Ime, priimek / Name : Urban Potocnik

02: Grafovski algoritmi v TK omrežjih
Graph algorithms for TC Networks

1 Uvod / Introduction

IKT omrežja lahko prestavimo s pomočjo grafa, kjer so mrežni elementi (usmerjevalnik..) predstavljeni kot vozlišča, ter komunikacijske povezave kot povezave grafa. Povezavam lahko določimo tudi usmeritev ali ceno prenosa oziroma kapaciteto. V vaji bomo konstruirali mrežne grafe in jih analizirali s pomočjo algoritmov.

ICT networks can frequently be presented as graphs where each network element (router, buffer…) translates into a graph node and each connection between elements translates into a graph edge. In some cases, edges can also be given direction (non-symmetric connections) and price / capacity. In this exercise we will learn to construct network graphs and analyse them using the breadth/depth-first search.

2 Uporaba / Using NetworkX library

Uporabljali bomo knjižnico `networkx`, ki omogoča ustvarjanje grafov in njihovo analizo. Navodila za uporabo so tukaj:

https://networkx.github.io/documentation/stable/tutorial.html

```
!pip install networkx
!pip install scipy
import matplotlib.pyplot as plt
import networkx as nx
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

2.1 Create graph, add nodes and edges

Kreiraj testni graf.

Networkx methods to create graph, add nodes and edges:

- nx.Graph() : creates empty graph
- nx.DiGraph() : creates empty directed graph

```
# Create empty graph
G = nx.Graph()
# Altermative:
#G = nx.empty_graph(8);

# Dodajaj povezave / Add edges
G.add_edge(0,1); # Add connection between node 0 and 1
G.add_edge(0,2);
G.add_edges_from([(1,3), (2,3), (3,4), (3,5), (4,6), (5,6), (6,7)]); #We can also add several connections at once

# add weight
G.edges[1,3]['weight'] = 0.3
```

Izpiši (print) lastnosti grafa G: edges, nodes.

Print edges and nodes variables of the graph G.

Vozlišča (Nodes): [0, 1, 2, 3, 4, 5, 6, 7]

Povezave (Edges): [(0, 1), (0, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 6), (5, 6), (6, 7)]

2.2 Izriši graf / Visualize graph

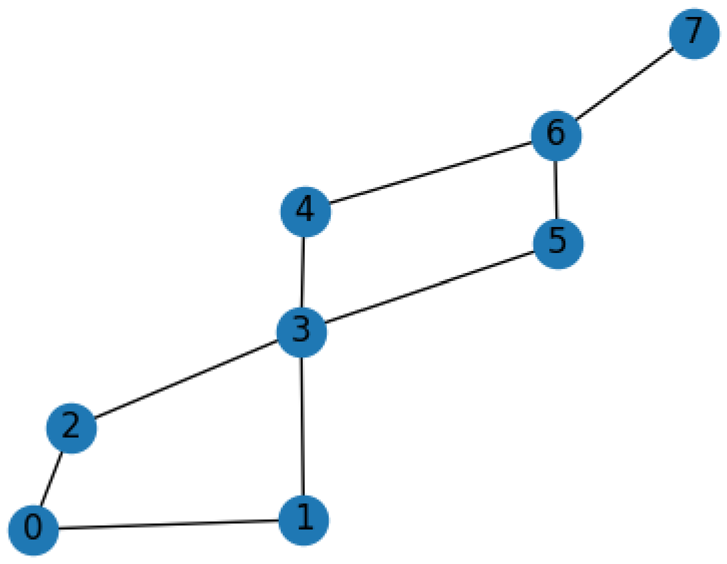
We need to define dictionary pos with [x,y] coordinates of the nodes. Optionally we can define dictionary of node labels.

```
# prepare an empty figure
plt.figure(figsize = [4,3])

## Graph layout / node positions
pos = nx.spring_layout(G)
#pos = nx.kamada_kawai_layout(G)
#pos = nx.circular_layout(G);
## Option: Define node positions manually as [x,y]
#pos = { 0 : [0,1], 1 : [1,2], 2 : [1,0], 3 : [2,1], 4 : [3,2], 5 : [3,0], 6 :
[4,1], 7 : [5,1] }

print(pos)
nx.draw(G,pos)

#Define node labels as a dictionary, key => node number, value => node name
#labels={}
#labels[0] = 'Node 0'
#labels[1] = 'Node 1'
```

```
#labels[2] = 'Node 2'
#labels[3] = 'Node 3'
#labels[4] = 'Node 4'
#labels[5] = 'Node 5'
#labels[6] = 'Node 6'
#labels[7] = 'Node 7'
#nx.draw_networkx_labels(G,pos,labels,font_size=10); #add labels

plt.show(); #show the graph
```

Test different predefined options of positions as well as manual positioning of nodes.

Resulting graph:



2.3 Matrika sosednosti / Adjacency matrix

Print adjacency matrix :

