Lab #2, 3: Uninformed Search Algorithms

The main aim of these labs is to deal with uninformed search algorithms using the graph search and tree search approaches. For a given state space with nodes and weights of each edge described in the following figure:
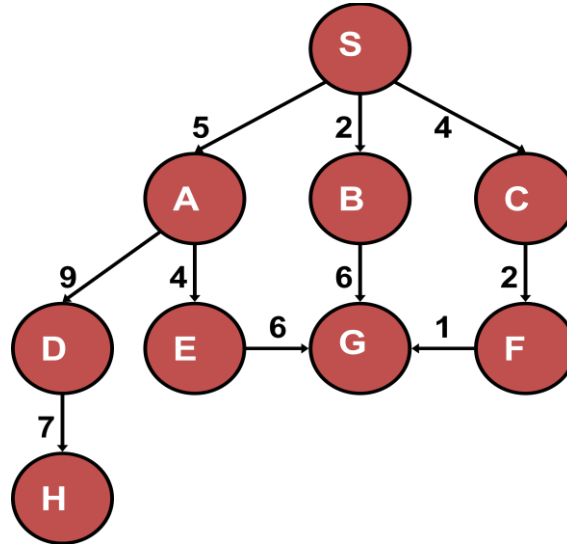

Fig. 1. State space 1

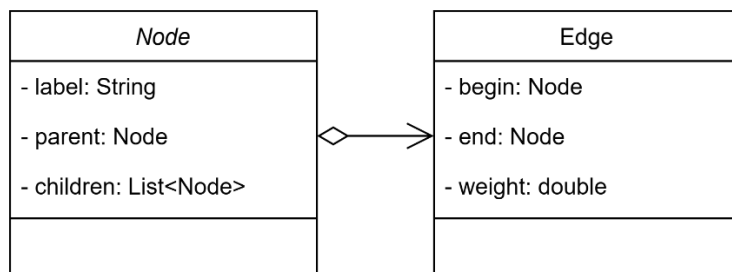The infrastructure of search algorithms is described in the following figure:

Fig 2. class diagram

Each Node includes:

- label
- parent
- children presented as edges.

Each edge includes

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

- Method: **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 space.**

Notice that, the traversal of nodes is in alphabetical order.

===

For **BreadthFirstSearchAlgo**, the frontier is a queue:

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

For **DepthFirstSearchAlgo**, the frontier is a 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

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. 3. Tree Search and Graph Search

===

Task 1: Implement execute(Node root, String goal) in **BreadthFirstSearchAlgo.java** and **DepthFirstSearchAlgo.java** using the graph search approach.

===

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);

        BreadthFirstSearchAlgo 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** using the graph search approach.

==

Task 3: Modify the implemented methods in **Task 1** and **Task 2** using the tree search approach.

==

Task 4: 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 to compare two nodes based on their path cost. 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. 4. Uniform cost search

===

Task 5: Similar to Task 2, implement method execute(Node root, String start, String goal) in **UniformCostSearchAlgo.java**

===

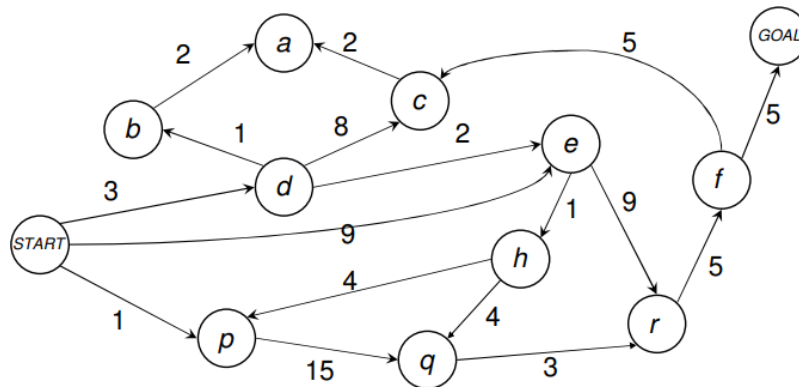**Additional task**: test all implementations with the following tree:

Fig. 5. State space 2

Task 6. 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
```

Fig. 6. Depth-limited tree search
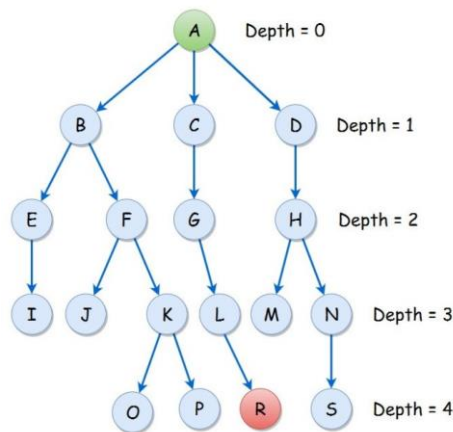

Fig. 7. State space 3