Problem Set 7

Due dates: Electronic submission of the pdf file of this homework is due on **3/21/2025 before 11:59pm** on canvas. The homework must be typeset with LaTeX to receive any credit. All answers must be formulated in your own words.

Watch out for additional material that will appear on Thursday! Deadline is on Friday, as usual.
Name: Chayce Leonard

Resources. (All people, books, articles, web pages, etc. that have been consulted when producing your answers to this homework)
Introduction to algorithms 4th-editions by Thomas C, Charles L, Ronald L, Clifford S
Geeksforgeeks.org
CSCE221 Notes (Graphs, Heaps, Dijkstra Kruskal)
CSCE110 Algorithms Notes
Class Videos
Author: Eric Matthes':
Python Crash Course, 3rd Edition: A Hands-On, Project-Based Introduction to Programming

On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment. Furthermore, I have disclosed all resources (people, books, web sites, etc.) that have been used to prepare this homework.

Signature: Chayce Leonard ____________________________

Read the chapters on "Elementary Graph Algorithms" and "Single-Source Shortest Paths" in our textbook before attempting to answer these questions.

Problem 1 (20 points)**.** Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of the number $|E|$ of edges.

Solution. **Algorithm Description**

We present a concise algorithm for detecting cycles in an undirected graph using depth-first search (DFS). The algorithm runs in O(V) time for sparse graphs where E = O(V).

Algorithm 1 Cycle Detection in Undirected Graph

```
function HasCycle(G = (V, E))
    visited ← set()
    for v ∈ V do
        if v ∉ visited and DFSCycle(G, v, None, visited) then return True
        end if
    end forreturn False
end function

function DFSCycle(G, v, parent, visited)
    visited.add(v)
    for u ∈ neighbors(v) do
        if u ∉ visited then
            if DFSCycle(G, u, v, visited) then return True
            end if
        else if u ≠ parent then return True
        end if
    end forreturn False
end function
```

Key Properties

- **Correctness:** The algorithm detects a cycle if and only if it encounters a visited vertex that is not the parent of the current vertex during DFS.
- **Time Complexity:** O(V + E), which is O(V) for sparse graphs where E = O(V).
- **Space Complexity:** O(V) for the visited set and recursion stack.

Proof Sketch

Proof. The algorithm is correct because:

1. It explores all vertices and edges in the graph.
2. A cycle is detected when a path leads back to a visited vertex through a non-parent edge.
3. If no such path exists, the graph is acyclic.

The time complexity is O(V + E) as each vertex and edge is visited at most once. For sparse graphs, this reduces to O(V). □

Problem 2 (20 points). Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Solution. Modified Bellman-Ford Algorithm:

```
function ModifiedBellmanFord(G(V, E), s)
    dist[v] ← ∞ for all v ∈ V
    dist[s] ← 0
    for i ← 1 to |V| − 1 do
        changes ← 0
        for each edge (u, v, w) ∈ E do
            if dist[u] + w < dist[v] then
                dist[v] ← dist[u] + w
                changes ← changes + 1
            end if
        end for
        if changes = 0 then return dist
        end if
    end for
    for each edge (u, v, w) ∈ E do
        if dist[u] + w < dist[v] then return "Negative cycle detected"
        end if
    end forreturn dist
end function
```

Proof of Correctness: The modification maintains the correctness of the original Bellman-Ford algorithm:

- If no changes occur in a pass, all shortest paths have been found.
- Early termination only happens when no further improvements are possible.
- The algorithm still performs up to $|V|$ passes, ensuring negative cycle detection.

Complexity Analysis:

- Time Complexity:
  - Worst-case: $O(|V||E|)$ - when changes occur in every pass
  - Best-case: $O(|E|)$ - when algorithm terminates after first pass
  - Average-case: $O(k|E|)$, where k is the number of passes before termination ($1 \leq k < |V|$)
- Space Complexity: $O(|V|)$ for distance array

The modification allows the algorithm to terminate in $m+1$ passes, where m is the maximum number of edges in the shortest path from the source to any vertex. This is because:

- After m passes, all shortest paths are guaranteed to be found.
- The $(m+1)$-th pass will have no changes, triggering early termination.

This early termination occurs naturally without prior knowledge of m, satisfying the problem requirement.

Problem 3 (20 points)**.** Suppose that we change line 6 of Dijkstra's algorithm in our textbook to the following.

6 **while** $|Q| > 1$

This change causes the while loop to execute $|V|-1$ times instead of $|V|$ times. Is this proposed algorithm correct? Explain. [Use the version of Dijkstra's algorithm from the textbook]

Solution. To determine if changing line 6 of Dijkstra's algorithm from `while` $|Q| > 0$ to `while` $|Q| > 1$ affects correctness, I'll analyze the algorithm's behavior with this modification.