```
# 2.4 : Adjacency matrix
adjM = nx.to_numpy_array(G)
print(adjM)
```

Insert result. What the numbers represent?

Matrika sosednosti (Adjacency Matrix): [[0. 1. 1. 0. 0. 0. 0. 0.] [1. 0. 0. 0.3 0. 0. 0. 0.] [1. 0. 0. 1. 0. 0. 0. 0.] [0. 0.3 1. 0. 1. 1. 0. 0.] [0. 0. 0. 1. 0. 0. 1. 0.] [0. 0. 0. 1. 0. 0. 1. 0.] [0. 0. 0. 0. 1. 1. 0. 1.] [0. 0. 0. 0. 0. 0. 1. 0.]]

2.4 Kreiraj in izriši usmerjen graf / Create and visualize directed graph

Copy code to generate graph, and change Graph to DiGraph and visualize resulting graph.

DG = nx.DiGraph()

DG.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 6), (5, 6), (6, 7)])

plt.figure(figsize=[4, 3])
nx.draw(DG, pos, with_labels=True, arrows=True)
plt.show()
adjM_directed = nx.to_numpy_array(DG)
print("Matrika sosednosti za usmerjen graf (Adjacency Matrix for Directed Graph):\n", adjM_directed)

Print adjacency matrix of a directed graph. What is the difference ?

Matrika sosednosti za usmerjen graf (Adjacency Matrix for Directed Graph): [[0. 1. 1. 0. 0. 0. 0. 0.] [0. 0. 0. 1. 0. 0. 0. 0.] [0. 0. 0. 1. 0. 0. 0. 0.] [0. 0. 0. 0. 1. 1. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 0.] [0. 0. 0. 0. 0. 0. 1. 0.] [0. 0. 0. 0. 0. 0. 0. 1.] [0. 0. 0. 0. 0. 0. 0. 0.]]

---

3 Algoritmi na grafih / Graph algorithms

3.1 Sprehod po drevesu v globino / Graph (tree) traversal: Depth First Search

Vhodni podatek je drevo, vozlišča (node, vertice) lahko predstavljajo poljubni podatek, v našem primeru je to mesto. Povezave (edges) predstavljajo ceste in razdalje med kraji.

Input data: graph (tree) with nodes (vertices) representing cities of Slovenia, and edges representing roads between cities.

Load and plot input graph:

```python
# Load graph
G2 = nx.read_graphml('Slo_cities.graphml')
print(G2.nodes)
print(G2.edges)

# prepare an empty figure
plt.figure()

#Plot the graph
nx.draw_networkx(G2, with_labels=True, font_size = 8)
```

Algoritem Sprehod v globino / Depth first search Algorithm:

Description: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/?ref=lbp

Code:

```python
# DFS Algorithm
from DataStructures import Stack

# Perform node visit
def visit_node(node, visited, visited_order, stack, graph, counter_list):
    visited.add(node)
    visited_order.append((counter_list[0], node))
    print("Visit: ", counter_list[0], node)
    counter_list[0]+=1
    for sosed in graph.adj[node]:
        if sosed not in visited:
            stack.push(sosed)
    # preglej kaj je na skladu
    # stack.print()

def dfs_traversal(inputGr, start_node):
    # Create a set to store visited vertices
    visited = set()
    # Create stack to store nodes to visit
    stack = Stack()
    # iterator over visits
    counter_list = [0]
    # Resulting sequence
    node_order = []

    # ALGORITHM DFS
    node = start_node
    visit_node(node, visited, node_order, stack, inputGr, counter_list)
    while stack.size() > 0:
        node = stack.pop()
        if node not in visited:
            visit_node(node, visited, node_order, stack, inputGr, counter_list)
    return node_order

nodes_dfs = dfs_traversal(G2, 'Ljubljana')
print("Visited nodes: ", nodes_dfs)
```

Test algorithm by selecting different starting points (cities) and check order of visited cities. Example result:

```
Obiskana vozlišča (Visited nodes): [(0, 'Ljubljana'), (1, 'Kocevje'), (2, 'Novo mesto'), (3, 'Krsko'), (4, 'Celje'), (5, 'Velenje'), (6, 'Dravograd'), (7, 'Maribor'), (8, 'Ptuj'), (9, 'Murska Sobota'), (10, 'Postojna'), (11, 'Koper'), (12, 'Nova Gorica'), (13, 'Jesenice'), (14, 'Kranj')]
```

3.2 Izrišite graf sprehoda v globino / Visualize DFS traversal

Create DiGraph and add edges from nodes_dfs, plot resulting graph.

```python
# Create a directed graph
G3 = nx.DiGraph()

# Add edges from the list nodes_dfs to the graph
currN = nodes_dfs[0][1]

# prepare an empty figure
plt.figure()

#Plot the graph
nx.draw_networkx(G3, with_labels=True, font_size = 8)
```

3.3 Sprehod v širino / Breadth First Search : number of edges to reach a node

Uporaba sprehoda v širino: izračun števila povezav, da dosežemo vsako mesto iz začetnega mesta.

Example of using BFS to calculate minimum number of edges to reach each node from the given starting point.

