FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / ~~We*~~, the undersigned, declare that the [assignment / ~~Assigned Practical Task report / Final Year Project report~~] submitted is my / ~~our*~~ work, except where acknowledged and referenced.

I / ~~We*~~ understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

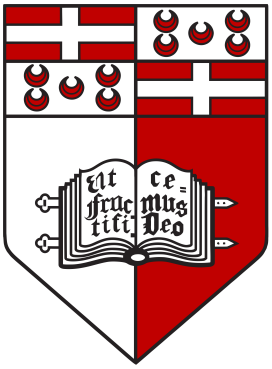
(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Miguel Dingli	M. Dingli
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS 2210	ICS2210 - Data Structures and Algorithms 2 - Course Project 2017
Course Code	Title of work submitted

26/05/2017
Date

This page intentionally left blank



ICS2210 - Data Structures and Algorithms 2
-
Course Project 2017

Full name: Miguel Dingli
I.D: 49997M
Course: B.Sc. (Hons) in Computing Science

Table of Contents

1 – Introduction

To start with, a general overview of the project's structure and organization will be presented. This will include the presentation of the Graph class, which makes use of the Node and Edge classes, the Search Algorithm class itself and all of its various sub-classes, and any other helper methods and classes.

The language chosen for this assignment is Java. An additional general note is that a decision was made to have *public final* class members instead of using getters and setters. This was done specifically so that the sections of code using these members are simplified. This decision does not affect the algorithms' space or time complexity as such. If the need arises, getters and setters could easily be added.

2 – The Graph Class

2.1 – The Graph Representation

The graph representation chosen for this assignment is the adjacency list graph representation. The main reason behind this decision is the fact that any graph that will be generated in this project is a sparse graph since each node can at most only have four adjacent nodes and the least amount of nodes in the graph is ten.

If an adjacency matrix were used, this would mean that, at most, only four out of ten entries in the matrix will have a value. In the worst case, this goes down to only one out of twelve entries. This led to the decision of instead using the adjacency list representation.

The second reason behind this decision comes from the adjacency list's ability to obtain the list of adjacent nodes for a given node in constant time. While searching, it is almost never the case that we want to find out whether there is an edge between two specific nodes. However, it is very common to require traversing a node's adjacent nodes. Hence, the adjacency matrix's advantage is nullified and the significance of the adjacency list's advantage makes it the ideal choice.

2.2 – Implementing the Representation

Figure 1 (in the next page) shows a UML representation of the Graph, Node, and Edge classes. Following the decision to use an adjacency lists, the Graph class contains an array of nodes, each of which (with reference to the Node class) has an array of edges that lead to adjacent nodes. The point of having a dedicated Edge class is that the real-valued edge weight can be stored alongside the destination node, instead of creating a separate data structure to store the weights. This is also the reason behind storing the *visited* state in the Node itself. The Graph class also keeps track of the start nodes and goal nodes.

The Graph constructor has the responsibility for initializing the graph based on the strict requirements set in the assignment. The private method *existsEdgeBetween(…)* is used only during initialization and indicates whether an edge exists between two nodes. The *resetVisitedNodes() method* is used to set the nodes' *visited* boolean value to false so that the graph can be re-used after a search is complete. Besides this, the method is also called in the iterative-deepening searches when re-starting a search at a different depth. The *isGoalNode()* method simply checks if the argument node is one of the nodes in the array of three goal nodes. Finally, *outputDetails()* is called to output the number of nodes, number of edges, and the adjacency list.

One last note is that the Node class uses an integer ID that is very important since besides serving as a unique node identifier, it is sometimes used as an index for arrays with a size equal to the number of nodes in the graph. Examples of this will be discussed later on. The *resetGlobalID()* method must be called by the graph before instantiating the nodes that will collectively make up the graph.

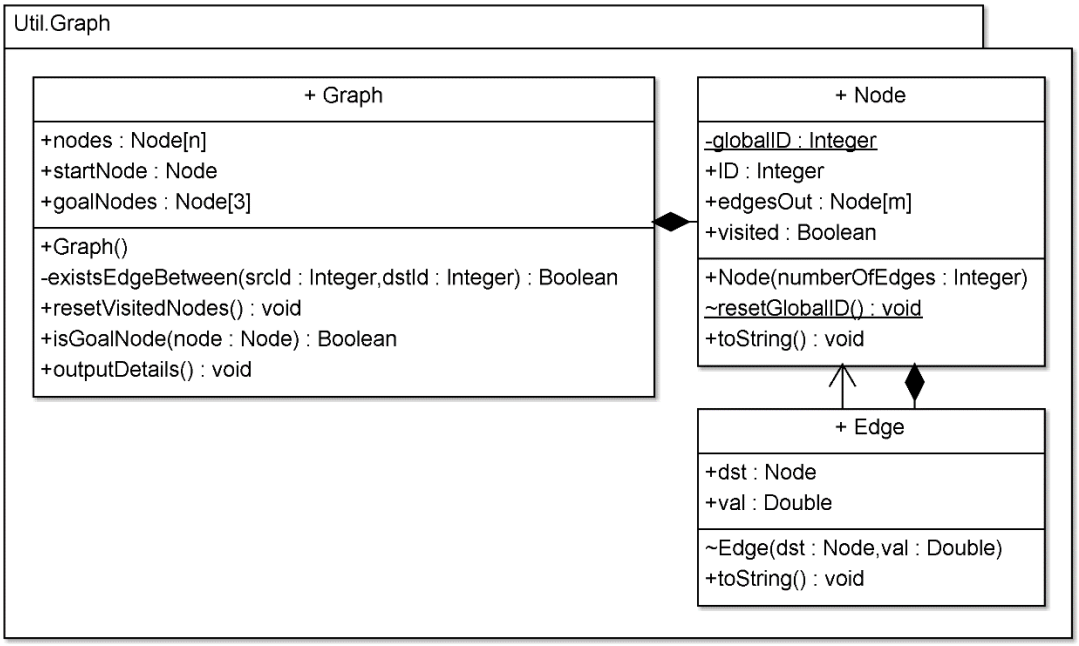


Figure 1 – UML of Util.Graph Package

2.3 – The Graph Constructor

The Graph constructor (presented in Listing 1) will now be discussed. As previously stated, the constructor is mainly responsible for the initialization of the graph's various aspects, based on the set requirements.

```
public Graph() {

    Node.resetGlobalID(); // Reset Node's global ID

    // "Create a graph containing n nodes..."
    nodes = new Node[UtilityMethods.randInt(10, 12)];
    for (int i = 0; i < nodes.length; i++) {
        nodes[i] = new Node(UtilityMethods.randInt(1, 4));
    }

    // "Designate a random node as the starting node."
    startNode = nodes[UtilityMethods.randInt(0, nodes.length - 1)];

    // "Designate 3 distinct random nodes as goal nodes. "
    goalNodes = new Node[]{null, null, null};
    for (int i = 0; i < 3; /*no increment*/) {
        final Node gNode = nodes[UtilityMethods.randInt(0, nodes.length - 1)];
        if (gNode != goalNodes[(i + 1) % 3] && gNode != goalNodes[(i + 2) % 3]) {
            goalNodes[i] = gNode;
            i++;
        }
    }

    // "For each node, create m distinct directed edges..."
    for (final Node from : nodes) {
        for (int edgeIndex = 0; edgeIndex < from.edgesOut.length; /*no increment*/) {
            final int dstIndex = UtilityMethods.randInt(0, nodes.length - 1);
            if (!existsEdgeBetween(from.ID, dstIndex)) {
                from.edgesOut[edgeIndex] = new Edge(nodes[dstIndex], Math.random());
                edgeIndex++;
            }
        }
    }
}
```

Listing 1 – Graph Class Constructor

Following from the discussion regarding the Node class' global ID, the first thing done in the Graph constructor is to call this method to set the global ID to zero. Throughout the rest of the constructor, a static method called *randInt(…)* will be used to generate a random number in the range based on the provided inclusive limits. This method is defined in a class called *UtilityMethods* that contains other helper methods.

