# THE WAREHOUSE LOCATION PROBLEM

Given $n$ cities with specified distances, one wants to build $k$ warehouses in different cities and $minimize$ the maximum distance of any city to a warehouse. This presentation seeks to solve the metric facility location problem by designing, testing and analyzing results from three (3) different algorithms based on an exhaustive search, a greedy heuristic, and a randomized implementation.

In graph theory this problem is referred to as $k-center$ problem which is NP-Hard, formulated as; given a weighted complete undirected graph $\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{W})$, with $\mathcal{V}$ the set of vertices, $\mathcal{E}$ the set of edges and $\mathcal{W}$ is the set of weights/distances (euclidean) between any two vertices of the graph, we aim to find a set of $k \leq m$ vertices for which the largest distance of any other vertex to its closest vertex in the k-set is minimum. More formally, we seek to find a set $\mathcal{V}^* \subseteq \mathcal{V} \ni \sup_{v\in\mathcal{V}} w_{\mathcal{G}(v,\mathcal{V}^*)}$ is minimized [1].

## Libraries

> time:
>
> - timing of function executions
>
> os:
>
> - interact with OS
> - file and directory path manipulations
>
> shutil.rmtree:
>
> - remove non empty and directories which is unavailable in os package
>
> numpy:
>
> - my student number was used as seed during initialization
> - array and matrix operations
> - random number generation
> - random selection of elements in a set
> - other mathematical functions
>
> itertools.combination:
>
> - generate combinations of elements in a container
>
> mathplotlib.pyplot:
>
> - graphing and plotting operations.
>
> networkx:
>
> - graph generation and manipulations
> - the erdos_renyi_graph graphs [2] were generated given the number of nodes to consider and probability of including an edge. Since we aim to generate complete graphs, this means every edge must be included (thus $p = 1$). The total number of edges for a comsplete graph is given as $\frac{n(n-1)}{2}$.

In [1]:

```
import os, time
from shutil import rmtree
import numpy as np
import networkx as nx
from itertools import combinations
from matplotlib import pyplot as plt
```

## BASE / AUXILLARY CLASS

> CreateDirectory
>
> - creates requested directory of the given directory name (dir_name)
> - returns the path of created directory
>
> GenerateNodeCoordinates
>
> - given graph nodes, generates random positions for each node
> - pairs node with corresponding position in space
>
> ComputeEdgeDistances
>
> - uses positions of node pairs in space connected by an edge to compute the euclidean distance
> - the euclidean distance is paired with corresponding edge connecting the nodes.
>
> GenerateGraphs
>
> - generates random complete, weighted and undirected erdos_renyi_graph for all edges (weights as euclidean distance between nodes).
> - specified number of vertices (nodes), and probability of edge inclusion.
> - save graph files in pickle into **./graphs/** directory. save as *.gpickle* file
> - plot graph if requested.
>
> LoadGraph
>
> - load graph of given node from the specified directory present in **./graphs/**. Example *./graphs/pickles/*
> - sets the loaded graph into context.
> - plot graph if graph file exists.
>
> PlotGraph
>
> - plots graph given various parameters.
> - generates 2 groups of colours for nodes if specified.
> - save graph to specified directory in **./graphs/**.
> - draw graph with edge weights or not if specified.
> - file_name to save graph if specified.

