are still inside a building. Every visitor inside the building should know where are the location of emergency exits there, supposedly. But in contrary, knowing emergency exits while they are not familiar enough of the building is difficult. Most of the time, they do not know which is the shortest and safest path they should follow in order to reach the exit because of the unfamiliarity. A survey for investigating exit choice decision in a Chinese supermarket by [1] shows that 48.6% of the respondent will use the nearest the emergency exit, but a considerable amount of people (20.9% of the respondent) will return to the main entrance. There are still many people who choose main entrance as their exit because it was the first thing that appears when thinking about evacuation route, since finding emergency exits is not that easy.

Making a decision in a critical time such as in a disaster often leads to bad decision, such as choosing a path that they should follow to go outside a building. Unfortunately, there has no time to make a good decision. Every decision has to be made quickly to make the evacuees as safe as possible. It also can reduce the number of the injuries because of the hazard. To help them in making a good decision, the author will provide a way to find a shortest evacuation route in a disaster using Breadth-First-Search algorithm.

II. Basic Graph Theories

A. Graph

A graph $G = (V, E)$ consists of a nonempty set of *vertices* / *nodes* called V and a nonempty set of *edges* called E, which each of edges has either one or two vertices associated with it, called its *endpoints* [2].

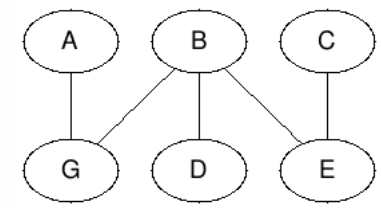


***Figure 1.** A graph that consists of 6 vertices and 5 edges. Source: author's document.*

B. Terminology

In graph theory, there are some terminologies that are important to describe anything related to graphs, such as behavior, properties, etc. Here are the lists of important terms that will be used.

1. Adjacent
   Two vertices u and v of a graph G are called *adjacent* / *neighbors* if both of the vertices are connected each other by an edge e.

2. Incident
   An edge e is called *incident with* the vertices u and v if it connects both of the two vertices.

3. Degree
   Degree of a vertex u in a non-directed graph G is the sum

of edges that are incident with that vertex.

4. Path
   A *path* in a graph G is a set of vertices that consists of any start vertex until the final vertex, creating a path.

5. Cycle / Circuit
   A *cycle* is path that the start and final vertices are the same vertex, creating a loop.

6. Isolated Vertex
   An *isolated vertex* is a vertex that has no edges that coincidence with it.

7. Subgraph
   A graph $H = (W, F)$ is called a *subgraph* of a graph $G = (V, E)$, where $F \subseteq E$ and $W \subseteq V$. A subgraph H is a *proper subgraph of* G if $H \neq G$.

8. Weighted Graph
   A *weighted graph* is a graph that has values (weight) on its edges. The values can represent relation between any two connected vertices, such as distance, time, production cost, etc.

C. Types of Graphs

Graphs can be classified into some categories based on their properties. Mainly, graphs can be classified into the types of graphs depending upon the number of edges, the number of vertices, interconnectivity, and their overall structure.

Based on the edges direction, graphs can be divided into two types, directed graph and non-directed graph.

1. Directed Graph
   A *directed graph* is a graph that contains edges that have directions to which vertex they are heading to.

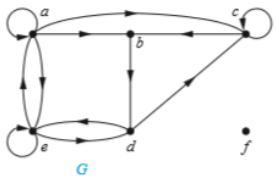


Figure 2. *An example of directed graph [2]*

2. Non-Directed Graph
   A *non-directed graph* is graph such the edges have no direction to any vertices.

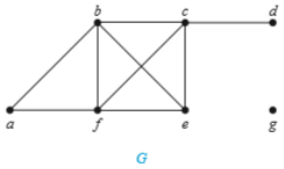


Figure 3. *An example of non-directed graph [2]*

Based on the edges types, graphs can be classified into three categories listed below:

1. Simple Graph
   A *simple graph* is a graph that has no parallel edges and has no loops.

2. Multigraph
   A *multigraph* is a graph that may has multiple edges connecting the same pair of vertices.

3. Pseudograph
   A *pseudograph* is a graph that may has multiple edges connecting the same pair of vertices as well as loops.