The array of nodes is initialized with a size from 10 to 12 (as per the requirement). Each node is given a random integer with a size from 1 to 4, corresponding to the number of outward edges (as per the requirement). Next, again as per the requirement, a single random start node and three random goal nodes are designated. For each potential goal node, this is compared to the other goal nodes, so that there are no duplicate goal nodes.

Finally, the edges array for each node in the *nodes* array is now populated with distinct nodes. The *existsEdgeBetween(…)* method is used to enforce the fact that there should not be multiple edges between the same ordered pairs of nodes. Edges are given a random weight between 0 and 1 using *Math.random()*.

3 – The Search Algorithms

3.1 – Introduction

The abstract Search Algorithm class and its subclasses (Figure 2) will now be discussed. Each subclass corresponds to a unique search algorithm; DFS corresponds to Depth-First Search, BFS corresponds to Breadth-First Search, and so on. The only abstract method in the abstract Search Algorithm class is the *startSearch(…)* method.

Upon instantiation of a search algorithm, the subclass constructors call the superclass' constructor in order to set the graph that will be searched. To perform the search, SearchAlgorithm's *performSearch()* method is called, which calls *preSearch()*, *startSearch(…)*, and *postSearch()* in that order. The two methods *preSearch()* and *postSearch()* are optionally overridden by concrete search algorithms to add extra behaviour before or after a search, respectively. Behaviour before a search usually involves initialization of any members in the class, while behaviour after a search usually consists of calculating the cost and forming the path, if this was not directly done during the search itself. The graph's start node is passed to *startSearch(…)* as an argument.

SearchAlgorithm also has *cost* and *path* which are expected to be set by the concrete search algorithms at the end of a search. The *prependToPath(…)* adds a node to the start of a path. This is useful since, in many cases, the path is formed by considering the goal node found and traversing in reverse until the start node is found. The *goalNodeFound()* method simply checks whether *theReachedGoalNode* is null or was actually set a node. Finally, the *outputResults()* method is used to output the search results (i.e. path and cost).

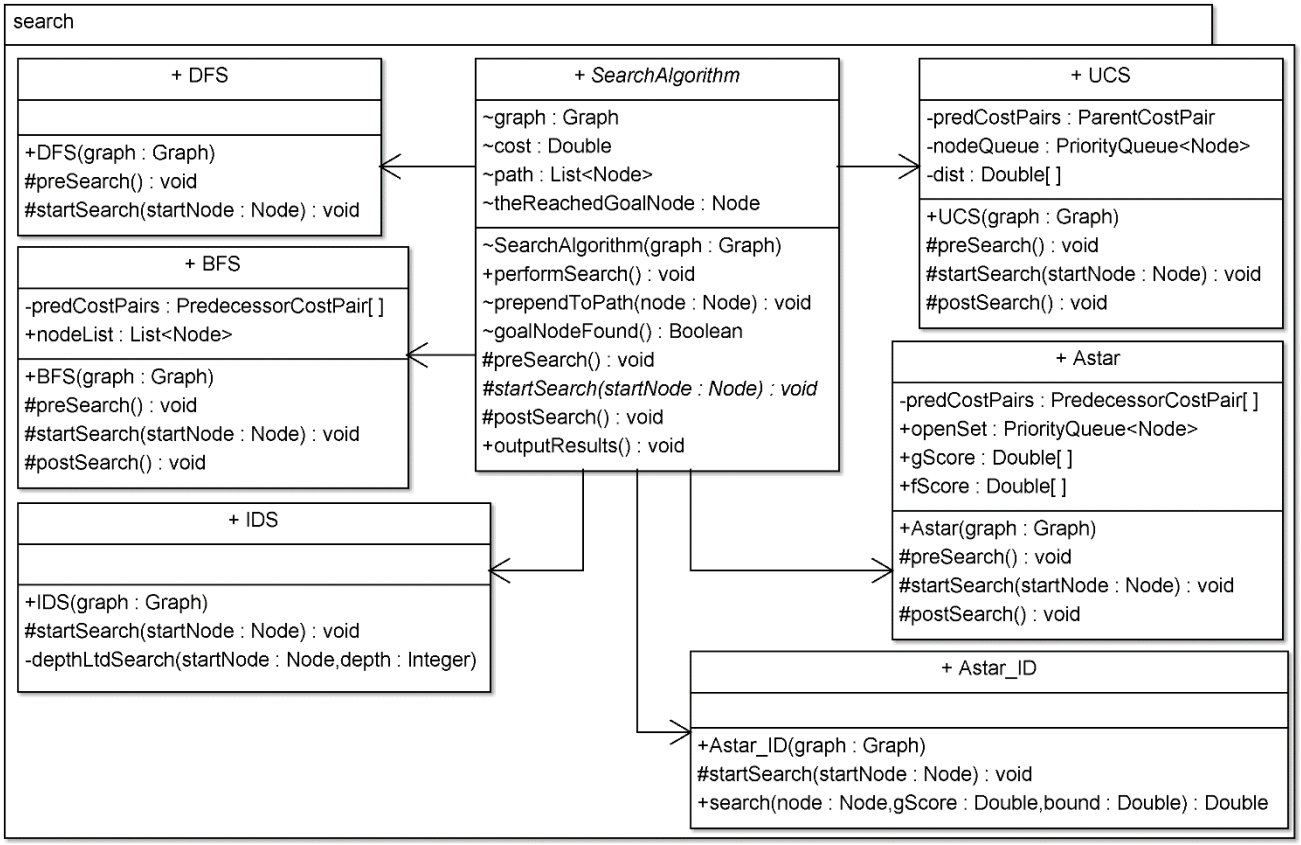


Figure 2 – UML of search Package

The search algorithms will now be individually discussed, starting from the Depth-First Search.

3.2 – Search Algorithms

In general, a search algorithm discussion will first focus on the *preSearch()* and *postSearch()* methods (if applicable), followed by any additional methods/features. The *startSearch(…)* method will then be discussed. The algorithm's time and space complexity, completeness and admissibility will then be discussed.

In general, when discussing the time and space complexity of the algorithms, $\boldsymbol{n}$ will refer to the total number of nodes, $\boldsymbol{m}$ will refer to the total number of edges, $\boldsymbol{b}$ will refer to the worst case branching factor, and $\boldsymbol{d}$ will refer to the worst case depth.