```python
# 3.3 BFS Algorithm

## BFS Problem : Minimum number of edges to each node
## https://www.geeksforgeeks.org/minimum-number-of-edges-between-two-vertices-of-a-graph/?ref=lbp
from DataStructures import Queue

# Perform node visit
def visit_node(node, node_level, visited, visited_order, queue, graph, counter_list):
    visited.add(node)
    visited_order.append([node, node_level])
    node_level += 1

    #print("Visit: ", counter_list[0], node)
    counter_list[0]+=1
    for sosed in graph.adj[node]:
        if sosed not in visited:
            #if stack.indexOf(sosed) == None:
            queue.enqueue([sosed,node_level])
    # preglej kaj je v vrsti
    # queue.print()

# BFS Algorithm
# inputGr : graph
# start_node : starting node
# return : list of nodes in the order of visits
def min_edges(inputGr, start_node):
    # Create a set to store visited vertices
    visited = set()
    # Store nodes to be visited in the queue
    queue = Queue()
    # counter : the number of visits
    counter_list = [0]
    # Resulting sequence
    node_order = []

    # ALGORITHM BFS
    node = start_node
    visit_node(node, 0, visited, node_order, queue, inputGr, counter_list)
    while queue.size() > 0:
        [node, level] = queue.dequeue()
        if node not in visited:
            visit_node(node, level, visited, node_order, queue, inputGr, counter_list)
    return node_order
```

Test min_edges on a graph, define start node. Print out the results.

Results:

Obiskana vozlišča (Visited nodes): [['Ljubljana', 0], ['Kranj', 1], ['Novo mesto', 1], ['Celje', 1], ['Postojna', 1], ['Kocevje', 1], ['Jesenice', 2], ['Krsko', 2], ['Maribor', 2], ['Velenje', 2], ['Nova Gorica', 2], ['Koper', 2], ['Murska Sobota', 3], ['Ptuj', 3], ['Dravograd', 3]]

3.4 Using networkx for graph traversal

Networkx library provides many methods to analyse and traverse graphs:

https://networkx.org/documentation/stable/reference/algorithms/traversal.html

- dfs_edges, dfs_tree: returns edges and tree according to DFS traversal
- dfs_successors

- bfs_edges, bfs_tree

```
startNode = 'Ljubljana'
print (list(nx.bfs_edges(G2,startNode)))

K = nx.bfs_tree(G2,startNode)

