Supporting Information Document

A Proposal for Optimal Emergency Vehicle Routing using Quantum Annealing.

Abhimanyu Deeraj

CCIR Future Scholars Programme, Cambridge Center of International Research, Cambridge, United Kingdom, CB4 0GA.

Data and Visualization for BQM Performance Statistics:

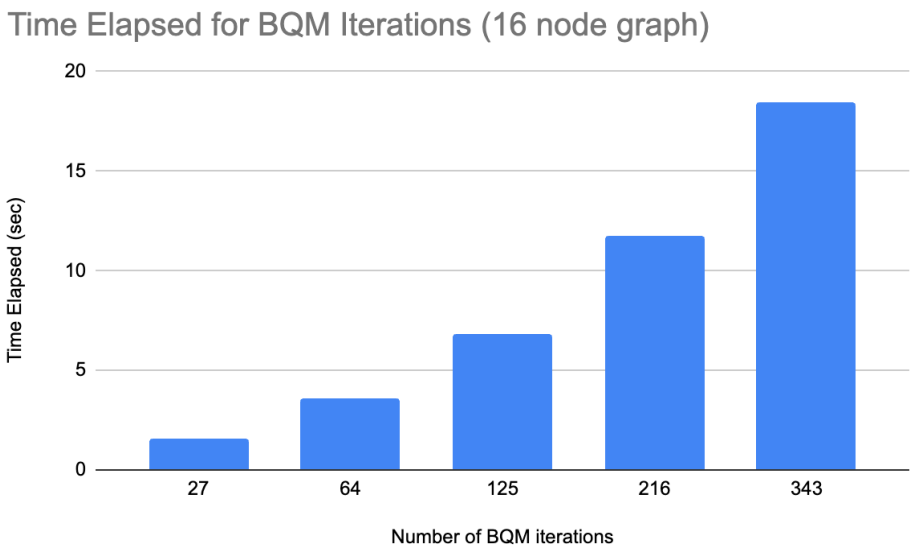


Figure S1. Created using Google Sheets. Time required to find the optimal route in a 16-node network graph under various BQM iteration sizes.

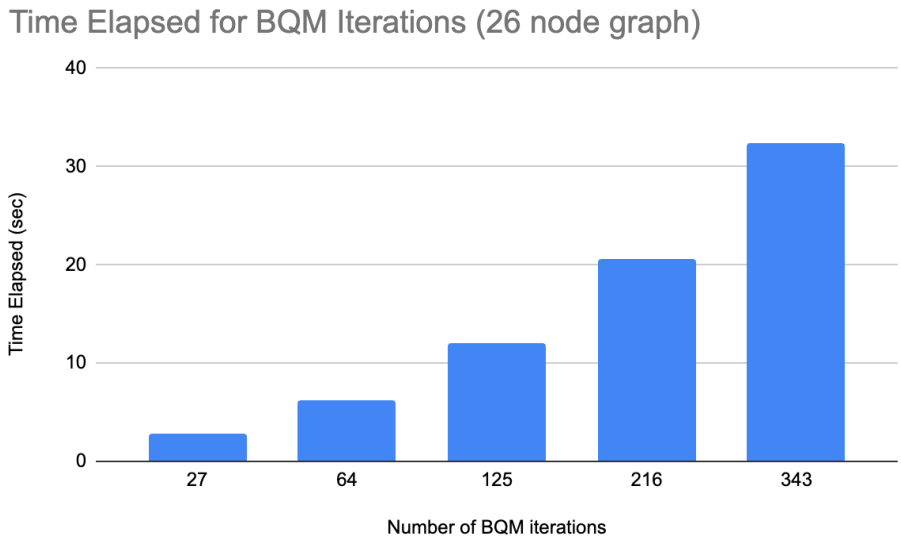


Figure S2. Created using Google Sheets. Time required to find the optimal route in a 26-node network graph under various BQM iteration sizes.

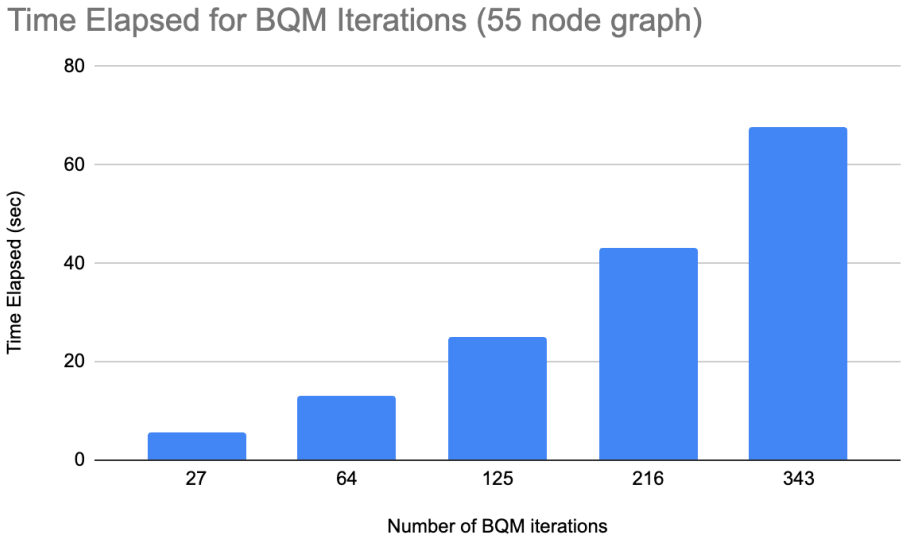


Figure S3. Created using Google Sheets. Time required to find the optimal route in a 55-node network graph under various BQM iteration sizes.