The original algorithm from the textbook is:

1: **function** DIJKSTRA(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     $S \leftarrow \emptyset$
4:     $Q \leftarrow V[G]$ ($\equiv \emptyset$...*initially*)
5:     **for** each vertex $u \in V[G]$ **do**
6:         INSERT(Q, u)
7:     **end for**
8:     **while** $|Q| > 0$ ($\equiv \neq 0$) **do**
9:         $u \leftarrow$ EXTRACT-MIN(Q)
10:         $S \leftarrow S \cup \{u\}$
11:         **for** each vertex $v \in \text{Adj}[u]$ **do**
12:             RELAX(u, v, w)
13:             **if** the call of RELAX decreased $v.d$ **then**
14:                 DECREASE-KEY$(Q, v, v.d)$
15:             **end if**
16:         **end for**

end while
end function

With the modification, the algorithm will terminate when there's exactly one vertex remaining in Q. Let's call this remaining vertex u_{last}.

Claim: The modified algorithm is correct.

Proof:

When the algorithm terminates with $|Q| = 1$, exactly one vertex (u_{last}) remains unprocessed. There are two possible cases for this remaining vertex:

Case 1: u_{last} is reachable from the source s.

Dijkstra's algorithm always extracts the vertex with minimum distance estimate. If u_{last} remains in Q while all other vertices have been processed, then u_{last} must have the largest distance estimate among all reachable vertices.

Since all other vertices with shorter distances have been processed, all edges leading to u_{last} from vertices with shorter distances have been relaxed. By the properties of Dijkstra's algorithm and the non-negative edge weight requirement, these relaxations have already correctly determined the shortest path to u_{last}.

Therefore, processing u_{last} would not change its distance estimate or any other vertex's estimate. Its shortest path is already correctly computed.

Case 2: u_{last} is unreachable from the source s.

If u_{last} is unreachable, its distance estimate remains ∞ throughout the algorithm. Processing it would not relax any edges (since relaxations only occur when the source vertex has a finite distance). Therefore, skipping its processing doesn't affect correctness.

Edge Case: Single-vertex graph

In the case of a single-vertex graph, the original algorithm processes the vertex and terminates. The modified algorithm terminates immediately without processing any vertex, as $|Q| = 1$ from the start. However, INITIALIZE-SINGLE-SOURCE already sets the correct distance (0) for the source vertex, so the modified algorithm still produces the correct result.

Conclusion:

The proposed modification to Dijkstra's algorithm is correct. The modification only skips the processing of the very last vertex in the priority queue, which either:

- Already has its correct final distance computed (if reachable), or
- Is unreachable from the source (distance = ∞)

In both cases, processing this last vertex would not change any distance values in the graph. All shortest paths from the source to all vertices will be correctly computed with the $|Q| > 1$ termination condition.

The theoretical running time remains asymptotically unchanged: the algorithm performs $|V| - 1$ iterations instead of $|V|$, which is still $O(|V|)$ iterations, keeping the overall time complexity the same.

Problem 4 (40 points)**.** Help Professor Charlie Eppes find the most likely escape routes of thieves that robbed a bookstore on Texas Avenue in College

Station. The map will be published on Thursday evening. In preparation, you might want to implement Dijkstra's single-source shortest path algorithm, so that you can join the manhunt on Thursday evening. Include your implementation of Dijkstra's algorithm and explain all details of your choice of the min-priority queue.

[Edge weight 1 means very desirable street, weight 2 means less desirable street]

Solution. For this assignment, we implemented Dijkstra's single-source shortest path algorithm to help Professor Charlie Eppes identify the most likely escape routes of thieves who robbed a bookstore on Texas Avenue in College Station. The robbers were last seen at waypoint 1, and our task was to determine the most likely destination among waypoints 6, 8, 9, 15, 16, and 22. We used a graph representation where edge weights of 1 indicate very desirable streets and weights of 2 indicate less desirable streets.

0.1 Graph Construction

The graph was constructed using the igraph library, with 22 vertices representing waypoints and edges representing road connections between them. Edge weights were assigned based on traffic conditions:

- Green road segments: weight = 1 (very desirable)
- Orange/yellow road segments: weight = 2 (less desirable)

0.2 Dijkstra's Algorithm Implementation

Our implementation of Dijkstra's algorithm uses a binary min-heap priority queue through Python's `heapq` module. The algorithm follows these steps:

1. Initialize distances to all vertices as infinity, except the source (waypoint 1) with distance 0
2. Initialize a priority queue with the source vertex and its distance
3. Until the queue is empty or all destinations are found:
   (a) Extract the vertex with minimum distance from the queue
   (b) For each unprocessed neighbor, calculate a potential new distance
   (c) If the new distance is shorter, update the distance and add the neighbor to the queue
4. Reconstruct paths using the previous vertex record

0.3 Min-Priority Queue Details

We chose to implement the min-priority queue using Python's `heapq` module for the following reasons:

- Efficiency: It provides O(log n) time complexity for both insertion and extraction operations
- Simplicity: The implementation is straightforward and built into Python's standard library
- Storage format: Queue elements are (distance, vertex) tuples which naturally maintain the priority ordering

The binary heap structure ensures that the vertex with the minimum distance is always at the root, making it optimal for Dijkstra's algorithm where we repeatedly need to extract the minimum-distance vertex.