3.2.1 – Depth-First Search

The Depth-First Search search algorithm class (Listing 2) was based on **[1]**. It overrides the *preSearch() and startSearch(…)* methods but not *postSearch()*. The ***preSearch()*** override is simply used to set the graph's start node to visited. This is done since during the search a node will only be set to visited when it is about to be visited, meaning that the start node would otherwise never be set to visited, which could cause problems.

```
public class DFS extends SearchAlgorithm {

    public DFS(final Graph graph) {
        super(graph);
    }

    @Override
    protected void preSearch() {
        graph.startNode.visited = true;
    }

    @Override
    protected void startSearch(final Node startNode) {

        if (graph.isGoalNode(startNode)) {
            theReachedGoalNode = startNode;
            path = new ArrayList<>();
            path.add(theReachedGoalNode);
            return;
        }

        for (final Edge e : startNode.edgesOut) {
            final Node n = e.dst;
            if (!n.visited) {
                n.visited = true;
                startSearch(n);
                if (goalNodeFound()) {
                    prependToPath(startNode);
                    cost += e.val;
                    return;
                }
            }
        }
    }
}
```

Listing 2 – The DFS Search Algorithm Class

DFS StartSearch Method

The ***startSearch(…)*** method implements the actual depth-first search algorithm. Given the graph's root as the starting node, the algorithm searches by prioritizing deep traversals over shallow traversals, meaning that it first traverses all nodes reachable from the first adjacent node before moving on to the second adjacent node. The effect of performing a deep search before moving on to other adjacent nodes is achieved by recursively calling the *startSearch* method on adjacent nodes sequentially.

The immediately obvious base case in the recursive calls is the case where the node argument is a goal node. During the "unwinding" stage of the recursion, if the goal node was found as a result of a recursive call, the remaining adjacent nodes are skipped by issuing a *return*. This speeds up the unwinding towards the start node. The unwinding from the found goal node to the start node is taken as an opportunity to form the path that leads from the start node to the goal node and accumulate its cost, so that these can be output later on.

Another base case is when all nodes reachable from the start node have been visited, but none of these is a goal node. In this case, there are no more unvisited nodes that can be visited and the recursion simply ends up unwinding without ever setting *theReachedGoalNode*, the *path*, and the *cost*.

DFS Time Complexity

For the time complexity of the algorithm, consider a general node. The number of adjacent nodes is equivalent to the branching factor b. For each node, all of these b adjacent nodes need to be traversed, irrelevant of whether they have been visited yet or not. In the worst case, none of these adjacent nodes is a goal node.

Now, for each of the b nodes, this process needs to be repeated, meaning that a further b nodes need to be traversed for every one of the original b nodes. This means that the total number of nodes explored increases to $b + b^2$. Recursively, each of the b^2 nodes has a further b adjacent nodes, and so on.

Clearly, the number of nodes checked is increasing polynomially based on the depth d of the search. In fact, in the general worst case, the time complexity will be $O(b + b^2 + b^3 + \cdots + b^d)$. Eliminating lower-order terms, this time complexity due to the searching can be expressed as $\boldsymbol{O(b^d)}$.

However, the above time complexity fails to account for the time complexity of adding the nodes to path from start to finish after a goal node has been found. Since the number of nodes that need to be added to the path is equivalent to the depth d reached, and since each add operation is essentially linear in time, then a more accurate time complexity for the overall search algorithm can be expressed as $\boldsymbol{O(b^d + d^2)}$.

DFS Space Complexity

For the space complexity of the algorithm, one needs to observe that the maximum number of nodes in consideration at any one point in time is equivalent to the depth d reached. The depth-first search only needs to store the nodes in the path of nodes in consideration, and hence the space complexity of this is $\boldsymbol{O(d)}$. Note that each node does not have to store the adjacent nodes; the array of outbound edges is merely referenced.

However, the space complexity needs to also take into consideration that **each** node needs to store a *visited* state to indicate whether it was visited yet or not. This feature has a space complexity of $\boldsymbol{O(n)}$. Therefore, excluding constant-size space aspects, the overall space complexity of the algorithm is a linear $\boldsymbol{O(n + d)}$.

DFS Completeness

Since the algorithm is able to reach all of the nodes that are reachable from the start node, then if at least one goal node is reachable, the algorithm will find it. This is only made possible because cycles are eliminated by keeping track of which nodes have been visited and which have not. Otherwise, the algorithm could get stuck by searching along a cycle, which means that no goal node would be found.

DFS Admissibility

Depth-first search is not admissible. This can be deduced from the fact that the algorithm does not consider the edge weight when searching. This means that it essentially picks the first goal node that it finds, irrespective of whether it is the optimal goal or not.

3.2.2 – Breadth-First Search

The Breadth-First Search search algorithm class (Listing 3) was based on **[2]**. It overrides the *preSearch()*, *startSearch(…)*, and *postSearch()* methods. It also makes use of a utility class called *PredecessorCostPair*, which stores a node (corresponding to the predecessor of a node) and a double *cost* (corresponding to the weight of the edge between the node and its predecessor).

```
public class BFS extends SearchAlgorithm {

    private PredecessorCostPair predCostPairs[];
    private List<Node> nodeList;

    public BFS(final Graph graph) {
        super(graph);
    }

    @Override
    protected void preSearch() {

        // Initializing predecessor-cost-pair array
        predCostPairs = new PredecessorCostPair[graph.nodes.length];
        for (int i = 0; i < predCostPairs.length; i++) {
            predCostPairs[i] = null;
        }

        // Initializing the node list
        nodeList = new ArrayList<>();
        nodeList.add(graph.startNode);
    }

    @Override
    protected void startSearch(final Node startNode) {

        graph.startNode.visited = true;
        if (graph.isGoalNode(startNode)) {
            theReachedGoalNode = startNode;
            return;
        }

        while (!nodeList.isEmpty()) {
            final Node head = nodeList.remove(0);
            for (final Edge e : head.edgesOut) {
                final Node n = e.dst;
                if (!n.visited) {
                    predCostPairs[n.ID] = new PredecessorCostPair(head, e.val);
                    if (graph.isGoalNode(n)) {
                        theReachedGoalNode = n;
                        return;
                    }
                    nodeList.add(n);
                    n.visited = true;
                }
            }
        }
    }

    @Override
    protected void postSearch() {

        final PathCostPair pathCost = UtilityMethods.toPathCostPair(
                predCostPairs, theReachedGoalNode);
        path = pathCost.path;
        cost = pathCost.cost;
    }
}
```

Listing 3 – The BFS Search Algorithm Class

The ***preSearch()*** method initializes an array of *PredecessorCostPair*s with null values which will be indexed by node IDs. During the search, when a node is set to be visited, the array element indexed by its ID is set as a new *PredecessorCostPair* with the node's predecessor and the weight of the edge in between. A list of nodes is also initialized with the start node. The ***postSearch()*** method uses the array to trace back the path from a goal node to the start to form the final list of nodes that will be output as a result. The stored edge weight is used to calculate the cost. The need for such an array will become clearer once the algorithm is described.

BFS StartSearch Method

Similar to the DFS, the ***startSearch(…)*** method implements the actual breadth-first search algorithm. Given the graph's root as the starting node, the algorithm searches by prioritizing shallow traversals as opposed to deep traversals, meaning that it first traverses all the adjacent nodes of a particular node before searching nodes reachable from the adjacent nodes themselves. Note that the algorithm was implemented iteratively.