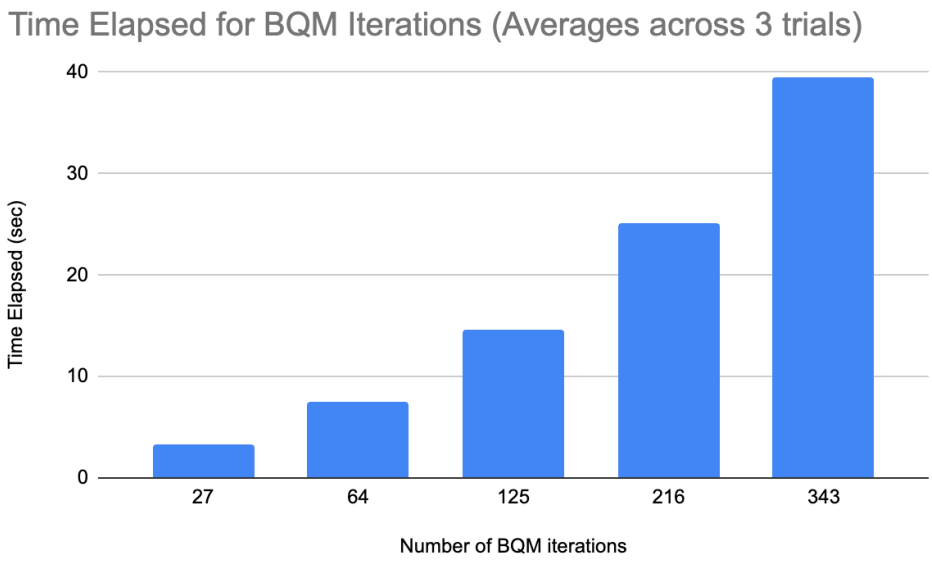


Figure S4. Created using Google Sheets. Average time required to find the optimal route across the three networks of varying complexities (16, 26, and 55 nodes) under various BQM iteration sizes.

Exhibit S5. Complete Source Code created using a Python Jupyter Notebook with D-Wave Ocean SDK and OpenStreetMap Python libraries.

Reference Sources for this codebase

- D-Wave Documentation and Tutorials: https://github.com/dwavesystems/dimod
- Lucas, A. (2014). "Ising formulations of many NP problems." Frontiers in Physics:
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Import all neccesary libraries for mapping the city, formulating and solving BQM, and to assist in aggregating results.¶

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import osmnx as ox
from dwave.system import DWaveSampler, EmbeddingComposite
import sklearn
```

```
---------------------------------------------------------------------------
---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

In [ ]:

```
import dimod
```

In [ ]:

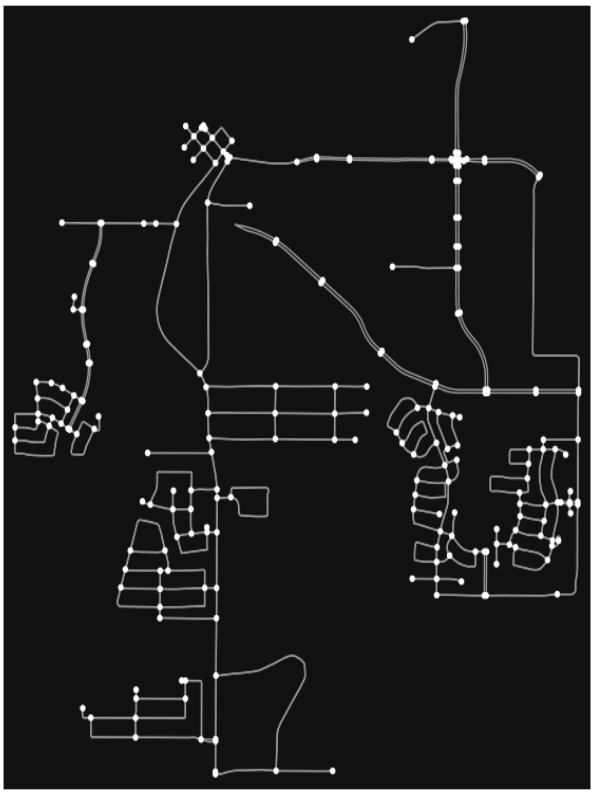
```
import itertools
```

In [ ]:

```
import neal
```

In [ ]:

```
#location for city is Haslet, TX
city_location = "Haslet, Texas, USA"

G = ox.graph_from_place(city_location, network_type="drive")
fig, ax = ox.plot_graph(G)
```



In [ ]:

```
#Observe nodes and edges
gdf_nodes, gdf_edges = ox.graph_to_gdfs(G)
# print stats
print('Node Stats')
print(gdf_nodes.count())
print('Edge S†ats')
print(gdf_edges.count())
```

```
Node Stats
y              230
x              230
street_count    230
highway         25
geometry       230
dtype: int64
Edge S†ats
osmid       564
highway     564
maxspeed     10
name        524
oneway      564
reversed    564
length      564
lanes        18
geometry    564
bridge       12
ref          36
dtype: int64
```

In [ ]:

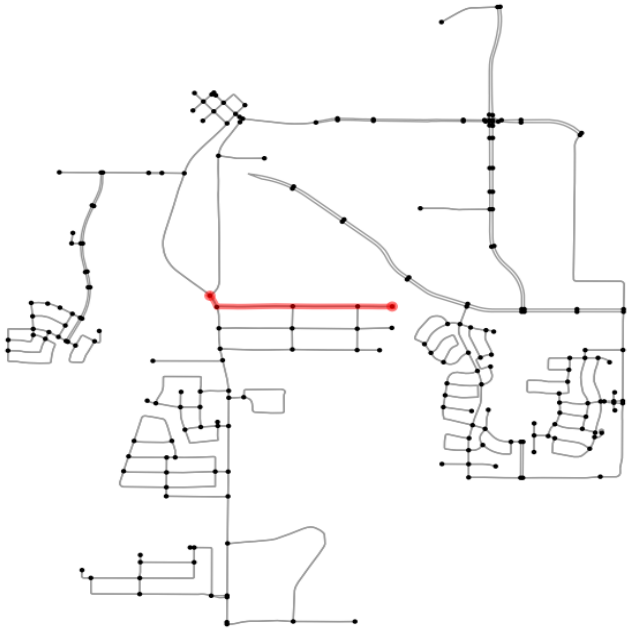
```
# Impute missing edge and travel times for calculating shortest distance later
G = ox.routing.add_edge_speeds(G)
G = ox.routing.add_edge_travel_times(G)
```

