Graph Searching Implementation in Game Programming Cases Using BFS and DFS Algorithms

Faiz Ilham Muhammad 13511080
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
faizilham@itb.ac.id

***Abstract*—Graphs are heavily used in video games; hence, it is not surprising that graph searching become an essential topic in game programming. This paper will show the implementation of the most basic graph searching algorithms, the Depth-First Search (DFS) and Breadth-First Search (BFS), in some game programming cases: minesweeper, turn-based tactics, and maze games.**

***Index Terms*—Graph Searching, Depth-First Search, Breadth-First Search, Game Programming.**

I. Introduction

Graphs, one of prime subject in discrete mathematics, have many applications and implementations in computer science world [1]. Graphs can be used to model relation, network, position, adjacency and many else. In game programming, graphs are extensively used, for example to model tiles, object position and image representation. Since video games heavily rely on graphs, graph search algorithms are necessarily needed. There are numerous graph search algorithms; the most basic ones are DFS (Depth-First Search) and BFS (Breadth-First Search) algorithm [2]. This paper will present how to implement DFS and BFS algorithm in some game programming cases: opening empty cells in minesweeper, creating turn based tactics' character movement area, calculating area damage, and solving and generating maze in maze games.

II. Theories and Terminologies

2.1. Graph Theory

[3] In discrete mathematics, graphs are collections of discrete objects and their relations. Graph can be visually represented by symbolizing its objects as vertexes / nodes and relations as edges / lines.

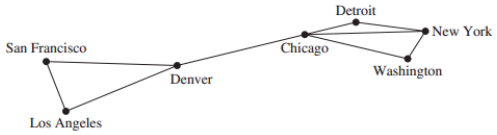

Figure 2.1. A Graph Example [4]

Mathematically, graph is defined as a pair of two sets V and E where:

V: a non-empty set of vertex/nodes = $\{v_1, v_2, \ldots, v_n\}$
E: a set of edges/lines connecting pairs of nodes = $\{e_1, e_2, \ldots, e_n\}$
In short, G = (V, E).
A directed graph is a graph which its edges are given directions. In mathematical definition, directed edge is a set of ordered pairs of vertexes, or

$$E = \{(v_a, v_b)\};\ v_a, v_b \in V.$$

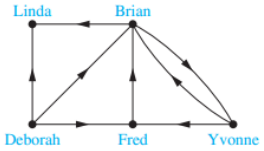

Figure 2.2. A Directed Graph Example [4]

Some terminology in graph theory:

1. Adjacent
   A pair of vertexes in undirected graph G is adjacent if there is an edge connecting both vertexes. All adjacent vertexes of vertex *u* are called neighbors of *u*.
2. Incident
   An edge *e* in undirected graph G is incident with vertex *u* and *v* if *e* connects *u* and *v*.
3. Isolated Vertex
   A vertex *v* is isolated if there is no edge in graph that incident with *v*. Isolated vertex can also be defined as a vertex which is not adjacent to any other vertexes in the graph.
4. Degree
   Degree of a vertex *v* in undirected graph G is the number of vertex adjacent to *v*.
5. Path
   A path of length n from vertex v_0 to v_n in graph G is a sequence of n edges $e_1, e_2, \ldots, e_n$ so that $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, $e_n = (v_{n-1}, v_n)$ are edges in graph G. In other words, a path is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph [4].
6. Circuit or Cycle
   A circuit or cycle is a path that starts and ends at the same vertex.

7. Connected
   An undirected graph G is called connected graph if for each pair of vertexes *u* and *v* in graph G there is a path connecting *u* and *v*.

2.2 Stack and Queue Data Structure

2.2.1. Stack

[5] A stack is a container of data that are added or deleted with last-in-first-out (LIFO) principle, which means an element can only be added to the top of the stack, and can only be deleted if it is the top-most element in the stack. Because of this characteristic, only the top of the stack that can be accessed; the others can only be accessed if the top one is removed. Stack can also be defined in recursive way:

1. A stack is empty, or
2. A top element and a stack below the top element.

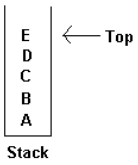

Figure 2.3 Visual Representation of Stack

There are four primitive functions for stack [2][5]:

1. Push
   Adds an element on the top of the stack
2. Pop
   Removes the top-most element of the stack
3. Top
   Returns the top-most element of the stack
4. Empty
   Checks whether the stack is empty or not

Stack can be used for implementing recursion, backtracking, and evaluating arithmetic expressions [6]. In graph searching, stacks are used in depth-first search [2].

