Lab #2, 3: Uninformed Search Algorithms

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

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

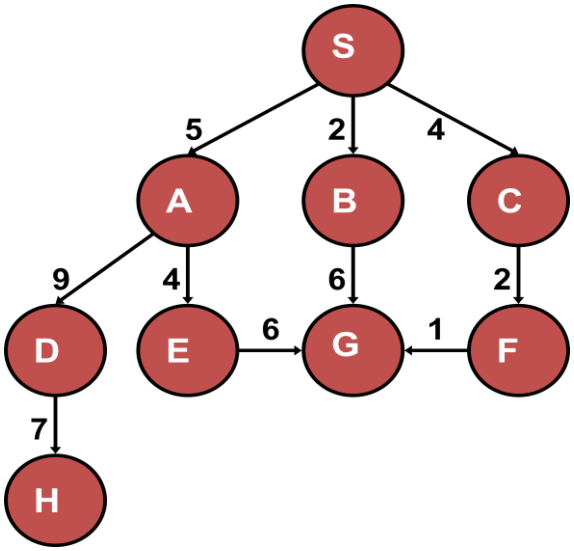

Fig. 1. Tree 1

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

```
public class Node implements Comparable<Node> {
	private String label;
	private Node parent; // for printing the path from the start
node to the goal node
	private double pathCost;// from the root node to this node
	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 state space 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 root, String goal);// find the path from root node
//to the goal node

public Node execute(Node root, 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 frontier is a queue:

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

For DepthFirstSearchAlgo, frontier is a stack:

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

Pseudocode for searching algorithms:

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  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
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Fig. 2. Graph search

==========

Task 1: Implement execute(Node root, 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 (**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 nodeB = new Node("B");
		Node nodeC = new Node("C");	Node nodeD = new Node("D");
		Node nodeE = new Node("E");	Node nodeF = new Node("F");
		Node nodeG = new Node("G");	Node nodeH = new Node("H");

		nodeS.addEdge(nodeA, 5);	nodeS.addEdge(nodeB, 2);
		nodeS.addEdge(nodeC, 4);	nodeA.addEdge(nodeD, 9);
		nodeA.addEdge(nodeE, 4);	nodeB.addEdge(nodeG, 6);

		nodeC.addEdge(nodeF, 2);	nodeD.addEdge(nodeH, 7);

		nodeE.addEdge(nodeG, 6);	nodeF.addEdge(nodeG, 1);

		ISearchAlgo algo1 = new BreadthFirstSearchAlgo();

		Node result = algo1.execute(nodeS, "G");
```

==========

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

==========

Task 3: Implement execute(Node root, 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.

Pseudocode for UCS (graph search):

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher PATH-COST **then**
      replace that *frontier* node with *child*

Fig. 3. Uniform cost search

==========

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

==========

Additional task: test all implementations with the following tree:

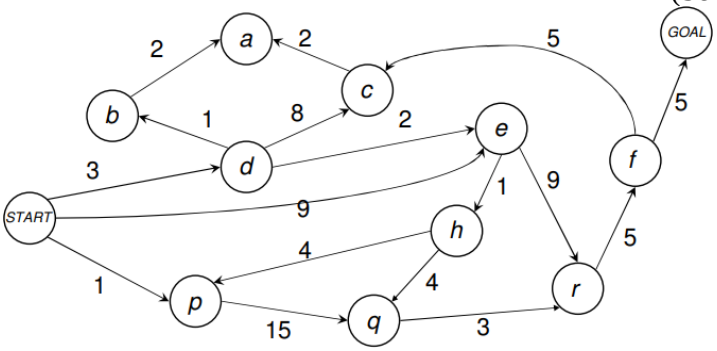

Fig. 4. Tree 2

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

Pseudocode:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.16 A recursive implementation of depth-limited tree search.

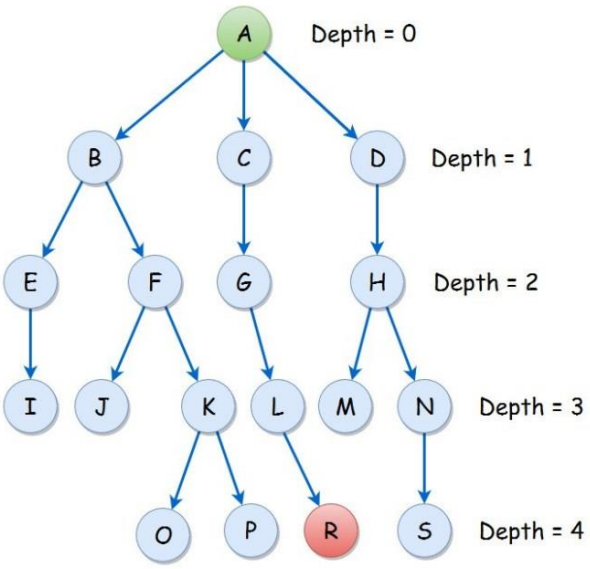

Fig. 5. Tree 3