MODUL 4
TEKNIK PENCARIAN BLIND SEARCH

4.1. Tujuan Praktikum

Mahasiswa mampu memahami konsep blind search dan dapat mengimplementasikan program salah satu algoritma blind search pada kasus tree. Program ini dibuat dengan menggunakan bahasa pemrograman Java.

4.2. Dasar Teori

4.2.1. Teknik Pencarian Blind Search

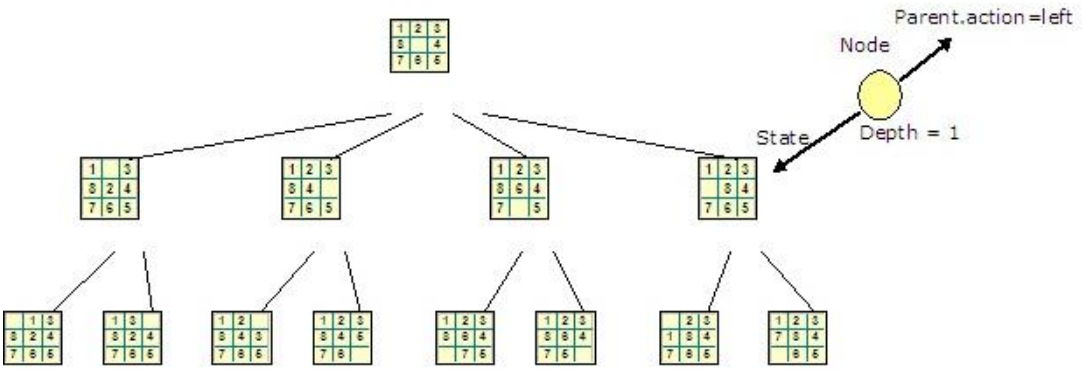
Ada beberapa algoritma yang dikategorikan ke dalam teknik pencarian blind search.

- Breadth First Search (BFS)
- Uniform Cost Search (UCS)
- Depth First Search (DFS)
- Depth Limited Search (DLS)
- Iterative Deepening Search (IDS)
- Bidirectional Search (BS)

Pada pertemuan praktikum ini, akan diperlihatkan salah satu algoritma blind search untuk mencari goal dari initial state yang diberikan. Terdapat 4 komponen yang harus didefinisikan ketika mencari solusi dari sebuah permasalahan.

1. Initial state
2. Goal State
3. Menentukan/menemukan urutan untuk mencapai Goal State
4. Biaya (cost) menemukan solusi.

Kesemua algoritma di atas mempunyai strategi dengan mencari goal yang dimulai dari initial state. Terminologi tree dipilih sebagai salah satu teknik pencarian untuk mencapai goal. Untuk lebih jelas, Gambar 4.1 memperlihatkan contoh kasus penyelesaian permainan 8-puzzle dengan mentransformasikan solusi permainan ke dalam topologi tree.



Gambar 4.1. Topologi Tree untuk permainan 8-puzzle[1]

Kumpulan node-node yang dibentuk tetapi belum disambungkan dengan node yang lain dinamakan dengan *fringe*. Setiap element dari fringe merupakan node left dari tree. Berikut akan dijelaskan algoritma-algoritma yang dikategorikan ke dalam kelas blind search.

- **Breadth First Search (BFS)**: adalah algoritma yang menjelajah node root pertama sekali, kemudian menjelajah semua successor dari node root, kemudian menjelajah semua successor dari successor, dan seterusnya sampai successor yang terakhir. Fringe merupakan struktur data queue First In First Out (FIFO).

- **Uniform Cost Search (UCS)**: merupakan modifikasi dari BFS dengan selalu menjelajah node yang paling sedikit cost-nya pada fringe menggunakan *path cost function g(n)* (misalnya biaya (banyaknya langkah) dari initial state ke node n). Node disusun dengan algoritma queue untuk menentukan jumlah (biaya) untuk mencapai node n.

- **Depth First Search (DFS)**: selalu menjelajah node yang paling dalam pada tree. Fringe merupakan struktur data queue (stack) Last In First Out (LIFO).

- **Depth Limited Search (DLS)**: Kegagalan algoritma DFS dalam menyediakan space (memory) dapat diatasi dengan menentukan terlebih dahulu depth limit l, yaitu node pada depth l diperlakukan seolah-olah mereka tidak memiliki successors.

