# Exercises: Minimum Spanning Tree; Shortest Paths

This document defines the **in-class exercises** assignments for the ["Algorithms" course @ Software University](https://softuni.bg/trainings/1194/Algorithms-September-2015).

For the following exercises you are given a Visual Studio solution "**Advanced-Graph-Algorithms-Lab**" holding portions of the source code + unit tests. You can download it from the course’s page.

# Part I – Minimum Spanning Tree (MST) – Kruskal’s Algorithm

If we have a weighted undirected graph we can extract a sub-graph where all nodes (vertices) of the original graph are connected by edges without any cycles. This is referred to as a **spanning tree**. A **minimum spanning tree (MST)** is the spanning tree with the smallest weight (several MST could exist in some graphs).

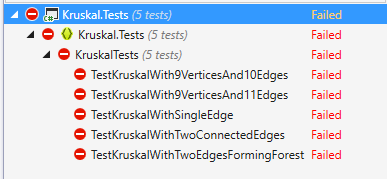
For example, a cable operator wants to connect its customers to a **cable network**. The homes of the customers are connected by streets (edges) with different lengths (weights). To find out how to connect all homes to its network most efficiently (least distance covered) you need to find the **MST**.

One simple algorithm to find the MST of given graph is [**Kruskal’s algorithm**](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm). Example:

|  |  |  |
| --- | --- | --- |
| **Graph** | **Output** | **MST** |
|  | Minimum spanning forest weight: 49  (3 5) -> 2  (0 8) -> 5  (1 7) -> 7  (6 8) -> 7  (1 4) -> 8  (3 6) -> 8  (2 6) -> 12 |  |

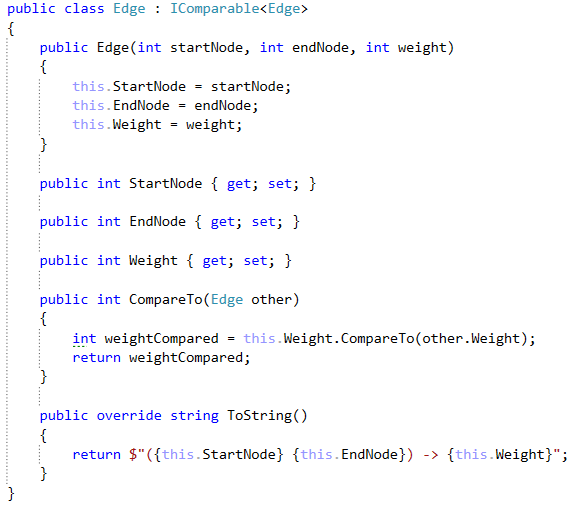
## Run the Unit Test to Make Sure They Initially Fail

Run the unit tests in the **Kruskal.Tests** project. Since we haven’t written the algorithm yet, so they should all fail:



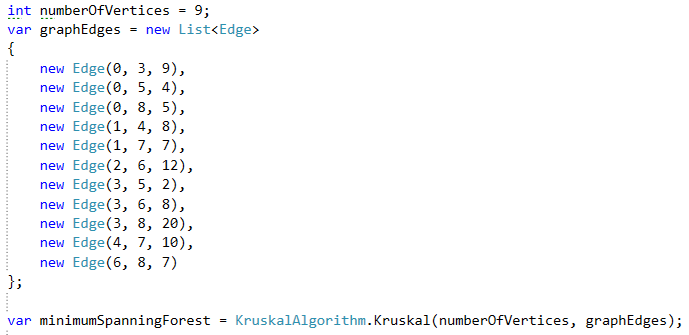
## Class Edge

In a weighted graph an edge holds a lot of information – it connects two nodes, one being the source and the other – the destination, and also has a weight. To simplify our work, we need a class Edge to hold this info. We can implement IComparable<Edge> in order to compare edges by weight. We can also override ToString() to be able to print an edge in a human-readable way. Assuming all weights are integers, we’ve added an implementation of the Edge class to the skeleton:

