Scientific Computation
Autumn 2024
Problem sheet 3

---

1. Consider the code below.

```python
from collections import defaultdict
import heapq
import networkx as nx

def find_shortest_path(G, start, end):
    """
    Find the shortest path between start and end nodes using Dijkstra's algorit

    Args:
        G: NetworkX graph object
        start: Starting node
        end: Target node

    Returns:
        Tuple containing:
        - The total distance of the shortest path (or None if no path exists)
        - The list of nodes in the shortest path (or None if no path exists)
    """
    # Initialize distances to infinity for all nodes except start
    distances = {start: 0}
    # Keep track of the previous node in the shortest path
    previous = {}
    # Keep track of visited nodes
    visited = set()
    # Priority queue to store (distance, node) pairs
    pq = [(0, start)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        # If we've reached the end node, construct and return the path
        if current_node == end:
            path = []
            while current_node in previous:
```

```
                path.append(current_node)
                current_node = previous[current_node]
            path.append(start)
            path.reverse()
            return current_distance, path

        # Skip if we've already found a shorter path to this node
        if current_node in visited:
            continue

        visited.add(current_node)

        # Check all neighbors of the current node
        for neighbor in G.neighbors(current_node):
            weight = G.edges[current_node, neighbor].get('weight', 1)
            if weight < 0:
                raise ValueError("Edge weights must be non-negative")

            distance = current_distance + weight

            # If we've found a shorter path to the neighbor
            if neighbor not in distances or distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = current_node
                heapq.heappush(pq, (distance, neighbor))

    # If we get here, no path was found
    return None, None

# Example usage
def example_usage():
    # Create a sample graph using NetworkX
    G = nx.Graph()

    # Add edges with weights
    edges = [
        (0, 1, 4),
        (0, 2, 2),
        (1, 2, 1),
        (1, 3, 5),
        (2, 3, 8),
        (2, 4, 10),
        (3, 4, 2)
    ]

    G.add_weighted_edges_from(edges)

    # Find shortest path from node 0 to node 4
```

```
distance, path = find_shortest_path(G, 0, 4)

if path:
    print(f"Shortest distance: {distance}")
    print(f"Shortest path: {' -> '.join(map(str, path))}")
else:
    print("No path exists")

# The same can be done using NetworkX's built-in function for comparison
nx_path = nx.shortest_path(G, 0, 4, weight='weight')
nx_distance = nx.shortest_path_length(G, 0, 4, weight='weight')
print("\nUsing NetworkX built-in functions:")
print(f"Shortest distance: {nx_distance}")
print(f"Shortest path: {' -> '.join(map(str, nx_path))}")
```

This code was created by claude 3.5 sonnet to find a shortest path between two nodes in a weighted graph where all edge weights are non-negative.

(a) Briefly (in 1 or 2 sentences) describe the strategy that find_shortest_path uses to solve this problem.

(b) The same node can appear multiple times in the priority queue. Briefly explain i) why this can happen and ii) whether or not this is a problem for the code's correctness.

(c) What is the big-O cost of this function?

Note: it may be helpful to run the code after adding a few print statements so that you can see what happens while it runs.

2. Consider the application of Dijkstra's algorithm to a weighted graph where all edge weights are non-negative. Let δ_x be the provisional distance of node x (as in lecture). Consider the set of all paths from x to the source node, s, where all nodes on the path other than x are finalized nodes. Show that δ_x is the length of a shortest path in this set. Assume that the distances of the finalized nodes have been set correctly.

3.
```
import numpy as np
from scipy.optimize import root
import matplotlib.pyplot as plt

def f1(X, p1, p2, p3, p4):
    x, y = X
    return [
        p1*x - p2*x*y,
        p3*p2*x*y - p4*y
    ]
```

```
def f2(p1, p2, p3, p4, initial_guess):
    points = []

    sol = root(f1, initial_guess, args=(p1, p2, p3, p4))

    if sol.success:
        point = np.round(sol.x, decimals=8)

        if np.all(point >= 0):
            if not any(np.allclose(point, p) for p in points):
                points.append(point)

    return points
```

Explain the functionality of the code above (e.g. what is the problem it is attempting to solve?). What do you expect to be returned by `f2` if `initial_guess=(0,0)`? Explain how to modify the initial guess so that the output will be different.