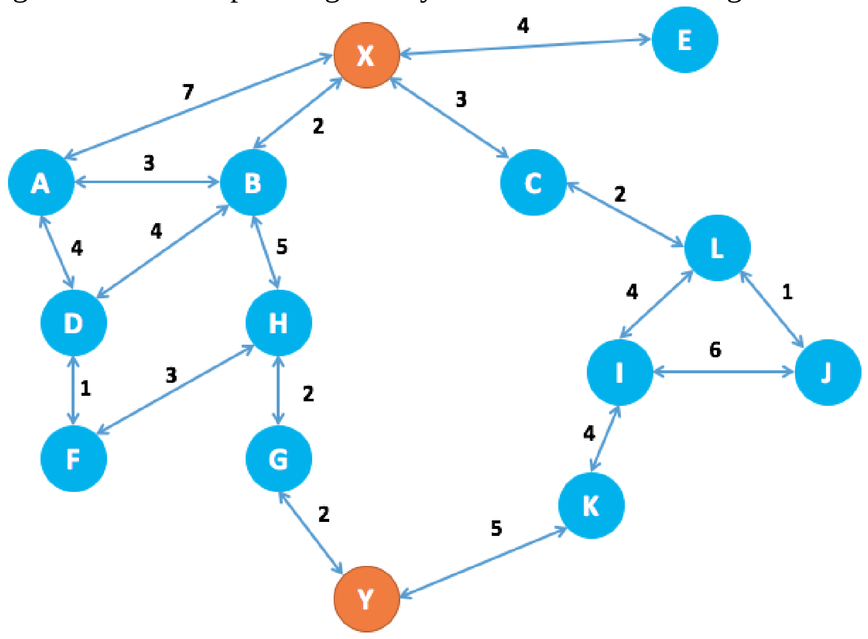
Implementing Djikstra's Shortest Path Algorithm with Python

Source: Ben Alex Keen

Djikstra’s algorithm is a path-finding algorithm, like those used in routing and navigation.
We will be using it to find the shortest path between two nodes in a graph.
It fans away from the starting node by visiting the next node of the lowest weight and continues to do so until the next node of the lowest weight is the end node.
We’ll go work through with an example, let’s say we want to get from X to Y in the graph below with the smallest weight possible. The weights in this example are given by the numbers on the edges between nodes.



We’ll start by constructing this graph in python:

In [1]:
```
from collections import defaultdict

class Graph():
    def __init__(self):
        """
        self.edges is a dict of all possible next nodes
        e.g. {'X': ['A', 'B', 'C', 'E'], ...}
        self.weights has all the weights between two nodes,
        with the two nodes as a tuple as the key
        e.g. {('X', 'A'): 7, ('X', 'B'): 2, ...}
        """
        self.edges = defaultdict(list)
        self.weights = {}

    def add_edge(self, from_node, to_node, weight):
        # Note: assumes edges are bi-directional
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.weights[(from_node, to_node)] = weight
        self.weights[(to_node, from_node)] = weight
```

In [2]:
```
graph = Graph()
```

In [3]:

```
edges = [
    ('X', 'A', 7),
    ('X', 'B', 2),
    ('X', 'C', 3),
    ('X', 'E', 4),
    ('A', 'B', 3),
    ('A', 'D', 4),
    ('B', 'D', 4),
    ('B', 'H', 5),
    ('C', 'L', 2),
    ('D', 'F', 1),
    ('F', 'H', 3),
    ('G', 'H', 2),
    ('G', 'Y', 2),
    ('I', 'J', 6),
    ('I', 'K', 4),
    ('I', 'L', 4),
    ('J', 'L', 1),
    ('K', 'Y', 5),
]

for edge in edges:
    graph.add_edge(*edge)
```

Now we need to implement our algorithm.
At our starting node (X), we have the following choice:

- Visit A next at a cost of 7
- Visit B next at a cost of 2
- Visit C next at a cost of 3
- Visit E next at a cost of 4

We choose the lowest cost option, to visit node B at a cost of 2.
We then have the following options:

- Visit A from X at a cost of 7
- Visit A from B at a cost of (2 + 3) = 5
- Visit D from B at a cost of (2 + 4) = 6
- Visit H from B at a cost of (2 + 5) = 7
- Visit C from X at a cost of 3
- Visit E from X at a cost of 4

The next lowest cost item is visiting C from X, so we try that and then we are left with the above options, as well as:

- Visit L from C at a cost of (3 + 2) = 5

Next we would visit E from X as the next lowest cost is 4.
For each destination node that we visit, we note the possible next destinations and the total weight to visit that destination. If a destination is one we have seen before and the weight to visit is lower than it was previously, this new weight will take its place. For example

- Visiting A from X is a cost of 7
- But visiting A from X via B is a cost of 5
- Therefore we note that the shortest route to X is via B

We only need to keep a note of the previous destination node and the total weight to get there.
We continue evaluating until the destination node weight is the lowest total weight of all possible options.