In [2]:

```
class IOBASE:
    """
    For input/output operations
    """

    graph = None
    adjacency_matrix = None
    graph_directory = os.path.join(os.getcwd(),'graphs')

    student_number = 106382
    np.random.seed(student_number)  # random number initializer

    def CreateDirectory(self, dir_name=None, remove_existing=False):
        """
        Args:
            dir_name [str]  - the directory to save graphs in. creates a directory name in 
self.graph_directory
            remove_existing [bool] - determines whether or not to remove existing folder.
        Returns:
            dir_path [str] - path to the directory just created.
        """
        dir_path = self.graph_directory
        if dir_name is not None:
            dir_path = os.path.join(dir_path, dir_name)

        # removes the directory if True. Does not, otherwise
        if remove_existing is True:
            if os.path.exists(dir_path):
                rmtree(dir_path)

        # creates directory if does not exist already
        if not os.path.exists(dir_path):
            os.makedirs(dir_path)

        return dir_path


    def GenerateNodeCoordinates(self, nodes):
        """
        As required, we ensure graph vertices are 2D points on the XOY plane, with integer 
valued coordinates between 1 and 1000.

        Args:
            nodes (np.ndarray): nodes of graph G

        Returns:
            dict: graph nodes along with coordinate/position in space.
        """
        nodes_with_pos = dict()  # initialize container
        for node in nodes:
            node_pos = np.random.randint(low=1, high=1000, size=(2))  # generate required 
position of node in space
            nodes_with_pos.update({node:node_pos})  # update container
        return nodes_with_pos


    def ComputeEdgeDistances(self, edges, node_coordinate):
        """
        Computes the euclidean distance for all edges

        Args:
            edges (np.ndarray): edges set of graph
            node_coordinate (dict): nodes with their associated position in space.

        Returns:
            dict: edge with associated euclidean distance
        """
        edge_distances = dict()  # initialize edge_distance container
        for edge in edges:
            source_pos = np.array(node_coordinate[edge[0]])  # starting node position
            sink_pos = np.array(node_coordinate[edge[1]])  # terminal node position
            euclidean_distance = np.sqrt(np.sum((source_pos - sink_pos) ** 2))  # compute 
euclidean distance between nodes of edge
            edge_distances.update({edge: round(euclidean_distance)})
        return edge_distances


    def GenerateGraphs(self, vertices_range=(2,5), prob_edge_inclusion=1, dir_name=None, 
plot_graph=False, remove_existing=False):
        """
        plots the generated graphs with their node and edge counts.
```

```
        saves generated graphs along with their plots in the graphs directory

        Args:
            vertices_range [tuple] - range of graphs to generate with number of vertices in
range. default 2-5
            prob_edge_inclusion [float] - for complete graphs, all edges have included.
            dir_name [str] - name of the folder to save graphs in.
            plot_graph [bool] - plots the graph if True.
            remove_existing [bool] - deletes existing directory.
        """
        save_path = self.graph_directory
        if dir_name is not None:
            save_path = self.CreateDirectory(dir_name=dir_name, remove_existing=remove_existing)

        low, upper = vertices_range  # minimum and maximum number of vertices of graphs to
generate.
        # assert number_of_graphs > 2, 'Number of Graphs to generate must be atleast 2'

        for node in range(low, upper+1):
            graph = nx.generators.random_graphs.erdos_renyi_graph(n=node, p=prob_edge_inclusion,
seed=self.student_number, directed=False)

            # generates coordinates in specified
            nodes_coordinates = self.GenerateNodeCoordinates(graph.nodes)
            nx.set_node_attributes(graph, values=nodes_coordinates, name='pos')  # attach
generated vertices position to graph
            edges_distances = self.ComputeEdgeDistances(graph.edges, nodes_coordinates)