plt.figure()
nx.draw_networkx(K, pos = nx.kamada_kawai_layout(K), with_labels=True, font_size = 8)
plt.show()
```

Insert BFS graph:

Test method dfs_tree and insert resulting DFS graph:

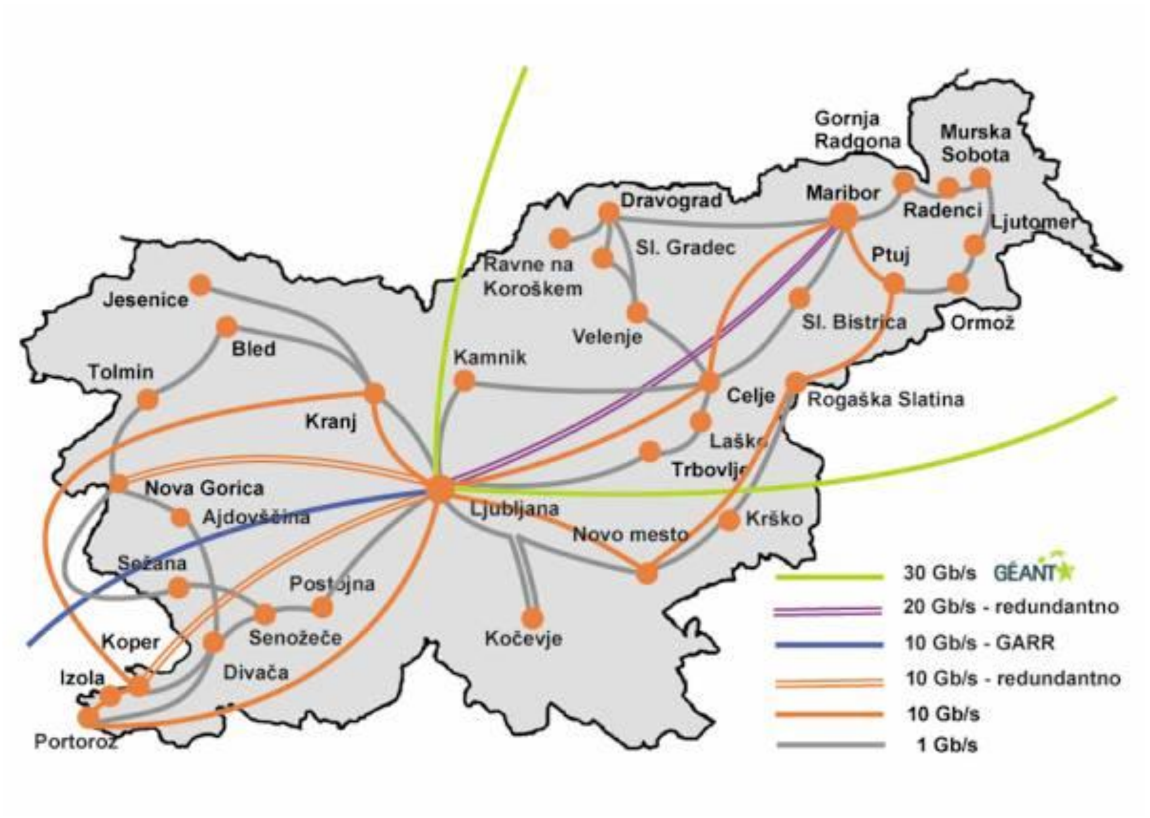
Povezave BFS (BFS edges): [('Ljubljana', 'Maribor'), ('Ljubljana', 'Celje'), ('Ljubljana', 'Koper'), ('Ljubljana', 'Kranj'), ('Ljubljana', 'Novo mesto'), ('Ljubljana', 'Portoroz')]

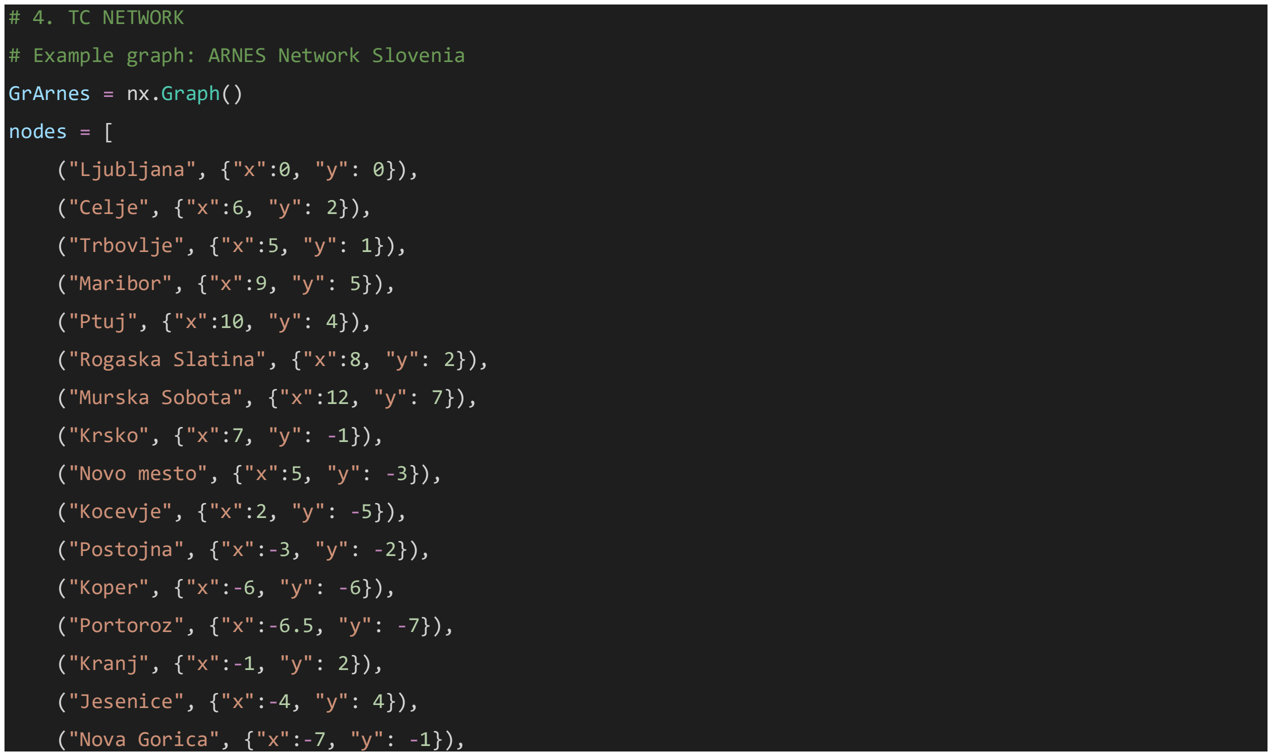
---

4 Analiza TK omrežij z grafi / TC Network analysis

Primer: Graf logičnih povezav omrežja Arnes

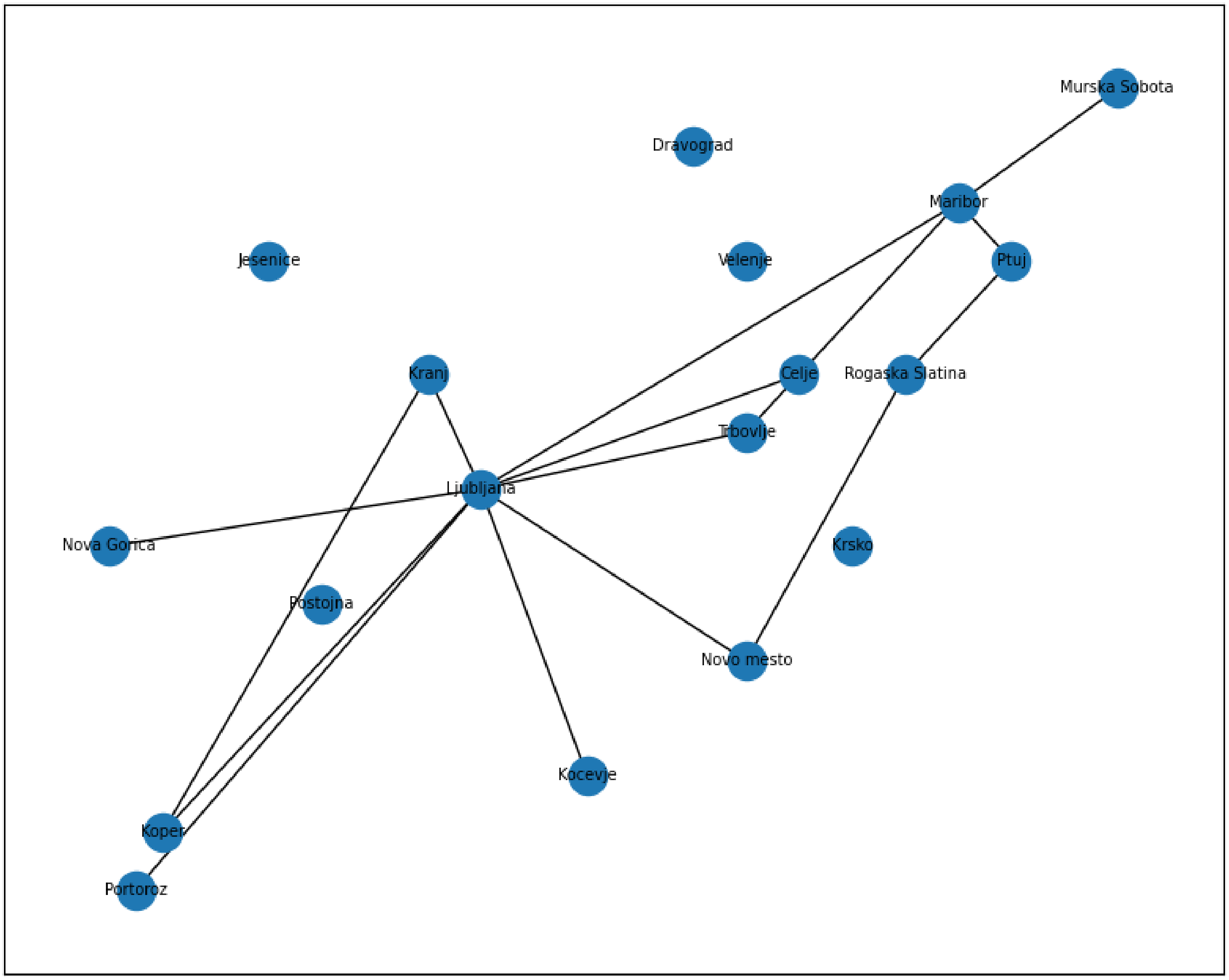
Example TC Network: ARNES network of Slovenia