D. Some Special Simple Graphs

There are some special, famous, or widely known types of graphs that are often to be found. Here are the lists of graphs.

1. Complete Graph
   A *complete graph on n vertices*, denoted by K_n, is a simple graph that contains exactly one edge connecting every pair of distinct vertices. In other words, a complete graph K_n with n vertices has exactly $n-1$ edges.

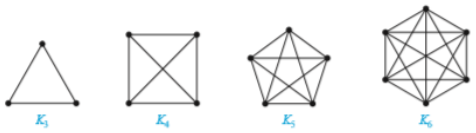


Figure 4. *Some examples of complete graph[2]*

2. Cycle Graph
   A *cycle graph* (or *cycle* only), denoted by C_n, $n \geq 3$, is a simple graph that consists of n vertices $v_1, v_2, \ldots, v_n$ and edges that connect between every pair of $\{v_{n-1}, v_n\}$.

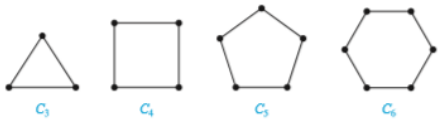


Figure 5. *Some examples of cycle graph [2]*

3. Wheels
   A *wheel*, denoted by W_n, actually a cycle graph C_n that is given additional vertex, and connect this new vertex to each vertex in the cycle.

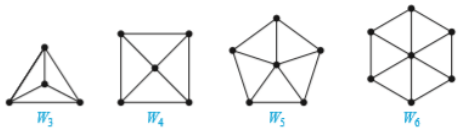


Figure 6. *Some examples of wheel graph [2]*

III. Graph Traversal

A. Definition

Graph traversal is a technique used for searching a vertex in a graph by traversing all of the vertices in a graph. This method obviously required considerable amount of computation time, since graph traversal may require some vertices to be visited more than once. Revisiting vertices is needed as sometimes it is not necessarily known whether a vertex has already been explored or not. As the graph becomes denser, this redundancy

becomes more often, thus increasing its computation time.

It is usually necessary to keep track of which vertices that has been visited before in the algorithm, so that the algorithm will not visiting the same vertices repeatedly, leading to infinite loop. This may be accomplished in some methods, such as associating each vertex of the graphs with some values or "colors" that distinguish them between the unvisited ones, indicating that certain vertex has been visited before.

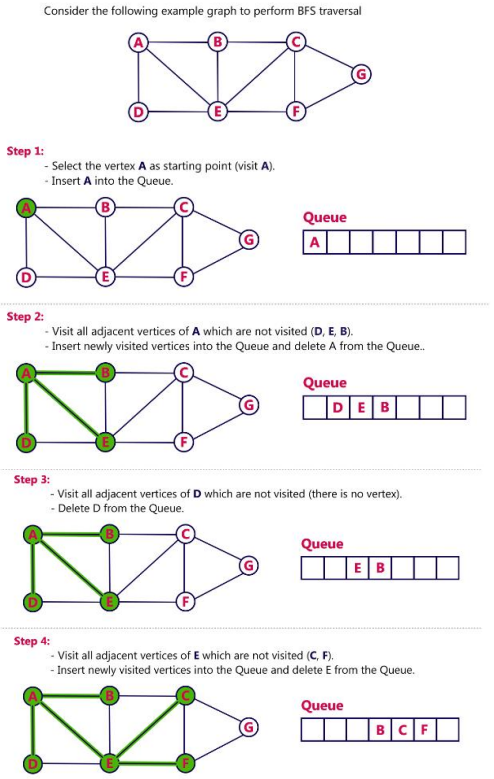
There are two famous graph traversal algorithms that can be used to achieve the goals, the two are Depth-First-Search (abbreviated as DFS) algorithm and Breadth-First-Search (abbreviated as BFS) algorithm. In this paper, BFS algorithm will be used later.

B. Breadth-First-Search (BFS)

Breadth-First-Search (abbreviated as BFS) is a kind of graph traversal algorithm. Give a graph *G = (V,E)* and a distinguished source vertex *s*, BFS explores every edges of *G* to discover every vertices that are accessible from *s*. In other words, BFS will visit vertices that are direct neighbors of *s* (first layer). Then, it will visit the neighbors of the direct neighbors that were visited before (second layer), and so on.