- **Iterative Deepening Depth First Search (IDS)**: secara umum strategi algoritma ini biasanya digunakan dengan mengkombinasikan algoritma depth first tree search yang mencari *the best depth limit*. Ini dilakukan dengan menambahkan limit dari 0, kemudian 1, kemudian 2, and dan seterusnya sampai goal-nya ditemukan.

[1] (Sumber: http://www.codeproject.com/Articles/203828/AI-Simple-Implementation-of-Uninformed-Search-Stra)

- **Bidirectional Search (BS)**: Ide dari algoritma ini adalah untuk mencari secara bersamaan baik dari goal ke initial state dan dari the initial state ke goal, dan berhenti ketika kedua langkah pencarian bertemu di pertengahan pencarian. Disini, terdapat dua fringe, fringe pertama digunakan untuk langkah dari initial state ke goal *(forward)* dan fringe satu lagi untuk langkah dari goal ke initial state *(backward)*. Setiap fringe diimplementasikan dengan algoritma LIFO atau FIFO tergantung dari strategi pencarian yang digunakan (misalnya Forward=BFS, Backward=DFS).

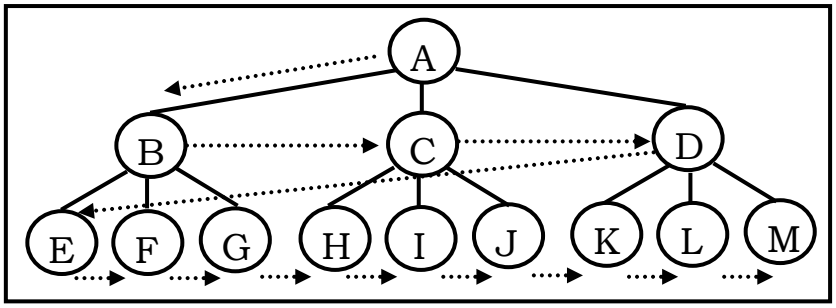
4.2.2. Metode Pencarian Dan Pelacakan

Pada dasarnya ada 2 teknik pencarian dan pelacakan yang digunakan, yaitu pencarian buta (*blind search*) dan pencarian terbimbing (*heuristic search*).

4.2.2.1. Pencarian Buta (*Blind Search*)

1. Pencarian Melebar Pertama *(Breadth-First Search)*

- Pada metode *Breadth-First Search*, semua node pada level n akan dikunjungi terlebih dahulu sebelum mengunjungi node-node pada level n+1.
- Pencarian dimulai dari node akar terus ke level ke-1 dari kiri ke kanan, kemudian berpindah ke level berikutnya demikian pula dari kiri ke kanan hingga ditemukannya solusi (Gambar 4.2).

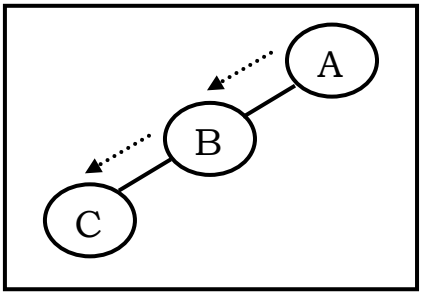


Gambar 4.2. Metode Breadth First Search

- Keuntungan
  a. Tidak akan menemui jalan buntu.
  b. Jika ada satu solusi, maka *breadth-first search* akan menemukannya. Dan jika ada lebih dari satu solusi, maka solusi minimum akan ditemukan.
- Kelemahan
  a. Membutuhkan memori yang cukup banyak, karena menyimpan semua node dalam satu pohon.
  b. Membutuhkan waktu yang cukup lama, karena akan menguji n level untuk mendapatkan solusi pada level yang ke-(n+1).

2. **Pencarian Mendalam Pertama *(Depth-First Search)***

- Pada *Depth-First Search*, proses pencarian akan dilakukan pada semua anaknya sebelum dilakukan pencarian ke node-node yang selevel.
- Pencarian dimulai dari node akar ke level yang lebih tinggi. Proses ini diulangi terus hingga ditemukannya solusi (Gambar 4.2).