0.4 Results and Analysis

After running our implementation, we determined:

- The most likely escape destination is waypoint 16 with a total distance of 3
- The escape path is: $1 \rightarrow 11 \rightarrow 17 \rightarrow 16$
- Distances to other potential destinations range from 4 to 7

We verified our results using igraph's built-in shortest path function, which confirmed our implementation's accuracy.

The full script and output are attached after the main part of the submission

0.5 Conclusion

Our implementation of Dijkstra's algorithm successfully identified the most likely escape routes for the robbers. The use of a binary heap priority queue provided an efficient solution with O((V + E) log V) time complexity, which is appropriate for the size of the road network in this problem.

Make sure that you derive the solutions to this homework by yourself without any outside help. Searching for solutions on the internet or asking any form of AI is not allowed. Write the solutions in your own words. Use version control for your program development and be prepared to show and explain any version of your code.

Checklist:

☐ Did you add your name?
☐ Did you disclose all resources that you have used?
(This includes all people, books, websites, etc. that you have consulted)
☐ Did you sign that you followed the Aggie honor code?
☐ Did you solve all problems?
☐ Did you typeset your answers entirely in LaTeX?
☐ Did you submit the pdf file of your homework?

```python
#using igraph to visualize effectively
import igraph as ig
import heapq


def create_escape_route_graph():
    """
    Creates a graph representing the road network based on the map.
    Vertices are waypoints 1-22, and edges represent direct road connections.
    Edge weights: 1 for green roads (desirable), 2 for orange roads (less desirable).
    """
    # Create empty graph with 22 vertices
    g = ig.Graph(n=22, directed=False)

    # Name the vertices
    g.vs["name"] = [str(i) for i in range(1, 23)]

    # Define edges with weights based on the map
    # Format: (source, target, weight)
    #indexing from zero so subtract one from mental index assumption
    edges_with_weights = [
        # Direct connections from waypoint 1
        (0, 1, 1),  # 1-2 (green)
        (0, 10, 1),  # 1-11 (green)

        # Connections from waypoint 2
        (1, 2, 1),  # 2-3 (green)

        # Connections from waypoint 3
        (2, 3, 1),  # 3-4 (green)

        # Connections from waypoint 4
        (3, 4, 1),  # 4-5 (green)

        # Connections from waypoint 5
        (4, 5, 2),  # 5-6 (orange)

        # Connections from waypoint 6
        (5, 6, 1),  # 6-7 (green)
        #considered (5, 6, 2) as yellow

        # Connections from waypoint 7
        (6, 7, 1),  # 7-8 (green)
        (6, 4, 1),  # 7-5 (green)

        # Connections from waypoint 8
        (7, 8, 2),  # 8-9 (yellow)

        # Connections from waypoint 9
        (8, 9, 2),  # 9-10 (orange)
        (8, 18, 1),  # 9-19 (green)

        # Connections from waypoint 10
```

```
        (9, 17, 2),  # 10-18 (orange)
        (9, 10, 1),  # 10-11 (green)

        # Connections from waypoint 11
        (10, 11, 2),  # 11-12 (orange)
        (10, 16, 1),    #11-17 (green)

        # Connections from waypoint 12
        (11, 12, 2),  # 12-13 (yellow)

        # Connections from waypoint 13
        (12, 13, 2),  # 13-14 (yellow)
        (12, 20, 1),  # 13-21 (green)

        # Connections from waypoint 14
        (13, 14, 1),  # 14-15 (green)
        (13, 19, 1),    #14-20 (green)
        (13, 15, 2),    #14-16 (yellow @ 16?)

        # Connections from waypoint 15
        #end node?
        #(14, 15, 1),  # 15-16 (green)

        # Connections from waypoint 16
        (15, 16, 1),  # 16-17 (green) [potentially yellow]
       #(15, 16, 2),

        # Connections from waypoint 17
        (16, 17, 2),  # 17-18 (yellow)

        # Connections from waypoint 18
        #end node (idk if 18-19 is connected but there is a spec of orange)
        (17, 18, 2),  # 18-19 (yellow)

        # Connections from waypoint 19
        #End node
        #(18, 21, 2)

        # Connections from waypoint 20
        (19, 20, 2),  # 20-21 (yellow)
        (19, 21, 1), #20-22 (green)

        # Connections from waypoint 21
        (20, 21, 2)  # 21-22 (some yellow)

    ]

    # Add edges to the graph
    # igraph uses 0-based indexing, but our waypoints are 1-based
    # so we've already adjusted the indices in edges_with_weights
    edges = [(e[0], e[1]) for e in edges_with_weights]
    weights = [e[2] for e in edges_with_weights]