The effect of performing a shallow search before moving on to "deeper" nodes is achieved by, for each node, traversing each adjacent node (in the process checking whether one is a goal node) and delaying the searching of deeper nodes by adding the checked adjacent node to the list of nodes, *nodesList*. Once the adjacent nodes are exhausted, the process loops and the first node in the nodes list is popped. This node's adjacent nodes are now traversed, and so on. Note that a node is set to visited immediately after it is added to the nodes list, and not when it is popped. This prevents a node from being added more than once to the nodes list.

The search terminates immediately if a goal node is found. Otherwise, all reachable adjacent nodes that were added to the nodes list have to be popped and checked until all reachable nodes are visited and no more nodes can be added to the nodes list, resulting in an empty nodes list which breaks the loop condition.

Consider once more the array of predecessor-cost pairs. When a goal node is found, there is insufficient information to deduce the path from the start node to the goal node. This is mainly due to the fact that the order in which nodes are traversed does not correspond to the order in which the nodes are connected with each other. This is precisely why the predecessor of each visited node needs to be recorded during the search.

BFS Time Complexity

The time complexity due to the searching deduced in a similar way to the depth-first search. For a general node, the number of further nodes that need to be traversed is equal to the branching factor b. For each of the b nodes added to the nodes list a further b nodes need to be traversed, increasing the total number of nodes explored increases to $b + b^2$. Furthermore, each of the b^2 nodes has b adjacent nodes, and so on. Now, at depth d, the nodes list will contain approximately b^d nodes, meaning that adding to nodes list in linear time using the ArrayList's *add* method will take a further $O(b^d)$ time. Since this is done for all of the $O(b^d)$ nodes, the time complexity due to the searching ends up increasing to $O(b^d \times b^d) = \boldsymbol{O(b^{2d})}$.

The above does not account for the time spent forming the path in the *postSearch()* method. Since the number of nodes to be added to the path is equivalent to the depth d reached, and since each *add* is linear in time, then a more accurate time complexity for the overall search algorithm can be expressed as $\boldsymbol{O(b^{2d} + d^2)}$.

Kindly refer to the space complexity discussion in the next page.

BFS Space Complexity

For the space complexity of the algorithm, one needs to observe that essentially only one node is in consideration at any one point in time, unlike depth-first search. The space taken up during the search is mostly due to the *predCostPairs* array, with a size of $O(n)$, and the nodes list, with a worst case size of $O(b^d)$.

However, the space complexity needs to also take into consideration that **each** node needs to store a *visited* state to indicate whether it was visited yet or not. This feature has a space complexity of $\boldsymbol{O(n)}$. Therefore, the overall space complexity of the algorithm is $O(b^d + n + n) = \boldsymbol{O(b^d + n)}$.

BFS Completeness

Since the algorithm is able to reach all of the nodes that are reachable from the start node, then if at least one goal node is reachable, the algorithm will find it. Unlike depth-first search, despite the fact that we still keep track of which nodes have been visited and which have not, this is not the aspect that makes the breads-first search complete. What makes BFS complete is the fact that rather than exploring along a single path of nodes, it instead searches all paths from the start node simultaneously at the same incremental depth.

BFS Admissibility

Depth-first search is not admissible. This can be deduced from the fact that, like depth-first search, the algorithm does not consider the edge weight when searching. This means that it essentially picks the first goal node that it finds, irrespective of whether it is the optimal goal or not.

However, what is known is that the goal node found will be an optimal goal with respect to the distance from the start node, given that BFS only searches at a particular depth if all shallower nodes have been explored. In fact, if the edge weights were constant, then the goal found is guaranteed to be an optimal goal.

3.2.3 – Iterative-Deepening Search

The Iterative-Deepening Search will now be discussed. The IDS search algorithm class (Listing 4) based on **[3]**, overrides only the *startSearch(…)* method but implements a *depthLtdSearch(…)* method which is essentially identical to the *startSearch(…)* implementation in *DFS* but with an added depth argument and condition.

```
public class IDS extends SearchAlgorithm {

    public IDS(final Graph graph) {
        super(graph);
    }

    @Override
    protected void startSearch(final Node startNode) {

        for (int depth = 0; depth < graph.nodes.length; depth++) {
            startNode.visited = true;
            depthLtdSearch(startNode, depth); // depth-limited search
            if (goalNodeFound()) {
                break;
            }
            graph.resetVisitedNodes();
        }
    }

    private void depthLtdSearch(final Node start, final int depth) {

        if (depth == 0 && graph.isGoalNode(start)) {
            theReachedGoalNode = start;
            path = new ArrayList<>();
            path.add(theReachedGoalNode);
        } else if (depth > 0) {
            for (final Edge e : start.edgesOut) {
                final Node n = e.dst;
                if (!n.visited) {
                    n.visited = true;
                    depthLtdSearch(n, depth - 1);
                    if (goalNodeFound()) {
                        prependToPath(start);
                        cost += e.val;
                        return;
                    }
                }
            }
        }
    }
}
```

Listing 4 – The IDS Search Algorithm Class

IDS StartSearch Method

In this case, the ***startSearch(…)*** method consists of a loop with an increasing depth as the loop counter, where the maximum depth is set as the number of nodes minus one. At each depth value, the start node is set as visited for the same reason as to why it was set as visited in the *preSearch()* of depth-first search, and the depth-limited search method is called with the depth as an argument along with the start node.

Essentially, the depth-limited search algorithm works exactly like depth-first search but is not allowed to search beyond the depth specified when it is called. Starting the depth at zero, this means that the algorithm will initially only search the start node. Moving on to a depth of one, the start node and its adjacent nodes will be searched. An increase depth means that the adjacent nodes of adjacent nodes can be searched, and so on.

Overall, when taking all the depth values into consideration, the algorithm actually searches in a breadth-first fashion since nodes at a particular depth are only searched if a previous call to the depth-limited search with

less depth allowance did not result in a found goal node. This means that the algorithm inherits advantages from both depth-first search and breadth-first search. From the DFS aspect, we will be able to figure out the path to a goal node without storing predecessor-cost pairs, while from the BFS aspect, we will be able to guarantee that the found goal node will be the closest to the start node, with respect to the number of edges.

Note that the depth is decremented with each recursive call, so that the goal node base case with a condition for zero depth can make sense. Like the depth-first search, the alternate base case is when the goal node is not found and the method returns. However, in this case, this merely means that a goal node was not found at the specified initial depth argument. It is confirmed that no goal node was found when the depth-limited search is supplied with the maximum depth and still no goal node is found.

IDS Time Complexity

Since the depth-limited search method is modelled after the depth-first search algorithm, then the time complexity due to this method is $\boldsymbol{O(b^d)}$, considering the worst case depth d. However, this method is called various times with different initial depth values.

Consider the depth-limited search called twice for two unique depths d_1 and d_2 where $d_1 + 1 = d_2$. This results in time complexities of $O(b^{d_1})$ and $O(b^{d_2})$, respectively. Summing this gives $O(b^{d_1} + b^{d_2})$. However, since $d_1 + 1 = d_2$, b^{d_1} is a lower-order term. This means that the overall complexity is actually $O(b^{d_2})$. If this is generalised, then the overall time complexity can be first expressed as a sum of lower-order complexities $O(b^0 + b^1 + \cdots + b^{d-1} + b^d)$, i.e. with depth values from 0 up to d, and then simplified to $\boldsymbol{O(b^d)}$.