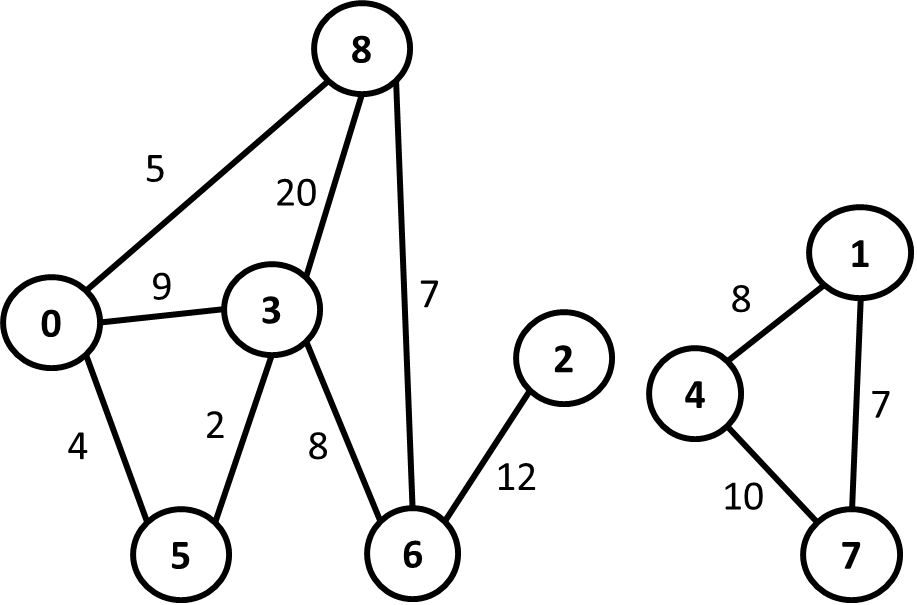


## Start with a Hardcoded Example

In the **KruskalMain** class you’re provided with a test case – a graph with 9 nodes and 11 edges kept in a list of edges. Your task is to complete the Kruskal() method in the **KruskalAlgorithm** class, which receives the number of nodes of the graph and the list of edges and returns a new list of edges representing the minimum spanning tree.



This is how this graph looks like:



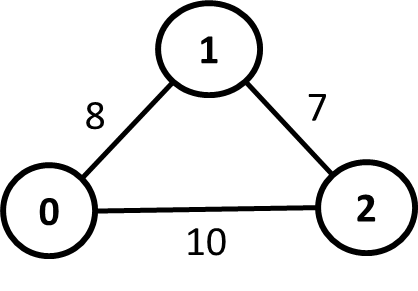
## Sort Edges

Kruskal’s algorithm works by taking all edges in turn, each time the one with smallest weight is picked. Having implemented IComparable<Edge>, we can simply sort the list of edges, calling the Sort() method of the list.

## Preventing Cycles – Concept

Kruskal’s algorithm is simple – having the sorted edges, we take each one in turn, check whether it causes a cycle if added to the current MST and if not – we add it to the MST. How do we check for cycles though?

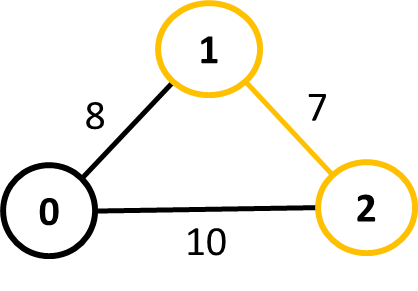
The algorithm works by taking the sorted edges and building the tree from scratch. In a tree, we have a root node and descendants; each descendant has a parent. Consider the following graph:



We have three edges. Following the format described in the Edge class, **(startNode endNode) -> weight**, these are:

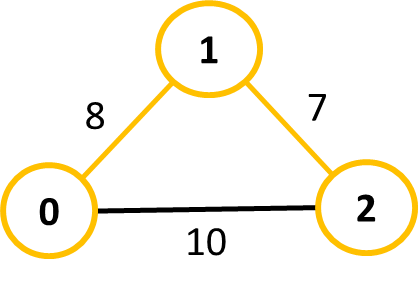
* (1 2) -> 7
* (1 0) -> 8
* (0 2) -> 10

At first, the tree is empty, so none of the nodes have parents. We begin by taking the first node, (1 2) -> 7.



Obviously, there are no cycles when adding the first edge. However, we now have a tree with two nodes, we need to mark one of the nodes as parent and the other as child. It doesn’t matter which will be which, so let’s say that 1 is parent of 2.

Next, we take the edge (1 0) -> 8.



