Solution Review: Implementing Dijkstra's

In this lesson, we'll look at a way to implement Dijkstra's Algorithm.

We'll cover the following

Solution #1: Calculating All Paths

```python
def Dijkstra(graph, src, dst):
    number_of_nodes = len(graph[0])   # Number of nodes in the graph
    parent = [-1] * number_of_nodes    # Setting up various lists
    visited = []
    unvisited = [i for i in range(number_of_nodes)]
    distance = [16] * number_of_nodes # The distance list initially has a distance of in
    distance[src] = 0
    shortest_path = []                # This list will have the shortest path at the end
    current = src                     # We start with the source

    while(len(unvisited)>0):
      # Visit all neighbors of current and update distance
      for i in range(number_of_nodes):
        if(graph[current][i]>=0 and distance[i] > graph[current][i]+distance[current]):
          distance[i] = graph[current][i]+distance[current] # Update distance
          parent[i] = current # Set new parent

      unvisited.remove(current) # Move current node from unvisited to visted
      visited.append(current)
      if(len(unvisited) != 0):
        current = unvisited[0] # New current node is an unvisited node with the smallest
        for n in unvisited:
          if(distance[n] < distance[current]):
            current = n

    curr = dst # Some code to get the shortest path from 'parent'
    shortest_path.append(curr)
    cost = 0
    while curr is not src:
      if parent[curr] == -1: # If there is no path to the source node
        return([[],-1])
```

Explanation

Let's go through this code line by line.

- **Lines 1-9**: we set up a few variables that are important for the implementation.
    1. The `number_of_nodes` is the number of nodes in the graph. It's equivalent to the number of rows/columns of the given `graph`. This variable is not necessary for the algorithm itself, but makes calculating other variables clear and easy.
    2. The `parent` list will map each node to its 'parent' or the previous node in the shortest path to the source node. Initialized to `-1`.
    3. The `visited` list is initially empty.
    4. The `unvisited` list contains all the nodes in the graph. Since the nodes in our graph are labeled by numbers, this list is simple to generate.
    5. The `distance` list has all the current distances of all the nodes from the `src` node. Note that all the distances besides the distance of the `src` node from itself are set to infinity, i.e., `16`.
    6. The `shortest_path` is initialized to an empty list.
    7. The `current` node is set to the `src` node since we are calculating the shortest paths from `src` to the rest of the nodes.
- **Lines 11-24**: The `while` loop.
    1. We iterate through all of the neighbors of every `current` node.
    2. If the current distance for a node is greater than the distance of the `current` node + the cost of the link between them, its distance is updated and its `parent` is changed to `current`.
    3. Once all of the neighbors of a node have been exhausted, the next `current` node is chosen to be an unvisited node with the smallest `distance`.
    4. Steps 1-3 are repeated until the while loop exits when no more nodes are left to visit.
- **Lines 26-34**: Calculating the shortest path and the cost. We traverse the `parent` list link by link until a path is generated. Since we start from the

destination, the path has to be reversed. While we calculate the path, we also sum up the cost of each link that is traversed.

- **Example**: Assume the source node is `0`. IF `parent[2]` has the value `1`, the previous node from `2` is `1` as part of the shortest path to a source. Then, suppose `parent[1]` is `0`. So the shortest path will turn out to be `[2, 1, 0]`.

Solution #2: Stopping When Route to `dst` is Calculated

```
def Dijkstra(graph, src, dst):
    number_of_nodes = len(graph[0])   # Number of nodes in the graph
    parent = [-1] * number_of_nodes    # Setting up various lists
    visited = []
    unvisited = [i for i in range(number_of_nodes)]
    distance = [16] * number_of_nodes # The distance list initially has a distance of infinit
    distance[src] = 0
    shortest_path = []                # This list will have the shortest path at the end
    current = src                     # We start with the source

    while(len(unvisited)>0):
      # Visit all neighbors of current and update distance
      for i in range(number_of_nodes):
        if(graph[current][i]>=0 and distance[i] > graph[current][i]+distance[current]):
          distance[i] = graph[current][i]+distance[current] # Update distance
          parent[i] = current # Set new parent

      if(current == dst):
        break

      unvisited.remove(current) # Move current node from unvisited to visted
      visited.append(current)
      if(len(unvisited) != 0):
        current = unvisited[0] # New current node is an unvisited node with the smallest 'dis
        for n in unvisited:
          if(distance[n] < distance[current]):
            current = n

    curr = dst # Some code to get the shortest path from 'parent'
    shortest_path.append(curr)
    cost = 0
    while curr is not src:
      if parent[curr] == -1: # If there is no path to the source node
        return([[],-1])
      cost = cost + graph[curr][parent[curr]] # The cost is the sum of the links in a path
      curr = parent[curr]
      shortest_path.append(curr)
    shortest_path.reverse()
    return([shortest_path, cost])

def main():
    graph = [
      [0,1,5,-1,-1],
      [1,0,3,-1,9],
      [5,3,0,4,-1],
```

```
        [-1,-1,4,0,2],
        [-1,9,-1,2,0]
    ]
    src = 0
    dst = 3
    print(Dijkstra(graph,src,dst))

if __name__ == "__main__":
    main()
```

Explanation

This solution differs from the previous one because it exits the while loop as soon as the destination node is visited via the if condition on **lines 18 and 19**. Since subsequent calculations for the rest of the graph do not change the shortest path to the destination node, there is no need to visit all of them.

In the next lesson, we'll learn about the Internet protocol!