Practical Task 10

Due date: 09:00pm Friday, September 29, 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 the Bellman-Ford algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted directed graph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. This assignment will use your previous achievements, specifically the generic Graph class that you have already completed in Practical Task 8.
3. Download the zip file named "Practical Task 10 (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 Week10 subfolder and the objective of the practical task is to add a new public class. There are no constraints on how to design and implement the new module internally, but its exterior must meet all the requirements of the task in terms of functionality accessible by a user.

   The second project has Runner10 _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 Bellman-Ford algorithm and its related program class. However, the runner already has a good structure able to check the implementation. 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. 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 version in order to test functionality of your code for common mistakes.
4. To get insights on how to implement the Bellman-Ford algorithm, refer to the lecture notes for week 9 available in CloudDeakin in Resources → Course materials.

Objectives

1. Learn implementation of the Bellman-Ford algorithm to the Single Source Shortest Path Problem with negative weight edges

Specification

Main task

In this practical task, your objective is to implement the Bellman-Ford algorithm and solve the Single Source Shortest Path Problem. Suppose that given is a weighted directed graph $G = (V, E)$ with n vertices and m edges, and a specified source vertex $s \in V$. Find the length of the shortest path from vertex s to every other vertex. Unlike Dijkstra's algorithm, the Bellman-Ford algorithm can also be applied to graphs containing negative weight edges. However, if the graph contains a negative cycle, then the shortest path to some vertices may not exist (due to the fact that the weight of the shortest path in this case must be equal to minus infinity). But, this algorithm can be modified to signal the presence of a cycle of negative weight, or even deduce this cycle.

The following depicts a pseudocode of the Bellman-Ford algorithm supporting retrieval of shortest paths with respect to a given source node s.

```
Function $BellmanFord(s : NodeId) : NodeArray \times NodeArray$
    $d = \langle \infty, \ldots, \infty \rangle$ : $NodeArray$ of $\mathbb{R} \cup \{-\infty, \infty\}$      // distance from root
    $parent = \langle \perp, \ldots, \perp \rangle$ : $NodeArray$ of $NodeId$
    $d[s] := 0;$   $parent[s] := s$                                        // self-loop signals root
    for $i := 1$ to $n - 1$ do
        forall $e \in E$ do $relax(e)$                                      // round $i$
    forall $e = (u, v) \in E$ do                                          // postprocessing
        if $d[u] + c(e) < d[v]$ then $infect(v)$
    return $(d, parent)$

Procedure $infect(v)$
    if $d[v] > -\infty$ then
        $d[v] := -\infty$
        foreach $(v, w) \in E$ do $infect(w)$
```

The algorithm operates on a set of distances $d[0, \ldots, n-1]$, which after execution of the algorithm will contain the answer to the problem. In the beginning, it fills the corresponding array as follows: $d[s] = 0$, and all other elements $d[v]$ for $v \neq s$ equal to infinity. The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries to produce relaxation along each edge $e = (v, u)$, $e \in E$, having weight $c(e)$. Relaxation along the edges is an attempt to improve the value $d[u]$ using value $d[v] + c(e)$. In fact, it means that we are trying to improve the answer for this vertex using edge (v, u) and the current answer for vertex v.

It is claimed that $n-1$ phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph. For unreachable vertices the distance $d[v]$ will remain equal to infinity (∞). This algorithm can be somewhat speeded up: often we already get the answer in a few phases and no useful work is done in remaining phases, just a waste visiting all edges. So, one may employ a *flag* variable to tell whether something changed in the current phase or not. Certainly, if nothing has been changed within the phase, the algorithm can be stopped. Note that this optimization does not improve the asymptotic behaviour, i.e., some graphs will still need all $n-1$ phases, but significantly accelerates the behaviour of the algorithm "on an average", i.e., on random graphs.

The algorithm not only finds the length of shortest paths, but also allows to reconstruct them. To do this, it employs another array $p[0,\ldots,n-1]$, where for each vertex we store its "predecessor" or "parent", i.e. the penultimate vertex in the shortest path leading to it. In fact, the shortest path to any vertex u is a shortest path to some vertex $p[v]$, to which we added u at the end of the path. Note that the algorithm works following the same logic: it assumes that the shortest distance to one vertex is already calculated, and, tries to improve the shortest distance to other vertices from that vertex. Therefore, at the time of improvement we just need to remember $p[v]$, i.e, the vertex from which this improvement has occurred.

