Modul 9

Breadth First Search & Depth First Search

9.1 Tujuan

Setelah mengikuti praktikum ini, diharapkan mahasiswa mampu mengenal konsep dan implementasi algoritma

a. BFS (*Breadth First Search*)
b. DFS (*Depth First Search*)

9.2 Alat dan Bahan

Alat & Bahan Yang digunakan adalah

a. Hardware : Perangkat computer berupa PC/Laptop
b. Sofware : Java SDK, Code editor

9.3 Dasar Teori

9.3.1 *Breadth First Search* (BFS)

Breadth First Search (BFS) adalah algoritma traversal graph. Tujuan dari algoritma ini adalah untuk mengunjugi seluruh simpul yang ada (yang terhubung) dengan mengimplementasikan penggunaan Queue.

Algoritma ini akan dimulai dari suatu simpul (*node*) s, kemudian akan dilanjutkan kepada simpul-simpul tetangga **level 1** selanjutnya dari s (anggap adalah “tetangga s-n”) dan daftar “tetangga s-n” tersebut akan dimasukkan ke dalam Queue. Jika daftar tetangga dari s telah dikunjungi semuanya, maka akan dilanjutkan kepada berlanjut pada level 2, yaitu simpul akar akan berada pada “tetangga s-1”. Proses tersebut berulang dengan mendaftar dan mengunjungi “tetangga s-1”, “tetangga s-2” dan seterusnya. Proses berulang hingga seluruh simpul telah dikunjungi.

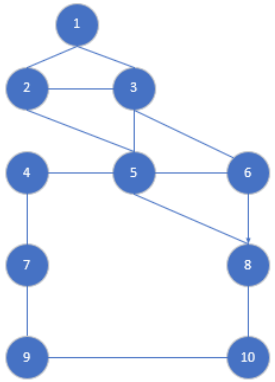
.

9.3.1.1 Algoritma BFS

Algoritma BFS adalah sebagai berikut

a. Tentukan matrik adjacency,
b. Tentukan simpul awal dan masukkan ke dalam Queue,
c. Remove dan ambil simpul pertama dari Queue, masukkan simpul tersebut kedalam `temp`,
d. Cari daftar tetangga yang belum pernah dikunjungi,
e. Masukkan daftar simpul tetangga yang belum dikunjungi kedalam Queue,
f. Cetak `temp`
g. Ulang ke c hingga daftar Queue habis

Perhatikan graph tak berarah berikut ini



Gambar 9.1 Contoh Graph

Dengan menggunakan algoritma BFS diatas, dengan simpul awal adalah simpul1, maka proses penelusuran adalah sebagai berikut

No	Simpul	Tetangga	Queue	Ter-cetak	Keterangan
0	~	~	1	~	Penentuan simpul awal
1	1	2, 3	2, 3	1	
2	2	~~1~~, 3, 5	3, 5	1 2	
3	3	~~1~~, 5, 6	5, 6	1 2 3	
4	5	~~3~~, 4, 6, 8	6, 4, 8	1 2 3 5	Pembacaan Tetangga akan tergantung kepada alur pembacaan urutan Dapat berupa [4, 8] atau [8, 4]
5	6	~~5~~, 8	4, 8	1 2 3 5 6	
6	4	~~5~~, 7	8, 7	1 2 3 5 6 4	
7	8	~~5~~, ~~6~~, 10	7, 10	1 2 3 5 6 4 8	
8	7	~~4~~, 9	10, 9	1 2 3 5 6 4 8 7	
9	10	~~8~~, 9	9	1 2 3 5 6 4 8 7 10	
10	9	~~7~~, ~~10~~	~	1 2 3 5 6 4 8 7 10 9	

Keterangan : simpul yang dicoret pada kolom tetangga, artinya simpul tersebut sebelumnya telah dikunjungi

9.3.2 *Depth First Search* (BFS)

Depth First Search (BFS) adalah algoritma traversal graph. Tujuan dari algoritma ini adalah untuk mengunjugi seluruh simpul yang ada (yang terhubung) dengan mengimplementasikan menggunakan Stack atau rekursif. Dalam Modul ini akan digunakan Stack.

Algoritma ini akan dimulai dari suatu simpul (*node*) s, kemudian akan dilanjutkan kepada simpul-simpul tetangga **level 1** selanjutnya dari s (anggap adalah "tetangga s-n") dan daftar "tetangga s-n" tersebut akan dimasukkan kedalam Stack. Selanjutnya, akan dilakukan proses pengecekan daftar "tetangga s-1", dan juga akan dimasukkan kedalam stack hingga mencapai ujung dari graph. Proses dilakukan secara berulang dengan melakukan **Pop** data dari Stack, hingga data dalam Stack habis.