Similar to the depth-first search, the above complexity does not take into consideration the time complexity of adding nodes to the path that will be presented as a result. Applying the same reasoning as that for the depth-first search, the overall time-complexity of the search algorithm can be assumed to be $\boldsymbol{O(b^d + d^2)}$.

IDS Space Complexity

Again, similar to DFS, the maximum number of nodes in consideration at any one point in time is equivalent to the depth d reached. Assuming the worst case depth, this results in a space complexity of $O(d)$. Note that a run of the depth-limited search at a lower depth simply produces coefficients and lower-order terms, meaning that the $\boldsymbol{O(d)}$ space complexity can easily be assumed to cater for the *startSearch(…)* part.

However, once more, the space complexity needs to take into consideration that **each** node needs to store a *visited* state. This feature has a space complexity of $\boldsymbol{O(n)}$. Therefore, excluding constant-size space aspects, the overall space complexity of the algorithm can be assumed to be a linear $\boldsymbol{O(n + d)}$.

IDS Completeness

The completeness of Iterative-Deepening Search, like the complexities, can be compared to that of depth-first search and breadth-first search. Like both DFS and BFS, since the algorithm is able to reach all of the nodes that are reachable from the start node, then if at least one goal node is reachable, the algorithm will find it.

IDS Admissibility

Since IDS, as previously stated, searches in a breadth-first fashion, it inherits the feature of BFS that if the edge weights were constant, then the goal node found is guaranteed to be the optimal goal, and hence IDS would be admissible. However, this is not the case, and so iterative-deepening search is not admissible.

3.2.4 – Uniform-Cost Search

The Uniform-Cost Search (UCS) search algorithm class is presented below (Listing 5) and was based on the course notes. This class overrides all three of the search methods: *preSearch()*, *startSearch(…)*, and *postSearch()*. It also has an array of predecessor-cost pairs (like BFS), a node queue, and an array of distances. Furthermore, it also has a private Comparator class. These will be discussed in the coming discussions.

```
public class UCS extends SearchAlgorithm {

    private PredecessorCostPair predCostPairs[];
    private PriorityQueue<Node> nodeQueue;
    private double dist[];

    public UCS(final Graph graph) {
        super(graph);
    }

    @Override
    protected void preSearch() {

        // Initializing predecessor-cost-pair array
        predCostPairs = new PredecessorCostPair[graph.nodes.length];
        for (int i = 0; i < predCostPairs.length; i++) {
            predCostPairs[i] = null;
        }

        // Initialize node queue
        nodeQueue = new PriorityQueue<>(new DistanceComparator());
        nodeQueue.addAll(Arrays.asList(graph.nodes));

        // Initializing distances array
        dist = new double[graph.nodes.length];
        for (int i = 0; i < dist.length; i++) {
            dist[i] = Double.MAX_VALUE;
        }
        dist[graph.startNode.ID] = 0;
    }

    @Override
    protected void startSearch(final Node startNode) {

        if (graph.isGoalNode(startNode)) {
            theReachedGoalNode = startNode;
            return;
        }
        while (nodeQueue.size() != 0) {
            final Node head = nodeQueue.getHighestPriority();
            head.visited = true;
            for (final Edge e : head.edgesOut) {
                final Node n = e.dst;
                if (!n.visited && (dist[n.ID] > dist[head.ID] + e.val)) {
                    predCostPairs[n.ID] = new PredecessorCostPair(head, e.val);
                    dist[n.ID] = dist[head.ID] + e.val;
                    if (graph.isGoalNode(n) && (!goalNodeFound()
                            || dist[n.ID] < dist[theReachedGoalNode.ID])) {
                        theReachedGoalNode = n;
                    }
                }
            }
        }
    }

    @Override
    protected void postSearch() {

        final PathCostPair pathCost = UtilityMethods.toPathCostPair(
                predCostPairs, theReachedGoalNode);
```

```
            path = pathCost.path;
            cost = pathCost.cost;
        }

        private class DistanceComparator implements Comparator<Node> {
            @Override
            public int compare(final Node n1, final Node n2) {
                // Result inverted since smaller distance is higher priority
                return -Double.compare(dist[n1.ID], dist[n2.ID]);
            }
        }
}
```

Listing 5 – The UCS Search Algorithm Class

The private variables will be discussed first. The array of predecessor-cost pairs **predCostPairs** will serve the same purpose as the array in the BFS. Without this array, once the goal node is found, not enough information will be available to deduce the path from the start node to the found goal node. The array of distances **dist** will be indexed by the ID of nodes and stores the estimate distance from the start node to the particular node. This information will be constantly updated and will be used by the node queue.

The node queue **nodeQueue** uses a custom implementation of a priority queue which instead of using the comparator during an enqueue, it uses the comparator with each dequeue. The purpose of this priority queue of vertices will be to always dequeue the node with the least distance value based on the **dist** array. In fact, a custom comparator **DistanceComparator** was implemented for this purpose. This comparator uses the latest distance values from the **dist** array so that nodes can be dequeued based on their distance. Since a priority queue by default gives the element with the highest key value, the comparator reverses the result of the distance comparison so that the queue instead gives the element with the lowest key (i.e. distance) value.

Moving on to the overridden methods, the ***preSearch()*** override initializes the arrays and queue. The queue is filled with all nodes, while the *dist* array is initialized with maximum double values which symbolise infinity. However, the distance corresponding to the start node is set to zero. This is because we know that the distance of the start node from itself has to be zero, however we have yet to calculate the distance of other nodes. The ***postSearch()***, like in BFS, simply uses the *predCostPairs* array to obtain the path and cost results.

UCS StartSearch Method

The actual algorithm will now be discussed. Uniform-Cost Search starts by dequeuing a node from the node queue. Since the start node has a distance value of zero, this is the first dequeued node.

Once a node is dequeued, it is set as visited and its adjacent nods are traversed. For each adjacent node, if the distance up to their *predecessor* plus the weight of the edge leading to the adjacent node is smaller than the current distance value of the adjacent node itself, then the adjacent node's distance value is updated to this value. This essentially indicates that a new shorter path was found from the start node to the particular node.

In this case, the predecessor-cost pair for the particular node is also updated. This may have been set before, but since a new shorter path was found, this is preferred. Additionally, it is checked whether the node is a goal node. If so, and if either no previous goal node was found or the distance to the previously found goal node is greater than the distance to the newly found goal node, the node is set as the new goal node.

Overall, what this algorithm tries to do is to calculate the distance from the start node to nodes as it goes along and uses this to obtain the shortest possible path to a goal node. An important note is that the word "distance" is actually referring to the cost of the edges along a path. "Distance" is being used since this algorithm is theoretically meant to be used to find the *shortest* path.

UCS Time Complexity

The time complexity of the implemented UCS algorithm will now be discussed. First of all, consider the node queue. The node queue will hold all of the nodes in the graph and the algorithm only terminates once the node queue is emptied. This means that the time complexity due to the loop of node queue dequeues is $\boldsymbol{O(n)}$.

For each of the n nodes in the queue, each one of the node's adjacent nods are checked, irrespective of whether the adjacent node has been visited or not. Since the number of adjacent nodes is equivalent to the branching factor b, the time complexity due to the checking of adjacent nodes for each node is $\boldsymbol{O(nb)}$.