In the presence of a negative cycle(s), there are further complications associated with the fact that distances to all vertices in this cycle, as well as the distances to the vertices reachable from this cycle are not defined — they should be equal to minus infinity (−∞). It is easy to see that the Bellman-Ford algorithm can endlessly do the relaxation among all vertices of this cycle and the vertices reachable from it. Therefore, if you do not limit the number of phases to $n-1$, the algorithm will run indefinitely, constantly improving the distance from these vertices. Hence we obtain the criterion for presence of a cycle of negative weights reachable for source vertex v:

- after $(n-1)$th phase, if we run algorithm for one more phase, and it performs at least one more relaxation, then the graph contains a negative weight cycle that is reachable from v;
- otherwise, such a cycle does not exist.

Moreover, if such a cycle is found, the Bellman-Ford algorithm can be modified so that it retrieves this cycle as a sequence of vertices contained in it. For this, it is sufficient to obtain the first vertex v for which there was a relaxation in nth phase. This vertex will either lie in a negative weight cycle, or is reachable from it. To get the vertices that are guaranteed to lie in a negative cycle, starting from the vertex v mark all the nodes reachable from v. The provided pseudocode handles this issue via the Infect(v) subroutine. Note that the algorithm explores negative cycles that are reachable from the starting vertex s, and, for unreachable cycles nothing changes in terms of the final solution.

Task 1. Implementation of the Bellman-Ford algorithm

Before starting your work on the Bellman-Ford algorithm, extend functionality of the INode<T> interface, which you have already developed as part of the Practical Task 8. There to be just one extra **public** property added meeting the following specification:

Label	Property. Gets or sets the integer type label associated with a particular node in a graph.

Note that you must subsequently define the same property in the Node private class of the Graph<T,K> as it implements INode<T>.

This property is to store auxiliary data associated with each of the nodes in the graph. Note that using integers gives us a convenient yet general way to encode any sort of data ranging from integer values as they are to generic type values recorded in an additional array and referred through integer indices stored as labels. The latter example realizes so-called look-up tables. In our case, the purpose of each label is to determine the index of a vertex as it appears in the ordered collection (for example, an internal doubly linked list) of nodes of a graph. In this sense, labels act as tag information providing connection between the instance of the Graph class and the logic of the Bellman-Ford algorithm, which eventually deals with vertices encoded as integers rather than objects. This could be clearly seen as it employs the two arrays required to compute the shortest distances to existing nodes and to reconstruct the shortest paths. In general, utilization of integer labels allows to incorporate the Graph class into particular algorithmic solution defined on the user's side.

Your next step is to develop a new **public** class named "BellmanFord" that must be placed into Week10 subfolder. The new class must have one **public static** function of the following signature:

```
public static void Solve( Graph<string, int> graph, int index,
                          out int[] d, out INode<string>[] p)
```

This method accepts an instance of the Graph<string,int> and a source node specified by the index. The index determines the source node's position as it appears in the order of the internal collection of nodes in the given graph. There are two outputs for this function. The first variable d is an array of shortest integer distances from the source node to all other nodes in the graph. The second variable p represents an array of parent nodes such that p[i] is the node visited immediately before vertex i on the shortest path from the source vertex to the vertex i. Clearly, d[index] = 0 and p[index] refers to the source node resulting in a self-loop. If nodes are unreachable in the graph or are affected by negative cycles, their corresponding values in arrays d and p must be **int.MinValue** (minus infinity) and **null**, respectively.

Finally, complete this method and check its correctness. You are certainly allowed to add any other private methods or variables within the BellmanFord class.

HINT: Remember that you may use int.MaxValue and int.MinValue to emulate the plus and minus infinity, respectively. However, you may face the problem known as **integer overflow** that occurs when the result of an arithmetic operation, such as multiplication or addition, exceeds the maximum size of the integer type used to store it. Therefore, be careful when you add a positive value to int.MaxValue or deduct a negative value from int.MinValue as you may see results that you do not expect. Indeed, you should rather check the values before performing integer addition and avoid it when one of the operands has the maximal possible value. You actually should not run **Relax** operation for the nodes that have not been reached yet (and thus have corresponding values infinite in the array d) and you should omit running

Infect operation for those that are not reachable or have been already infected (thus their values will be set either as plus or minus infinity in the array d).