Namun hal ini dapat menyebabkan permasalahan, jika graph terlalu kompleks dan terlalu dalam sehingga proses pencarian satu utas simpul akan menjadi terlalu panjang. Oleh karena itu umumnya akan dilakukan pembatasan hingga seberapa dalam proses tersebut dilakukan. Sehingga pembatasan tersebut tersebut akan menyebabkan kemungkinan ada simpul yang tidak pernah terkunjungi.

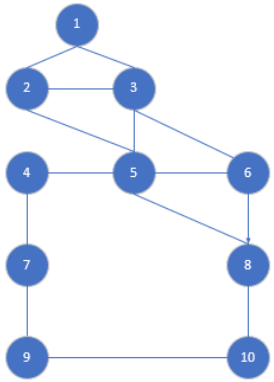
Perbedaan utama antara BFS dan DFS terletak pada bagaimana cara mengunjungi simpul. Jika BFS akan menghabiskan simpul pada level 1, sedangkan DFS akan merunut satu simpul hingga ujung graph.

9.3.2.1 Algoritma DFS

Algoritma BFS adalah sebagai berikut

a. Tentukan matrik adjacency,
b. Tentukan simpul awal dan masukkan kedalam Stack,
c. Pop simpul pertama dari Stack, masukkan simpul tersebut kedalam `temp`,
d. Cari daftar tetangga dari simpul tersebut yang belum pernah dikunjungi,
e. Masukkan daftar simpul tetangga yang belum dikunjungi kedalam Stack,
f. Cetak `temp`
g. Ulang ke c hingga daftar Stack habis

Perhatikan graph tak berarah berikut ini



Gambar 9.2 Contoh Graph

Dengan menggunakan algoritma DFS, dengan simpul awal adalah simpul1, maka proses penelusuran adalah sebagai berikut

No	Simpul	Tetangga	Stack	Ter-cetak	Keterangan
0	~	~	1	~	Penentuan simpul awal
1	1	2, 3	3, 2	1	Push 2, push 3 → posisi menjadi terbalik 3, 2
2	3	~~1,~~ ~~2,~~ 5, 6	6, 5, 2	1 3	Push 5, push 6 → posisi menjadi terbalik 6, 5, 2
3	6	~~3,~~ ~~5,~~ 8	8, 5, 2	1 3 6	Dst
4	8	~~5,~~ ~~6,~~ 10	10, 5, 2	1 3 6 8	
5	10	~~8,~~ 9	5, 2	1 3 6 8 10	9 tidak dimasukkan kedalam stack, perhatikan pemetaan adjacency matriks
6	5	~~2,~~ ~~3,~~ 4, ~~6~~	4, 2	1 3 6 8 10 5	
7	4	~~5,~~ 7	7, 2	1 3 6 8 10 5 4	
8	7	~~4,~~ 9	9, 2	1 3 6 8 10 5 4 7	
9	9	~~7,~~ ~~10~~	2	1 3 6 8 10 5 4 7 9	
10	2	~~1,~~ ~~3,~~ ~~5~~	~	1 3 6 8 10 5 4 7 9 2	

Keterangan :

a. Urutan dalam kolom stack akan menjadi terbalik jika dibandingkan dengan kolom tetangga, karena tetangga terakhir akan di-push paling akhir, sehingga menjadi Top dari Stack
b. simpul yang dicoret pada kolom tetangga, artinya simpul tersebut sebelumnya telah dikunjungi

9.3.2.2 Implementasi Algoritma

9.3.2.2.1 Kelas Simpul (*NodeAdjacent*)

Berikut adalah kelas yang digunakan untuk melakukan proses penyimpanan data simpul.

Variable

a. data, value dari node akan disimpan,
b. visited, status sudah pernah dikunjungi atau belum.

```
1.    class NodeAdjacent {
2.        private int data;
3.        private boolean visited=false;
4.
5.        NodeAdjacent(int data) {
6.            this.data = data;
7.        }
8.
9.        public int getData() {
10.           return data;
11.       }
12.
13.       public boolean isVisited() {
14.           return visited;
15.       }
16.
17.       public void setVisited(boolean visited) {
18.           this.visited = visited;
19.       }
20.   }
```