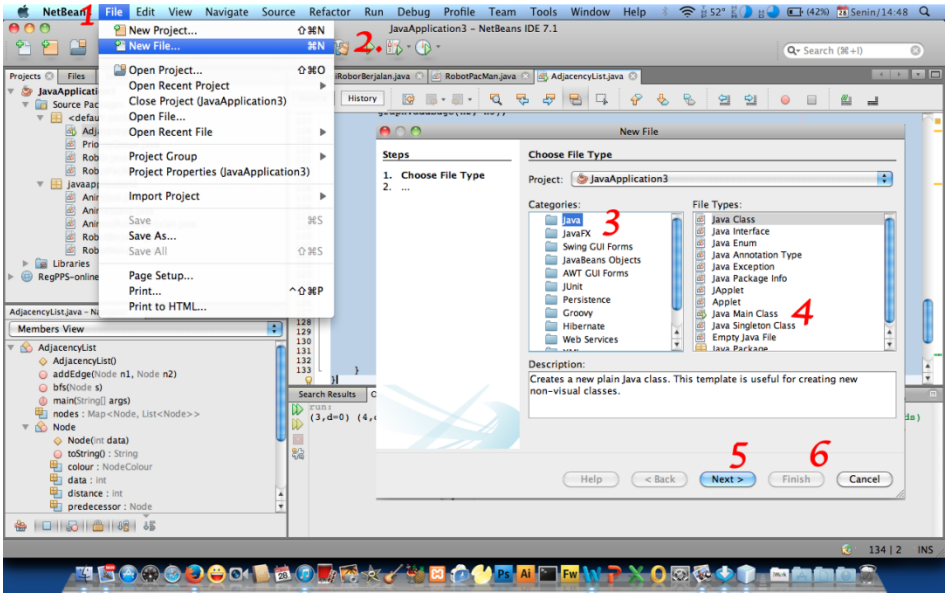


Gambar 4.3. Depth First Search

- Keuntungan
  a. Membutuhkan memori yang relatif kecil, karena hanya node-node pada lintasan yang aktif saja yang disimpan.
  b. Secara kebetulan, metode *depth-first search* akan menemukan solusi tanpa harus menguji labih banyak lagi dalam ruang keadaan.
- Kelemahan
  a. Memungkinkan tidak ditemukannya tujuan yang diharapkan.
  b. Hanya akan mendapatkan 1 solusi pada setiap pencarian.

4.3. Mengimplementasikan Algoritma Bfs.

Ketiklah *source code* Program 3.1 pada perangkat lunak Netbeans 7.0 pada bagian teks editor Java Main Class. Pilih Menu “File”, lalu pilih submenu “New File”. Kemudian pilih Categories “Java” dengan FileTypes-nya adalah “Java Main Class”. Setelah itu, tekan tombol “Next” dan masukkan nama file AdjacencyList, dan terakhir tekan tombol “Finish”. Alur langkah untuk membuat algoritma BFS dapat diikuti melalui Gambar 4.3



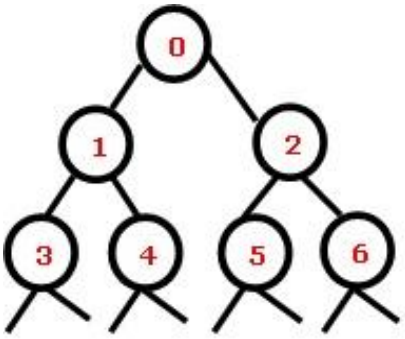
Gambar 4.4. Teks Editor Netbeans 7.0

Kode berikut merupakan urutan angka dari 0, 1, 2, .N. Initial state-nya adalah 0 dan goal state-nya adalah angka yang ditetapkan oleh user. Setiap langkah dibangkitkan secara acak atau 1. *Method* `addEdge(Node n1, Node n2)` adalah method untuk menghubungkan dua buah edge sedangkan *method* `bfs(Node s)` adalah method untuk menentukan teknik pencarian menggunakan algoritma Breadth First Search..