            nx.set_edge_attributes(graph, values=edges_distances, name='w')  # attach computed
euclidean distance of edges to graph

            nx.write_gpickle(graph, os.path.join(save_path, f'{node}_nodes.gpickle'))

            if plot_graph is True:
                self.PlotGraph(graph=graph, save_plot=True)


    def LoadGraph(self, nodes, dir_name=None):
        """
        loads saved graph of given node.
        plots the graph with the specified node if exists.

        Args:
            nodes [int, str] - graph file containing specified number of nodes.
            graph_path [str] - path to graph file directory to load from. defaults to
graphs_directory.
        Sets:
            self.graph [nx.Graph] - the graph with corresponding number of nodes fetched
        """
        try:
            dir_path = self.graph_directory
            if dir_name is not None:
                dir_path = os.path.join(dir_path, dir_name)

            graph_files = [filename for filename in os.listdir(dir_path) if
filename.endswith('.gpickle')]

            # load the last file in the directory if file not found
            if nodes is None:
                nodes = random.choice(graph_files)[-1]

            # check if queried graph exists or not. return error if not
            graph_name = f'{nodes}_nodes.gpickle'
            assert graph_name in graph_files, f'graph file with nodes {nodes} not found'

            graph = nx.read_gpickle(os.path.join(dir_path, graph_name))
            self.graph = graph # set graph of class
            self.adjacency_matrix = nx.adjacency_matrix(graph, weight='w').todense()

            print(f'Graph with {nodes} vertices loaded!!')
        except FileNotFoundError:
            return f'No graph file with {nodes} vertices!!'


    def PlotGraph(self, graph=None, save_plot=False, dir_name=None, remove_existing=False,
with_weights=True, color_diff=None, method=None):
        """
        plot the given graph and return 1 on success
        Args:
            graph [nx.Graph] - the graph to plot
            save_graph [bool] - if true, saves the graph. default not to save graph.
            save_path [tuple] - directories to save file in
        """
        if graph is None:
            graph = self.graph

        title = f'Graph with {len(graph.nodes)} nodes and {len(graph.edges)} edges'  # title for
base graphs w/o coloring
        filename = f'n{len(graph.nodes)}.png'  # for non colored

        # Defines color for some selected nodes to be colored differently
        color_map = None
        if color_diff is not None:

            title = f'Graph with {len(graph.nodes)} nodes, {len(graph.edges)} edges and k=
{self.k}'  #for colored graphs
            filename = f'n{len(graph.nodes)}_k{self.k}.png'  # filename for colored graphs

            if not isinstance(color_diff, list):
                color_diff = list(color_diff)
            color_map = []
            for node in graph:
                if node in color_diff:
                    color_map.append('red')
                else:
                    color_map.append('tab:blue')

        # plot graph positions of vertices and edges
        pos=nx.get_node_attributes(graph, 'pos')

        # node coloring if requested.
        fig = plt.figure(figsize=(10,10))  # size of the plot displayed
        ax = fig.add_axes([0.1,0.1,0.75,0.75]) # axis starts at 0.1, 0.1
        if color_diff is not None:
            nx.draw_networkx(graph, pos, node_color=color_map, with_labels=True, node_size=700,
font_size=18, ax=ax)
        else:
            nx.draw_networkx(graph, pos, with_labels=True, node_size=700, font_size=18, ax=ax)

        # plot with weighted edges if requested
        if with_weights is True:
            nx.draw_networkx_edge_labels(graph, pos, ax=ax)

        ax.set_title(title)
        ax.tick_params(left=True, bottom=True, labelleft=True, labelbottom=True)

        # build path and save graph
        if save_plot is True:
            save_path = self.graph_directory
            if dir_name is not None:
                save_path = self.CreateDirectory(dir_name=dir_name,
remove_existing=remove_existing)
            if method is not None:
                filename = method + '_' + filename
            fig.savefig(os.path.join(save_path, filename))
```