9.3.2.3 Implementasi BFS

```
1           import java.util.ArrayList;
2           import java.util.LinkedList;
3           import java.util.Queue;

4           public class BFS_ADJACENT {

5           private Queue<NodeAdjacent> queue = new LinkedList<>();      //daftar Queue
6           static ArrayList<NodeAdjacent> nodes = new ArrayList<NodeAdjacent>();     //daftar simpul

            /*
            Method ini digunakan untuk mencari daftar tetangga dari suatu simpul.
            Daftar tetangga didapatkan dari matrik adjacent.
            */
7           public ArrayList<NodeAdjacent> listTetangga(int matriks[][], NodeAdjacent x) {
8                 int idx = -1;
9                 ArrayList<NodeAdjacent> tetangga = new ArrayList<>();
10                for (int i = 0; i < nodes.size(); i++) {
11                      if (nodes.get(i).getData() == x.getData())) {
12                            idx = i;
13                            break;
14                      }
15                }

16                if (idx != -1) {
17                      for (int j = 0; j < matriks[idx].length; j++) {
18                            if (matriks[idx][j] == 1) {
19                                  tetangga.add(nodes.get(j));
20                            }
21                      }
22                }
23                return tetangga;
24          }
```

```
            /*
            Algoritma utama dari BFS. Sesuai dengan Algoritma pada bagian 9.3.1.1
            Parameter
            a. matriks : daftar matriks adjacent
            b. node : simpul awal
            */

25          public void bfs(int matriks[][], NodeAdjacent node) {
26               queue.add(node);
27               node.setVisited(true);
28               while (!queue.isEmpty()) {
29                    NodeAdjacent element = queue.remove();
30                    System.out.print(element.getData() + "_");
31                    ArrayList<NodeAdjacent> tetangga = listTetangga(matriks, element);
32                    for (int i = 0; i < tetangga.size(); i++) {
33                         NodeAdjacent n = tetangga.get(i);
34                         if (n != null && !n.isVisited()) {
35                              queue.add(n);
36                              n.setVisited(true);
37                         }
38                    }
39               }
40          }
```

```
            /*
            Method utama, berisi penentuan matriks dan pemanggilan algoritma.
            Daftar simpul, dengan asumsi value unik
            */

41          public static void main(String arg[]) {
42                NodeAdjacent node1 = new NodeAdjacent(1);
43                NodeAdjacent node2 = new NodeAdjacent(2);
44                NodeAdjacent node3 = new NodeAdjacent(3);
45                NodeAdjacent node4 = new NodeAdjacent(4);
46                NodeAdjacent node5 = new NodeAdjacent(5);
47                NodeAdjacent node6 = new NodeAdjacent(6);
48                NodeAdjacent node7 = new NodeAdjacent(7);
49                NodeAdjacent node8 = new NodeAdjacent(8);
50                NodeAdjacent node9 = new NodeAdjacent(9);
51                NodeAdjacent node10 = new NodeAdjacent(10);

52                nodes.add(node1);
53                nodes.add(node2);
54                nodes.add(node3);
55                nodes.add(node4);
56                nodes.add(node5);
57                nodes.add(node6);
58                nodes.add(node7);
59                nodes.add(node8);
60                nodes.add(node9);
61                nodes.add(node10);
```

```
62              int matriks[][] = {
63              //   Simpul 1  2  3  4  5  6  7  8  9  10
64                           {0, 1, 1, 0, 0, 0, 0, 0, 0, 0}, // Simpul 1 : value 1
65                           {1, 0, 1, 0, 1, 0, 0, 0, 0, 0}, // Simpul 2 : value 2
66                           {1, 1, 0, 0, 1, 1, 0, 0, 0, 0}, // Simpul 3 : value 3
67                           {0, 0, 0, 0, 1, 0, 1, 0, 0, 0}, // Simpul 4 : value 4
68                           {0, 1, 1, 1, 0, 1, 0, 1, 0, 0}, // Simpul 5 : value 5
69                           {0, 0, 0, 0, 1, 0, 0, 1, 0, 0}, // Simpul 6 : value 6
70                           {0, 0, 0, 1, 0, 0, 0, 0, 1, 0}, // Simpul 7 : value 7
71                           {0, 0, 0, 0, 1, 1, 0, 0, 0, 1}, // Simpul 8 : value 8
72                           {0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Simpul 9 : value 9
73                           {0, 0, 0, 0, 0, 0, 0, 1, 1, 0}, // Simpul 10: value 10
74              };


75              System.out.print("Traversal Graph : ");
76              BFS_ADJACENT bfs_adjacent = new BFS_ADJACENT();
77              bfs_adjacent.bfs(matriks, node1);
78              System.out.println("");
79            }
80          }
```

