Report Project 3

Suhaib Abdi Muhummed, muhummed@kth.se

Course code: IX1500_HT2024
Date: 2024-10-21

1 Sammanfattning med Resultat

I detta projekt har vi implementerat två huvuduppgifter. I **Task 1** implementerades Kruskals algoritm för att konstruera ett Minimum Spanning Tree (MST) från en graf. Algoritmen fungerar genom att successivt välja kanter med lägst vikt och lägga till dem i trädet, samtidigt som cykler undviks med hjälp av Union-Find datastrukturen.

Resultaten visar att algoritmen korrekt identifierar ett MST för den givna grafen, vilket visualiserades med hjälp av NetworkX. Både ursprungsgrafen och det slutgiltiga MST visualiserades för att verifiera resultatet. Den teoretiska tidskomplexiteten för Kruskals algoritm är $O(E \log E)$, där E är antalet kanter i grafen, vilket är effektivt för glesa grafer.

I **Task 2** undersökte vi både användningen av specifika länkar mellan städer och konsekvenserna av ett länkfel i nätverket. För de givna länkarna antogs en trafikkapacitet, och för övriga länkar genererades kapaciteter slumpmässigt mellan 1 och 10. Därefter beräknades vikterna som inversen av kapaciteten. Vi använde olika algoritmer för att hitta den mest optimerade rutten från Stockholm till varje stad med minimal vikt (kostnad).

Vi analyserade även vad som händer vid ett länkfel genom att simulera ett avbrott mellan Göteborg och Lund. Genom att använda Dijkstra's algoritm visade vi hur trafiken automatiskt omdirigerades via Stockholm, vilket påverkade den totala vikten av rutten.

NOTERA att data och resultat kan variera varje gång algoritmen körs eftersom kapaciteten för de övriga länkarna genereras slumpmässigt. Detta innebär att de specifika vikterna och rutterna kan se olika ut vid varje körning.

1.1 Resultat Task 1

Kruskals algoritm väljer kanterna i stigande ordning baserat på deras vikter. Med hjälp av Union-Find säkerställs att inga cykler skapas när nya kanter läggs till. Det resulterande trädet förbinder alla noder och minimerar den totala summan av kantvikter, vilket gör det till ett MST.

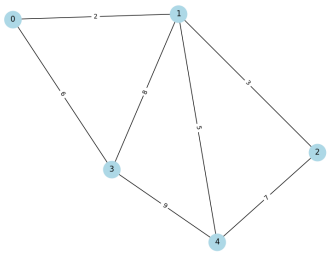

Figure 1: Initial Graf (före MST)

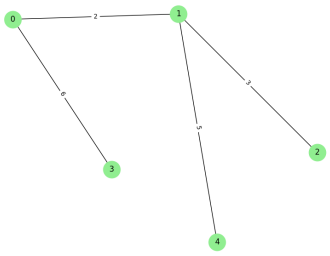

Figure 2: Minimum Spanning Tree (efter MST)

1.2 Resultat Task 2

1.2.1 Karta över Sverige

I figuren nedan visas Sveriges karta, där varje stad representerar en nod och varje länk representerar en väg mellan två städer.