## MAIN CLASS

> ExhaustiveSearch
>
> - generate a list of all potential optimal locations sets $p\_sets$. An expression for the number of elements in this list is expressed as
>
> $$|p\_sets| = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$
>
> - initialize largest distances $pv\_max\_d$ for all $p\_sets$ elements to very small values.
> - for each element $p\_set$ of the $p\_sets$ elements (potential optimal locations), find the largest distance of the closest city to the p_set.
> - update the corresponding distances for the specific $p\_set$ in $pv\_max\_d$.
> - choose the set with the smallest distance distance in $pv\_max\_d$ as the optimal locations set $(VStar)$.
>
> GreedyHeuristic [1]
>
> - initialize $\mathcal{V}^*$ to $\emptyset$
> - start by choosing an arbitrary/random vertex for the first warehouse location from $\mathcal{V}$ and add it to $\mathcal{V}^*$.
> - foreach vertex of $\mathcal{V}$ add the vertex to $\mathcal{V}^*$ if its distance is farthest away from already chosen vertices in $\mathcal{V}^*$.
> - repeat the above step until the $k$ locations are realised.
> - report the vertices in $\mathcal{V}^*$ as optimal locations suitable for the warehouse contruction of minimum distance of reach from the cities.
>
> Randomized Approach [3]
>
> - this implements a lass vegas search algorithm
> - we initialize V_star to emptyset.
> - select at random a vertex from the vertices set.
> - perform (iii) repeatedly (each time picking a new element) until the cardinality of V_star is k (stopping criteria).
> - the true optimal location set is an element of the $\binom{n}{k}$ location sets of size $k$.