2.2.2 Queue

[5] A queue is a container of data that are added or deleted with first-in-first-out (FIFO) principle, which means an element can only be added to the rear of the queue, and can only be deleted if it is the front-most element in the queue.

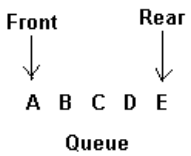

Figure 2.4. Visual Representation of Queue

There are four primitives function for queue [2][5]:

1. Enqueue
   Adds an element to the rear of the queue
2. Dequeue
   Removes the front-most element of the queue
3. Front
   Returns the front-most element of the queue
4. Empty
   Checks whether the queue is empty or not

Queues can be used for implementing operating system task [6]. In graph searching, queues are used in breadth-first search [2].

2.3 DFS and BFS Algorithm

2.3.1. Depth-First Search (DFS) Algorithm

[2] Depth-First Search algorithm works by pushing all adjacent nodes to a stack, and then processed by popping them. In other words, the process will visit the deepest node from a branch before visit the other branch. Since DFS uses stacks, it can simulate backtracking and can be defined in recursive way. DFS node-visiting process can be visualized as in Figure 2.5

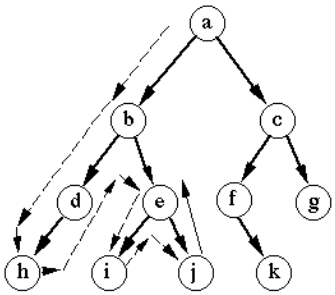

Figure 2.5. Depth-First Search Node-Visiting Process

Suppose the process starts from node a. Then, it will respectively visit b, d, h, e, i, j, c, f, k, and g.

DFS can be defined as follows:

```
create empty stack S
u ← starting node
push u to S
mark u as visited
while S is not empty
    top ← pop(S)
    if top fulfills the search condition
       //some expression
       //stop function
    for each v neighbor of u
       if v is unvisited
           mark v as visited
           push v to S
```

DFS can also be defined recursively as follows:

```
DFS(u):
   if u fulfills the search condition
       //some expression
       //stop function
   mark u as visited
   for each v neighbor of u
      if v is unvisited
        DFS(v)
   mark u as unvisited
```

2.3.2 Breadth-First Search (BFS) Algorithm

[2] Breadth-First Search (BFS) algorithm works by visiting a node, enqueue all adjacent nodes to a queue, and process them by dequeueing them. In other words, the process will visit every node in the same depth; let's say

depth n, before visit nodes in depth n+1. BFS node-visiting process can be visualized as in Figure 2.6

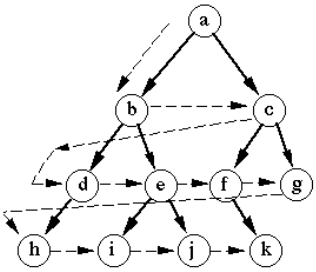

Figure 2.6. Breadth-First Search Node-Visiting Process

Suppose the process starts from node a. Then, it will respectively visit b, c, d, e, f, g, h, i, j, and k.

BFS can be defined as follows:

```
create empty queue Q
u ← starting node
enqueue u to Q
mark u as visited
while Q is not empty
    top ← dequeue(Q)
    if top fulfills the search condition
       //some expression
       //stop function
    for each v neighbor of u
       if v is unvisited
           mark v as visited
           enqueue v to Q
```

III. IMPLEMENTING DFS AND BFS IN GAMES

3.1. Case 1: Minesweeper Empty Tile Click

In minesweeper, if player clicks an empty tile (tile that do not have number or bomb on it) all other empty tiles adjacent to it will also be opened.

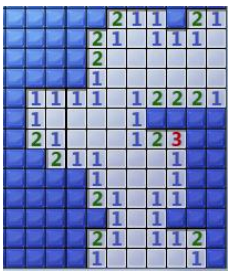

Figure 3.1 Opening an Empty Tile in Minesweeper

So how does the opening algorithm works? The main objective of the algorithm is to visit all empty tiles and open it. If it encounters a numbered tile, it opens the tile but not looks further. Since the main objective is to visit all tiles (or, in graph theory term, nodes), both DFS and BFS can be implemented in this problem.