BFS algorithm mainly uses *queue* data structure in this application. It first starts by inserting the source vertex *s* to the queue, then process the queue as follows:

1. Take the front most vertex *u* from the queue
2. Insert all the unvisited neighbors of *u* to the queue, then mark them as visited
3. Repeat step 1 and 2 until all of the vertices had already been visited



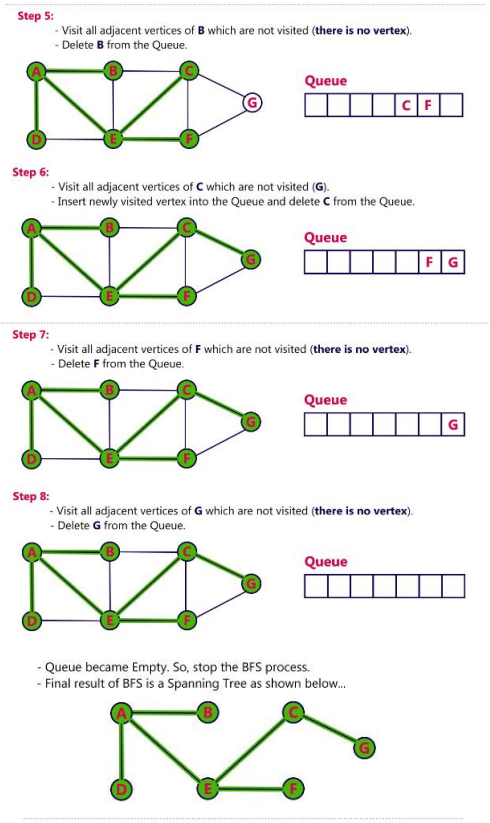


Figure 7. *Graphical simulation of BFS algorithm [4]*

The algorithm will visit vertex *s* and all vertices in the connected component that contains *s* layer by layer. BFS algorithm complexity is *O(V+E)* where *V* is the number of vertices and *E* is the number of edges in the graph. Below is the pseudo-code of BFS algorithm.

```
BFS(G,s) {
  initialize vertices;
  Q = {s};
  while (Q not empty) {
    u = Dequeue(Q);
    for each v adjacent to u do {
      if (color[v] == WHITE) {
      color[v] = GRAY;
      d[v] = d[u]+1; // compute d[]
      p[v] = u;  // build BFS tree
      Enqueue(Q,v);
    }
  }
  color[u] = BLACK;
}
```

BFS algorithm implementation[5].

IV. Implementation of BFS Algorithm on Finding Shortest Escape Route

A. Building Layout

BFS algorithm obviously needs graph to be traversed. So, the building layout that will be used for this implementation needs to be converted to a graph. The graph that will be used is in the form of grid map with the size of *R* x *C*, where *R* is the row size of the grid, indicating building's length, and *C* is the column size of the grid, indicating building's width.

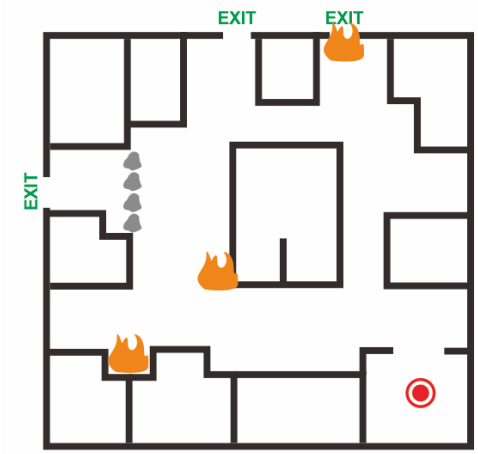


Figure 8. *Sample layout. Source: author's document*

The picture above represents a layout building with some emergency exits blocked with rocks or fires. The red dot indicating the evacuee that wants to escape from the building as quickly as possible, before the fire spreads to another emergency exit and the evacuee cannot escape.

This building layout is just a simple example. In reality, real building layout is much more complex that this example, thus needs more memory space and more time consuming.