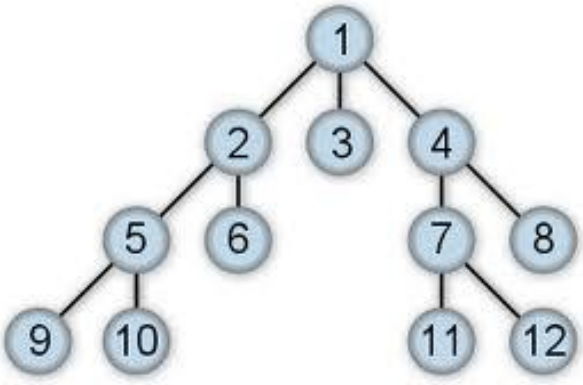
```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Queue;
import java.util.Set;

/**
 * Graph represented by an adjacency list.
 *
 * Reference: Introduction to Algorithms - CLRS.
 *
 * @author
 * @since: 26/02/2011
 */

public class AdjacencyList
{
    public enum NodeColour { WHITE, GRAY, BLACK }

    public static class Node
    {
        int data;
        int distance;
        Node predecessor;
        NodeColour colour;

        public Node(int data)
```

```
    {
        this.data = data;
    }

    public String toString()
    {
     return "(" + data + ",d=" + distance + ")";
    }
}

Map<Node, List<Node>> nodes;

public AdjacencyList()
{
    nodes = new HashMap<Node, List<Node>>();
}

public void addEdge(Node n1, Node n2)
{
    if (nodes.containsKey(n1)) {
        nodes.get(n1).add(n2);
    } else {
   ArrayList<Node> list = new ArrayList<Node>();
        list.add(n2);
        nodes.put(n1, list);
    }
}

public void bfs(Node s)
{
    Set<Node> keys = nodes.keySet();
    for (Node u : keys) {
        if (u != s) {
            u.colour = NodeColour.WHITE;
            u.distance = Integer.MAX_VALUE;
            u.predecessor = null;
        }
    }
    s.colour = NodeColour.GRAY;
    s.distance = 0;
    s.predecessor = null;
    Queue<Node> q = new ArrayDeque<Node>();
    q.add(s);
    while (!q.isEmpty()) {
        Node u = q.remove();
        List<Node> adj_u = nodes.get(u);
        if (adj_u != null) {
            for (Node v : adj_u) {
                if(v.colour == NodeColour.WHITE) {
                    v.colour = NodeColour.GRAY;
                    v.distance = u.distance + 1;
                    v.predecessor = u;
                    q.add(v);
                }
            }
        }
        u.colour = NodeColour.BLACK;
        System.out.print(u + " ");
    }
}
public static void main(String[] args)
```

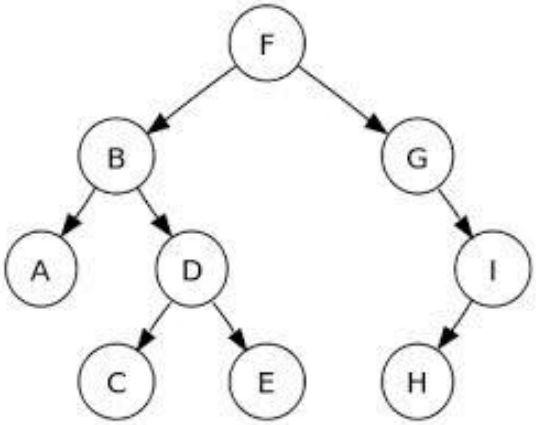
```
    {
        AdjacencyList graph = new AdjacencyList();
        Node n1 = new Node(1);
        Node n2 = new Node(2);
        Node n3 = new Node(3);
        Node n4 = new Node(4);
        Node n5 = new Node(5);
        Node n6 = new Node(6);
        Node n7 = new Node(7);
        Node n8 = new Node(8);

        graph.addEdge(n1, n2);

        graph.addEdge(n2, n1);
        graph.addEdge(n2, n3);

        graph.addEdge(n3, n4);
        graph.addEdge(n3, n2);

        graph.addEdge(n4, n3);
        graph.addEdge(n4, n5);
        graph.addEdge(n4, n6);

        graph.addEdge(n5, n4);
        graph.addEdge(n5, n6);
        graph.addEdge(n5, n7);

        graph.addEdge(n6, n4);
        graph.addEdge(n6, n5);
        graph.addEdge(n6, n7);
        graph.addEdge(n6, n8);

        graph.addEdge(n7, n5);
        graph.addEdge(n7, n6);
        graph.addEdge(n7, n8);

        graph.addEdge(n8, n6);
        graph.addEdge(n8, n7);

        graph.bfs(n3);

    }
}
```



Gambar 4.5. Tree 1



Gambar 4.6. Tree 2



Gambar 4.7. Tree 3

Tugas:

1. Tentukan bagaimana algoritma BFS di atas dapat menentukan node ke 8, 6, dan 7.
2. Ubahlah method static void main sehingga bentuk tree seperti Gambar 4.4 dapat dibentuk. Kemudian tentukan bagaimana algoritma BFS dapat menemukan node 5.
3. Ubahlah method static void main sehingga bentuk tree seperti Gambar 4.5 dapat dibentuk. Kemudian tentukan bagaimana algoritma BFS dapat menemukan node 9.

Ubahlah kode program di atas sehingga bentuk tree seperti Gambar 6 dapat dibentuk. Kemudian tentukan bagaimana algoritma BFS dapat menemukan node C.