```
# 4. TC NETWORK
# Example graph: ARNES Network Slovenia
GrArnes = nx.Graph()
nodes = [
    ("Ljubljana", {"x":0, "y": 0}),
    ("Celje", {"x":6, "y": 2}),
    ("Trbovlje", {"x":5, "y": 1}),
    ("Maribor", {"x":9, "y": 5}),
    ("Ptuj", {"x":10, "y": 4}),
    ("Rogaska Slatina", {"x":8, "y": 2}),
    ("Murska Sobota", {"x":12, "y": 7}),
    ("Krsko", {"x":7, "y": -1}),
    ("Novo mesto", {"x":5, "y": -3}),
    ("Kocevje", {"x":2, "y": -5}),
    ("Postojna", {"x":-3, "y": -2}),
    ("Koper", {"x":-6, "y": -6}),
    ("Portoroz", {"x":-6.5, "y": -7}),
    ("Kranj", {"x":-1, "y": 2}),
    ("Jesenice", {"x":-4, "y": 4}),
    ("Nova Gorica", {"x":-7, "y": -1}),
    ("Velenje", {"x":5, "y": 4}),
    ("Dravograd", {"x":4, "y": 6}),
]
GrArnes.add_nodes_from(nodes)

# Pridobi pozicije iz podatkov
position = {}
for i in GrArnes.nodes:
    position[i] = [GrArnes.nodes[i]['x'], GrArnes.nodes[i]['y']]
print(position)

# Add edges with weights (Gb)
edge_list = [
    ("Ljubljana","Maribor", {"Gb":20}),
    ("Ljubljana","Koper", {"Gb":20}),
    ("Ljubljana","Nova Gorica", {"Gb":20}),
    ("Ljubljana","Kranj", {"Gb":10}),
    ("Ljubljana","Celje", {"Gb":10}),
    ("Ljubljana","Novo mesto", {"Gb":10}),
    ("Ljubljana","Portoroz", {"Gb":10}),
    ("Koper","Kranj", {"Gb":10}),
    ("Novo mesto","Rogaska Slatina", {"Gb":10}),
    ("Ptuj","Rogaska Slatina", {"Gb":10}),
    ("Celje","Maribor", {"Gb":10}),
    ("Ptuj","Maribor", {"Gb":10}),

