Practical Task 9

Due date: 09:00pm Friday, September 22, 2018

(Very strict, late submission is not allowed)

General Instructions

1. You may need to read Chapter 11 of SIT232 Workbook available in Resources → Course materials → SIT232 Workbook to remind / study Generics and their application in object-oriented programming. You may also need to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you need excellent understanding of these topics to progress well through the practical tasks of the course. Make sure you are proficient with them as they form a basis to design and develop programming modules in this and all the subsequent tasks. You may find other important topics required to complete the task, like exceptions handling, in other chapters of the workbook.
2. This time, your task is to implement two classical graph traversal algorithms, namely the Depth First Search and the Breadth First Search. This assignment is based on your previous achievements in Practical Task 8. Specifically, you must extend the Graph class that you should have completed by now. However, if you struggle with its implementation or are not able to finish the Practical Task 8, then official working solution will be available right after the submission deadline for Task 8; that is, at 10:00pm Friday, September 14, 2018. However, in this case, you have to wait till the release date and have only one week ahead to complete Practical Task 9. So, do your best to complete Task 8 as soon as possible to potentially get full scores and start the next task earlier.
3. Download the zip file named "Practical Task 9 (template)" attached to this task. It contains a template for the program that you need to work on. Open the solution file in Microsoft Visual Studio. There are two connected projects inside: DataStructures_Algorithms and Runner. There are currently no files in the Week09 subfolder and the objective of the practical task is to add a new partial class. There are no constraints on how to design and implement the new modules internally, but their exterior must meet all the requirements of the task in terms of functionality accessible by a user.

   The second project has Runner09_Task1.cs, which is new. It implements IRunner interface and acts as a main method in case of this practical task by means of the Run(string[] args) method. Make sure that you set the right runner class in Program.cs. You may modify this class as you need as its main purpose is to test the correctness of the graph traversal algorithms. However, it has already a good structure to check the expected methods. Therefore, you should first explore the content of the checking function and see whether it satisfies your goals in terms of testing. Some lines are commented out to make the current version compilable, so assure that you make them active as you progress with the coding part. In general, you should use this class and its method to thoroughly test your code. Your aim here is to cover all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the code. We will later on replace your runner by our special versions in order to test functionality of your code for common mistakes.

4. To strengthen your understanding of the Depth First Search and the Breadth First Search algorithms, Read Chapter 14.3 of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser. You may access the book on-line for free from the reading list application in CloudDeakin available in Resources → Course materials → Course Book: Data structures and algorithms in Java.

Objectives

1. Learn implementation of the Depth First Search and the Breadth First Search graph traversal algorithms.

Specification

Main task

In this practical task, your objective is to implement two graph traversal strategies: the Depth First Search (DFS) and the Breadth First Search (BFS).

The DFS algorithm starts at the specified node of a graph (or at the root if the graph is a tree) and explores all the descendant nodes as far as possible along each branch before backtracking. When visiting a node, the DFS marks it as examined. Therefore, the algorithm remembers previously visited nodes and will not repeat them. The DFS typically traverses an entire graph. Note that traversal performed without memorizing previously visited nodes results in visiting the nodes infinitely as it is caught in a cycle. This is a common error and you must focus on this issue to succeed with your implementation. The DFS is a **recursive** algorithm.

The BFS also starts at the specified node of a graph (or at the root if the graph is a tree) and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. It uses a strategy opposite to the DFS, which instead explores the highest-depth nodes first before being forced to backtrack and expand shallower nodes. The BFS progresses with the existing nodes iteratively level by level. The first level of a zero distance consists of a traversal queue that has only the starting node. This is a base case that initializes the algorithm. The general case builds a new traversal queue, numbered as $i + 1$ and corresponding to the distance of $i + 1$ from the starting node, based on the current queue consisting of nodes of distance i. It explores all the nodes adjacent to those in the queue i and adds to the new queue $i + 1$ those of them that have not been explored yet and, in case of a directed graph, serve as target nodes. When in some iteration k the queue k is empty, the algorithm terminates. Similarly to the DFS, the BFS remembers previously visited nodes so that they are not repeated again during the search. The BFS is an **iterative** algorithm.

Your task is to extend the existing Graph<T,K> class that you have completed in Practical Task 8. Rather than creating a new class, this time you must create a new C# source code file named as "GraphTraversal.cs" in Week09 subfolder. Add **'partial'** keyword to the declaration of the Graph<T,K> in the code finished in Week08 subfolder and copy the declaration of the Graph<T,K> to GraphTraversal.cs. You may read more about this keyword following the link:

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods

Now, proceed with implementation of the algorithms.

Task 1. Implementation of the Depth First Search algorithm

In the part of the Graph<T,K> class in GraphTraversal.cs, create a new **public** method meeting the following requirements.

INode<T>[] DFS(INode<T> node)	Traverses the current Graph<T,K> according the Depth First Search strategy starting at the specified node. Returns an array of nodes casted to the INode<T> interface, where the order of the elements in the array is determined by the order that they have been visited (i.e. entered) during the search. If Graph<T,K> is empty, i.e. the graph has no nodes, the resulting array is to be empty (note that this is different to null), too. The resulting array may contain only a subset of nodes if there are more than one connected component in the graph, thus if some nodes are not reachable from the given node.