The operations done for each node that satisfied the non-visited and distance conditions essentially do not depend on the size of the input since this involves the setting of variables and an addition. Hence, these operations can be ignored in the time complexity. However, like in breadth-first search, the above $O(nd)$ does not account for the time spent forming the path in the *postSearch()* method, i.e. $O(d^2)$, from previously. Therefore, a more accurate time complexity for the overall search algorithm can be expressed as $\boldsymbol{O(nb + d^2)}$.

UCS Space Complexity

The space complexity will now be discussed. The space taken up during the uniform-cost search is mostly due to the *predCostPairs* array, with a size of $O(n)$, the nodes queue, with a worst case size of $O(n)$, and array of distance values, also with a size of $O(n)$. Since these complexities are of the same degree, then the overall space complexity due to these three aspects is a linear $\boldsymbol{O(n)}$.

In the loops throughout the search, only constant-size space is essentially required since the *head* and *n* values are not kept between loops and only serve as references to one node each at a time.

Once more, the space complexity needs to take into consideration that **each** node needs to store a *visited* state. However, this feature has a space complexity of $\boldsymbol{O(n)}$, which when added to the above $O(n)$, does not affect the complexity. Therefore, the overall space complexity of the algorithm is $\boldsymbol{O(n)}$.

UCS Completeness

Uniform-Cost Search is complete. This can be deduced from the fact that the algorithm goes through the adjacent nodes of each node. If a path from the start node to a goal node exists, then this will have been found by the time the algorithm terminates and can be traced in reverse using the predecessor-cost pairs array.

UCS Admissibility

Given the fact that the algorithm does not settle for the first goal node found and instead opts to find the goal node with the shortest distance (or cost) from the start node, the Uniform-Cost Search is in fact admissible. At the end of the algorithm, the stored goal node will be the goal node with the smallest *dist* value.

Kindly refer to section 3.3 for a comparison between the results of UCS, A*, and Iterative-Deepening A*, which is meant to show that since all of these three are admissible, then their results should match exactly.

3.2.5 – A* Search

The A* search algorithm class is presented below (Listing 6) and was based on the pseudocode presented in **[4]**. This class overrides all three of the search methods: *preSearch()*, *startSearch(…)*, and *postSearch()*. It also has an array of predecessor-cost pairs (identical to the one in BFS and UCS), a node queue representing the collection of discovered but not yet visited nodes, and two arrays of *scores* that will be discussed further on. Like UCS, this search algorithm also makes use of a private Comparator class.

```
public class Astar extends SearchAlgorithm {

    private PredecessorCostPair predCostPairs[];
    private PriorityQueue<Node> openSet;
    private double gScore[];
    private double fScore[];

    public Astar(final Graph graph) {
        super(graph);
    }

    @Override
    protected void preSearch() {

        // Initializing predecessor-cost-pair array
        predCostPairs = new PredecessorCostPair[graph.nodes.length];
        for (int i = 0; i < predCostPairs.length; i++) {
            predCostPairs[i] = null;
        }

        // Initialize openSet queue
        openSet = new PriorityQueue<>(new FScoreComparator());
        openSet.add(graph.startNode);

        // Initialize gScore and fScore arrays
        gScore = new double[graph.nodes.length];
        fScore = new double[graph.nodes.length];
        for (int i = 0; i < graph.nodes.length; i++) {
            gScore[i] = Double.MAX_VALUE;
            fScore[i] = Double.MAX_VALUE;
        }
        gScore[graph.startNode.ID] = 0; // Distance from start to itself is zero
        fScore[graph.startNode.ID] = heuristic_cost_estimate(graph.startNode);
    }

    @Override
    protected void startSearch(final Node startNode) {

        Node node;
        while (!openSet.isEmpty()) {

            // Get node in openSet having the lowest fScore
            node = openSet.getHighestPriority();
            if (graph.isGoalNode(node)) {
                theReachedGoalNode = node;
                return;
            }

            // Remove node and set as visited
            openSet.remove(node);
            node.visited = true;

            // Traverse adjacent nodes
            for (final Edge e : node.edgesOut) {
                final Node n = e.dst;
                if (!n.visited) {
```

```
                    // Distance from start to the adjacent node
                    double tentative_gScore = gScore[node.ID] + e.val;
                    if (!openSet.contains(n)) {
                        openSet.add(n); // New node discovered
                    } else if (tentative_gScore >= gScore[n.ID]) {
                        continue; // Not a better path
                    }

                    // Until now, this path is the best, so it is stored
                    predCostPairs[n.ID] = new PredecessorCostPair(node, e.val);
                    gScore[n.ID] = tentative_gScore;
                    fScore[n.ID] = gScore[n.ID] + heuristic_cost_estimate(n);
                }
            }
        }
    }

    @Override
    protected void postSearch() {

        final PathCostPair pathCost = UtilityMethods.toPathCostPair(
                predCostPairs, theReachedGoalNode);
        path = pathCost.path;
        cost = pathCost.cost;
    }

    private class FScoreComparator implements Comparator<Node> {
        @Override
        public int compare(final Node n1, final Node n2) {
            // Result inverted since smaller fScore is higher priority
            return -Double.compare(fScore[n1.ID], fScore[n2.ID]);
        }
    }
}
```

Listing 6 – The A Search Algorithm Class*

The private variables will be discussed first. The array of predecessor-cost pairs **predCostPairs** will serve the same purpose as the array in BFS and UCS. The queue of nodes **openSet** will store nodes that have been discovered but which have not yet been visited. It uses the custom priority queue implementation with an **FScoreComparator** comparator to always dequeue the node with the lowest *fScore* value.

The array of scores **gScore** will be indexed by the ID of nodes and stores the estimate distance from the start node to the particular node. The array of scores **fScore** will also be indexed by the node IDs but will store the estimated distance from the start node to a goal node, if the particular node is one of the nodes in the path between the start and goal. Both arrays are constantly updated and will be used throughout the algorithm.

Moving on to the overridden methods, the ***preSearch()*** override initializes the arrays and queue. Only the start node is added to the queue since we know that it exists so it has been "discovered". All fScores and gScores are set to the maximum double value since we do not yet know the distances. However, since the distance from start to start is zero, and so fScore for the start node is zero, and since the distance from start to a goal node is completely unknown, this is initialized as the value of the "heuristic function" (discussed below) for the start node. The ***postSearch()*** override, like in BFS and UCS, sets the path and cost results.

The Heuristic Function

The **heuristic function** is meant to estimate the distance from the particular node to a goal node. This function must never overestimate the distance and is essentially application dependent. For this assignment, this value was set as zero for all nodes. Since zero distance is never an overestimate, this does not break the admissibility of the algorithm. Trying to come up with a distance estimate with the given graph definition without actually searching is likely to increase the algorithm cost significantly, which defeats the purpose of the estimate. If, on the other hand, nodes in the graph were labelled with 'x' and 'y' coordinates, and assuming that the goal

nodes are known without having to search for them, the distance can be estimated as the Euclidean distance between the two points represented by the 'x' and 'y' coordinates of the particular node and the closest goal.