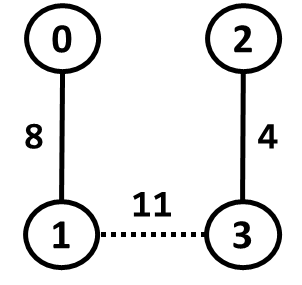
0 has no parent and neither has 1. We can add the edge to the MST. Again, we have to mark which node is the parent. Let’s say 1 is the parent again.

In the final step, we take (0 2) -> 10. This will obviously cause a cycle, but how do we tell? We compare parents for the two nodes of the edge, if they are the same, this means there is a cycle. The parent of 0 is 1; the parent of 2 is also 1, therefore we have a cycle and skip this edge.

What if we chose 0 to be parent of 1 during step 2? When considering the final edge, 0 has no parent. The parent of 2 is 1, but the parent of 1 is 0, therefore 2 is descendant of 0.

In essence, we go up the tree following the parents of each node until we reach the **root**. We **compare the roots for both nodes of an edge and if they have the same root then we have a cycle**.

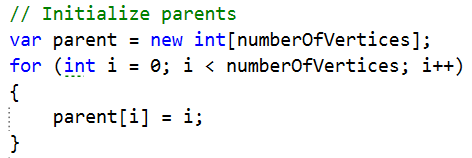
If we have two trees and add a node that connects them, how do we merge the trees? Consider the following example where we have added to the MST two edges (0 1) -> 8 and (2 3) -> 4. We’ve marked that 0 is parent (in this case also root node) of 1 and 2 is parent (root node) of 3. When we add (1 3) -> 11 which doesn’t cause a cycle, we need to set parents in such a way that the tree will have a single root.



Having the two roots, 0 and 2, of the nodes from the newly added edge, we can just mark one as parent of the other, e.g. mark that 0 is parent of 2. This means that, later on, when checking any of the four nodes, they will all have a single root - 0. 0, 1 and 2 all have 0 as parent; 3 has 2 as parent which has 0 as parent, so the root of 3 is also 0. Therefore, **when adding a new edge to the MST, we need to mark one of the roots as parent of the other.**

## Keeping Track of Parents - Setup

In a tree, a node can have at most one parent, therefore, we can keep information about each node’s parent in an array. To mark that a node has no parent, we can simply say that the node is its own parent. Initializing the array is trivial:

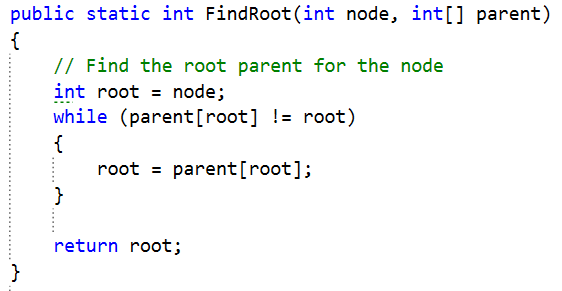


When adding an edge to the MST we’ll mark one of the nodes as parent of the other node.

## **FindRoot()** Method - Implementation

We saw that when taking an edge, we need to find the roots for both nodes. We’ll need a method to find the root of a given node.

Finding the root is pretty easy. If the node is its own parent, then it is the root. If the node has a different parent, we go to the parent and check again; we repeat until we find the root. We need to traverse the parent[] array until we reach an element which has equal index and value:



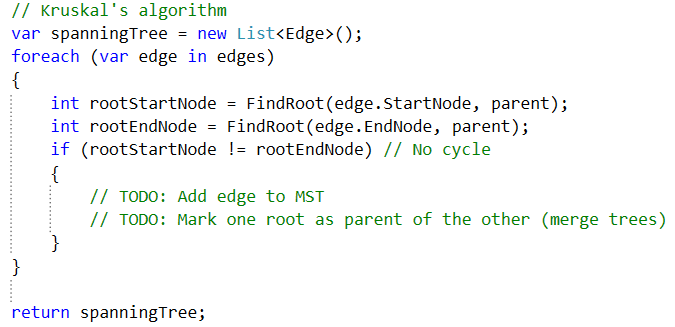
Note that if we have a forest to start with, we’ll have one root per connected component. The FindRoot() method and the algorithm itself will still work correctly.