3.1.1. Implementation

Suppose there are these functions and variables in the game code:

- Tiles (variable)
  A matrix that resembles minesweeper game board. It may has three kind of value : bomb, number and empty. Tiles[i,j] means tiles at column i row j.
- Mark (procedure) : marks Tiles[i,j] as visited / unvisited
- Open (procedure) : opens Tiles[i,j]

DFS implementation is defined as follows:

- Create empty stack
- Mark all tiles as unvisited
- Push starting tile <i,j> to stack
- Mark <i,j> as visited
- While stack not empty
  - Pop top element to <k,l>
  - Open <k,l>
  - If tile[k,l] is empty tile then
    Check for every valid index neighboring <k,l>. If it is unvisited, push it to stack and mark it as visited.

In minesweeper, there are 8 neighbors of <i,j> = {<i-1,j>, <i+1,j>, <i,j-1>, <i,j+1>, {<i-1,j-1>, <i+1,j-1>, <i-1,j-1>, <i+1,j+1>}, in simple, all 8 tiles surrounding it. A valid neighbor is defined as a neighbor that its index is within the index bound of the matrix, i.e. if matrix's index is [1...100, 1...100], then <x, y> is valid if and only if 1 < x < 100 and 1 < y < 100.

BFS implementation is almost exactly same as DFS one, but one needs to use queue instead of stack.

3.2 Case 2: Turn Based Tactics / Strategy Games

3.2.1 Movement Area

In turn based tactics / strategy games, characters can move for a certain distance of tiles. If player selects a character, the game shows which tiles that are available to be set on. Tiles that are outside of character's maximum distance, or have obstacle or other character on will not be shown as available. Notice that in Figure 3.2. available tiles are shown as blue tiles. Far tiles and tiles that have characters or obstacles on are not colored blue.

Figure 3.2 Available Tiles Shown as Blue Tiles.

So how does the coloring algorithm works? The main objective of the coloring algorithm is to visit all tiles that are available and in range, and color them. Tiles that are not in range or unavailable will not be visited.

At the first sight, both DFS and BFS seem can be implemented in this problem by limiting its range of checking. However, because of the range limitation, BFS is more suitable to be implemented than DFS as BFS visits all nodes in the same depth before visiting any

nodes in the next depth. DFS, on the other hand, may produce incorrect results because of the range limitation.

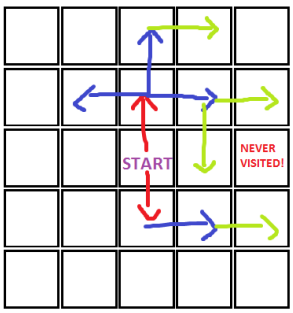

Figure 3.3 Incorrect Results in Limiting DFS Range

In Figure 3.3., DFS visiting process is shown running clock wise; respectively visits top, right, bottom and finally left neighbor. DFS visiting directions are shown as arrows, the colors represent depth / range; red means depth 1, blue 2, and green 3. Since DFS look to the depth first, it may result a non-optimum solution of node depth / range [8]; Tile[4,3] range from "start" is should be 1, but it is counted as 3. Because DFS searching range is limited to a number, in this case 3, there are some nodes that are never visited, but actually they should be visited. In Figure 3.3., such node is shown in Tile[5,3]. Tile[5,3] never visited because all neighboring tiles are regarded as ranged 3. There are methods to use DFS in a limited range called Depth-First Iteration Deepening [7], but it won't be discussed in this paper.

3.2.1.1. Implementation

Suppose there are these functions and variables in the game code:

- Tiles (variable)
  Matrix that represent game tiles. Tiles[i,j] means tiles at column i row j.
- Color (procedure) : colors tile at column i row j.

BFS implementation is defined as follows:

- Create a queue
- Set m with number of maximum movement
- Initialize all tiles as uncolored
- Enqueue starting tile <i,j>
- Color <i,j>
- While queue is not empty and m > 0
  - Dequeue front element to <k,l>
  - Decrease m by 1
  - Check for every valid index neighboring <k,l>. If it is not colored and available, enqueue it and color it.

In many turn base tactics games, there are 4 neighbors of <i,j> = {<i-1,j>, <i+1,j>, <i,j-1>, <i,j+1>}, in simple, the top, bottom, left, and right tiles surrounding it. A valid neighbor is defined as a neighbor that its index is within the index bound of the matrix, i.e. if matrix's index is [1...100, 1...100], then <x, y> is valid if and only if 1 < x < 100 and 1 < y < 100.

In this BFS implementation, colored is used instead of mark. This is done because coloring the tiles can be represented as marking the tiles; tiles that already colored will not be visited again.

3.2.2. Area Damage

In turn based tactics / strategy games, some attacks may affect a range of area, i.e. bombs, missiles, or explosions. This kind of attacks is called area damage or splash damage. As the name suggest, area damage affects all characters in the area; a character will get more damage if it closer to the center of the area.