B. BFS Algorithm

The author uses C++ language for implementing BFS algorithm. This implementation requires the map to be converted into grid with the following conditions:

1. Character "#" indicating obstacles, such as rocks, fires, walls, etc.
2. Character "0" indicating path that can be passed through.
3. Character "S" is the start vertex, or the position of the evacuee.
4. Character "F" indicating the emergency exits or any exits available in the map.

Using the following rules above, the map on figure 3 before is converted into this grid:

#	#	#	#	#	#	#	F	#	#	#	F	#	#	#	#	#
#	0	0	#	0	#	0	0	#	0	#	#	#	#	0	0	#
#	0	0	#	0	#	0	0	#	0	#	0	0	#	0	0	#
#	0	0	#	0	#	0	0	#	#	#	0	0	#	#	0	#
#	0	0	#	#	#	0	0	0	0	0	0	0	0	#	0	#
#	#	#	#	0	0	0	#	#	#	#	#	0	0	#	#	#
#	0	0	#	0	0	0	#	0	0	0	#	0	0	0	0	#
F	0	0	#	0	0	0	#	0	0	0	#	0	0	0	0	#
#	#	#	#	0	0	0	#	0	0	0	#	0	#	#	#	#
#	0	#	#	0	0	0	#	0	#	0	#	0	#	0	0	#
#	0	0	#	0	0	#	#	0	#	0	#	0	#	0	0	#
#	#	#	#	0	0	#	#	#	#	#	#	0	#	#	#	#
#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	#
#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	#
#	#	#	0	#	#	#	0	0	0	0	0	#	#	0	#	#
#	0	#	#	#	0	#	#	#	#	#	#	#	0	0	0	#
#	0	0	#	0	0	0	#	0	0	0	0	#	0	S	0	#
#	0	0	#	0	0	0	#	0	0	0	0	#	0	0	0	#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Figure 9. *Converted map, viewed in Microsoft Excel. Source: author's document*

The grid itself is actually a graph, with every character are its vertices. Character "0" denoting path that can be passed through by the evacuee, and every adjacent "0" is connected by each other, thus making them neighbors. Every character "0" is connected only with eight other "0" around it, in the position of eight wind directions.

The evacuee can go through the path by traversing the edges of any two "0" vertices from the start position (start vertex, denoted by character "S").

Then, the implementation of the BFS algorithm is shown as follows.

```
#include <bits/stdc++.h>

#define fi first
#define se second
using namespace std;

typedef pair<int,int>node;

int main(){
  queue<node>q;
  node start,finish,now,next;
  #define sfi start.fi
  #define sse start.se
  int r,c,a,b,step[120][120];
  int dr[8] = {-1,0,1,0,1,1,-1,-1};
  int dc[8] = {0,-1,0,1,1,-1,1,-1};
  char grid[120][120];
  bool visited[120][120];

  cin>>r>>c; //input map size
  //input grid
  for (int i=0;i<r;i++)cin>>grid[i];

  for (int i=0;i<r;i++){
    for (int j=0;j<c;j++){
      if (grid[i][j]=='S'){ //start vertex
        sfi = i;
        sse = j;
      }
    }
  }

  //start the BFS
  q.push(make_pair(sfi,sse)); //enqueue start vertex
  visited[sfi][sse] = true; //start vertex visited
  step[sfi][sse] = 0; //step masih 0

  while (!q.empty()){

    now = q.front();
    q.pop();

    /* search neighbors in every directions */
    for (int i=0;i<8;i++){
      next.fi = now.fi + dr[i]; //x-position
      next.se = now.se + dc[i]; //y-position

      /* check if not traversing exceed the boundaries
*/
      if (next.fi>=0 && next.fi<=r && next.se>=0 &&
next.se<=c){
        /* check if vertex can be visited (not an
obstacle) */
        if (grid[next.fi][next.se]!='#'){
          /* check if it hasn't been visited yet */
          if (!visited[next.fi][next.se]){
            /* mark as visited */
            visited[next.fi][next.se] = true;
            step[next.fi][next.se] =
step[now.fi][now.se]+1;
            q.push(make_pair(next.fi,next.se));
          }
        }
      }
    }
  }
```

```
  /* output the resulting grid */
  FILE *output;
  output = fopen("output.csv", "w+");
  cout<<"\n";
  for (int i=0;i<r;i++){
    for (int j=0;j<c;j++){
      if(grid[i][j] == '#') fprintf(output, "%c",
grid[i][j]);
      else fprintf(output, "%d", step[i][j]);
      if(j<c) fprintf(output, ",");
    }
    fprintf(output, "\n");
  }
  fclose(output);

  return 0;
}
```