In this trivial case it is easy to work out that the shortest path will be:
X -> B -> H -> G -> Y

For a total weight of 11.

In this case, we will end up with a note of:

- The shortest path to Y being via G at a weight of 11
- The shortest path to G is via H at a weight of 9
- The shortest path to H is via B at weight of 7
- The shortest path to B is directly from X at weight of 2

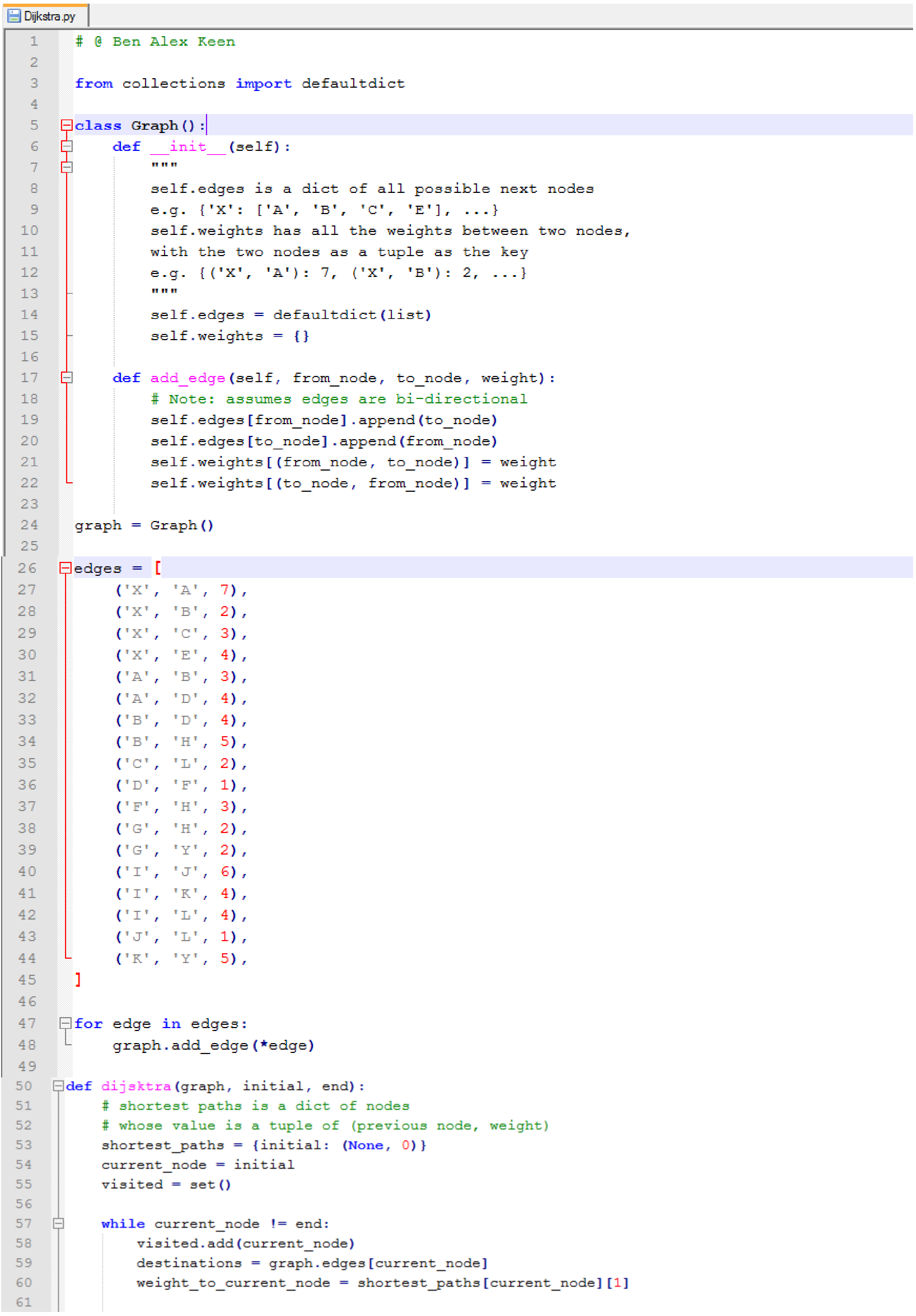
And we can work backwards through this path to get all the nodes on the shortest path from X to Y.
Once we have reached our destination, we continue searching until all possible paths are greater than 11; at that point we are certain that the shortest path is 11.