    ("Ljubljana","Trbovlje", {"Gb":1}),
    ("Celje","Trbovlje", {"Gb":1}),
    ("Ljubljana","Kocevje", {"Gb":1}),
    ("Maribor","Murska Sobota", {"Gb":1})
   # ("Kranj","Jesenice", {"Gb":1}),
]

# Prepare edges with weights
w_edges = []
for edge in edge_list:
    w_edges.append((edge[0],edge[1],edge[2]["Gb"]))
# Add edges to the graph
GrArnes.add_weighted_edges_from(w_edges)

# Test print
print(GrArnes.edges)
print(GrArnes.edges[('Ljubljana', 'Maribor')])

edge_labels = {}
```

```
for e in edge_list:
    #print(e[2]["Gb"])
    edge_labels[(e[0],e[1])] = e[2]["Gb"]
#print(edge_labels )

# Plot the graph
plt.figure(figsize=[10,8])
nx.draw_networkx(GrArnes, position, with_labels=True, font_size = 7)
#nx.draw_networkx_edge_labels(GrArnes, position, edge_labels=edge_labels, font_size = 9)
```

Insert resulting plot (graph):



Print adjacency matrix of the graph:

4.1 Connected components

https://networkx.org/documentation/stable/reference/algorithms/component.html

Preveri povezanost omrežja, ali so vsa vozlišča povezana v en graf.

Test connectedness of the network nodes.

```
# Povezanost
print('Graph is connected ? ', nx.is_connected(GrArnes))
print(list(nx.connected_components(GrArnes)))

connComp = list(nx.connected_components(GrArnes))
for comp in connComp:
    print('Connected subgraph: ', comp)
```

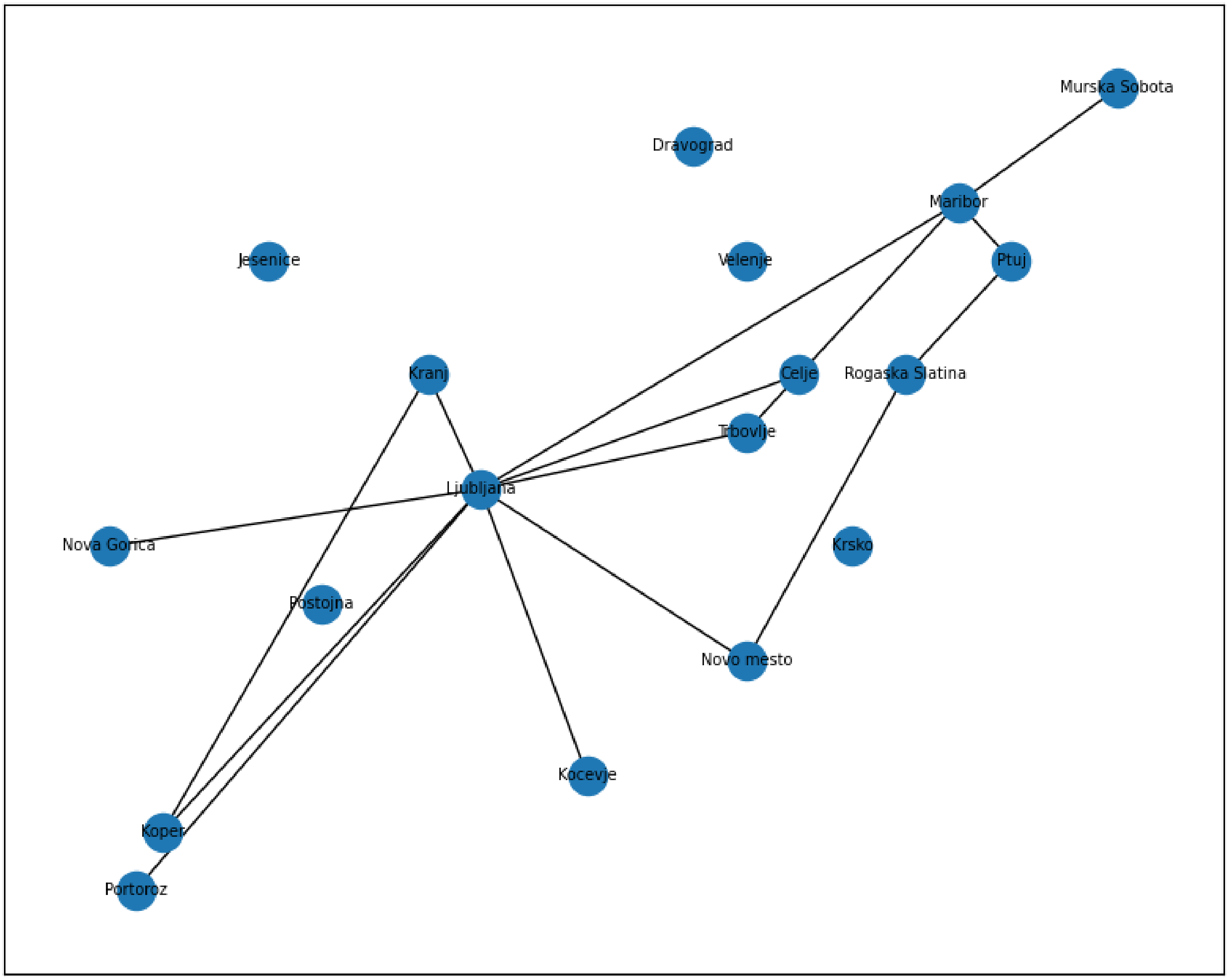
Result:

Obiskana vozlišča (Visited nodes): [(0, 'Ljubljana'), (1, 'Kocevje'), (2, 'Novo mesto'), (3, 'Krsko'), (4, 'Celje'), (5, 'Velenje'), (6, 'Dravograd'), (7, 'Maribor'), (8, 'Ptuj'), (9, 'Murska Sobota'), (10, 'Postojna'), (11, 'Koper'), (12, 'Nova Gorica'), (13, 'Jesenice'), (14, 'Kranj')]

4.2 Dodaj povezave / Add edges between nodes

According to image, add additional (missing) links between given cities (nodes), so that the network is fully connected.

Insert updated plot of the network:



4.3 Kritične točke / Articulation points

Zelo dober primer uporabe sprehoda v globino je algoritem za analizo omrežja in detekcijo kritičnih vozlišč. Kritična vozlišča so vozlišča, katerih izpad bi omrežje razdelil v dve podomrežji, med katerima ni povezave.

A good example of efficient use of DFS is detection of articulation points in a network.

An articulation point or cut vertex is any node whose removal (along with all its incident edges) increases the number of connected components of a graph. Articulation points are critical nodes in a network since their failure results in the network breaking apart into several disconnected subnetworks.