- the probability of realizing this true optimal location set is therefore given as

$$\mathcal{P} = \frac{1}{\binom{n}{k}}$$

In [3]:

```
class WarehouseLocation(IOBASE):
    """
    - This module implements a Brute-Force, Greedy-Heuristic and Randomized techniques to find a
solution for the warehouse location problem.
    - The Greedy algorithm is an improvement of the naive Randomized technique proposed.
    - The Exhaustive-Search finds the optimal locations where the warehouses should be
constructed.
    """


    def __init__(self):
        """
        initialize parameters.
        """
        self.k = 1

    def GeneratePrimes(self, k):
        """
        To stay inline with the project objective, this method helps to generate
        the required prime numbers for k.

        Returns:
            list: an array of prime numbers
        """
        primes = []
        for n in range(2, k):
            for i in range(2, n):
                if(n%i == 0):  # if number has factors.
                    break
            else:
                primes.append(n)  # update container
        return primes

    def PotentialVStars(self, V=None):
        """
        This function returns all k-set non-empty non-alternating partitions of the given
vertices set.

        Implementation:
            - We generate all partitions of the vertices set V via subsets of V.
            - Filter out the empty set and the subset with all vertices.

        Args:
            V [list] - array of graph vertices.

        Returns:
            partitions [list] - all k-set of valid partitions of the vertices set V.
        """
        if V == None:
            V = list(self.graph.nodes())

        # Ensures input vertices set is list fornatted
        if not isinstance(V, list):
            V = list(V)

        all_partitions = [] # complementary partitions container
        Powerset = [subset for subset_index in range(len(V)+1) for subset in
combinations(V,subset_index)] # len(Powerset) = 2**len(V)
        for subset in Powerset:
            # self.counter += 1 # increase counter
            if (len(subset) == 0) or (len(subset) == len(V)):
                continue
            all_partitions.append(subset)

        # Take first half since second half is simply permutations of first half.
        all_partitions = [part for part in all_partitions if len(part) == self.k]
        return all_partitions


    def ExhaustiveSearch(self, graph=None, k=None, plot_graph=False):
        """
        Implementation:
            - Get all subsets of the vertex set
            - Initialize distance registers for each subset to a very small number (maximum
trick)
            -
        Args:
            k [integer] - cardinality of optimal locations set.
            graph [nx.Graph] - graph to locate optimal warehouse locations.
            plot_graph [bool] - indicates whether to plot the resulting graph

        Returns:
            [set] - optimal solution suitable for warehouse construction.

        """
        if graph is None:
            graph = self.graph

        # when no k is given use default of 1
        if k != None:
            self.k = k

        assert isinstance(graph, nx.classes.graph.Graph), 'specified graph is not valid!'

        self.counter = 0
        pot_vstars = self.PotentialVStars(V=graph.nodes) # list of all k-set possible partitions
of the vertices set

        part_max_dist = {part: -np.infty for part in pot_vstars}  # initialize warehouse:largest
distance of closest city.
        for p_vstar in pot_vstars:

            closest_dists = []  # initialize closest distances from a warehouse. for all
cities/vertices.
            for vertex in graph.nodes:
                dists_from_vstar = [dist for (index, dist) in
enumerate(self.adjacency_matrix[vertex].tolist()[0]) if index in p_vstar]  # distance of vertex
from selected nodes
                min_dist_from_vstar = min(dists_from_vstar)  # distance of city from a closest
warehouse/node in vstar
                closest_dists.append(min_dist_from_vstar)  # update closest distance container

                self.counter += 1

            largest_dist = max(closest_dists)  # fetch maximum distance of cities from closest
warehouse

            # using the maximum trick to record the largest distances for each potential vstar
            if largest_dist > part_max_dist[p_vstar]:
                part_max_dist[p_vstar] = largest_dist

        # select the smallest distanced of largest distanced warehouse from closest city
        self.V_star = min(part_max_dist, key=part_max_dist.get)

        # initial graph saved before merging starts
        if plot_graph is True:
            self.PlotGraph(graph=graph, color_diff=list(self.V_star), save_plot=True,
dir_name='plot_output', method='exhaustive')


    def GreedyHeuristic(self, k=None, graph=None, plot_graph=False):
        """
        Implementation:
            - Initialize V_star
            - Update V_star with k locations/nodes suitable for the warehouse contruction such
that
                - newly found vertices from the vertices set are farthest away from vertices in
V_star
                - Using the maximum trick was used to obtain the maximum distanced vertices.

        Args:
            k [integer] - cardinality of optimal locations set.
            graph [nx.Graph] - graph to locate optimal warehouse locations.
            plot_graph [bool] - indicates whether to plot the resulting graph

        Returns:
            [set] - optimal solution suitable for warehouse construction.

        """
        # when no graph instance is given
        if graph == None:
            graph=self.graph

        # when no k is given use default of 1
        if k != None:
            self.k = k

        self.counter = 1
        self.V_star = set()  # initialize V_star container
```

```
        self.V_star.add(np.random.choice(graph))  # select random node/warehouse location, add to V_star

        # find all k locations farthest away from already selected locations.
        for k in range(self.k-1):  # pick exactly k-1 nodes plus 1 is k

            # we will be applying the maximum trick
            max_distance = - np.infty
            max_distanced_vertex = None

            for node_g in self.graph:  # scan through graph nodes

                # consider only cities not yet considered
                if node_g in self.V_star:
                    continue

                for selected_location in self.V_star:  #  scan V_star
                    dist = self.adjacency_matrix[node_g, selected_location]  # get distance between node and selected location

                    if dist > max_distance:  # using the maximum trick.
                        max_distance = dist
                        max_distanced_vertex = node_g

                    self.counter += 1

            self.V_star.add(max_distanced_vertex)  # update V_star

        # initial graph saved before merging starts
        if plot_graph is True:
            self.PlotGraph(graph=graph, color_diff=list(self.V_star), save_plot=True, dir_name='plot_output', method='greedy')


    def Randomized(self, k=None, graph=None, plot_graph=False):
        """
        Implementation:
            - initialize V_star to an empty set.
            - randomly select a vertex from the vertices set and add it to V_star.
            - repeat (ii) until there are k elements/locations in V_star

        Args:
            k [integer] - cardinality of optimal locations set.
            graph [nx.Graph] - graph to locate optimal warehouse locations.
            plot_graph [bool] - indicates whether to plot the resulting graph

        returns:
            [set] - optimal solution suitable for warehouse construction.
        """
        if graph is None:
            graph = self.graph.copy()

        # when no k is given use default of 1
        if k != None:
            self.k = k

        self.counter = 0

        assert isinstance(graph, nx.classes.graph.Graph), 'specified graph is not valid!'

        # initialize V_star
        self.V_star = set()

        for _ in range(self.k):
            self.counter += 1
            V_without_Vstar = list(set(graph.nodes).difference(self.V_star))
            self.V_star.add(np.random.choice(V_without_Vstar))

        # initial graph saved before merging starts
        if plot_graph is True:
            self.PlotGraph(graph=graph, color_diff=list(self.V_star), save_plot=True, dir_name='plot_output', method='randomized')


 # instantiate an object
WL = WarehouseLocation()
```