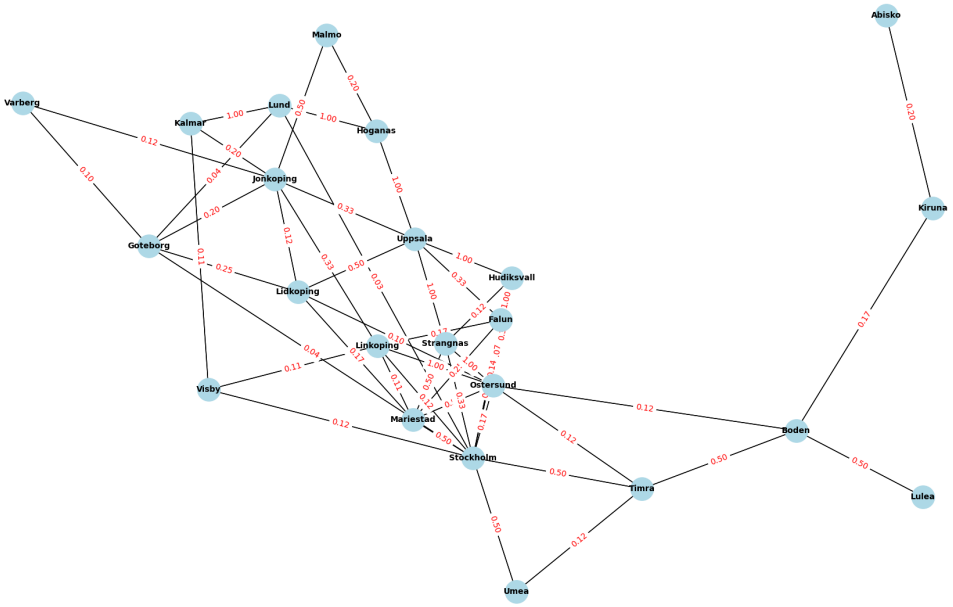

Figure 3: Sveriges karta

1.2.2 Kortaste vägen mellan Stockholm och Abisko

Tabellen nedan visar den kortaste vägen mellan Stockholm och Abisko, inklusive vikterna (invers kapacitet) för varje del av rutten.

Route (Stockholm to Abisko)	**Weight (Inverse Capacity)**
Stockholm → Umeå	0.11
Umeå → Timrå	0.11
Timrå → Boden	0.10
Boden → Kiruna	0.17
Kiruna → Abisko	0.12
Total Weight	**0.61**

Table 1: Kortaste vägen från Stockholm till Abisko och total vikt.

Tabellen nedan visar körningstiderna mellan Stockholm och Abisko för de olika algoritmerna som användes för att beräkna den kortaste vägen. Dijkstra's algoritm, Bellman-Ford-algoritmen och Floyd-Warshall-algoritmen jämfördes för att se vilken som var mest effektiv.

Algorithm	**Execution Time (seconds)**
Dijkstra's Algorithm	0.0000523972
Bellman-Ford Algorithm	0.0005442380
Floyd-Warshall Algorithm	0.0018004534

Table 2: Körningstider för olika algoritmer.

1.2.3 Länkfel mellan Göteborg och Lund

I sista delan av Task 2 har vi analyserat kortaste vägen mellan Göteborg och Lund med och utan ett länkfel mellan de två städerna.

Scenario	**Tid (sek)**	**Kortaste väg och totalvikt**
Före länkfel	0.0000618	Gbg → Lund (Vikt: 0.04) Totalvikt: 0.04
Efter länkfel	0.0000578	Gbg → Sthlm → Lund Vikt: 0.04 + 0.03 = 0.07

Table 3: Resultat före och efter länkfel mellan Göteborg och Lund.

2 Formler och Ekvationer

2.1 1. Minimum Spanning Tree (MST)

En viktad graf består av noder och kanter, där varje kant $e(u, v)$ har en vikt $w(u, v)$. Ett MST är ett träd som förbinder alla noder med den minimala summan av kantvikter utan att skapa cykler.

$$\text{MST}(G) = \min \sum_{e \in T} w(e)$$

där T är uppsättningen av kanter som förbinder alla noder i grafen G.

2.2 2. Kruskals Algoritm

Kruskals algoritm fungerar genom att sortera alla kanter efter vikt och iterativt lägga till dem i MST om de inte skapar cykler, med hjälp av Union-Find-strukturen.

2.2.1 Union-Find Datastruktur

Union-Find används för att hålla reda på sammanhängande komponenter och undvika cykler. Den består av två operationer:

$\text{Find}(u)$: Returnerar roten av mängden som innehåller noden u.

$\text{Union}(u, v)$: Slår samman två mängder om de tillhör olika komponenter.