Shortest distance between 2 selected nodes¶

In [ ]:

```
# pick any 2 nodes, set them as orig and destination and find shortest distance based on travel time, plot this unconstrained route
orig, dest = 82623141, 82595228

# find shortest distance route
sd_route = ox.shortest_path(G, orig, dest, weight='travel time')
# plot the unconstrained route on the graph
fix, ax = ox.plot_graph_route(G, sd_route, orig_dest_size=50, node_size=10,
bgcolor = 'white', node_color = 'black')
```

```
/Users/abhimanyudeeraj/research/dwave_bqm_exp1/lib/python3.13/site-packages/osmnx/routing.py:335: UserWarning: The attribute 'travel time' is missing or null on some edges.
  _verify_edge_attribute(G, weight)
```

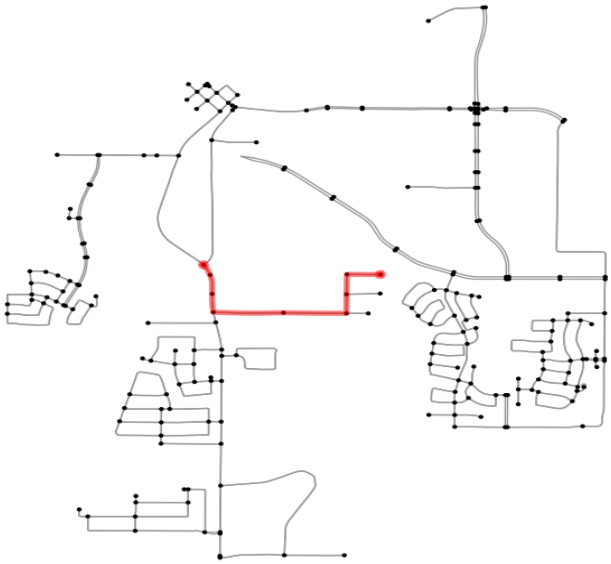


Find an alternate route by removing 2 nodes from the shortest path for validation¶

In [ ]:

```
# clone the graph and remove three nodes in the shortest route to simuate constraints
# validation graph
VG = G.copy()
nodes_to_remove = [82453018, 82453019]

# remove the nodes from the graph
VG.remove_nodes_from(nodes_to_remove)
# Calculate the shortest path again with the nodes removed
# First impute missing edge speeds and travel times
VG = ox.routing.add_edge_speeds(VG)
VG = ox.routing.add_edge_travel_times(VG)
```

```
# rerecalculate the shodtest path between the original origin and destination with the new constraints - nodes removed
# find shortest distance route
new_sd_route = ox.shortest_path(VG, orig, dest, weight='travel_time')
# plot the constrained route on the graph
fix, ax = ox.plot_graph_route(VG, new_sd_route, orig_dest_size=50,
node_size=10, bgcolor = 'white', node_color = 'black')
```



Use D-Wave BQM to identify a refined shortest path given a series of constraints¶

For the purpose of this illustration, we simulate traffic incidents by apply the constraints to two nodes from the original shortest path. We select the same three nodes from the validation step above.¶

We apply constraints as a BQM and solve using D-Wave Solver.¶

The goal is to ensure nodes along the best solution from the BQM will closely follow the alternate constrained shortest path from the validation step above.¶

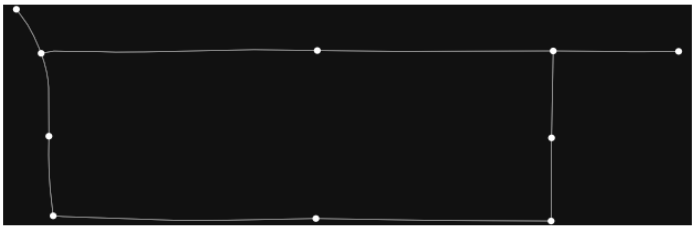
In [ ]:

```
# Create a sub-graph of just the nodes of interest (from the two alternate routes) to improve the efficiency of the D-Wave BQM

dwave_nodes = list(set(sd_route + new_sd_route))
print (dwave_nodes)
```

[82623137, 82623141, 82688780, 82484748, 82684495, 82484752, 82623129, 82453018, 82595228, 82453020]

In [ ]:

```
# create subgraph with the selected nodes
SG = G.subgraph(dwave_nodes)
# display subgraph
fig, ax = ox.plot_graph(SG, node_size=20, edge_linewidth=0.5)
```



In [ ]:

```
## create a network graph corresponding to the subgraph with travel time for edges as the weights

# convert graph to node and edge GeoPandas GeoDataFrames
sgdf_nodes, sgdf_edges = ox.graph_to_gdfs(SG)

# Create two new graphs from the nodes GeoDataFrame - G1 to hold the routes and G2 to store the distance
G1 = nx.Graph() # initialize a new graph to store route
G2 = nx.Graph() # initialize a new graph to store distance

for i, row in sgdf_nodes.iterrows():
    G1.add_node(i, pos=(row.geometry.x, row.geometry.y))
    G2.add_node(i, pos=(row.geometry.x, row.geometry.y))


#Iterate through all nodes and find shortest paths between them minimizing travel time
for i, row1 in sgdf_nodes.iterrows():
    for j, row2 in sgdf_nodes.iterrows():
        if i != j:
            # temp_route = ox.shortest_path(SG, orig=j, dest=i, weight="travel_time") # shortest route between i and j
            temp_route = ox.shortest_path(SG, orig=j, dest=i, weight="length") # shortest route between i and j
            if temp_route is not None:
                temp_distance = round(sum(ox.routing.route_to_gdf(SG, temp_route)["length"]))
            G1.add_edge(i, j, weight=temp_route)
            G2.add_edge(i, j, weight=temp_distance)
```