Matriks adjacency diatas menggunakan Graph yang ada pada gambar-gambar 9.1 dan 9.2

```
1.    import java.util.ArrayList;
2.    import java.util.Stack;

3.    public class DFS_ADJACENT {

4.       static ArrayList<NodeAdjacent> nodes = new ArrayList<>();

      /*
      Method ini digunakan untuk mencari daftar tetangga dari suatu simpul.
      Daftar tetangga didapatkan dari matrik adjacent.
      */
5.       public ArrayList<NodeAdjacent> findNeighbours(int matriks[][], NodeAdjacent x) {
6.          int idx = -1;
7.          ArrayList<NodeAdjacent> tetangga = new ArrayList<>();
8.          for (int i = 0; i < nodes.size(); i++) {
9.             if (nodes.get(i).equals(x)) {
10.               idx = i;
11.               break;
12.            }
13.         }
14.
15.         if (idx != -1) {
16.            for (int j = 0; j < matriks[idx].length; j++) {
17.               if (matriks[idx][j] == 1) {
18.                  tetangga.add(nodes.get(j));
19.               }
20.            }
21.         }
22.         return tetangga;
23.      }
```

```
/*
Algoritma utama dari DFS. Sesuai dengan Algoritma pada bagian 9.3.2.1
Parameter
matriks : daftar matriks adjacent
node : simpul awal
*/

24.     public void dfsUsingStack(int matriks[][], NodeAdjacent node) {
25.        Stack<NodeAdjacent> stack = new Stack<>();
26.        stack.add(node);
27.        while (!stack.isEmpty()) {
28.           NodeAdjacent element = stack.pop();
29.           if (!element.isVisited()) {
30.              System.out.print(element.getData() + " ");
31.              element.setVisited(true);
32.           }

33.           ArrayList<NodeAdjacent> tetangga = findNeighbours(matriks, element);
34.           for (int i = 0; i < tetangga.size(); i++) {
35.              NodeAdjacent n = tetangga.get(i);
36.              if (n != null && !n.isVisited()) {
37.                 stack.add(n);
38.              }
39.           }
40.        }
41.     }
```

```
            /*
            Method utama, berisi penentuan matriks dan pemanggilan algoritma.
            Daftar simpul, dengan asumsi value unik
            */

42.      public static void main(String arg[]) {
43.         NodeAdjacent node1 = new NodeAdjacent(1);
44.         NodeAdjacent node2 = new NodeAdjacent(2);
45.         NodeAdjacent node3 = new NodeAdjacent(3);
46.         NodeAdjacent node4 = new NodeAdjacent(4);
47.         NodeAdjacent node5 = new NodeAdjacent(5);
48.         NodeAdjacent node6 = new NodeAdjacent(6);
49.         NodeAdjacent node7 = new NodeAdjacent(7);
50.         NodeAdjacent node8 = new NodeAdjacent(8);
51.         NodeAdjacent node9 = new NodeAdjacent(9);
52.         NodeAdjacent node10 = new NodeAdjacent(10);

53.         nodes.add(node1);
54.         nodes.add(node2);
55.         nodes.add(node3);
56.         nodes.add(node4);
57.         nodes.add(node5);
58.         nodes.add(node6);
59.         nodes.add(node7);
60.         nodes.add(node8);
61.         nodes.add(node9);
62.         nodes.add(node10);
```

```
63.          int matriks[][] = {
64.         //   Simpul 1  2  3  4  5  6  7  8  9  10
81                      {0, 1, 1, 0, 0, 0, 0, 0, 0, 0}, // Simpul 1 : value 1
82                      {1, 0, 1, 0, 1, 0, 0, 0, 0, 0}, // Simpul 2 : value 2
83                      {1, 1, 0, 0, 1, 1, 0, 0, 0, 0}, // Simpul 3 : value 3
84                      {0, 0, 0, 0, 1, 0, 1, 0, 0, 0}, // Simpul 4 : value 4
85                      {0, 1, 1, 1, 0, 1, 0, 1, 0, 0}, // Simpul 5 : value 5
86                      {0, 0, 0, 0, 1, 0, 0, 1, 0, 0}, // Simpul 6 : value 6
87                      {0, 0, 0, 1, 0, 0, 0, 0, 1, 0}, // Simpul 7 : value 7
88                      {0, 0, 0, 0, 1, 1, 0, 0, 0, 1}, // Simpul 8 : value 8
89                      {0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Simpul 9 : value 9
65.                     {0, 0, 0, 0, 0, 0, 0, 1, 1, 0}, // Simpul 10: value 10
66.         };

67.          DFS_ADJACENT dfsExample = new DFS_ADJACENT();
68.          System.out.print("Traversal Graph : ");
69.          dfsExample.dfsUsingStack(matriks, node1);
70.          System.out.println();
71.       }
72.    }
```

Matriks adjacency diatas menggunakan Graph yang ada pada gambar-gambar 9.1 dan 9.2

Untuk selanjutnya silahkan kerjakan dengan menggunakan matriks yang berbeda