In this task, when there are multiple alternative nodes to be visited next from the current node, follow the alphabetical order of the data assigned to the nodes. This policy should be applied as in this task we assume that data has text representation. This policy implies sorting of nodes to support the mentioned selection rule. Therefore, the Graph<T,K> should have an existing internal comparator capable to order two incident edges of type IEdge<T, K> with respect to their target nodes. Note that both edges will actually have the same origin. To solve this sorting sub-problem, develop a private class that can be declared as follows:

```
private class DFSComparator : IComparer<IEdge<T, K>>
```

You may then apply the DFSComparator every time the sorting sub-problem must be resolved. Clearly, you still must implement the DFSComparator class so that it provides you a proper ordering rule. Furthermore, remember that this method is to be recursive. So DFS(INode<T> node) should be just a wrapper function that calls another (private) recursive one where the main work is to be done.

HINT: To introduce marking of nodes and edges, you may also need to extend the existing internal Node and Edge classes of the Graph<T,K> by adding new properties.

Task 2. Implementation of the Breadth First Search algorithm

Now, add the second graph traversal algorithm to the **partial** Graph<T,K> class in Graph-Traversal.cs as a new **public** method meeting the following requirements.

INode<T>[][] BFS(INode<T> node)	Traverses the current Graph<T,K> according the Breadth First Search strategy starting at the specified node. Returns a jagged two-dimensional array of nodes casted to the INode<T> interface. Each subarray of the resulting array corresponds to a layer of nodes of certain distance from the specified node. The first subarray contains only the specified node as its distance is zero, while subarray i has nodes which distance from the node is of i edges. All nodes in a same subarray are equally-distant from the given node. If Graph<T,K> is empty, i.e. the graph has no nodes, the resulting array is to be empty (note that this is different to null), too. The resulting array may contain only a subset of nodes if there are more than one connected component in the graph, thus if some nodes are not reachable from the given node.

Testing your code

This section displays the printout produced by the given Runner based on the official solution. Note that you will likely get different printout as the ToString() methods written by you might differ from those in the official solution. This is clearly valid and allowed. This printout is given to facilitate testing of your code for potential logical errors. It demonstrates the correct logic rather than the expected printout in terms of text and alignment.

```
:: Existing Nodes: [ Adelaide,Alice Springs,Brisbane,Broome,Cairns,Canberra,Dar-
win,Hobart,Melbourne,New Castle,Perth,Sydney ]
:: edge added: Adelaide -> Brisbane
:: edge added: Adelaide -> Canberra
:: edge added: Adelaide -> New Castle
:: edge added: Adelaide -> Perth
:: edge added: Alice Springs -> Brisbane
:: edge added: Alice Springs -> Broome
:: edge added: Alice Springs -> Canberra
:: edge added: Alice Springs -> Melbourne
:: edge added: Alice Springs -> New Castle
:: edge added: Alice Springs -> Perth
:: edge added: Alice Springs -> Sydney
:: edge added: Brisbane -> Hobart
:: edge added: Brisbane -> New Castle
:: edge added: Broome -> Adelaide
:: edge added: Broome -> Brisbane
:: edge added: Broome -> Darwin
:: edge added: Broome -> Melbourne
:: edge added: Broome -> New Castle
:: edge added: Broome -> Perth
:: edge added: Cairns -> Brisbane
:: edge added: Cairns -> Broome
:: edge added: Cairns -> Darwin
:: edge added: Cairns -> Hobart
:: edge added: Cairns -> New Castle
:: edge added: Cairns -> Sydney
:: edge added: Canberra -> Alice Springs
:: edge added: Canberra -> Broome
:: edge added: Canberra -> Darwin
:: edge added: Canberra -> Hobart
:: edge added: Canberra -> Melbourne
```

```
:: edge added: Canberra -> Perth
:: edge added: Canberra -> Sydney
:: edge added: Darwin -> Adelaide
:: edge added: Darwin -> Canberra
:: edge added: Darwin -> Sydney
:: edge added: Hobart -> Adelaide
:: edge added: Hobart -> Alice Springs
:: edge added: Hobart -> Broome
:: edge added: Hobart -> New Castle
:: edge added: Hobart -> Perth
:: edge added: Melbourne -> Adelaide
:: edge added: Melbourne -> Broome
:: edge added: Melbourne -> Cairns
:: edge added: Melbourne -> Hobart
:: edge added: Melbourne -> New Castle
:: edge added: New Castle -> Adelaide
:: edge added: New Castle -> Brisbane
:: edge added: New Castle -> Broome
:: edge added: New Castle -> Cairns
:: edge added: New Castle -> Hobart
:: edge added: New Castle -> Perth
:: edge added: Perth -> Brisbane
:: edge added: Perth -> Cairns
:: edge added: Perth -> Canberra
:: edge added: Perth -> Darwin
:: edge added: Perth -> Melbourne
:: edge added: Sydney -> Adelaide
:: edge added: Sydney -> Broome
:: edge added: Sydney -> Cairns
:: edge added: Sydney -> Canberra
:: edge added: Sydney -> Perth

Printing the content of traverse_DFS...
[ Adelaide,Brisbane,Hobart,Alice Springs,Broome,Darwin,Canberra,Melbourne,Cairns,New
Castle,Perth,Sydney ]
Printing the content of traverse_BFS (level 0)...
[ Adelaide ]
Printing the content of traverse_BFS (level 1)...
[ Brisbane,Canberra,New Castle,Perth ]
Printing the content of traverse_BFS (level 2)...
[ Hobart,Alice Springs,Broome,Darwin,Melbourne,Sydney,Cairns ]
```

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your GraphTraversal.cs and Graph.cs files only and name the zip file as PracticalTask9.zip. Please, **make sure the file name is correct**. You must not submit the whole MS Visual Studio project / solution files.

Marking scheme

You will get 1 mark for outstanding work, 0.5 mark only for reasonable effort, and zero for unsatisfactory work or any compilation errors.