The algorithm works by traversing all reachable vertices that can be visited from the start position. First, the position of the start vertex is inserted to the queue, or can be said, the start vertex is enqueued and marked as visited. Then, the algorithm will check if there are any vertices that had not been visited before. If yes, it will check every neighbor of the current processed vertex, whether they can be visited or not. If a new vertex can be visited, it will enqueue the vertex for the next calculation. If there is no neighbor that can be visited from the current processed vertex, the loop will stop and the algorithm will proceed to the next vertex that is available in the queue.

While the queue is not empty, it will pop out the front most element of the queue, or known as dequeue, to be the next vertex that will be processed. Then it will again check every neighbor that can be traversed, and so on. It will keep track of how many steps that are needed to visit a certain vertex in a two-dimensional array (matrix) representing the grid map. The process will be repeated until all of the vertices has been visited by the algorithm.

If the queue is empty, then the traversing process is finished. The next step is showing the matrix that contains steps that are needed for every vertex visited. Then, it will be found how many steps that are needed to reach a certain exit. The exit that can be reached with minimal steps is the shortest one. In some cases, maybe the are more than one nearest exits since the position of the evacuee (the start vertex) could be anywhere.

C. Result

With the following codes and the converted grid before, the resulting grid is generated. The output is the same grid as the input grid but the character "0" is replaced with how many steps it takes to reach certain vertex "0" from the start vertex. Using the algorithm, the output is then saved into .csv file format to make it easier to analyze the resulting grid.

Below is the picture of the program's main interface. It first receives an integer *R* and *C*, both representing the sum of rows and columns of the grid, respectively. Then, it receives an input grid that wanted to be analyzed.

```
D:\HMIF\Tugas\Matdis\Paper>bfs
19 17
#######F###F#####
#00#0#00#0####00#
#00#0#00#0#00#00#
#00#0#00###00##0#
#00####00000000#0#
#####000######00##
#00#000#000#0000#
F00#000#000#0000#
#####000#000#0###
#0##000#0#0#0#00#
#00#00##0#0#0#00#
#####00######0###
#000000000000000#
#000000000000000#
###0###00000##0##
#0####0#######000#
#00#000#0000#0S0#
#00#000#0000#000#
#################
```

Figure 10. *The program is receiving an input, viewed in Windows command prompt. Source: author's document*

The resulting grid is shown below. The author colorized it to make it easier to distinguish between every element that exist in the grid.

#	#	#	#	#	#	#	19	#	#	#	0	#	#	#	#	#
#	0	0	#	0	#	18	18	#	0	#	#	#	#	0	0	#
#	0	0	#	0	#	17	17	#	0	#	14	14	#	0	0	#
#	0	0	#	0	#	17	16	#	#	#	13	13	#	#	0	#
#	0	0	#	#	#	17	16	15	14	13	12	12	12	#	0	#
#	#	#	#	17	17	17	#	#	#	#	#	11	11	#	#	#
#	0	0	#	16	16	16	#	0	0	0	#	10	10	10	11	#
0	0	0	#	15	15	15	#	0	0	0	#	9	9	10	11	#
#	#	#	#	14	14	14	#	0	0	0	#	8	#	#	#	#
#	0	#	#	13	13	13	#	0	#	0	#	7	#	0	0	#
#	0	0	#	12	12	#	#	0	#	0	#	6	#	0	0	#
#	#	#	#	12	11	#	#	#	#	#	#	5	#	#	#	#
#	15	14	13	12	11	10	9	8	7	6	5	4	4	4	4	#
#	15	14	13	12	11	10	9	8	7	6	5	4	3	3	3	#
#	#	#	13	#	#	#	9	8	7	6	5	#	#	2	#	#
#	0	#	#	#	0	#	#	#	#	#	#	#	1	1	1	#
#	0	0	#	0	0	0	#	0	0	0	0	#	1	0	1	#
#	0	0	#	0	0	0	#	0	0	0	0	#	1	1	1	#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Figure 11. *The resulting grid, viewed in Microsoft Excel. Source: author's document*