In [ ]:

```
# create a dataframe of edge distances in the format: origin, destination, dist_in_meters
# where
# origin = start node
# desitination = end node
# dist_in_meters = distance in meters from start node to end node - for edge weights
edge_routes = nx.get_edge_attributes(G1, 'weight')
edge_distances = nx.get_edge_attributes(G2, 'weight')

distance_data = []

for key, value in edge_distances.items():
    if G.has_edge(key[0], key[1]) or G.has_edge(key[1], key[0]):
        distance_data.append(list(key) + [value])

distance_df = pd.DataFrame(distance_data)
distance_df.columns = ['origin', 'destination', 'dist_in_meters']
print(len(distance_df))
print(distance_df)
```

```
10
     origin  destination  dist_in_meters
0  82623137     82623141             257
1  82623137     82484752             184
2  82623137     82453018             484
```

3	82484748	82684495	169
4	82484748	82623129	177
5	82688780	82484752	176
6	82688780	82453020	483
7	82684495	82453020	539
8	82623129	82453018	567
9	82623129	82595228	107

Set some of the global variables as well as complete some preprocessing of edge weight assignment¶

In [ ]:

```
# Some varialbes are global, as they will be used throughout multiple functions and are often repeated throghout the algorithm

#Initialize the BQM
bqm = dimod.BinaryQuadraticModel("BINARY")

 # Identify nodes with traffic incidents based on validation criteria
incident_nodes = nodes_to_remove

# Extract nodes from the distance_df dataframe and add as variables to the BQM
nodes = pd.concat([distance_df['origin'], distance_df['destination']]).unique()
```

In [ ]:

```
#Initialize penalty coefficients
#From top to bottom of below: penalty coeffcients for constraints that ensure:
#aggregation of different runs of the BQM
#each node has a degree of 2 (one incoming and outgoing edge)
#start node had one outgoing edge, end node has one incoming edge
#penalizes inclusion of incident nodes
#connectivity reward (emphasis on connection between nodes to ensure there are no fragmented paths in the solution graph)

aggregation_penalty = 0
lambda_penalty = 0
lambda_2_penalty = 0
connectivity_penalty = 0

#separate penalty to ensure selection of start and end nodes and omission of incident nodes (both stay pretty much constant)
startNode_endNodePenalty = 999999
incident_penalty = 999999

# initilize start and end nodes based on our orignal criteria
start_node, end_node = orig, dest
```

Initially create the BQM from the dataset above.¶

In [ ]:

```
def create_BQM(G, config, start_node, end_node):
    lambda_penalty = config["lambda_penalty"]
    lambda_2_penalty = config["lambda_2_penalty"]
    connectivity_penalty = config["connectivity_penalty"]

    # for each node, add variables for each internal node in the graph. o.e, nodes that are not the start, end, or incident nodes
    for node in nodes:
        if node not in [start_node, end_node] + incident_nodes:
            # add variable corresponding to this node in the BQM
            bqm.add_variable(node, -1)  # Degree variable for node
            # find neighbouring node
            neighboring_nodes_df = distance_df[(distance_df['origin'] == node) | (distance_df['destination'] == node)]
```

```
        # intially, regardless of distance, considers nodes in orign and destination as neighbors
        # takes each row in neighbors, if it is not orign (start of an edge), assign this node as the origin, if it is origin, assign it as the destination (end of edge).
        # iterate through all nodes
        #Add new interactions which is the degree depicting number of hops from the node


        for index, row in neighboring_nodes_df.iterrows():

            origin = row['origin']
            destination = row['destination']
            weight = row['dist_in_meters']

            if origin == node:
                neighbor = destination
            else:
                neighbor = origin

            bqm.add_variable(f"y_{node}_{neighbor}", weight)

            bqm.add_interaction(f"x_{node}", f"y_{node}_{neighbor}", connectivity_penalty) # maps origin and destination nodes to an edge.
            bqm.add_interaction(f"x_{neighbor}", f"y_{node}_{neighbor}", connectivity_penalty)


            bqm.add_interaction(f"y_{node}_{neighbor}", f"y_{node}",-2* connectivity_penalty) # lowers energy for solution where the edge AND both nodes are selected
            bqm.add_interaction(f"y_{node}_{neighbor}", f"y_{neighbor}",-2* connectivity_penalty)

            ##add comments
            outgoing_edges = [f"y_{node}_{neighbor}" for neighbor in nodes if G.has_edge(node, neighbor)]
            incoming_edges = [f"y_{neighbor}_{node}" for neighbor in nodes if G.has_edge(neighbor, node)]

            for edge in outgoing_edges:
                if node == origin:
                    bqm.add_interaction(f"x{origin}", edge, -lambda_penalty)
                elif node == destination:
                    bqm.add_interaction(f"x{destination}", edge, -lambda_penalty)
                ##else:
                    ##bqm.add_interaction(f"x_{node}", edge, -lambda_penalty)

            for edge in incoming_edges:
                if node == origin:
                    bqm.add_interaction(f"x{origin}", edge, -lambda_penalty)
                elif node == destination:
                    bqm.add_interaction(f"x{destination}", edge, -lambda_penalty)
                ##else:
                    ##bqm.add_interaction(f"x_{node}", edge, -lambda_penalty)

        bqm.add_linear(f"x_{node}", lambda_penalty)
    else:

        for node in [start_node, end_node]:
            if node == start_node:
                bqm.add_interaction(f"y {start_node}_{destination}", f"x{start_node}", -lambda_2_penalty)
            elif node == end_node:
                bqm.add_interaction(f"y {origin}_{end_node}", f"x{end_node}", -lambda_2_penalty)

        for node in incident_nodes:
            bqm.add_linear(f"deg_{node}", incident_penalty)
            incident_outgoing_edges = [f"y_{node}_{neighbor}" for neighbor in nodes if G.has_edge(node, neighbor)]
            incident_incoming_edges = [f"y_{neighbor}_{node}" for neighbor in nodes if G.has_edge(neighbor, node)]
```

```
            for edge in incident_outgoing_edges:
                if node == origin:
                    bqm.add_interaction(f"x{origin}", edge, incident_penalty)
                elif node == destination:
                    bqm.add_interaction(f"x{destination}", edge, incident_penalty)
                ##else:
                    ##bqm.add_interaction(f"x_{node}", edge, incident_penalty)

            for edge in incident_incoming_edges:
                if node == origin:
                    bqm.add_interaction(f"x{origin}", edge, incident_penalty)
                elif node == destination:
                    bqm.add_interaction(f"x{destination}", edge, incident_penalty)
                ##else:
                    ##bqm.add_interaction(f"x_{node}", edge, incident_penalty)

            # 4. Degree constraint: Sum of all incident edges must be 0
            bqm.add_linear_equality_constraint(
                [(edge, 1) for edge in incident_incoming_edges+incident_outgoing_edges],
                lagrange_multiplier=incident_penalty,
                constant=0)


    return bqm
```