A* StartSearch Method

The actual algorithm will now be discussed. A* starts by dequeuing a node from the openSet queue. Since the start node is the only node in the queue, this is the first node to get dequeued. Note that a loop will iterate until the openSet queue is empty, dequeuing a node each time the loop condition is not false. The dequeued node is immediately checked to see whether it is a goal node. If so, the algorithm ends.

Once a node has been dequeued from the queue, it is set as visited. Next, for each non-visited adjacent node, a tentative gScore is calculated by adding the *predecessor* node's gScore to the weight of the edge between this node and the adjacent node in consideration. This means that the distance from the start node to the adjacent node, through a particular path of nodes, is being calculated.

Next, if the adjacent node is not a part of the openSet queue, it is added to it so that it can be visited later on, if a goal node would not have already been found by that point. If the node was not in the queue, this means that this is the first gScore calculation for the node. Otherwise, if the node was in the queue, it is checked whether the new gScore value is greater than the previous gScore value. A smaller gScore value is preferred since this means a shorter path from the start node to the node. In fact, if either the node was not in the queue or the new tentative gScore value is smaller than the previous, the gScore value for the node is set.

This is taken as an opportunity to also set the predecessor-cost pair for the node, since this predecessor provides the node with the best path so far from the start node, and the fScore value, which is set as the distance from the start to the node (gScore) plus the estimated distance to the closest goal node.

Overall, this algorithm is constantly updating the three arrays; predCostPairs, gScore, and fScore so that the scores can be used in future checks and so that when a goal node is found, the shortest path back to the start node can be traced using the updated predCostPairs. An important note is that since visited nodes are not re-visited, and nodes that are already in the openSet queue are not re-added, issues due to cycles are avoided.

A* Time Complexity

The time complexity of the implemented A* algorithm will now be discussed. It will be assumed that the time complexity of the heuristic function is $O(1)$, since it essentially depends on the application. In this implementation, the function simply returns a zero.

Consider the *startSearch(…)* method. In the worst case, the number of nodes visited will depend on the depth reached and the branching factor of the nodes, and so the time complexity of this aspect is $O(b^d)$. This complexity can be deduced in a similar way as was done for other algorithms.

However, for each node visited, a series of further operations dependent on the number of nodes in the queue take place, such as getting the *highest priority* node from the queue and removing this node later on. Since these operate on a queue with worst case space of $O(b^d)$, then this aspect also has an $O(b^d)$ time complexity.

Considering that operations of time complexity $O(b^d)$ need to be performed for each of the $O(b^d)$ nodes, this indicates that the overall time complexity is $O(b^d \times b^d) = \boldsymbol{O(b^{2d})}$.

A* Space Complexity

The space complexity will now be discussed. The space taken up during the A* search is mostly due to the *predCostPairs* array, with a size of $O(n)$, the *openSet* queue, with a worst case size of $O(b^d)$ since it contains the adjacent nodes of all nodes in the worst case, and the two arrays of scores, with a collective size of $O(2n)$. Adding these complexities together produces $O(3n + b^d)$ which can be simplified to $\boldsymbol{O(n + b^d)}$.

In the loops throughout the search, only constant-size space is essentially required since the *node*, *n, and tentative gScore* values are only used within the loop iteration in which they are given a value.

Once more, the space complexity needs to take into consideration that **each** node needs to store a *visited* state. However, this feature has a space complexity of $\boldsymbol{O(n)}$, which when added to the above $O(n + b^d)$, does not affect the complexity. Therefore, the overall space complexity of the algorithm is $\boldsymbol{O(n + b^d)}$.

A* Completeness

A* Search is complete. This can be deduced from the fact that the algorithm goes through the adjacent nodes of each node and adds these to the openSet queue, from which each node, until a goal is found, is dequeued. If from a particular node an adjacent node is not visited, this is only because it was already visited. Note that if a path from the start node to a goal node exists, this can be traced using the predecessor-cost pairs array.

A* Admissibility

A* Search is also admissible. This is due to the fact that the node dequeued from the openSet queue is always the one with the lowest fScore value compared to the other nodes in the queue, i.e. the one through which the path is shortest from a start node to a goal node. This means that when the goal node is found, this would be the goal node that results in the shortest path from the start node to a goal node.

Kindly refer to section 3.3 for a comparison between the results of UCS, A*, and Iterative-Deepening A*, which is meant to show that since all of these three are admissible, then their results should match exactly.

3.2.6 – Iterative-Deepening A* Search

The Iterative-Deepening A* (IDA*) search algorithm class is presented below (Listing 8) and was based on the pseudocode presented in **[5]**. This class overrides only the *startSearch(…)* method. Similar to the non-deepening A* class, it also has an array of **gScore** values.

```
public class Astar_ID extends SearchAlgorithm {

    private static final double FOUND = -1;
    private double gScores[];
    private double depth;

    public Astar_ID(final Graph graph) {
        super(graph);
    }

    @Override
    protected void startSearch(final Node startNode) {

        depth = heuristic_cost_estimate(startNode);
        do {
            // Reset gScores values
            gScores = new double[graph.nodes.length];
            for (int i = 0; i < gScores.length; i++) {
                gScores[i] = Double.MAX_VALUE;
            }
            gScores[graph.startNode.ID] = 0;

            // Perform depth-limited search
            depth = depthLtdSearch(startNode);
        } while (depth != FOUND && depth != Double.MAX_VALUE);
    }

    private double depthLtdSearch(final Node node) {

        // Calculate fScore
        final double fScore = gScores[node.ID] + heuristic_cost_estimate(node);
        if (fScore > depth) {
            return fScore; // fScore exceeded the depth
        } else if (graph.isGoalNode(node)) {
            // A goal node was found
            theReachedGoalNode = node;
            path = new ArrayList<>();
            path.add(theReachedGoalNode);
            return FOUND;
        }

        // Calculate the minimum depth greater than the max depth allowed
        double min = Double.MAX_VALUE;
        for (final Edge e : node.edgesOut) {
            if (gScores[node.ID] + e.val < gScores[e.dst.ID]) {
                gScores[e.dst.ID] = gScores[node.ID] + e.val;
                final double result = depthLtdSearch(e.dst);
                if (result == FOUND) {
                    // Goal node was found
                    prependToPath(node);
                    cost += e.val;
                    return FOUND;
                } else if (result < min) {
                    min = result; // Update minimum
                }
            }
        }
        return min;
    }
}
```

Listing 7 – The Iterative-Deepening A Search Algorithm Class*

Considering the private members, the purpose of the array of gScore values **gScores**, is identical to that of the gScore array in normal A*, i.e. for a node, it stores the estimated distance from the start node to the particular node. The **depth** variable will store the maximum depth that the depth-limited section of the algorithm is allowed to reach, which will be slowly increased until a goal node is found or until a dead end is reached. A value **FOUND** was also defined and will indicate that a goal node was found. Note that the heuristic function was again given a value of zero for each node so that it does not overestimate distances to goal nodes.

IDA* StartSearch Method

The actual search algorithm will now be discussed. Since this algorithm needs a depth-limited search, another method called **depthLtdSearch** was implemented for this purpose. The **startSearch(…)** method consists of a loop which iterates until the depth returned from calling the depth-limited search method is either FOUND, indicating that a goal node was found, or the maximum double value, indicating that a dead end was reached and that the goal node is not reachable at any depth. Otherwise, the value returned by the depth-limited search is the next *suggested* depth that the algorithm will run at to try to reach a goal node.