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, Darwin, Ho-
bart, Melbourne, New Castle, Perth, Sydney ]
:: edge added: Adelaide -> Brisbane : distance 7
:: edge added: Adelaide -> Cairns : distance 10
:: edge added: Adelaide -> Hobart : distance 2
:: edge added: Adelaide -> Perth : distance 5
:: edge added: Alice Springs -> Adelaide : distance 9
:: edge added: Alice Springs -> Broome : distance 1
:: edge added: Alice Springs -> Cairns : distance 4
:: edge added: Alice Springs -> Sydney : distance 7
:: edge added: Brisbane -> Adelaide : distance 8
:: edge added: Brisbane -> Broome : distance 6
:: edge added: Brisbane -> Cairns : distance 9
:: edge added: Brisbane -> Darwin : distance 8
:: edge added: Brisbane -> Hobart : distance 5
:: edge added: Brisbane -> Melbourne : distance 6
:: edge added: Brisbane -> Sydney : distance 1
:: edge added: Broome -> Alice Springs : distance 1
:: edge added: Broome -> Cairns : distance 9
:: edge added: Broome -> Canberra : distance 6
:: edge added: Broome -> Darwin : distance 7
:: edge added: Broome -> Hobart : distance 6
:: edge added: Broome -> Melbourne : distance 1
:: edge added: Broome -> New Castle : distance 8
:: edge added: Broome -> Perth : distance 2
:: edge added: Broome -> Sydney : distance 9
:: edge added: Cairns -> Broome : distance 6
:: edge added: Cairns -> Melbourne : distance 4
:: edge added: Cairns -> New Castle : distance 7
:: edge added: Cairns -> Perth : distance 9
:: edge added: Canberra -> Brisbane : distance 5
:: edge added: Canberra -> Cairns : distance 10
:: edge added: Canberra -> Hobart : distance 2
:: edge added: Canberra -> Perth : distance 1
:: edge added: Darwin -> Alice Springs : distance 9
:: edge added: Darwin -> Brisbane : distance 1
:: edge added: Darwin -> Broome : distance 9
:: edge added: Darwin -> Canberra : distance 7
:: edge added: Darwin -> Hobart : distance 7
:: edge added: Darwin -> Perth : distance 8
:: edge added: Darwin -> Sydney : distance 2
:: edge added: Hobart -> Adelaide : distance 8
:: edge added: Hobart -> Alice Springs : distance 8
:: edge added: Hobart -> Broome : distance 7
:: edge added: Hobart -> Canberra : distance 5
:: edge added: Hobart -> Darwin : distance 6
:: edge added: Hobart -> Sydney : distance 8
:: edge added: Melbourne -> Brisbane : distance 8
:: edge added: Melbourne -> Darwin : distance 9
:: edge added: Melbourne -> Perth : distance 9
:: edge added: New Castle -> Adelaide : distance 10
:: edge added: New Castle -> Broome : distance 5
:: edge added: New Castle -> Darwin : distance 9
:: edge added: New Castle -> Hobart : distance 10
:: edge added: New Castle -> Melbourne : distance 5
:: edge added: Perth -> Alice Springs : distance 1
:: edge added: Perth -> Brisbane : distance 5
:: edge added: Perth -> Cairns : distance 4
:: edge added: Perth -> Hobart : distance 7
:: edge added: Sydney -> Alice Springs : distance 7
:: edge added: Sydney -> Cairns : distance 7
:: edge added: Sydney -> Canberra : distance 10
:: edge added: Sydney -> Hobart : distance 4
:: edge added: Sydney -> Melbourne : distance 7


Searching for shortest paths from node 0 <=> Adelaide...