2.3 Dijkstra's Algoritm: Matematiska Formulering

2.3.1 1. Inledande tillstånd

Låt $G(V, E)$ vara en viktad graf, där V är mängden av noder och E är mängden av kanter. För varje nod u, låt $d(u)$ representera det kortaste avståndet från startnoden till noden u. I början sätts:

$$d(u) = \begin{cases} 0 & \text{om } u = \text{start} \\ \infty & \text{om } u \neq \text{start} \end{cases}$$

2.3.2 2. Algoritmens huvudsakliga steg

Algoritmen upprätthåller en prioritetskö som alltid väljer noden u med lägst kostnad $d(u)$. För varje nod u, itererar algoritmen över alla dess grannar v (som är kopplade via en kant (u, v)) och uppdaterar kostnaden för att nå v om den nya vägen via u är kortare än den tidigare beräknade kostnaden:

$$d(v) = \min(d(v), d(u) + w(u, v))$$

Här är:

- $d(u)$: Kortaste avståndet från startnoden till u.
- $w(u, v)$: Vikten (kostnaden) på kanten mellan u och v.
- $d(v)$: Kortaste avståndet till v som ska uppdateras.

2.3.3 3. Process

När en nod u har besökts och alla dess grannar har kontrollerats, sätts noden som bearbetad, och algoritmen flyttar till noden v med lägst $d(v)$ i prioritetskön. Algoritmen fortsätter tills alla noder har besökts, eller tills prioritetskön är tom.

3 Diskussion

I Task 2 visade resultaten att, baserat på körningstiderna som visas i tabellen, Dijkstra's algoritm är den bäst optimerade för att beräkna den kortaste vägen i denna typ av problem där vi har icke-negativa vikter på länkarna. Dijkstra's algoritm är särskilt effektiv i grafen vi har använt eftersom den har en relativt liten komplexitet:

$$O((V + E) \log V)$$

Där V är antalet noder och E är antalet kanter i grafen.

Bellman-Ford-algoritmen hanterar negativa vikter, men eftersom våra vikter är positiva, är den överflödig för detta specifika problem och har en högre komplexitet:

$$O(VE)$$

vilket resulterar i en längre körningstid jämfört med Dijkstra's algoritm.

3.1 Varför Dijkstra's algoritm är snabbare efter länkfel?

Det kan verka kontraintuitivt att Dijkstra's algoritm tog längre tid före länkfelet än efter länkfelet, trots att vägen efter länkfelet är längre.

När länken mellan Göteborg och Lund är tillgänglig (före länkfelet), behövde algoritmen bara utforska en direkt väg med en lägre kostnad. Trots detta kan fler interna operationer ha utförts på grund av strukturen i prioritetskön. Efter länkfelet omdirigerades trafiken genom Stockholm, vilket innebar att algoritmen kunde bearbeta och uppdatera noderna mer effektivt. Detta minskade den totala körningstiden trots att den slutliga vägen var längre.

Detta belyser att algoritmens körningstid inte enbart beror på längden på den slutliga vägen utan också på antalet noder som utforskas och hur dessa uppdateras under körningen.

4 Kod

Här nedan följer den fullständiga koden (2 sidor). Den finns också tillgänglig på Github: https://github.com/suhaib921/IX150$_H$T24_50140 − *.git*

4.1 Task 1

```
import networkx as nx
import matplotlib.pyplot as plt

# Union-Find data structure for cycle detection
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    # Find with path compression
    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    # Union by rank
    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            # Union by rank to keep tree flat
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

# Kruskal's algorithm to find MST
def kruskal(n, edges):
    # Sort edges by their weights
    edges.sort(key=lambda x: x[2])

    uf = UnionFind(n)
    mst = []  # To store the minimum spanning tree

    for u, v, weight in edges:
```

```
        # If u and v are in different sets, add the edge to the MST
        if uf.find(u) != uf.find(v):
            mst.append((u, v, weight))
            uf.union(u, v)

    return mst

# Example graph with 5 vertices and edges with weights
# (u, v, weight) where u and v are vertices
edges = [
    (0, 1, 2), (0, 3, 6),
    (1, 2, 3), (1, 3, 8), (1, 4, 5),
    (2, 4, 7), (3, 4, 9)
]

n = 5  # Number of vertices in the graph

# Running Kruskal's algorithm to find the MST
mst = kruskal(n, edges)

# Output the edges in the MST
print("Minimum Spanning Tree edges (with weights):")
for u, v, weight in mst:
    print(f"{u} -- {v} == {weight}")

# Create a graph with NetworkX
G = nx.Graph()

# Add edges to the original graph
for u, v, weight in edges:
    G.add_edge(u, v, weight=weight)

# Draw the original graph with weights
plt.figure(figsize=(8, 6))
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=700, font_size=12)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.title("Original Graph with Weights")
plt.show()

# Create another graph for the MST
MST = nx.Graph()
for u, v, weight in mst:
    MST.add_edge(u, v, weight=weight)
```

```
# Draw the Minimum Spanning Tree (MST)
plt.figure(figsize=(8, 6))
nx.draw(MST, pos, with_labels=True, node_color='lightgreen', node_size=700, font_size=12)
mst_edge_labels = nx.get_edge_attributes(MST, 'weight')
nx.draw_networkx_edge_labels(MST, pos, edge_labels=mst_edge_labels)
plt.title("Minimum Spanning Tree (MST)")
plt.show()
```