Function to run multiple BQMs¶

In [ ]:

```
def multiple_BQM_solve (G, penalty_combinations, sampler):
    bqm_solutions = []
    for config in penalty_combinations:
        clone_bqm = create_BQM(G, config, start_node, end_node)
        bqm_solution = sampler.sample(clone_bqm, num_reads=100).first.sample

        # Extract the selected nodes and edges from each run
        all_selected_nodes = [node for node in nodes if bqm_solution.get(f"x_{node}", 0) == 1]
        # [print(node) for node in nodes if bqm_solution.get(f"x_{node}", 0) == 1]
        all_selected_edges = [(origin, destination) for (origin, destination) in zip (distance_df["origin"], distance_df["destination"])
if bqm_solution.get(f"y_{origin}_{destination}", 0) == 1]
        # [print(str(origin) + '_' + str(destination)) for (origin, destination) in zip (distance_df["origin"], distance_df["destination"])
if bqm_solution.get(f"y_{origin}_{destination}", 0) == 1]
        all_nodes_from_edges = set()
        for selected_edge in all_selected_edges:
            all_nodes_from_edges.add(selected_edge[0])
            all_nodes_from_edges.add(selected_edge[1])

        ##print (all_nodes_from_edges)
        # add a constraint to ensure start and end nodes always selected
        bqm.add_linear(f"x_{start_node}", -startNode_endNodePenalty)
        bqm.add_linear(f"x_{end_node}", -startNode_endNodePenalty)


        bqm_solutions.append((all_selected_nodes, all_selected_edges))
        # print(bqm_solutions)
    return bqm_solutions
```

Classical Node Aggregation¶

In [ ]:

```
def node_aggregation (bqm_solutions):

    node_counter = {}

    for solution in bqm_solutions:
        solution_edges = set(solution[1])# to store edges from each solution
        solution_nodes = set()# to store nodes from each solution
        for edge in solution_edges:
            solution_nodes.add(edge[0])
            solution_nodes.add(edge[1])

        for node in solution_nodes:
            node_counter[node] = node_counter.get(node, 0)+1

    #print(node_counter)

    aggregated_nodes = [key for key, value in node_counter.items() if value/len(bqm_solutions) > 0.9] ## consider the node if it is
in 90% of the solutions
    print("Aggregated nodes:")
    print(aggregated_nodes)
    return aggregated_nodes
```

Main program to solve BQM using an iterative process and node aggregation for the best constrained path from source to destination.¶

In [ ]:

```
#intialize a range of values for each penalty.
#aggregation_penalty_list = [1]
lambda_penalty_list = [100, 500, 1000, 1500]
lambda_2_penalty_list = [100, 500, 1000, 1500]
connectivity_penalty_list = [1000, 1500, 2000, 2500]

# Generate all possible combinations of all penalties
penalty_combinations = [
    {"lambda_penalty": lp, "lambda_2_penalty": l2p, "connectivity_penalty": cp}
    for lp, l2p, cp in itertools.product(lambda_penalty_list, lambda_2_penalty_list, connectivity_penalty_list)
]

print (penalty_combinations)

# sampler = EmbeddingComposite(DWaveSampler()) ### do not use till you get accepted to dwave launchpad program
## instead use the simulated annealer below
sampler = neal.SimulatedAnnealingSampler()
total_solutions = multiple_BQM_solve(G, penalty_combinations, sampler)

#for solution in total_solutions:
    #print (solution)

final_nodes = node_aggregation(total_solutions)
# Aggregate results and visualize the resultant graph
#final_graph_visualization(node_aggregation(total_solutions))
##print(final_nodes)
```

```
[{'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'lambda_penalty': 100, 'lambda_2_penalty':
100, 'connectivity_penalty': 1500}, {'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 2000},
{'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 2500}, {'lambda_penalty': 100, 'lambda_2_penalty':
500, 'connectivity_penalty': 1000}, {'lambda_penalty': 100, 'lambda_2_penalty': 500, 'connectivity_penalty': 1500},
{'lambda_penalty': 100, 'lambda_2_penalty': 500, 'connectivity_penalty': 2000}, {'lambda_penalty': 100, 'lambda_2_penalty':
500, 'connectivity_penalty': 2500}, {'lambda_penalty': 100, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1000},
{'lambda_penalty': 100, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1500}, {'lambda_penalty': 100, 'lambda_2_penalty':
1000, 'connectivity_penalty': 2000}, {'lambda_penalty': 100, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2500},
```