Node Adelaide : shortest distance is 0 via parent Adelaide [ Adelaide ]
Node Alice Springs : shortest distance is 6 via parent Perth [ Adelaide, Perth, Alice Springs
]
Node Brisbane : shortest distance is 7 via parent Adelaide [ Adelaide, Brisbane ]
Node Broome : shortest distance is 7 via parent Alice Springs [ Adelaide, Perth, Alice
Springs, Broome ]
Node Cairns : shortest distance is 9 via parent Perth [ Adelaide, Perth, Cairns ]
Node Canberra : shortest distance is 7 via parent Hobart [ Adelaide, Hobart, Canberra ]
Node Darwin : shortest distance is 8 via parent Hobart [ Adelaide, Hobart, Darwin ]
Node Hobart : shortest distance is 2 via parent Adelaide [ Adelaide, Hobart ]
Node Melbourne : shortest distance is 8 via parent Broome [ Adelaide, Perth, Alice Springs,
Broome, Melbourne ]
Node New Castle : shortest distance is 15 via parent Broome [ Adelaide, Perth, Alice Springs,
Broome, New Castle ]
Node Perth : shortest distance is 5 via parent Adelaide [ Adelaide, Perth ]
Node Sydney : shortest distance is 8 via parent Brisbane [ Adelaide, Brisbane, Sydney ]


Searching for shortest paths from node 6 <=> Darwin...

Node Adelaide : shortest distance is 9 via parent Brisbane [ Darwin, Brisbane, Adelaide ]
Node Alice Springs : shortest distance is 8 via parent Broome [ Darwin, Brisbane, Broome, Al-
ice Springs ]
Node Brisbane : shortest distance is 1 via parent Darwin [ Darwin, Brisbane ]
Node Broome : shortest distance is 7 via parent Brisbane [ Darwin, Brisbane, Broome ]
Node Cairns : shortest distance is 9 via parent Sydney [ Darwin, Sydney, Cairns ]
Node Canberra : shortest distance is 7 via parent Darwin [ Darwin, Canberra ]
Node Darwin : shortest distance is 0 via parent Darwin [ Darwin ]
Node Hobart : shortest distance is 6 via parent Sydney [ Darwin, Sydney, Hobart ]
Node Melbourne : shortest distance is 7 via parent Brisbane [ Darwin, Brisbane, Melbourne ]
Node New Castle : shortest distance is 15 via parent Broome [ Darwin, Brisbane, Broome, New
Castle ]
Node Perth : shortest distance is 8 via parent Darwin [ Darwin, Perth ]
Node Sydney : shortest distance is 2 via parent Darwin [ Darwin, Sydney ]


Searching for shortest paths from node 11 <=> Sydney...

Node Adelaide : shortest distance is 12 via parent Hobart [ Sydney, Hobart, Adelaide ]
Node Alice Springs : shortest distance is 7 via parent Sydney [ Sydney, Alice Springs ]
Node Brisbane : shortest distance is 11 via parent Darwin [ Sydney, Hobart, Darwin, Brisbane ]
Node Broome : shortest distance is 8 via parent Alice Springs [ Sydney, Alice Springs, Broome
]
Node Cairns : shortest distance is 7 via parent Sydney [ Sydney, Cairns ]
Node Canberra : shortest distance is 9 via parent Hobart [ Sydney, Hobart, Canberra ]
Node Darwin : shortest distance is 10 via parent Hobart [ Sydney, Hobart, Darwin ]
Node Hobart : shortest distance is 4 via parent Sydney [ Sydney, Hobart ]
Node Melbourne : shortest distance is 7 via parent Sydney [ Sydney, Melbourne ]
Node New Castle : shortest distance is 14 via parent Cairns [ Sydney, Cairns, New Castle ]
Node Perth : shortest distance is 10 via parent Broome [ Sydney, Alice Springs, Broome, Perth
]
Node Sydney : shortest distance is 0 via parent Sydney [ Sydney ]


!!! We now change the graph and add a couple of negative weights ...