Note that in each iteration, the gScores array is reset. This is important since if the goal node is not found after a depth-limited search, the algorithm shouldn't use gScore values from the previous depth-limited run.

Moving on to the depth-limited search method; for a particular node, this starts by immediately calculating an fScore value based on the node's gScore value plus the heuristic function's value for an estimate of the distance from the start node to a goal node, if the path includes the node in consideration. If this exceeds the allowed depth, the fScore is returned to indicate that an increased depth allowance is required to reach the node. Otherwise, if the node is a goal node, this is set as so and added to the path, and the method returns.

Note that the depth-limited search method is recursive. If the bound was not exceeded and the node was not a goal node, the next step is to calculate a minimum next depth that is greater than the current maximum depth allowed. This is done by considering each adjacent node for which the gScore is higher than that of its predecessor plus the weight of the edge between them. This means that only adjacent nodes that have not been visited through a shorter path from the start node are considered. For each of these adjacent nodes, a new gScore value is calculated (since a shorter path was found) and the depth-limited search is called.

The minimum of the return values from these calls is returned from the method to indicate that this is the next maximum depth that the depth-limited search should run at. Note that if one of these values is FOUND, then this indicates that a goal node was found in the latest recursive call and so the current node is added to the path and FOUND is returned so that the remaining functions in the call stack also get this value. Since the algorithm searches by depth, the order of the method calls in the call stack can be used as an opportunity to record the path back to the start node by prepending nodes to the path and adding edge weight to the cost.

Overall, the algorithm searches increasingly deeper, each time increasing to the next depth requested by the depth-limited search, until either a goal node is found or until the minimum next depth returned by the method is the maximum double value, which is the value that it is initialized as, indicating a dead end.

An important note is that due to the condition (`gScores[node.ID] + e.val < gScores[e.dst.ID]`) which does not visit adjacent nodes when their gScore values are already small, cycles are avoided since the gScore after going through a cycle will certainly be greater than the gScore value of the node/s involved in the cycle.

Kindly refer to the time complexity discussion in the next page.

IDA* Time Complexity

The time complexity of the implemented iterative-deepening A* algorithm will now be discussed. Consider the *startSearch(…)* method. In the worst case, the loop in this method will iterate d times, assuming that one extra node will be reachable for each extension of the allowed maximum depth.

Now consider the *depthLtdSearch(…)* method. In the worst case, for each node in the depth d reached, all of the b adjacent nodes need to be explored, meaning a time complexity of $O(b^d)$.

Overall, considering that the *depthLtdSearch(…)* needs to be performed at each of the d iterations of the loop in the *startSearch(…)* method, the overall time complexity is $\boldsymbol{O(d \times b^d)}$.

IDA* Space Complexity

The space complexity will now be discussed. The space taken up during the iterative-deepening A* search is mostly due to the *gScores* array, with a size of $O(n)$, and the values used in the recursive depth-limited method, including the *fScore*, *min*, and *result* values, which results in a size of $O(3d)$ where d refers to the number of nodes at the maximum depth reached. This gives an overall linear space complexity of $\boldsymbol{O(n + 3d)}$. The *depth* and *FOUND* values only contribute $O(1)$ to the space complexity and hence do not affect it.

Although each node stores a *visited* state, since this is not used in this algorithm, it will not be considered in the space complexity. This means that the overall space complexity of the algorithm is a linear $\boldsymbol{O(n + d)}$.

IDA* Completeness

Iterative-deepening A* Search is complete. This can be deduced from the fact that the algorithm goes through the adjacent nodes of each node. If from a particular node an adjacent node is not visited, this is only because the adjacent node's gScore value (initialized to Double.MAX_VALUE) is already low, meaning that it was already visited. If a path from the start node to a goal node exists, this will be found eventually.

IDA* Admissibility

Iterative-Deepening A* Search is also admissible. This can be deduced from the fact that the algorithm only visits adjacent nodes for which the gScore plus the edge weight is lower than the adjacent node's gScore, meaning that at the point when a goal node is found, it would be because it was visited through the path along which the nodes have their lowest gScore value possible up to the first goal node. Since the depth is increased according to the estimated distance to at the closest goal node, the goal node reached must be the goal node to which the path is shortest from the start node.

Kindly refer to section 3.3 for a comparison between the results of UCS, A*, and Iterative-Deepening A*, which is meant to show that since all of these three are admissible, then their results should match exactly.

3.3 – Admissibility Test for UCS, A*, and Iterative-Deepening A*

As stated in sections 3.2.4, 3.2.5, and 3.2.6, the results of UCS, A*, and Iterative-Deepening A* will be compared to show that since all of these three search algorithms are admissible, then their results should match exactly. This was done by arranging a second main method in an AdmissibilityTest class which runs these three algorithms for 100,000 times, each time with a new graph.

If at any point the paths from these three algorithms do not match with eachother, the test immediately terminates and displays the graph details and paths which caused the failure. Fortunately, this method runs the three algorithms 100,000 times successfully and outputs: "All 100000 runs were successful!", which further indicates that the implementations of these three algorithms resulted in admissible searches.

```
public class AdmissibilityTest {

    private static int count = 0;

    public static void main(String[] args) {

        final int ITERATIONS = 100000;
        for (count = 0; count < ITERATIONS; count++) {
            if (!run()) {
                return;
            }
        }
        System.out.println("All " + count + " runs were successful!");
    }

    private static boolean run() {

        // Run searches using UCS, Astar, Astar_ID
        final Graph g = new Graph();
        final SearchAlgorithm sa[] = { new UCS(g), new Astar(g), new Astar_ID(g) };
        for (final SearchAlgorithm a : sa) {
            a.performSearch();
        }

        // Check that none of the paths are null and that their length match
        if (sa[0].path != null && sa[1].path != null && sa[2].path != null &&
                sa[0].path.size() == sa[1].path.size() &&
                sa[0].path.size() == sa[2].path.size()) {
            // Compare all nodes of each path
            for (int i = 0; i < sa[0].path.size(); i++) {
                if (sa[0].path.get(i).ID != sa[1].path.get(i).ID ||
                        sa[0].path.get(i).ID != sa[2].path.get(i).ID) {
                    // Failed (one node is not equal to the others)
                    g.outputDetails();
                    Arrays.asList(sa).forEach(SearchAlgorithm::outputResults);
                    System.out.println("Failed (type 1 fail) at run " + count);
                    return false;
                }
            }
            return true; // Successful
        } else if (sa[0].path == null && sa[1].path == null && sa[2].path == null) {
            return true; // Successful
        } else {
            // Failed
            g.outputDetails();
            Arrays.asList(sa).forEach(SearchAlgorithm::outputResults);
            System.out.println("Failed (type 2 fail) at run " + count);
            return false;
        }
    }
}
```

Listing 8 – Admissibility Test

4 – Reference List

[1] ***Data Structure - Depth First Traversal***. Available: https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm.

[2] ***Data Structure - Breadth First Traversal***. Available: https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm.

[3] ***Iterative deepening depth-first search***. Available: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search.

[4] ***A* search algorithm***. Available: https://en.wikipedia.org/wiki/A*_search_algorithm.

[5] ***Iterative deepening A****. Available: https://en.wikipedia.org/wiki/Iterative_deepening_A*.