```
nx.draw(GrArnes,position)
nx.draw_networkx_labels(GrArnes,position); #dodamo oznake vozlisc
apts = list(nx.articulation_points(GrArnes))
print( 'Articulation points: ' , apts)
```

Which are critical nodes in the ARNES netwo rk?

Kritična vozlišča (Articulation points): ['Maribor', 'Ljubljana']

4.4 Optimizacija poti po grafu / Optimization: Shortest path algorithm

https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html

Naloga: želimo najti optimalno pot od začetne do končne točke. Uporabimo algoritem iskanja najkrajše poti (interno uporablja Dijsktra algoritem), ki vrne pot z najmanjšo vsoto uteži povezav.

Compute the shortest path between start and end node, so that the sum of edge weights is minimal.

```
# SHORTEST PATHS
start_node = 'Koper'
end_node = 'Maribor'
edgew = 'weight'

sp = nx.shortest_path(GrArnes, start_node, end_node, weight=edgew)
print('shortest_path: ', sp)

sp_length = nx.shortest_path_length(GrArnes, start_node, weight=edgew)
print('shortest_path_length: ', sp_length)

for p in nx.all_shortest_paths(GrArnes, start_node, end_node, weight=edgew):
    print(p)
```

Testiraj algoritem (različne začetne in končne točke) in ugotovi, kakšno pot je vrnil in kaj izpiše. Ali lahko obstaja več ekvivalentnih poti? Vstavi primer in komentiraj, kaj pomeni vsak izpis.

Test algorithm and present an example of resulting shortest path. Comment, what the printout of each method represents.

start_node = 'Koper' end_node = 'Maribor' sp = nx.shortest_path(GrArnes, start_node, end_node, weight='weight') print("Najkrajša pot (Shortest path):", sp)

Najkrajša pot (Shortest path): ['Koper', 'Ljubljana', 'Maribor']

4.5 Najnižja cena prenosa / Optimization of transfer costs

Modificiraj zgornji primer, tako da povezavam dodaš ceno prenosa. Ta naj bo določena kot 100 / Gb . Sedaj poišči optimalno pot z najmanjšo skupno ceno prenosa podatkov.

Kolikšna je najnižja cena prenosa med Koper in Murska Sobota?

Modify shortest path example. Add cost to each edge, calculated as 100 / Gb. Find path with minimum total cost.

What is the lowest transfer cost between Koper and Murska Sobota?

```
# Add costs to edges
for edge in GrArnes.edges:
    GrArnes.edges[edge]['cost'] = 100.0 / GrArnes.edges[edge]['weight']
```

Result:

Pot z najnižjo ceno (Path with lowest cost): ['Koper', 'Ljubljana', 'Maribor', 'Murska Sobota']

4.6 Algoritmi pretoka po grafu / Finding maximum flows

https://networkx.org/documentation/stable/reference/algorithms/flow.html

Želimo najti skupno največjo kapaciteto prenosa, ki ga omogočajo povezave med dvema mestoma. Podatki lahko tečejo po vzporednih povezavah.

The task is to find maximum flow (transfer capacity) possible between the start and end nodes, using all available parallel paths. Edge weight represents link transfer capacity in Gb.

