

Lab #2, 3: Uninformed Search Algorithms

The main aim of these labs are to deal with uninformed search algorithms using tree search approach.

For a given tree with nodes and weights of each edge described in the following figure:

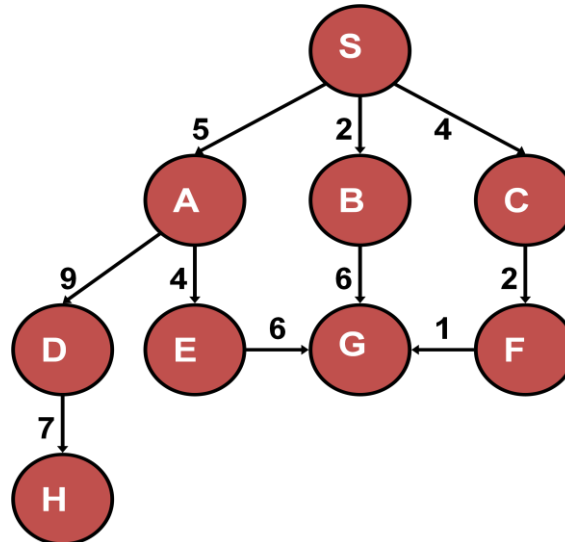


Fig. 1. Tree 1

The node structure used for the tasks of uninformed search is defined as follows:

```
public class Node implements Comparable<Node>, Cloneable {
    private String label;
    private Node parent; // for print the path from the start node
    to goal node
    private double pathCost; // from the root node to this node
    private int depth; // used for compute the depth of a node in a
    tree search
    private List<Edge> children = new ArrayList<Edge>();

    public Node(String label) {
        this.label = label;
    }

    public Node(String label, int h) {
        this.label = label;
    }

    //...
}
```

Each Node has a **label** and a **path cost** (computed from the start node) and a list of **children** presented as edges. Each edge includes **begin** and the **end** nodes. We can build the tree by the **addEdge** methods as follows:

- **addEdge (Node that, double weight):** add an edge connecting the current node with that node. The weight is given as the second parameter.

- **addEdge(Node that):** add an edge connecting the current node with that node. The default weight is 1. **This option is used for non-weighted tree.**

parent attribute is used to **track the found path** from the **Start node to the Goal node** (or from the **Root node to the Goal node**).

```
public class Edge implements Comparable<Edge>{
    private Node begin;
    private Node end;
    private double weight;

    public Edge(Node begin, Node end, double weight) {
        super();
        this.begin = begin;
        this.end = end;
        this.weight = weight;
    }

    public Edge(Node begin, Node end) {
        this.begin = begin;
        this.end = end;
        this.weight = 1;
    }
    //...
```

Next, the interface **ISearchAlgo.java** defined 2 execute methods:

```
public Node execute(Node tree, String goal); // find the path from root node
//to the goal node

public Node execute(Node tree, String start, String goal); // find the path
//from start node to the goal node
```

Notice that, traversal of nodes in alphabetical order.

=====

For BreadthFirstSearchAlgo, the structure of the frontier is Queue:

```
Queue<Node> frontier = new LinkedList<Node>();
```

For DepthFirstSearchAlgo, the structure of the frontier is Stack:

```
Stack<Node> frontier = new Stack<Node>();
```

Pseudocode for searching algorithms:

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

Fig. 4. Tree search algorithm

Task 1: Implement `execute(Node tree, String goal)` in `BreadthFirstSearchAlgo.java` and `DepthFirstSearchAlgo.java`

In this task, the **parent** attribute is used to track the found path from the Root node (or the Start node) to the Goal node.

The method for convert from a GOAL node with setting its parent node is defined as follows (`NodeUtils.java`):

```
public static List<String> printPath(Node node) {
    List<String> result = new ArrayList<String>();
    result.add(node.getLabel());
    Node tmp;
    while ((tmp = node.getParent()) != null) {
        result.add(tmp.getLabel());
        node = tmp;
    }
    Collections.reverse(result);
    return result;
}
```