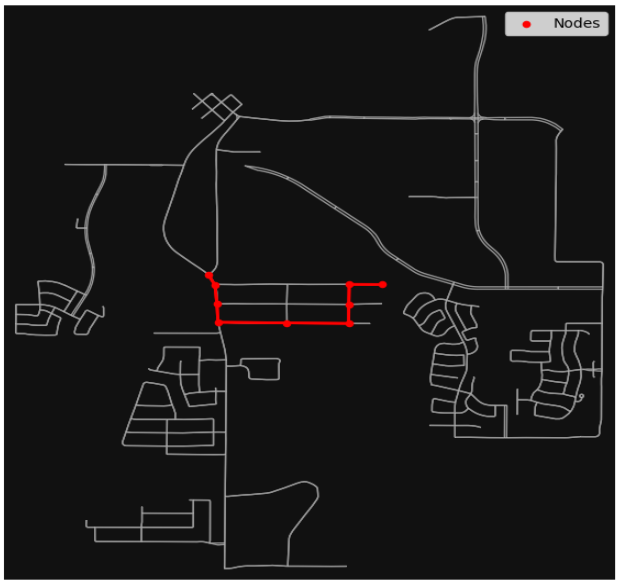
{'lambda_penalty': 100, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1000}, {'lambda_penalty': 100, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1500}, {'lambda_penalty': 100, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2000}, {'lambda_penalty': 100, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2500}, {'lambda_penalty': 500, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'lambda_penalty': 500, 'lambda_2_penalty': 100, 'connectivity_penalty': 1500}, {'lambda_penalty': 500, 'lambda_2_penalty': 100, 'connectivity_penalty': 2000}, {'lambda_penalty': 500, 'lambda_2_penalty': 100, 'connectivity_penalty': 2500}, {'lambda_penalty': 500, 'lambda_2_penalty': 500, 'connectivity_penalty': 1000}, {'lambda_penalty': 500, 'lambda_2_penalty': 500, 'connectivity_penalty': 1500}, {'lambda_penalty': 500, 'lambda_2_penalty': 500, 'connectivity_penalty': 2000}, {'lambda_penalty': 500, 'lambda_2_penalty': 500, 'connectivity_penalty': 2500}, {'lambda_penalty': 500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1000}, {'lambda_penalty': 500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1500}, {'lambda_penalty': 500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2000}, {'lambda_penalty': 500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2500}, {'lambda_penalty': 500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1000}, {'lambda_penalty': 500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1500}, {'lambda_penalty': 500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2000}, {'lambda_penalty': 500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 100, 'connectivity_penalty': 1500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 100, 'connectivity_penalty': 2000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 100, 'connectivity_penalty': 2500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 500, 'connectivity_penalty': 1000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 500, 'connectivity_penalty': 1500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 500, 'connectivity_penalty': 2000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 500, 'connectivity_penalty': 2500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1500}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2000}, {'lambda_penalty': 1000, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 100, 'connectivity_penalty': 1500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 100, 'connectivity_penalty': 2000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 100, 'connectivity_penalty': 2500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 500, 'connectivity_penalty': 1000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 500, 'connectivity_penalty': 1500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 500, 'connectivity_penalty': 2000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 500, 'connectivity_penalty': 2500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 1500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1000, 'connectivity_penalty': 2500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 1500}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2000}, {'lambda_penalty': 1500, 'lambda_2_penalty': 1500, 'connectivity_penalty': 2500}]
Aggregated nodes:
[82623137, 82623141, 82484748, 82688780, 82684495, 82484752, 82595228, 82623129, 82453020]

In [ ]:

```
# Define place to retrieve OSM data
place_name = "Haslet, Texas, USA"  # Change based on location

# Download graph from OSM
G = ox.graph_from_place(place_name, network_type="drive")
print(final_nodes)


node_x_list = [G.nodes[n]['x'] for n in final_nodes]
node_y_list = [G.nodes[n]['y'] for n in final_nodes]

fig, ax = ox.plot_graph(G, show=False, close=False, node_size=0.2)

ax.scatter(node_x_list, node_y_list, c='red', s=20, zorder=3, label="Nodes")

final_edges = set()

for i in final_nodes:
    for j in final_nodes:
        if i != j:
            if G.has_edge(i, j) or G.has_edge(j, i):
                final_edges.add((i,j))
```

```
print(final_edges)

for edge in final_edges:
    node1, node2 = edge
    x_coords = [G.nodes[node1]['x'], G.nodes[node2]['x']]
    y_coords = [G.nodes[node1]['y'], G.nodes[node2]['y']]
    ax.plot(x_coords, y_coords, color='red', linewidth=2, zorder=2)

plt.legend()
plt.show()
```

```
[82623137, 82623141, 82484748, 82688780, 82684495, 82484752, 82595228, 82623129, 82453020]
{(82484752, 82623137), (82688780, 82484752), (82484752, 82688780), (82623137, 82623141), (82623129, 82484748),
(82453020, 82688780), (82484748, 82684495), (82684495, 82453020), (82453020, 82684495), (82684495, 82484748),
(82623141, 82623137), (82623129, 82595228), (82688780, 82453020), (82484748, 82623129), (82595228, 82623129),
(82623137, 82484752)}
```