```
# MAXIMUM FLOW
start_node = 'Koper'
end_node = 'Maribor'

flow_value, flow_dict = nx.maximum_flow(GrArnes, start_node, end_node, capacity='weight')
print('Total flow ', start_node, ' to ', end_node, ': ', flow_value)

for node in flow_dict:
    print(node, flow_dict[node])
```

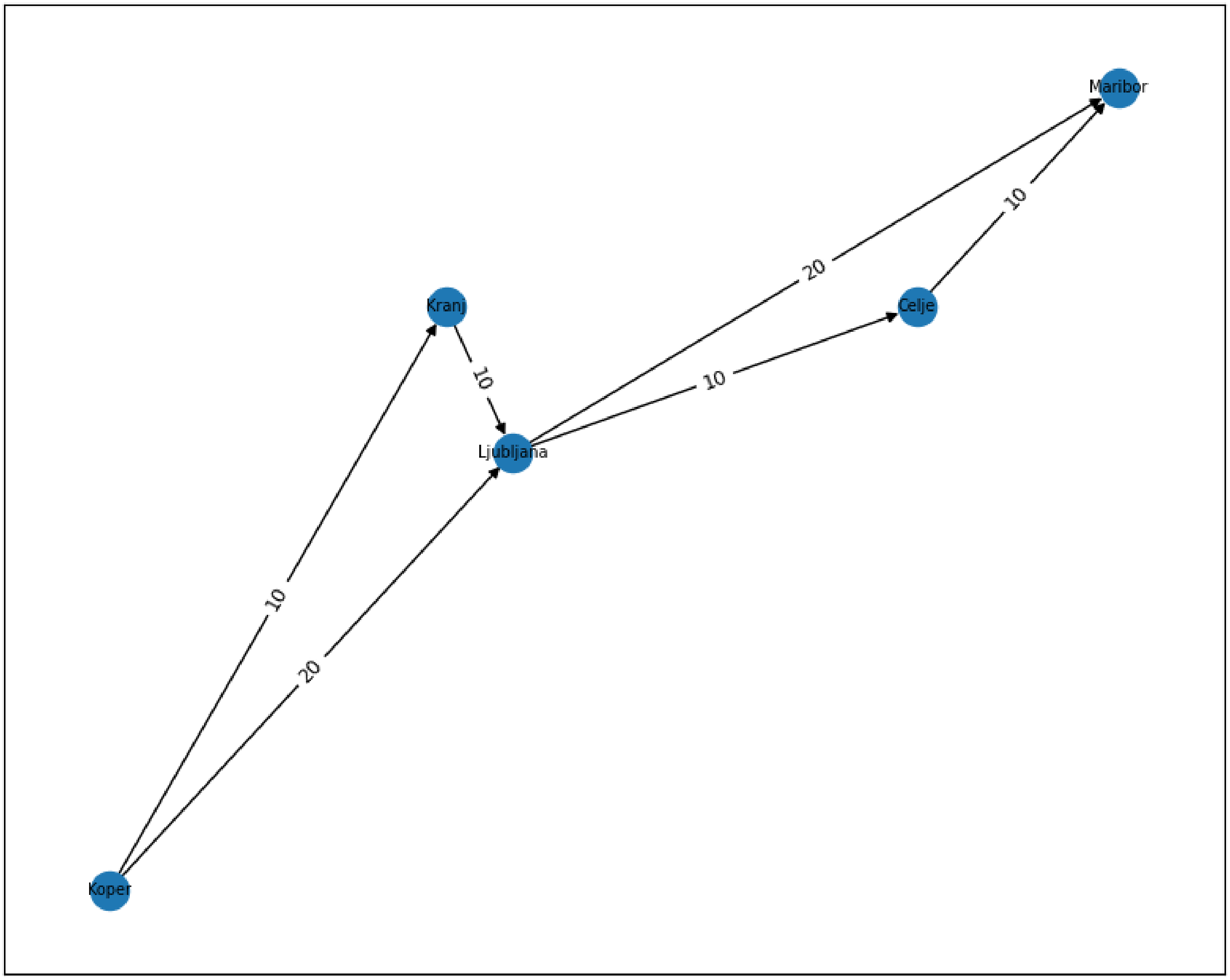
Results (max transfer) for pair of cities:

Maksimalni pretok (Maximum flow): 20.0

Ali ste opazili kakšno pomanjkljivost ali nelogični rezultat, če zamenjamo start in cilj, kaj je vzrok? *Is there something unexpected in the results, if we change starting and ending node, what is the issue?*

Pri algoritmu za iskanje maksimalnega pretoka lahko opazimo, da rezultati niso vedno simetrični, če zamenjamo začetno in končno vozlišče. To je posledica tega, da je pretok usmerjen in je odvisen od kapacitet povezav v smeri od začetnega do končnega vozlišča. Če so povezave asimetrične (npr. različne kapacitete v obe smeri), lahko dobimo različne rezultate.

4.7 Dodatno: vizualiziraj maksimalni pretok / Additional task: visualize max flow as directed graph



Graf prikazuje povezave, ki so bile uporabljene za maksimalni pretok med izbranima vozliščema. Številke na povezavah predstavljajo količino pretoka, ki teče skozi vsako povezavo.