7SENG010W Data Structures and Algorithms
Week 10 Tutorial Exercises
Topics: Graph Shortest Paths using Dijkstra's Shortest Path Algorithm

These exercises cover the *Shortest Path* problem for weighted edge digraphs using Edsger W. Dijkstra's Shortest Path Algorithm. You can use either the Adjacency Matrix or Adjacency Lists representation of a graph developed in the Week 9 Tutorial. However, for this exercise it is recommended that you use Adjacency Lists to represent the graph.

Exercise 1.
Use David Galles' visualisation tool of Dijkstra's Shortest Path algorithm to explore how it works in practice, it is available at:

https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

Using this visualisation tool do the following experiment:

Generate a graph that is "**Directed**", "**Small**", and select either the "**Logical**" (diagram of graph), "**Adjacency Matrix**" or "**Adjacency List**" representation of the graph, and select "**Start Vertex**" as **0.**

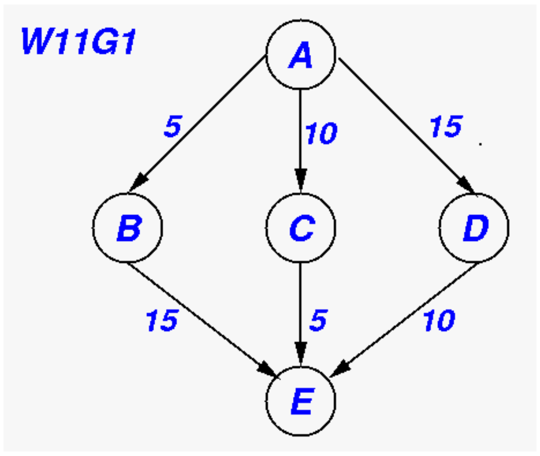
Then run the animation by **stepping through the algorithm**, but before the step is performed try to predict what it will do in that step and its effect, i.e. which vertex it "*visits*" or "*finds*", what data is updated and how, and which vertex it "*moves to*" to continue the search. To help you predict what will happen, follow the pseudo code for Dijkstra's Shortest Path algorithm given in the Week 10 Lecture.

Notes on the animation:

- "***Cheapest Unknown Vertex***" means the nearest vertex to the "**Start Vertex**" you selected, that has been ***found***, but not yet *explored*, i.e. not searched from yet, in other words the first vertex in the sorted queue.
- In the table "**Path**" plays the role of "**edgeTo**" and "**Cost**" is "**distTo**". The edge to your "**Start Vertex**" is defined as "-1".
- If "**Known**" is true then the vertex has been *explored*, i.e. removed from the vertex queue and searched from, false otherwise.
- At the end it constructs all the shortest paths by using the "**Path**" vertices and working backwards from each vertex to your "**Start Vertex**".

You should take screen shots or do a screen capture to record the structure of the graph and the result of the traversal as a sequence of the vertices 0 – 7 that have been traversed.

Exercise 2.
Using the following graph ***W11G1***



V = { *A, B, C, D, E* }

E = { (*A, B*, 5), (*A, C*, 10), (A, *D*, 15), (*B, E*, 15), (C, *E*, 5), (*D, E*, 10) }

Using ***A*** as the *source* vertex ***s*** perform Dijkstra's Shortest Path algorithm on the graph.

You should produce (pen and paper or an online tool) a "log" of the state of the search as it is performed, i.e. use a table similar to the one used in the lecture notes that records the: `edgeTo[v]`, `distTo[v]`, `PriQueue` and `nearestVertex`.

After the algorithm has been completed reconstruct the shortest paths from ***A*** to each vertex from the `edgeTo[v]` edges for each vertex.