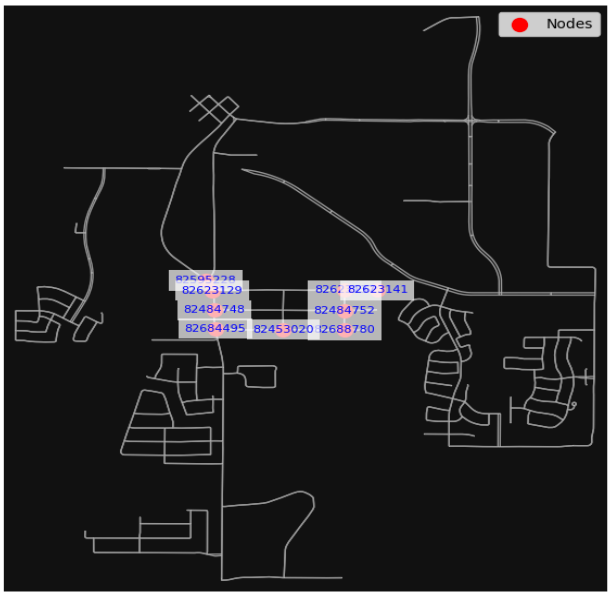
Old graphing code.¶

In [ ]:

```
# Define place to retrieve OSM data
place_name = "Haslet, Texas, USA"  # Change based on location

# Download graph from OSM
G = ox.graph_from_place(place_name, network_type="drive")

print(set(total_solutions[0][1]))

solution_edges = set(total_solutions[0][1])

solution_nodes = set()

for edge in solution_edges:
    solution_nodes.add(edge[0])
    solution_nodes.add(edge[1])
print(solution_nodes)
```

```
node_x_list = [G.nodes[n]['x'] for n in solution_nodes]
node_y_list = [G.nodes[n]['y'] for n in solution_nodes]

fig, ax = ox.plot_graph(G, show=False, close=False, node_size=0.2)

ax.scatter(node_x_list, node_y_list, c='red', s=100, zorder=3, label="Nodes")

for node, x, y in zip(solution_nodes, node_x_list, node_y_list):
    ax.text(x,y,str(node), fontsize = 8, color ='blue', ha = 'center', va = 'center', bbox=dict(facecolor='white', alpha=0.7, edgecolor =
'none'))

plt.legend()
plt.show()
```

```
{(82688780, 82484752), (82623137, 82623141), (82484748, 82684495), (82684495, 82453020), (82623129, 82595228),
(82688780, 82453020), (82484748, 82623129), (82623137, 82484752)}
{82623137, 82623141, 82484748, 82688780, 82684495, 82484752, 82595228, 82623129, 82453020}
```

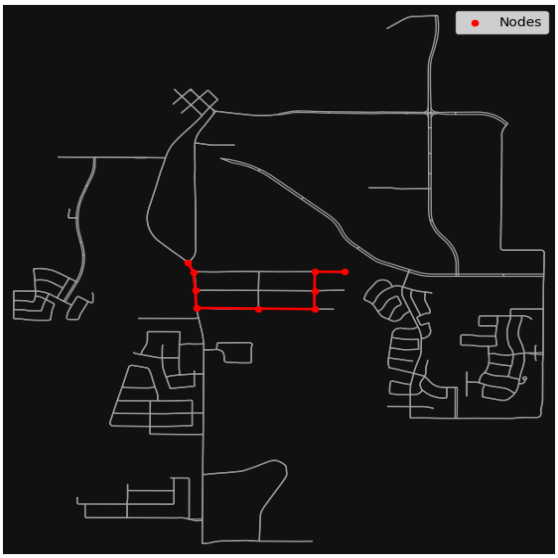


In [ ]:

```
# Define place to retrieve OSM data
place_name = "Haslet, Texas, USA"  # Change based on location

# Download graph from OSM
G = ox.graph_from_place(place_name, network_type="drive")

print(set(total_solutions[0][1]))

solution_edges = set(total_solutions[0][1])

solution_nodes = set()

for edge in solution_edges:
    solution_nodes.add(edge[0])
    solution_nodes.add(edge[1])
print(solution_nodes)

node_x_list = [G.nodes[n]['x'] for n in solution_nodes]
node_y_list = [G.nodes[n]['y'] for n in solution_nodes]
```

```
fig, ax = ox.plot_graph(G, show=False, close=False, node_size=0.2)

ax.scatter(node_x_list, node_y_list, c='red', s=20, zorder=3, label="Nodes")


for edge in solution_edges:
    node1, node2 = edge
    x_coords = [G.nodes[node1]['x'], G.nodes[node2]['x']]
    y_coords = [G.nodes[node1]['y'], G.nodes[node2]['y']]
    ax.plot(x_coords, y_coords, color='red', linewidth=2, zorder=2)

plt.legend()
plt.show()
```

```
{(82688780, 82484752), (82623137, 82623141), (82484748, 82684495), (82684495, 82453020), (82623129, 82595228),
(82688780, 82453020), (82484748, 82623129), (82623137, 82484752)}
{82623137, 82623141, 82484748, 82688780, 82684495, 82484752, 82595228, 82623129, 82453020}
```



Full combination of penalties -- consider later¶

In [ ]:

```
#intialize a range of values for each penalty.
aggregation_penalty_list = [1, 10, 50, 100]
lambda_penalty_list = [100, 200, 300, 400]
lambda_2_penalty_list = [100, 200, 300, 400]
connectivity_penalty_list = [200, 500, 1000, 10000]

# Generate all possible combinations of all penalties
penalty_combinations = [
    {"aggregation_penalty": ap, "lambda_penalty": lp, "lambda_2_penalty": l2p, "connectivity_penalty": cp}
    for ap, lp, l2p, cp in itertools.product(aggregation_penalty_list, lambda_penalty_list, lambda_2_penalty_list,
connectivity_penalty_list)
]

print (penalty_combinations)

sampler = EmbeddingComposite(DWaveSampler())
total_solutions = multiple_BQM_solve(G, penalty_combinations, sampler)
```

```
# Aggregate results and visualize the resultant graph
final_graph_visualization(node_aggregation(total_solutions))
```

[{'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 1, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 100,

'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 10, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty':

100, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000}, {'aggregation_penalty': 50, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 200}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 500}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 200}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 500}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 200}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 500}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 200}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 500}, {'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000},

{'aggregation_penalty': 100, 'lambda_penalty': 100, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 200, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 300, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 100, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 200, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 300, 'connectivity_penalty': 10000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 200},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 500},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 1000},
{'aggregation_penalty': 100, 'lambda_penalty': 400, 'lambda_2_penalty': 400, 'connectivity_penalty': 10000}]