Figure 3.4 A Simple Area Damage Representation

Notice in Figure 3.4. the farther to the center (Tile[3,3]) the lower the damage. The blue line denotes the bound of tile searching (in this example, the area radius is 2), while tile searching starts from the center. A simple damage percentage calculation as shown in Figure 3.4. can be done by using formula as follows:

$$\%\ Damage = \frac{radius - tile\_range\ + 1}{radius + 1}$$

Note that this formula only calculates damage within the area radius; tiles outside of it will simply take no damage.

Area damage algorithm works like movement area; it checks every tile in range. The difference is the area damage algorithm still visits tiles that have obstacle or character, and checks if there is a character in a tile it visits, calculate and inflict damage to the character. Because it works like movement area algorithm, area damage algorithm also employs BFS algorithm.

3.2.2.1. Implementation

Suppose there are these functions, data types, and variables in the game code:

- Tiles (variable)
  Matrix that represent game tiles. Tiles[i,j] means tiles at column i row j.
- Mark (procedure) : marks Tiles[i,j] as visited / unvisited
- damage(procedure) : inflict damage to character by *value*

BFS implementation is defined as follows:

- create a queue
- set m with radius
- mark all tiles as unvisited
- Enqueue starting tile <i,j>
- Mark <i,j> as visited
- While queue is not empty and m > 0
    - Dequeue front element to <k,l>
    - if there is character on <k,l>, damage it by (m + 1) / (radius + 1) * damage power
    - m ← m - 1
    - Check for every valid index neighboring <k,l>. if it is not visited, enqueue it and mark it as visited.

3.3. Case 3: Maze Games

3.3.1 Maze Solver

Maze games can be solved using both DFS and BFS. In multi-solution maze games, DFS will always generate a solution, but it won't be guaranteed as the optimal one [9]. BFS, on the other hand, will always generate the optimal solution –the shortest one [9].

A maze can be represented in an m x n matrix of boolean. Maze[x,y] will be evaluated as true if it is passable (a path); otherwise, it will be false (a wall).

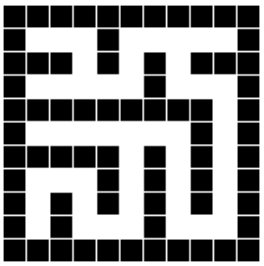

Figure 3.5. Visual Representation of Maze Matrix; White: True (path), Black: False (wall)

3.3.1.1. DFS Implementation

To implement the maze solving algorithm using DFS, the recursive approach is easier to implement. A global stack called path is also required to keep track the observed path.

Suppose there is an empty global stack variable called PATH. DFS implementation is defined as follows:

- Push current_tile to the stack PATH
- Mark current_tile as visited
- If current tile is the destination, then return true
- For each next_tile as valid neighbors of current_tile
    - If next_tile unvisited
        - Recursively call DFS with next_tile as its parameter
        - If it returns true, then return true
- Pop(PATH)
- return false

The solution will be recorded in stack PATH, with the bottom-most element as the starting tile and the top-most one as the destination.

3.3.1.2. BFS Implementation

To implement the maze solving algorithm using BFS, some modifications are needed. Since BFS don't automatically backtrack like DFS [10], a list called directions will be added. Every directions list's member consists of two elements: tile coordinate <x,y> and direction value of its "parent" tile <dx,dy>. The direction value is obtained as follows: if <x,y> is the child of <x, y + 1>, then its direction is <0,1>; if it is the child of <x - 1,y>, then its direction is <-1, 0>.

Suppose there is a list of (<x,y>, <dx, dy>) called DIRECTION. BFS implementation is defined as follows:

- create a queue
- mark all tiles as unvisited
- enqueue starting tile <i,j>
- mark <i,j> as visited
- while queue is not empty and destination not found
    - dequeue front element to <k,l>
    - for each <x,y> valid neighbors of <k,l>
        - if <x,y> is unvisited
            - mark <x,y> as visited
            - enqueue <x,y>
            - add <x,y> to DIRECTION. set its direction value as <k - x, l - y>
            - if <x,y> is the destination, then found and break the loop.

The BFS implementation above doesn't generate the path; it only generates optimal backward directions from destination to the starting tile. Generating the path can simply be done by tracing the directions from destination to starting tile like this:

- <x,y> ← destination tile
- Push <x,y> to stack PATH
- while <x,y> is not the starting tile
    - find <x,y> in DIRECTION. Get its direction value as <dx, dy>
    - <x,y> ← <x+dx, y+dy>
    - Push <x,y> to stack PATH

This algorithm will generate the path solution in stack PATH, with the bottom-most element as the destination tile and the top-most one as the starting tile. Notice that the stack PATH generated by this algorithm is reversed to the DFS one.