In [4]:

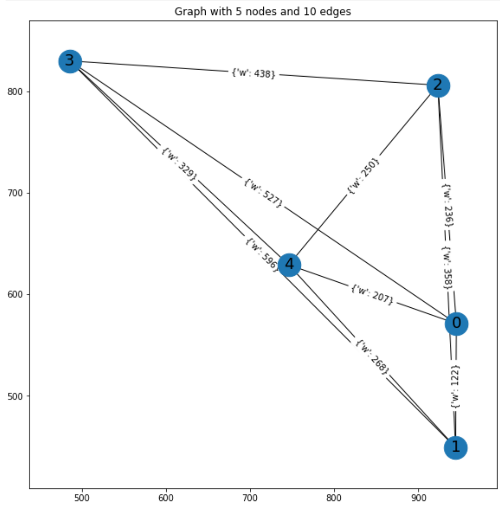
```
# GENERATE GRAPHS
# generates graphs and save in specified output folder
WL.GenerateGraphs(vertices_range=(2,20), dir_name='pickles', remove_existing=True)  # vertices from 2-10 graphs

# vertices of graphs generated
graph_vertices = [_ for _ in range(2, 20+1)]
```

### Example Instance $n = 5, k \in [2, 3]$

In [5]:

```
WL.LoadGraph(nodes=5, dir_name='pickles')
WL.PlotGraph(with_weights=True, save_plot=True, dir_name='plot_output')
```

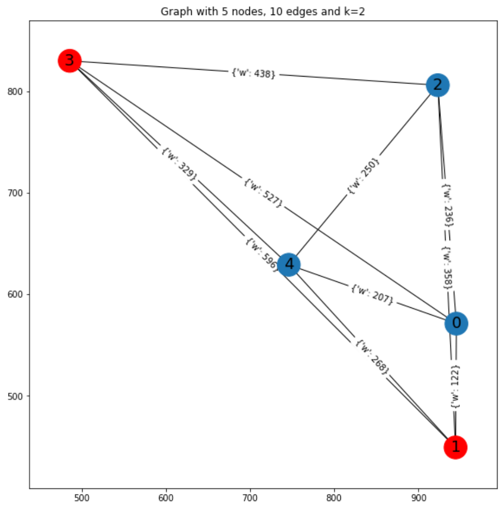
```
Graph with 5 vertices loaded!!
```



### Greedy Heuristics

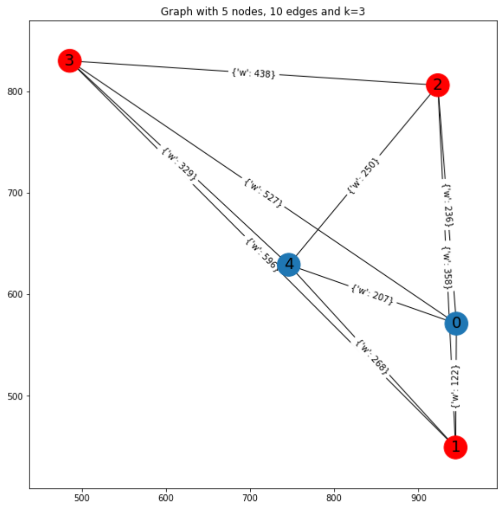
In [6]:
```
WL.GreedyHeuristic(k=2, plot_graph=True)
WL.V_star
```

Out[6]:
```
{1, 3}
```



In [7]:
```
WL.GreedyHeuristic(k=3, plot_graph=True)
WL.V_star
```

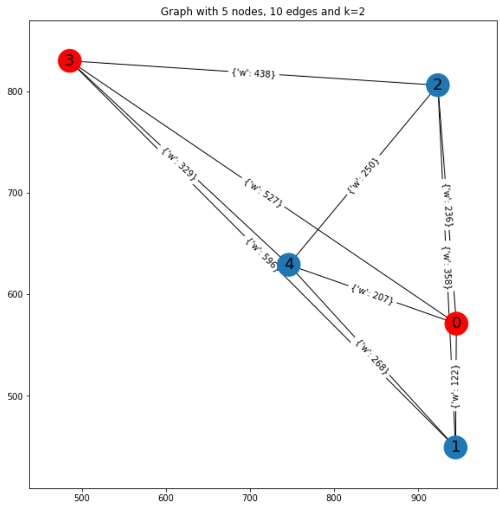
Out[7]:
```
{1, 2, 3}
```



### Exhaustive Search / Bruteforce

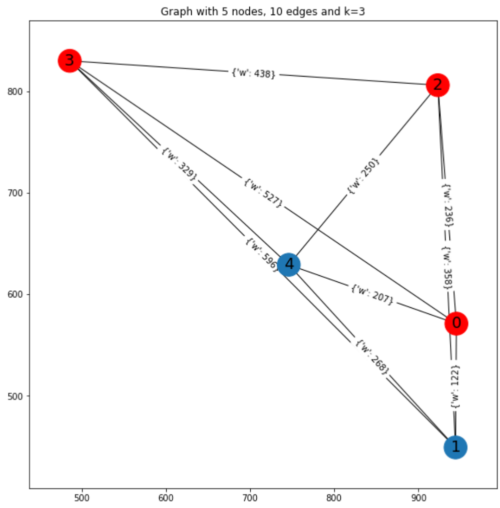
In [8]:
```
WL.ExhaustiveSearch(k=2, plot_graph=True)
WL.V_star
```