KeyboardInterrupt Traceback (most recent call last)

Cell In[57], line 16

13 print (penalty_combinations)

15 sampler = EmbeddingComposite(DWaveSampler())

---> 16 total_solutions = multiple_BQM_solve(G, penalty_combinations, sampler)

```
 18 # Aggregate results and visualize the resultant graph
 19 final_graph_visualization(node_aggregation(total_solutions))


Cell In[56], line 5, in multiple_BQM_solve(G, penalty_combinations, sampler)
  3 for config in penalty_combinations:
  4     clone_bqm = create_BQM(G, config, start_node, end_node)
----> 5     bqm_solution = sampler.sample(clone_bqm, num_reads=100).first.sample
  7     # Extract the selected nodes and edges from each run
  8     all_selected_nodes = [node for node in nodes if bqm_solution.get(f"x_{node}", 0) == 1]


File ~/research/dwave_bqm_exp1/lib/python3.13/site-packages/dwave/system/composites/embedding.py:240, in EmbeddingComposite.sample(self, bqm, chain_strength, chain_break_method, chain_break_fraction, embedding_parameters, return_embedding, warnings, **parameters)
 231 else:
 232     # we want the parameters provided to the constructor, updated with
 233     # the ones provided to the sample method. To avoid the extra copy
 234     # we do an update, avoiding the keys that would overwrite the
 235     # sample-level embedding parameters
 236     embedding_parameters.update((key, val)
 237                                 for key, val in self.embedding_parameters
 238                                 if key not in embedding_parameters)
--> 240 embedding = self.find_embedding(source_edgelist, target_edgelist,
 241                                 **embedding_parameters)
 243 if bqm and not embedding:
 244     raise ValueError("no embedding found")


File ~/research/dwave_bqm_exp1/lib/python3.13/site-packages/minorminer/minorminer.py:41, in find_embedding(S, T, max_no_improvement, random_seed, timeout, max_beta, tries, inner_rounds, chainlength_patience, max_fill, threads, return_overlap, skip_initialization, verbose, interactive, initial_chains, fixed_chains, restrict_chains, suspend_chains)
 21 @__wraps(__find_embedding)
 22 def find_embedding(S, T,
 23                    max_no_improvement=10,
(...)
```

```
    39                 suspend_chains=(),
    40                 ):
---> 41     return __find_embedding(S, T,
    42                         max_no_improvement=max_no_improvement,
    43                         random_seed=random_seed,
    44                         timeout=timeout,
    45                         max_beta=max_beta,
    46                         tries=tries,
    47                         inner_rounds=inner_rounds,
    48                         chainlength_patience=chainlength_patience,
    49                         max_fill=max_fill,
    50                         threads=threads,
    51                         return_overlap=return_overlap,
    52                         skip_initialization=skip_initialization,
    53                         verbose=verbose,
    54                         interactive=interactive,
    55                         initial_chains=initial_chains,
    56                         fixed_chains=fixed_chains,
    57                         restrict_chains=restrict_chains,
    58                         suspend_chains=suspend_chains,
    59                         )

File ~/research/dwave_bqm_exp1/lib/python3.13/site-packages/minorminer/_minorminer.pyx:293, in
minorminer._minorminer.find_embedding()


KeyboardInterrupt: embedding cancelled by keyboard interrupt
```

In [ ]:

BQM-Driven Node aggregation process (Will consider this for future enhancements)¶

In [ ]:

```
def node_aggregation (bqm_solutions):
    second_bqm = dimod.BinaryQuadraticModel("BINARY")
    # to store final nodes
```

```
    for solution in bqm_solutions:
        solution_edges = set(solution)# to store edges from each solution
        solution_nodes = set()# to store nodes from each solution
        for edge in solution:
            solution_nodes.add(edge[0])
            solution_nodes.add(edge[1])

        # again, a constraint is added to always include start and end nodes.
        second_bqm.add_linear(f"x_{start_node}", -startNode_endNodePenalty)
        second_bqm.add_linear(f"x_{end_node}", -startNode_endNodePenalty)


        # loop through all origin and destination nodes.
        for origin in solution_nodes:
            for destination in solution_nodes:
                if origin != destination:
                    x_origin = f"x_[origin]_{item}" #set variable for origin node in this list
                    x_destination = f"x_[destination]_{item}"#set variable for destination node in this list
                    second_bqm.add_variable(x_origin, aggregation_penalty) # add a variable with an enforced penalty for origin node to be represented in the node aggregation constriant of BQM
                    second_bqm.add_variable(x_destination, aggregation_penalty)# add a variable with an enforced penalty for destination node to be represented in the node aggregation constriant of BQM
                    # quadratic constraint: takes the difference between origin and destination nodes between consecutive runs to guage disagreement between all solutions
                    #aggregation penalty is there to minimize disagreement of origin and destination nodes between runs, therefore nodes with minimal disagreement are chosen.
                    ##bqm.add_interaction(f"x_{origin}" - f"x_{destination}"*aggregation_penalty)
                    second_bqm.add_interaction

                        #add valid nodes (origin and destination nodes) that have minimal disagreement between each run of the BQM
                        if item[f"x_{origin}"] == 1:
                            final_nodes.add(origin)
                        if item[f"x_{destination}"] == 1:
                            final_nodes.add(destination)

    for origin, destination in G.edges: # connectivity constraint that prevents fragmented paths between origin and destination nodes.
        second_bqm.add_interaction(f"y_{origin}_{destination}", f"x{origin}" -2*connectivity_penalty)
        second_bqm.add_interaction(f"y_{origin}_{destination}", f"x{destination}" -2*connectivity_penalty)

        for item in bqm_solutions: # if origin and destination are in the
            if item[f"y_{origin}_{destination}"] == 1:
                final_edges.add((origin, destination))


    final_node_list = list(final_nodes)
    final_edge_list = list(final_edges)

    return final_node_list, final_edge_list
```