## Kruskal’s Algorithm – Implementation

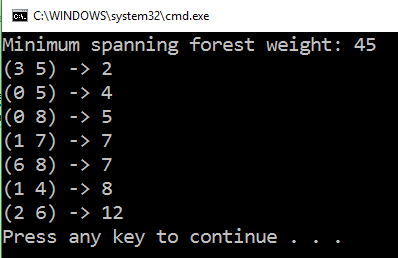
Having the sorted edges and a way to prevent cycles, it’s time to implement the algorithm itself.

It’s pretty simple: 1) instantiate a list to hold the edges of the MST; 2) traverse all edges; 3) find the roots for both nodes in an edge; 4) if the roots are different – add the edge to the MST and mark one node as parent of the other.

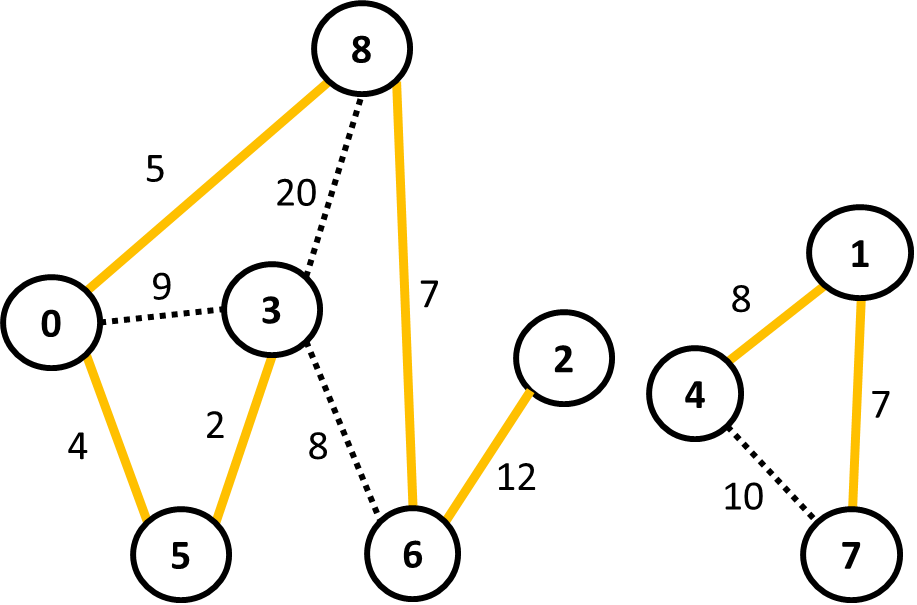
Complete the TODOs in the code below:



Once you’re done, run the program. It should produce the following result:

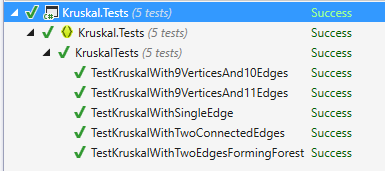


And this is the result visually:



## Run the Unit Tests Again

This time all tests should pass:

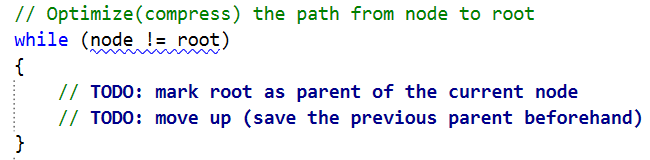


We’re not done yet though. We can optimize things a bit.

## **FindRoot()** – Path Compression

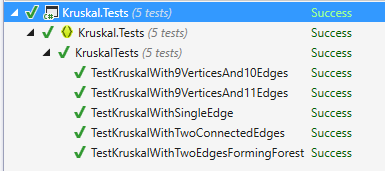
Perhaps you’ve noticed that when searching for the root of a given node we always move up the tree one step at a time. This isn’t really necessary though. In a tree, the root is only one and we can mark that it’s the parent of all its descendants, so, instead of climbing each node in succession, we can go straight to the root. This is what’s called path compression.

We need to modify the FindRoot() method. Once we find the root, we’ll mark it as parent for all nodes we’ve traversed in order to find it. Start with the node we were given initially and move up until we reach the root, making the root a parent for all nodes we move through along the way:



## Run the Unit Tests One Last Time

Make sure the algorithm still works. Running the unit tests after each change we make ensures we don’t break the code while trying to fix or improve it. The tests should still pass.