Out[8]:
```
(0, 3)
```



In [9]:
```
WL.ExhaustiveSearch(k=3, plot_graph=True)
WL.V_star
```

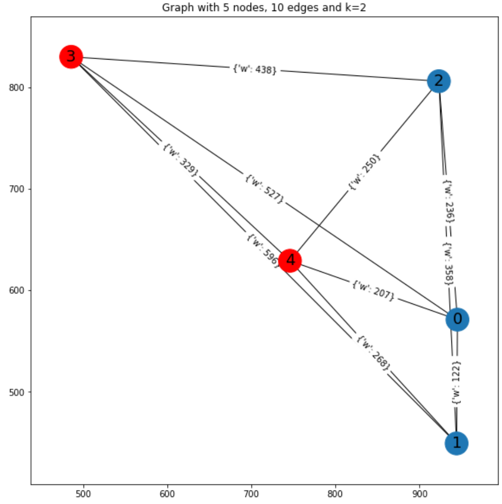
Out[9]:
```
(0, 2, 3)
```



### Randomized Approach

In [10]:
```
WL.Randomized(k=2, plot_graph=True)
WL.V_star
```

Out[10]:
```
{3, 4}
```

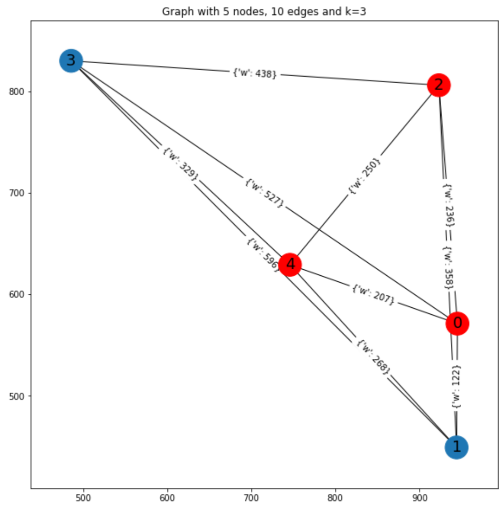


In [11]:

```
WL.Randomized(k=3, plot_graph=True)
WL.V_star
```

Out[11]:

```
{0, 2, 4}
```



## Exhaustive-Search

In [12]:

```
# COUNTER GRAPH FOR EXHAUSTIVE SEARCH.
WL.CreateDirectory(dir_name='executions', remove_existing=False)
with open('graphs/executions/exhaustive.csv','a') as f:
    f.write('nodes,k,output,counter,time(sec)\n')
    for node in graph_vertices:
        WL.LoadGraph(nodes=node, dir_name='pickles')
        # WL.PlotGraph(with_weights=True, save_plot=False, dir_name='plot_output')

        for k in WL.GeneratePrimes(k=node):
            start_ = time.time()
            WL.ExhaustiveSearch(k=k, plot_graph=False)
            line = f'{node},{k},{" ".join([str(_) for _ in WL.V_star])},{WL.counter},
{time.time() - start_}\n'

            f.write(line)
```

```
Graph with 2 vertices loaded!!
Graph with 3 vertices loaded!!
Graph with 4 vertices loaded!!
Graph with 5 vertices loaded!!
Graph with 6 vertices loaded!!
Graph with 7 vertices loaded!!
Graph with 8 vertices loaded!!
Graph with 9 vertices loaded!!
Graph with 10 vertices loaded!!
Graph with 11 vertices loaded!!
Graph with 12 vertices loaded!!
Graph with 13 vertices loaded!!
Graph with 14 vertices loaded!!
Graph with 15 vertices loaded!!
Graph with 16 vertices loaded!!
Graph with 17 vertices loaded!!
Graph with 18 vertices loaded!!
Graph with 19 vertices loaded!!
Graph with 20 vertices loaded!!
```