From the resulting grid above, the number of steps taken to reach certain exits is shown. In the sample case, there is only one accessible emergency exit, while the rest two is blocked by some obstacles, whether it is fire or rocks. It may have multiple paths to reach the exit from the start position, but the algorithm shows only the minimum steps that are needed. From the grid above, 19 steps are needed to reach the exit from the start position (denoted as the character "0" in a red cell). The unreachable vertices are marked as zero steps by the algorithm, because it is not possible to walk to them from the start position, because of blocked by the obstacles. The shortest path is shown as follows:

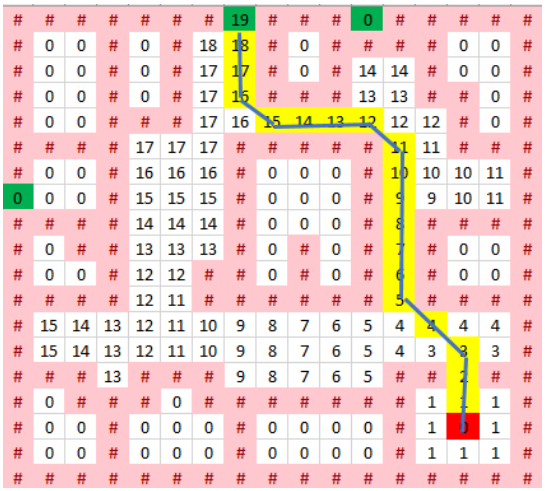


Figure 12. *The shortest path, viewed in Microsoft Excel.*
Source: author's document

Using the shortest path the algorithm has generated before, it only needs 19 steps taken from the start vertex. It also efficiently generates diagonal paths, makes the path shorter. This algorithm also can generate multiple paths leading to different exits, if there are more than one emergency exits available in the map.

V. Conclusion

In conclusion, it is possible to find shortest evacuation route using Breadth-First-Search algorithm. Whether the objective is to find one or more paths leading to multiple exits, this algorithm is useful as well. However, it is necessary to have the building layout first before using this algorithm, because this algorithm generates shortest path output based on the grid that has been given before.

VI. Acknowledgment

The first and the foremost thanks from the author is to our God, Allah (swt) for giving inspiration and chance to be able to create and finish this paper successfully. Special acknowledgement belongs to Dr. Ir. Rinaldi, MT. as the lecturer of the author's IF2120 Discrete Mathematics class, for guidance and every knowledge given so that the author can understand about computer science deeper and also for preparing this paper. Last but not least, the author thanks deeply to the author's parents for

References

[1] E.-W. Augustijn-Beckers, J. Flacke, and B. Retsios, *Investigating the Effect of Different Pre-Evacuation Behavior and Exit Choice Strategies Using Agent-Based Modeling*. Procedia Engineering, vol. 3, pp. 23–35, Jan. 2010.

[2] K.H. Rosen, *Discrete Mathematics and its Applications*, 7th ed. New York: McGraw-Hill, 2012, pp. 641-802.

[3] S. Halim and F. Halim, *Competitive programming 3: The New Lower Bound of Programming Contests*. S.l.: Lulu.com, 2013.

[4] Btech Smart Class. "Graph Traversals – BFS". Internet: http://btechsmartclass.com/DS/U3_T11.html, [Dec 2, 2017].

[5] A.S. Arifin, *Art of Programming Contest*. [On-line]. Bangladesh: Gyankosh Prokashoni. [Dec 2, 2017].

[6] N.A.M. Sabri et al. (2014) "Simulation Method of Shortest and Safest Path Algorithm for Evacuation in High Rise Building". *Applied Mathematical Science*. [On-line]. 8, pp 2-3. Available: http://www.m-hikari.com/ams/ams-2014/ams-101-104-2014/45384.html [Dec 1, 2017].

Pernyataan

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

Bandung, 3 Desember 2017

Ahmad Fahmi Pratama - 13516139