Scientific Computation
Autumn 2024
Problem sheet 3 solution

1.

```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))}")
```

The code above 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.
Solution: The code is using Dijkstra's algorithm, and the priority queue is managed using a binary heap.

(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.
Solution: When a node in the priority queue has its provisional distance (PD) updated, a new element is added to the binary heap with the new PD as key and the node number as the value. The element with the old PD remains in the heap, so multiple elements corresponding to the same node can be present in the heap. This is not a problem for the code's correctness because the binary heap design ensures that the element with the smaller provisional distance will be popped first, and the node will be labeled as "visited". If other elements for the same node are popped in the future, nothing will be done since the code checks if the popped node has been visited before carrying out any substantive calculations.

(c) What is the big-O cost of this function?
Solution: Since a node can appear in the heap k times where k is the degree of the node, the size of the heap is $\mathcal{O}(L)$ rather than $\mathcal{O}(N)$, and the costs of min-removal and insertion will be $\mathcal{O}(\log L)$ rather than $\mathcal{O}(\log N)$. Modifying the result from lecture to account for the "extra" elements in the heap and the fact that a node can be popped from the heap multiple times, the overall cost of the algorithm will be $\mathcal{O}(L \log L)$. Note however that $L < N^2/2$, so this is equivalent to $\mathcal{O}(L \log N)$. Note that the cost of constructing the path is $O(N)$ since there can be a maximum of $N-1$ nodes in a path, only $\mathcal{O}(1)$ operations are used in the while loop where the path is constructed, reversing a list is $O(N)$, and only one path is constructed using the information stored when nodes are popped/finalized.

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.
Solution: Say that x has k neighbors in F, the set of finalized nodes, with $k \geq 1$, and label these nodes as $i_1, i_2, ..., i_k$ with the subscript indicating the order in which the nodes were finalized. We need to consider paths which start from x, go directly to one of these neighbors, and then continue on to the source, s. Say the length of such a path via neighbor i_y is l. Then $l \geq d_{x,i_y}$ since otherwise, there would be a path from i_y to s with length shorter than $d_{i_y,s}$. The length of a shortest path from x to s where all nodes other than x are in F is then, $\min(d_{si_1} + w_{i_1,x}, d_{si_2} + w_{i_2,x}, ..., d_{si_k} + w_{i_k,x})$, and Dijkstra's algorithm ensures that this is equal to δ_x. (δ_x is initially $d_{si_1} + w_{i_1,x}$, then $\min(d_{si_1} + w_{i_1,x}, d_{si_2} + w_{i_2,x})$, then $\min(d_{si_1} + w_{i_1,x}, d_{si_2} + w_{i_2,x}, d_{si_3} + w_{si_3}, x)$ and so on.)

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.
Solution: Function `f1` defines a system of two equations with unknowns x and y. The function `f2` finds a solution to these equations using the scipy function,

`root`. If the initial guess is $(0, 0)$ we expect the function to return $(0, 0)$ as this is solution to the system defined in `f1`. The system also has the non-trivial solution, $x = p4/(p2\ p3), y = p1/p2$, so choosing a guess closer to this solution should return this solution. Note that this system of equations can easily be simplified to a linear system, so `root` isn't really needed. It is only for cases where the nonlinearity cannot be removed that it tends to be helpful.