## Greedy-Search

In [13]:

```
# COUNTER GRAPH FOR GREEDY-HEURESTICS.
WL.CreateDirectory(dir_name='executions')
with open('graphs/executions/greedy.csv','a') as f:
    f.write('nodes,k,output,counter,time(sec)\n')
    for node in graph_vertices:
        WL.LoadGraph(nodes=node, dir_name='pickles')
        # WL.PlotGraph(with_weights=True, save_plot=False, dir_name='plot_output')

        for k in WL.GeneratePrimes(k=node):
            start_ = time.time()
            WL.GreedyHeuristic(k=k, plot_graph=False)

            line = f'{node},{k},{" ".join([str(_) for _ in WL.V_star])},{WL.counter},
{time.time() - start_}\n'

            f.write(line)
```

```
Graph with 2 vertices loaded!!
Graph with 3 vertices loaded!!
Graph with 4 vertices loaded!!
Graph with 5 vertices loaded!!
Graph with 6 vertices loaded!!
Graph with 7 vertices loaded!!
Graph with 8 vertices loaded!!
Graph with 9 vertices loaded!!
Graph with 10 vertices loaded!!
Graph with 11 vertices loaded!!
Graph with 12 vertices loaded!!
Graph with 13 vertices loaded!!
Graph with 14 vertices loaded!!
Graph with 15 vertices loaded!!
Graph with 16 vertices loaded!!
Graph with 17 vertices loaded!!
Graph with 18 vertices loaded!!
Graph with 19 vertices loaded!!
Graph with 20 vertices loaded!!
```

## Randomized Approach

In [14]:

```
# COUNTER GRAPH FOR RANDOMIZED APPROACH.
WL.CreateDirectory(dir_name='executions', remove_existing=False)
with open('graphs/executions/randomized.csv','a') as f:
    f.write('nodes,k,output,counter,time(sec)\n')
    for node in graph_vertices:
        WL.LoadGraph(nodes=node, dir_name='pickles')
        # WL.PlotGraph(with_weights=True, save_plot=False, dir_name='plot_output')

        for k in WL.GeneratePrimes(k=node):
            start_ = time.time()
            WL.Randomized(k=k, plot_graph=False)
            line = f'{node},{k},{" ".join([str(_) for _ in WL.V_star])},{WL.counter},
{time.time() - start_}\n'

            f.write(line)
```

```
Graph with 2 vertices loaded!!
Graph with 3 vertices loaded!!
Graph with 4 vertices loaded!!
Graph with 5 vertices loaded!!
Graph with 6 vertices loaded!!
Graph with 7 vertices loaded!!
Graph with 8 vertices loaded!!
Graph with 9 vertices loaded!!
Graph with 10 vertices loaded!!
Graph with 11 vertices loaded!!
Graph with 12 vertices loaded!!
Graph with 13 vertices loaded!!
Graph with 14 vertices loaded!!
Graph with 15 vertices loaded!!
Graph with 16 vertices loaded!!
Graph with 17 vertices loaded!!
Graph with 18 vertices loaded!!
Graph with 19 vertices loaded!!
Graph with 20 vertices loaded!!
```

## REFERENCES

[1] Padh, Kirtan, Ola Svensson, and Ashkan Norouzi-Fard. The k-center problem. Diss. Thesis, 2015. Available: https://www.researchgate.net/profile/Kirtan-Padh/publication/287621460_The_k-center_problem/links/5677dae308ae125516ee435e/The-k-center-problem.pdf

[2] Erdos, Paul, and Alfréd Rényi. "On the evolution of random graphs." Publ. Math. Inst. Hung. Acad. Sci 5.1 (1960): 17-60. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.348.530&rep=rep1&type=pdf

[3] Robič, B. i Mihelič, J. (2005). Solving the k-center Problem Efficiently with a Dominating Set Algorithm. Journal of computing and

information technology, 13 (3), 225-234. https://doi.org/10.2498/cit.2005.03.05

[4] J. Garcia-Diaz, R. Menchaca-Mendez, R. Menchaca-Mendez, S. Pomares Hernández, J. C.Pérez-Sansalvador and N. Lakouari, "Approximation Algorithms for the Vertex K-Center Problem: Survey and Experimental Evaluation," in IEEE Access, vol. 7, pp. 109228-109245, 2019, doi: 10.1109/ACCESS.2019.2933875.