In [4]:
```
def dijsktra(graph, initial, end):
    # shortest paths is a dict of nodes
    # whose value is a tuple of (previous node, weight)
    shortest_paths = {initial: (None, 0)}
    current_node = initial
    visited = set()

    while current_node != end:
        visited.add(current_node)
        destinations = graph.edges[current_node]
        weight_to_current_node = shortest_paths[current_node][1]

        for next_node in destinations:
            weight = graph.weights[(current_node, next_node)] + weight_to_current_node
            if next_node not in shortest_paths:
                shortest_paths[next_node] = (current_node, weight)
            else:
                current_shortest_weight = shortest_paths[next_node][1]
                if current_shortest_weight > weight:
                    shortest_paths[next_node] = (current_node, weight)

        next_destinations = {node: shortest_paths[node] for node in shortest_paths if node 
not in visited}
        if not next_destinations:
            return "Route Not Possible"
        # next node is the destination with the lowest weight
        current_node = min(next_destinations, key=lambda k: next_destinations[k][1])

    # Work back through destinations in shortest path
    path = []
    while current_node is not None:
        path.append(current_node)
        next_node = shortest_paths[current_node][0]
        current_node = next_node
    # Reverse path
    path = path[::-1]
    return path
```

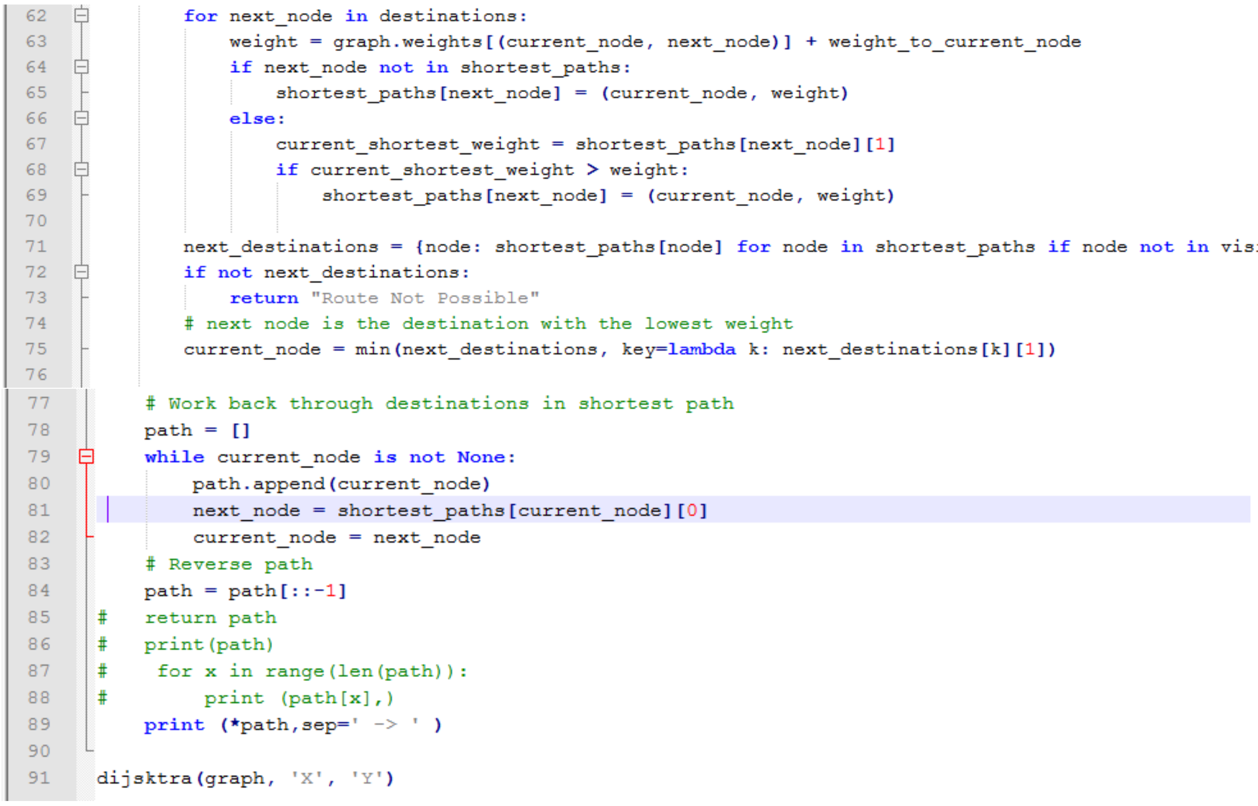
In [5]:
```
dijsktra(graph, 'X', 'Y')
```