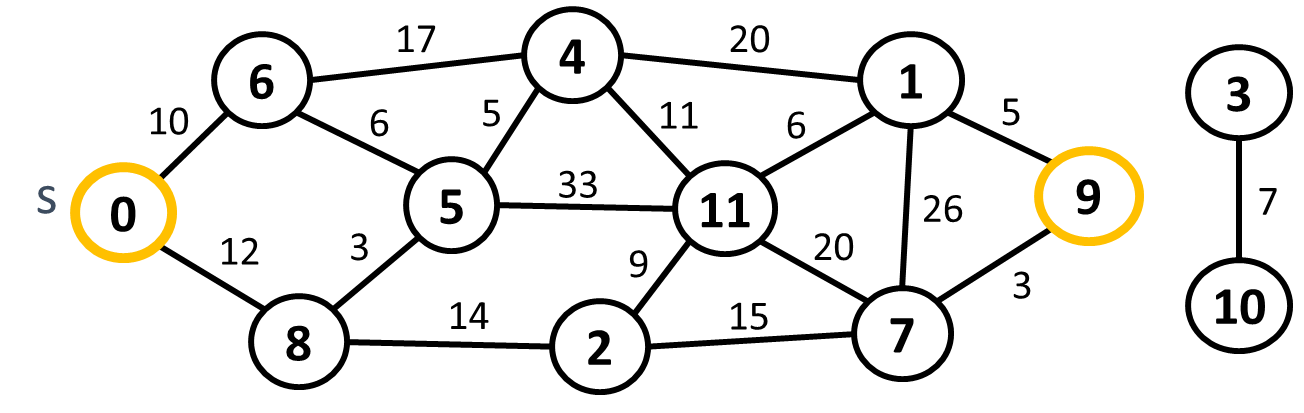
You’ve successfully implemented Kruskal’s algorithm!

# Part II – Dijkstra’s Shortest Path Algorithm

Finding the **shortest path between two nodes** in an unweighted graph is done by applying simple BFS. When we’re working with weighted graphs though, things get more complicated. **Dijkstra’s algorithm** is one of the most famous ones that solves this task.

A classical application of the shortest path algorithm might be to find the shortest path between two towns on a map holding towns connected with roads where each road holds the distance between two towns.

Example: Find the shortest path between **node 0** and **node 9** in the following weighted undirected graph:



The result is: 0->8->5->4->11->1->9 (length 42).

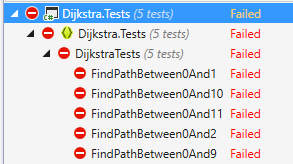
## Provided Assets

In the **Dijkstra** project you are provided with a test case in the Main() method and a method for printing the shortest path between two vertices.

The graph is given as a square matrix where the indices of the rows and columns represent nodes; each cell represents the weight of the edge between two nodes – the row and the column. If the cell has value of 0 this means that there is no edge connecting the two nodes (or the row and column indices are the same).

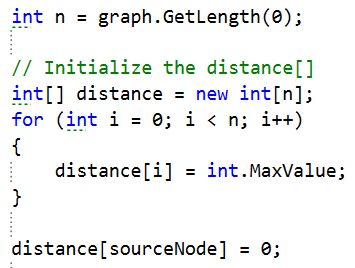
## Run the Unit Tests to Make Sure They Initially Fail

The algorithm is still not implemented, so all unit tests should fail:



## Initialize Distances

We need an array to hold the minimum distance to each node. Initially, the distance to the source node is 0 and the distance to all other nodes is set to infinity (or, in our case, the maximal value for the int type):



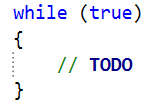
## Initialize the **previous[]** and **used[]** arrays

We also need to keep track of the nodes we’ve visited and, in order to reconstruct the path later, the previous node. The source node has no previous, the value for it will be **null**, so we need a **nullable** type:

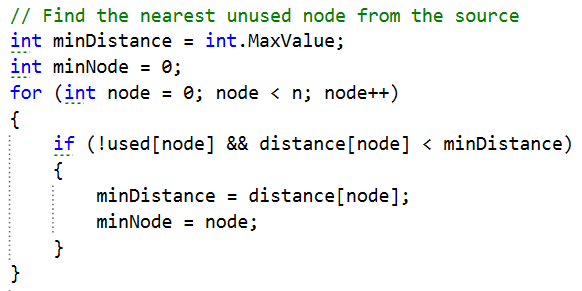


## Find Nearest Unvisited Node

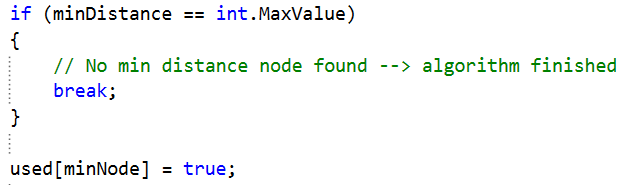
The next steps take place in a loop – we find the nearest unvisited node and start from there. When all possible nodes are traversed, we’ll break the loop:



Finding the nearest unvisited node is simple – loop through all nodes; for each node check whether it’s been visited (this info is kept in the used[] array), check if the distance between the source node and the current one is smaller than the current shortest distance:



When we’re done, if minDistance is still equal to int.MaxValue, this means all possible nodes have been traversed and we need to exit the loop; if not, mark the minNode as visited:



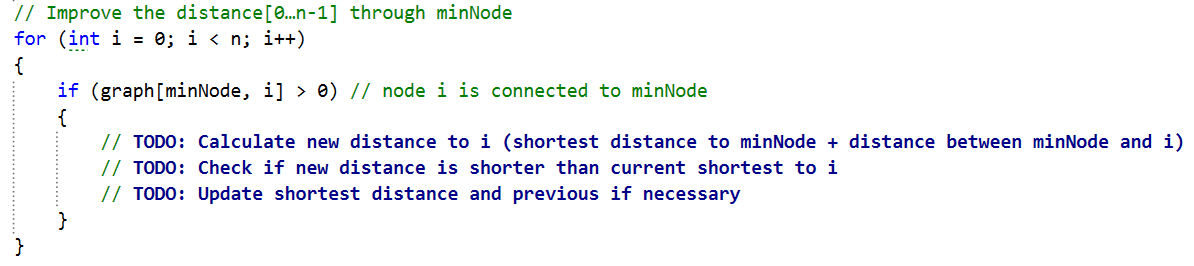
## Improve Shortest Distances

Using the node we just found, we need to go through all nodes connected to it and improve the shortest distances. A node is connected to another node if there is an edge between them; in the context of our matrix, this means a cell with value greater than 0.

You need to do the following:

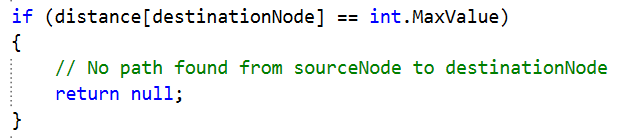
1. Loop through each node
2. Check the ones that are connected to the minNode
3. Calculate the distance from the source node to the current node – just add the shortest distance to minNode and the distance between minNode and the node
4. If the calculated distance is shorter than the current shortest distance for the current node – update the shortest distance and make minNode the previous element

Complete the TODOs in order to improve the shortest distances:

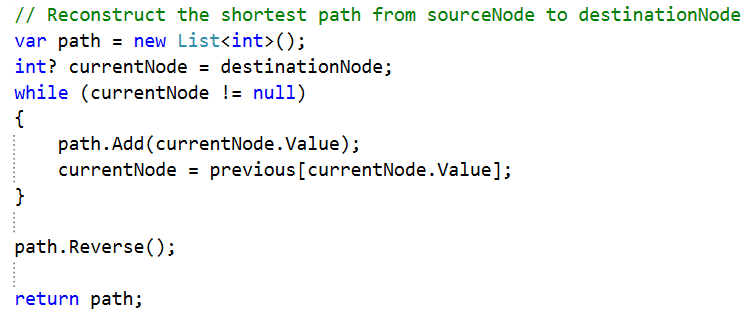


## Reconstruct Shortest Path

First thing’s first – if the shortest distance to the destination is still infinity then we haven’t found a path:

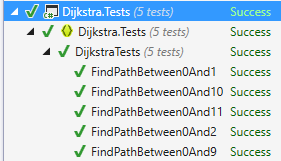


To reconstruct the path, we start from the destination node and move back (using the info kept in the previous[] array) until we reach the source – the source has no previous, so the value will be null. At each step, add the node to a list. Reverse the list in the end and return it:



## Run the Unit Tests Again

If you’ve completed all tasks correctly, the unit test should now pass:



Congratulations! You’ve completed one of the most famous graph algorithms – Dijkstra’s!