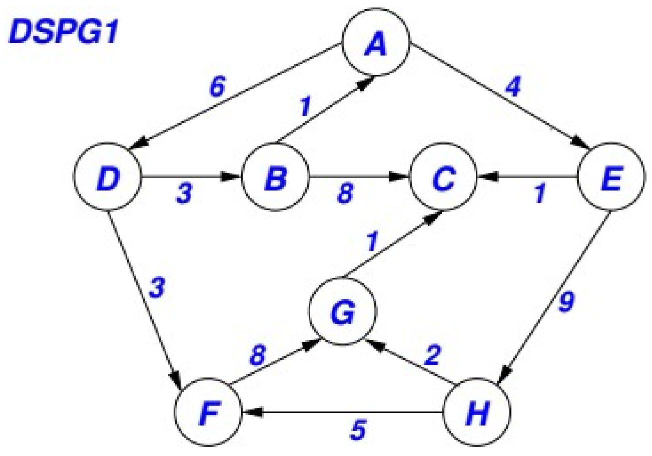
Compare your answers with other students in the tutorial. If there are any differences try to work out why.

OPTIONAL Exercise 3 - Group Coursework

In the rest of the tutorial you can work with your CW group on the group coursework.
You can ask questions about the CW or attempt the two Optional Exercises 4 & 5 which are relevant to the coursework.

OPTIONAL Exercise 4.

The graph ***DSPG1*** from the Week 11 Lecture:



V = { *A, B, C, D, E, F, G, H* }

E = { (*A, D*, 6), (*A, E*, 4), (*B, A*, 1), (*B, C*, 8), (*D, B*, 3), (*D, F*, 3),
(*E, C*, 1), (*E, H*, 9), (*F, G*, 8), (*G, C*, 1), (*H, F*, 5), (*H, G*, 2) }

(a) Create a new class `DijkstraSPGraph` class based on one of your existing graph classes and an `Edge` class to represent a weighted digraph edge, you can also use the `Vertex` class form Week 10.

(b) Using the pseudo code for Dijkstra's Shortest Path algorithm given in the Week 11 Lecture to implement the algorithm using an adjacency list representation. Add the following methods to your `DijkstraSPGraph` class:

```
public void DijkstraShortestPath( Vertex sourceVertex )

private void RelaxEdge( Edge edge )
```

The `DijkstraShortestPath` method should print out for each vertex its shortest path distance and the path.

For the graph representation and traversal of the graph you should re-use the code developed in the Week 9 Tutorial, in particular your graph adjacency list representation class `GraphAdjList` and the Week 10 Tutorial `BreadthFirstSearch(int vertex)` method.

You will also need to create and manage a queue of the *found* vertices, e.g. a sorted queue of "`(vertex, distance)`" pairs sorted on "`distance`". To do this you may want to re-use and adapt your Linked-List code from the Week 3 Tutorial, and/or your Queue code from the Week 4 Tutorial. Alternatively, you could look for a suitable class in the System.Collections or System.Collections.Generic namespaces.

(c) Create a test program `TestingDSP` to test your implementation of Dijkstra's Shortest Path algorithm by running it on graph ***DSPG1.*** The program should print out the distance from the source vertex and the shortest path for all vertices. Check the results produced correspond to those given in the lecture, if they do not work out why and correct the mistakes.

(d) To further check your Dijkstra's Shortest Path algorithm add code to your `TestingDSP` program to:

- create the ***W11G1*** graph and run your implementation on it.
- create the graph that was generated by the DSP visualisation tool in **Exercise 1** and run your implementation on it.

Compare the results with your results in **Exercise 2** and those produced by the visualisation tool respectively. If they are not the same then work out why and correct any mistakes.

OPTIONAL Exercise 5.

(a) Add a method to your DijkstraSP class to solve the following variant of the shortest path problem: ***Source-sink****: from one vertex s to another vertex t.*

```
public Path FindShortestPath( sourceVertex, sinkVertex )
```

(b) Add a method to your DijkstraSP class that can be used to modify the *weight* of an edge:

```
public Path ModifyEdge( sourceVertex, destVertex, weight )
```

(c) Test the two methods by adding code to your test program to find a path between several pairs of vertices in one of the above graphs, then modify some edge weights and re-calculate the shortest paths. Check that the results are as expected, if not then find the errors and correct them.