:: edge : Adelaide -> Brisbane : distance -50
:: edge : Alice Springs -> Broome : distance 1
:: edge : Alice Springs -> Cairns : distance 4
:: edge : Alice Springs -> Sydney : distance 7
:: edge : Brisbane -> Broome : distance 6
:: edge : Brisbane -> Cairns : distance 9
:: edge : Brisbane -> Darwin : distance 8
:: edge : Brisbane -> Hobart : distance 5
:: edge : Brisbane -> Melbourne : distance 6
:: edge : Brisbane -> Sydney : distance 1
:: edge : Broome -> Alice Springs : distance 1
:: edge : Broome -> Cairns : distance 9
:: edge : Broome -> Canberra : distance 6
:: edge : Broome -> Darwin : distance 7
:: edge : Broome -> Hobart : distance 6
:: edge : Broome -> Melbourne : distance 1
:: edge : Broome -> New Castle : distance 8
:: edge : Broome -> Perth : distance 2
:: edge : Broome -> Sydney : distance 9
:: edge : Cairns -> Broome : distance 6
:: edge : Cairns -> Melbourne : distance 4
:: edge : Cairns -> New Castle : distance 7
:: edge : Cairns -> Perth : distance 9
:: edge : Canberra -> Brisbane : distance 5
:: edge : Canberra -> Cairns : distance 10
:: edge : Canberra -> Hobart : distance 2
:: edge : Canberra -> Perth : distance 1
:: edge : Darwin -> Alice Springs : distance 9
:: edge : Darwin -> Brisbane : distance 1
:: edge : Darwin -> Broome : distance 9
:: edge : Darwin -> Canberra : distance 7
:: edge : Darwin -> Hobart : distance 7
:: edge : Darwin -> Perth : distance 8
:: edge : Darwin -> Sydney : distance 2
:: edge : Hobart -> Alice Springs : distance 8
:: edge : Hobart -> Broome : distance 7
:: edge : Hobart -> Canberra : distance 5
:: edge : Hobart -> Darwin : distance 6
:: edge : Hobart -> Sydney : distance 8
:: edge : Melbourne -> Brisbane : distance 8
:: edge : Melbourne -> Darwin : distance 9
:: edge : Melbourne -> Perth : distance 9
:: edge : New Castle -> Broome : distance 5
:: edge : New Castle -> Darwin : distance 9
:: edge : New Castle -> Hobart : distance 10
:: edge : New Castle -> Melbourne : distance 5
:: edge : Perth -> Alice Springs : distance 1
:: edge : Perth -> Brisbane : distance 5
:: edge : Perth -> Cairns : distance 4
:: edge : Perth -> Hobart : distance 7
:: edge : Sydney -> Alice Springs : distance 7
:: edge : Sydney -> Cairns : distance 7
:: edge : Sydney -> Canberra : distance 10
:: edge : Sydney -> Hobart : distance 4
:: edge : Sydney -> Melbourne : distance 7
:: edge : Adelaide -> Victor Harbour : distance 5
:: edge : Victor Harbour -> Adelaide : distance -10

Searching for shortest paths from node 0 <=> Adelaide...

Node Adelaide : unreachable
Node Alice Springs : unreachable
Node Brisbane : unreachable
Node Broome : unreachable
Node Cairns : unreachable
Node Canberra : unreachable
Node Darwin : unreachable
Node Hobart : unreachable
Node Melbourne : unreachable
Node New Castle : unreachable
Node Perth : unreachable
Node Sydney : unreachable
Node Victor Harbour : unreachable


Searching for shortest paths from node 6 <=> Darwin...

Node Adelaide : unreachable
Node Alice Springs : shortest distance is 8 via parent Broome [ Darwin, Brisbane, Broome, Al-
ice Springs ]
Node Brisbane : shortest distance is 1 via parent Darwin [ Darwin, Brisbane ]
Node Broome : shortest distance is 7 via parent Brisbane [ Darwin, Brisbane, Broome ]
Node Cairns : shortest distance is 9 via parent Sydney [ Darwin, Sydney, Cairns ]
Node Canberra : shortest distance is 7 via parent Darwin [ Darwin, Canberra ]
Node Darwin : shortest distance is 0 via parent Darwin [ Darwin ]
Node Hobart : shortest distance is 6 via parent Sydney [ Darwin, Sydney, Hobart ]
Node Melbourne : shortest distance is 7 via parent Brisbane [ Darwin, Brisbane, Melbourne ]
Node New Castle : shortest distance is 15 via parent Broome [ Darwin, Brisbane, Broome, New
Castle ]
Node Perth : shortest distance is 8 via parent Darwin [ Darwin, Perth ]
Node Sydney : shortest distance is 2 via parent Darwin [ Darwin, Sydney ]
Node Victor Harbour : unreachable


Searching for shortest paths from node 12 <=> Victor Harbour...

Node Adelaide : unreachable
Node Alice Springs : unreachable
Node Brisbane : unreachable
Node Broome : unreachable
Node Cairns : unreachable
Node Canberra : unreachable
Node Darwin : unreachable
Node Hobart : unreachable
Node Melbourne : unreachable
Node New Castle : unreachable
Node Perth : unreachable
Node Sydney : unreachable
Node Victor Harbour : unreachable
```

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your BellmanFord.cs files only and name the zip file as Practi-calTask10.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.