3.3.2. Maze Generator

Mazes can also be generated by directly using DFS [11] or using its working principle, recursive backtracking [10]. The general idea of generating mazes by using DFS is to "carve" the walls into maze [10][11].

The general algorithm runs as follow:

- Suppose there is a maze matrix as defined in Section 3.3.1. Maze Solver. Let the maze matrix size be N x N with N is an odd number. Let the index of maze matrix be [0...N-1, 0...N-1].

- Set all cells in the matrix be false (wall).
- Choose a starting cell <i,j> with i and j be odd numbers
- Starts a DFS procedure from <i,j>.
- After the DFS finishes making the maze, choose two carved cells (may be random or not) as the start point and the finish point of the maze.

3.3.2.1. Implementation

DFS implementation is defined as follows:

- Create a stack
- Set <i,j> as path
- Push <i,j> to stack
- While stack is not empty
  - Pop the stack to <k,l>
  - While <k,l> still has valid "wall" neighbors(*)
    - <x,y> ← Randomly selects one of the valid "wall" neighbors(*) of <k,l>
    - Set <x,y> as path
    - Set the cell between <k,l> and <x,y> as path(**)
    - Push <x,y> to stack

Notes:

(*) In this implementation, neighboring cells is defined two cells up, right, down and left of the current cell; that is <i,j-2>, <i+2,j>, <i,j+2> and <i-2,j>.

(**) The cell between <k,l> and <x,y> can be calculated as <k + (k – x)/2, l + (y – l)/2>

In this implementation, odd numbers are chosen to make sure that every tunnel branch will be separated at least by one-block thick of walls. This algorithm can also be implemented with BFS by using queue instead of stack.

IV. Conclusion

Graph searching is one of the most essential algorithms in game programming. By implementing the Depth-First Search and Breadth First Search, two basic graph searching algorithms, some game programming cases like opening empty cells in minesweeper, making character movement area and calculating area damage in turn based tactics and strategy games, and solving and generating mazes. Since DFS and BFS are the simplest graph searching algorithms, both can be replaced with a more advanced graph searching algorithm to increase its efficiency.

V. Acknowledgment

Faiz Ilham Muhammad as the author expresses his deepest gratitude to Allah SWT, the only God of the Universe, for His everlasting mercy He gave during the writing process of this paper. The author also expresses his sincere thanks to Mr Rinaldi Munir and Mrs. Harlili as the lecturers of Discrete Structure Course, to his beloved family, and to his entire comrade in Axivic Lunarismosinerati and ASCII 2011.

References

[1] Shariefuddin Pirsada and Ashay Dharwadker, "Applications of Graphs Theory" in *Journal of The Korean Society for Industrial and Applied Mathematics*. vol. 11 no. 4. 2007.

[2] Introduction to Graphs and Their Data Structures: Section 2. http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=graphsDataStrucs2. Accessed at December 16th 2012 2.51 p.m.

[3] Rinaldi Munir, *Diktat Kuliah IF2091: Struktur Diskrit*. 4th edition. Bandung: Program Studi Teknik Informatika STEI, 2008, ch. 10.

[4] Kenneth H Rosen, *Discrete Mathematics and Its Application 7th*. New York: McGraw-Hill Companies Inc., 2012, ch 10.

[5] Stacks and Queues. http://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html. Accessed at December 16th 2012 5.42 p.m.

[6] Inggriani Liem. *Diktat Struktur Data*. Bandung: Program Studi Teknik Informatika STEI, 2008, pp 41-60.

[7] Depth-First Iterative Deepening. http://intelligence.worldofcomputing.net/ai-search/depth-first-iterative-deepening.html. Accessed at December 17th 9.30 a.m.

[8] Game Trees. http://www.cs.cmu.edu/~./adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html. Accessed at December 17th 9.30 a.m.

[9] Maze Algorithms. http://www.astrolog.org/labyrnth/algrithm.htm. Accessed at December 18th 10.00 a.m.

[10] BFS, DFS and SCC Algorithm. http://www.cs.bgu.ac.il/~visproj/romanr/bfs_dfs_scc.html. December 18th 10.00 a.m.

[11] Depth First Search, Maze Algorithm. http://www.migapro.com/depth-first-search/. Accessed at December 18th 10.00 a.m

PERNYATAAN

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, 18 Desember 2011

Faiz Ilham Muhammad
13511080