Out[5]:
```
['X', 'B', 'H', 'G', 'Y']
```

So there we have it, confirmation that the shortest path from X to Y is:
X -> B -> H -> G -> Y

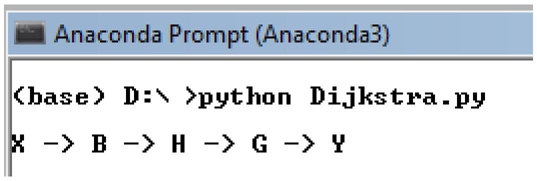

Dijkstra.py

```python
# @ Ben Alex Keen

from collections import defaultdict

class Graph():
    def __init__(self):
        """
        self.edges is a dict of all possible next nodes
        e.g. {'X': ['A', 'B', 'C', 'E'], ...}
        self.weights has all the weights between two nodes,
        with the two nodes as a tuple as the key
        e.g. {('X', 'A'): 7, ('X', 'B'): 2, ...}
        """
        self.edges = defaultdict(list)
        self.weights = {}

    def add_edge(self, from_node, to_node, weight):
        # Note: assumes edges are bi-directional
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.weights[(from_node, to_node)] = weight
        self.weights[(to_node, from_node)] = weight

graph = Graph()

edges = [
    ('X', 'A', 7),
    ('X', 'B', 2),
    ('X', 'C', 3),
    ('X', 'E', 4),
    ('A', 'B', 3),
    ('A', 'D', 4),
    ('B', 'D', 4),
    ('B', 'H', 5),
    ('C', 'L', 2),
    ('D', 'F', 1),
    ('F', 'H', 3),
    ('G', 'H', 2),
    ('G', 'Y', 2),
    ('I', 'J', 6),
    ('I', 'K', 4),
    ('I', 'L', 4),
    ('J', 'L', 1),
    ('K', 'Y', 5),
]

for edge in edges:
    graph.add_edge(*edge)

def dijsktra(graph, initial, end):
    # shortest paths is a dict of nodes
    # whose value is a tuple of (previous node, weight)
    shortest_paths = {initial: (None, 0)}
    current_node = initial
    visited = set()

    while current_node != end:
        visited.add(current_node)
        destinations = graph.edges[current_node]
        weight_to_current_node = shortest_paths[current_node][1]

```



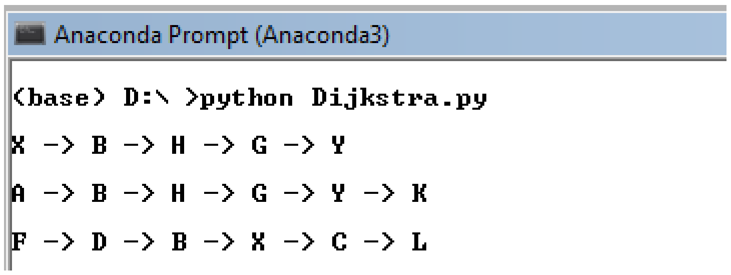
```
        for next_node in destinations:
            weight = graph.weights[(current_node, next_node)] + weight_to_current_node
            if next_node not in shortest_paths:
                shortest_paths[next_node] = (current_node, weight)
            else:
                current_shortest_weight = shortest_paths[next_node][1]
                if current_shortest_weight > weight:
                    shortest_paths[next_node] = (current_node, weight)

        next_destinations = {node: shortest_paths[node] for node in shortest_paths if node not in visited}
        if not next_destinations:
            return "Route Not Possible"
        # next node is the destination with the lowest weight
        current_node = min(next_destinations, key=lambda k: next_destinations[k][1])

    # Work back through destinations in shortest path
    path = []
    while current_node is not None:
        path.append(current_node)
        next_node = shortest_paths[current_node][0]
        current_node = next_node
    # Reverse path
    path = path[::-1]
#   return path
#   print(path)
#    for x in range(len(path)):
#        print (path[x],)
    print (*path,sep=' -> ' )

dijsktra(graph, 'X', 'Y')
```

Results:



Anaconda Prompt (Anaconda3)

```
(base) D:\ >python Dijkstra.py
X -> B -> H -> G -> Y
```



Anaconda Prompt (Anaconda3)

```
(base) D:\ >python Dijkstra.py
X -> B -> H -> G -> Y
A -> B -> H -> G -> Y -> K
F -> D -> B -> X -> C -> L
```