4.2 Task 2

4.2.1 Task 2

```
import networkx as nx
import random

# Define the cities
cities = ["Abisko", "Boden", "Falun", "Goteborg", "Hoganas", "Hudiksvall", "Jonkoping",
          "Kalmar", "Kiruna", "Lidkoping", "Linkoping", "Lulea", "Lund", "Malmo",
          "Mariestad",  "Ostersund", "Stockholm", "Strangnas", "Timra", "Uppsala",
          "Umea", "Varberg", "Visby"]

# Define the links between cities (connections only, no capacity yet)
links = [
    (15, 18), (13, 17), (3, 16), (6, 3), (18, 6), (12, 2), (15, 11), (19, 21), (7, 8),
    (19, 2), (7, 4), (21, 17), (17, 11), (23, 11), (10, 7), (16, 17), (10, 20),
    (13, 5), (3, 17), (7, 22), (15, 10), (16, 18), (11, 16), (17, 23), (18, 17),
    (20, 6), (11, 7), (9, 2), (3, 20), (4, 17), (19, 17), (7, 20), (22, 4), (14, 7),
    (15, 17), (6, 17), (20, 5), (16, 15), (11, 3), (9, 1), (4, 10), (5, 14), (6, 16),
    (16, 19), (13, 8), (8, 23), (16, 10), (4, 13), (2, 16), (15, 3), (20, 18)
]

# Predefined capacities for specific links (provided in the task)
predefined_capacities = {
    ('Stockholm', 'Goteborg'): 25,
    ('Stockholm', 'Lund'): 30,
    ('Goteborg', 'Lund'): 25,
    ('Stockholm', 'Falun'): 15,
    ('Falun', 'Ostersund'): 15,
    ('Ostersund', 'Umea'): 15
}
# Function to calculate the inverse of capacity
def calculate_inverse_capacity(links, predefined_capacities):
    weighted_links = []
    for link in links:
        city1_index, city2_index = link
        city1 = cities[city1_index - 1]  # Adjust to 1-based indexing
```

```
        city2 = cities[city2_index - 1]

        # Check if the capacity for this link is predefined
        if (city1, city2) in predefined_capacities:
            capacity = predefined_capacities[(city1, city2)]
        elif (city2, city1) in predefined_capacities:
            capacity = predefined_capacities[(city2, city1)]
        else:
            # Assign random capacity between 1 and 10 for undefined links
            capacity = random.randint(1, 10)

        # Calculate the weight (inverse of capacity)
        weight = 1 / capacity

        # Add the link with weight to the list
        weighted_links.append((city1_index - 1, city2_index - 1, weight))

    return weighted_links

# Function to create the graph with weighted edges
def create_graph_with_weights():
    G = nx.Graph()
    G.add_nodes_from(range(len(cities)))

    # Generate the weighted links with calculated inverse capacities
    weighted_links = calculate_inverse_capacity(links, predefined_capacities)

    # Add the edges with weights to the graph
    for city1, city2, weight in weighted_links:
        G.add_edge(city1, city2, weight=weight)

    return G
```

4.2.2 Task 2

```
import heapq

def dijkstra_algorithm(graph, start):
    queue = [(0, start)]  # (distance, node)
    distances = {node: float('inf') for node in graph.nodes}
    distances[start] = 0

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

        # Skip processing if we already have a shorter distance recorded
```

```
        if current_distance > distances[current_node]:
            continue

        for neighbor, attributes in graph[current_node].items():
            distance = current_distance + attributes['weight']
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances
```