Using the following nodes (see Fig. 1) for testing:

```
Node nodeS = new Node("S");
Node nodeA = new Node("A");
Node nodeC = new Node("C");
Node nodeE = new Node("E");
Node nodeG = new Node("G");

Node nodeB = new Node("B");
Node nodeD = new Node("D");
Node nodeF = new Node("F");
Node nodeH = new Node("H");

nodeS.addEdge(nodeA, 5);
nodeS.addEdge(nodeC, 4);
nodeA.addEdge(nodeE, 4);

nodeC.addEdge(nodeF, 2);
nodeE.addEdge(nodeG, 6);

nodeS.addEdge(nodeB, 2);
nodeA.addEdge(nodeD, 9);
nodeB.addEdge(nodeG, 6);

nodeD.addEdge(nodeH, 7);
nodeF.addEdge(nodeG, 1);

ISearchAlgo algo1 = new BreadthFirstSearchAlgo();
```

```
algo1.execute(nodeS);
```

Task 2: Similar to Task 1, implement method `execute(Node tree, String start, String goal)` in **BreadthFirstSearchAlgo.java**, **DepthFirstSearchAlgo.java**

Task 3: Implement `execute(Node tree, String goal)` in **UniformCostSearchAlgo.java** (implements **ISearchAlgo**)

For **UniformCostSearchAlgo**, the frontier is **PriorityQueue**:

```
PriorityQueue<Node> frontier = new PriorityQueue<Node>(new
NodeComparator());
```

NodeComparator is used for comparing the pathcosts of two nodes.

Task 4: Similar to Task 2, implement method `execute(Node tree, String start, String goal)` in **UniformCostSearchAlgo.java**

Additional task: test all implementations with the following tree:

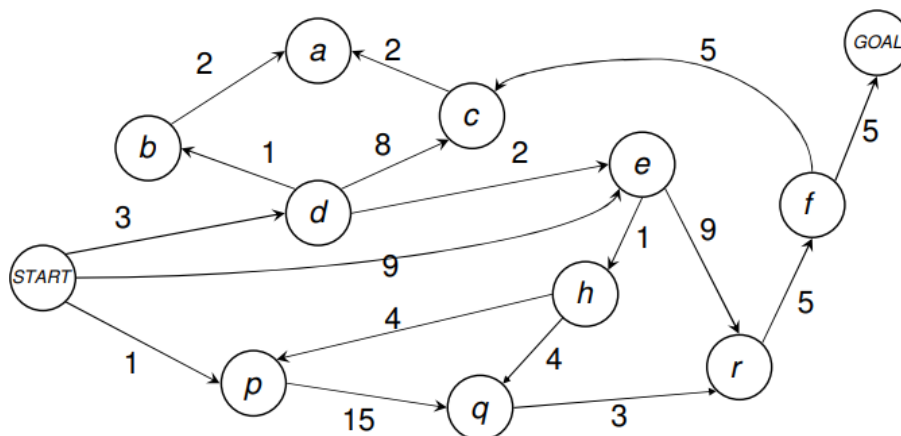


Fig. 5. Tree 2

Task 5. Using **Depth limited search**, implement the methods `public Node execute(Node tree, String goal, int limitedDepth):`

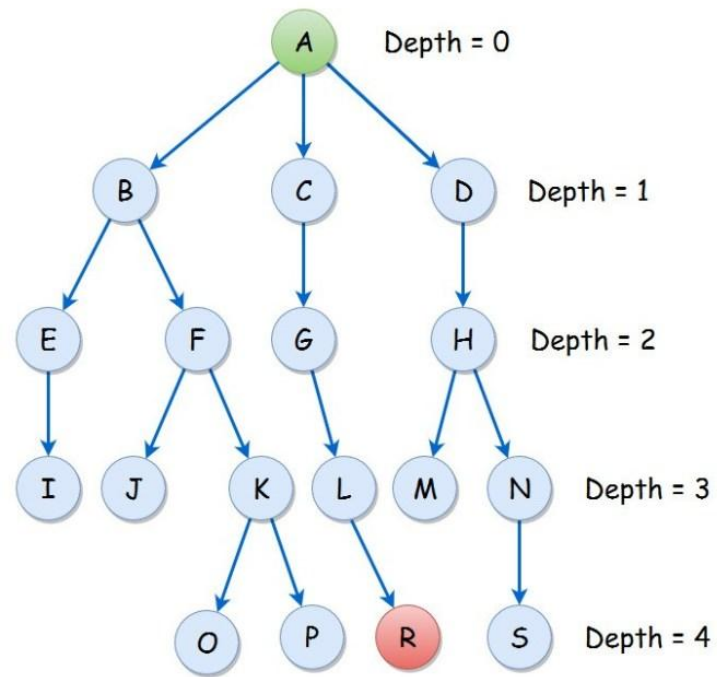


Fig. 6. Tree 3