```

```
    g.add_edges(edges)
    g.es["weight"] = weights

    return g


def dijkstra(graph, source, destinations):
    """
    Custom implementation of Dijkstra's algorithm using a priority queue

    Args:
        graph: igraph Graph object
        source: Index of source vertex
        destinations: List of destination vertex indices

    Returns:
        distances: Dictionary mapping vertex indices to distances from source
        previous: Dictionary for reconstructing shortest paths
    """
    n = graph.vcount()

    # Initialize distances with infinity for all vertices except source
    distances = {i: float('infinity') for i in range(n)}
    distances[source] = 0

    # Dictionary to store the previous vertex in the shortest path
    previous = {i: None for i in range(n)}

    # Priority queue for efficient minimum distance extraction
    # Format: (distance, vertex)
    priority_queue = [(0, source)]

    # Set to track processed vertices
    processed = set()

    # Track number of destinations found
    destinations_found = 0

    while priority_queue and destinations_found < len(destinations):
        # Get vertex with minimum distance
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # Skip if already processed
        if current_vertex in processed:
            continue

        # Mark as processed
        processed.add(current_vertex)

        # Check if this is a destination
        if current_vertex in destinations:
            destinations_found += 1

```

```python
        # Process all adjacent vertices
        for edge in graph.incident(current_vertex):
            # Get the neighbor vertex
            neighbor = graph.es[edge].target if graph.es[edge].source == current_vertex else graph.es[edge].source

            # Skip if already processed
            if neighbor in processed:
                continue

            # Get edge weight
            weight = graph.es[edge]["weight"]

            # Calculate potential new distance
            distance = current_distance + weight

            # Update if we found a better path
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = current_vertex
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances, previous


def reconstruct_path(previous, start, target):
    """
    Reconstruct the path from start to target vertex

    Args:
        previous: Dictionary of previous vertices
        start: Starting vertex
        target: Target vertex

    Returns:
        List representing the path from start to target
    """
    path = []
    current = target

    while current is not None:
        # Add 1 to convert 0-based indices back to waypoint numbers
        path.append(current + 1)
        current = previous[current]

    # Reverse to get path from start to target
    return path[::-1]


def main():
    # Create graph based on the map
    g = create_escape_route_graph()

```

```python
    # Define source (waypoint 1) and potential destinations
    source = 0  # 0-based index for waypoint 1
    destinations = [5, 7, 8, 14, 15, 21]  # 0-based indices for waypoints 6, 8, 9, 15, 16, 22

    # Run our custom Dijkstra's algorithm
    distances, previous = dijkstra(g, source, destinations)

    # Print distances to potential destinations
    print("Distances from waypoint 1 to potential destinations:")
    for dest in destinations:
        waypoint_num = dest + 1  # Convert to 1-based for display
        print(f"Waypoint {waypoint_num}: {distances[dest]}")

    # Find the destination with minimum distance (most likely escape route)
    min_distance = float('infinity')
    most_likely_destination = None

    for dest in destinations:
        if distances[dest] < min_distance:
            min_distance = distances[dest]
            most_likely_destination = dest

    # Convert back to 1-based waypoint number for display
    most_likely_waypoint = most_likely_destination + 1

    # Reconstruct and display the most likely escape route
    most_likely_path = reconstruct_path(previous, source, most_likely_destination)

    print(f"\nMost likely escape route: Waypoint {most_likely_waypoint}")
    print(f"Total distance (sum of weights): {min_distance}")
    print(f"Path: {' → '.join(map(str, most_likely_path))}")

    # Print all paths to potential destinations
    print("\nAll escape routes:")
    for dest in destinations:
        waypoint_num = dest + 1  # Convert to 1-based for display
        path = reconstruct_path(previous, source, dest)
        print(f"To waypoint {waypoint_num} (distance {distances[dest]}): {' → '.join(map(str, path))}")

    # Alternative: Use igraph's built-in shortest_paths function
    print("\nVerifying with igraph's built-in function:")
    for dest in destinations:
        waypoint_num = dest + 1  # Convert to 1-based for display
        path = g.get_shortest_paths(source, dest, weights='weight')[0]
        path_weights = [g.es[g.get_eid(path[i], path[i + 1])]["weight"] for i in range(len(path) - 1)]
        total_weight = sum(path_weights)
        # Convert 0-based indices to 1-based waypoint numbers
        path_waypoints = [p + 1 for p in path]
        print(f"To waypoint {waypoint_num} (distance {total_weight}): {' → '.join(map(str, path_waypoints))}")
```

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