The Warehouse Location Problem

ABLORDEPPEY Prosper (106382)

Abstract **–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.

The project further compares the solutions generated by the different approaches for some example problem instances. Afterwards, we carry out systematic computational experiments and analyze the computational complexity of the developed algorithms for prime numbers $k \in \{2, 3, 5, 7, \dots\}$.

Keywords –

- **$n = |\mathcal{V}|$ - the number of vertices of graph $\mathcal{G}$.**
- **$m = |\mathcal{E}|$ - the number of edges of graph $\mathcal{G}$.**
- **$(u, v) \in \mathcal{E}$ - is an edge of $\mathcal{G}$.**
- **$w((u, v)) \in \mathcal{W}$ - is the euclidean distance between the edge connecting $u, v \in \mathcal{V}$ for $(u, v) \in \mathcal{E}$. Thus $\mathcal{W} : \mathcal{E} \to \mathbb{Z}^+$**
- **$\mathcal{V}^*$ - a subset of $\mathcal{V}$ of size $(k \leq n)$ where the warehouses should be located/built, in order to minimize the maximum distance of the cities from a warehouse. More formally, $\mathcal{V}^* \subseteq \mathcal{V} \ni \sup_{v \in \mathcal{V}} w_{\mathcal{G}(v, \mathcal{V}^*)}$ is minimized [9].**
- **$Adj_{\mathcal{G}}$ - the adjacency matrix of graph $\mathcal{G}$, is a two dimensional (2D) matrix of weights/euclidean distances connecting pair of vertices in the network. [3].**
- **counter - the number of operations carried out for each k.**

I. Outline Of Implementation

The outline of this project is given as follows;

II. Project Directory Structure
III. Brute-Force or Exhaustive-Search
IV. Greedy-Heuristic
V. Randomized Approach
VI. Formal Analysis
VII. Practical Analysis
VIII. Example Case ($n = 5$, $k \in [2, 3]$)

II. Project Directory Structure

The folder structure for this project is graphically given in Fig. (1). From the representation, the project has one (1) main directory and the *Warehouse_Location* python notebook file. Detailed description is presented below;

```
Project
 |
 ├─ graphs
 |   ├─ pickles
 |   |   ├─ 2_nodes.gpickle
 |   |   ├─ 3_nodes.gpickle
 |   |   ├─ ...
 |   |   ├─ 19_nodes.gpickle
 |   |   └─ 20_nodes.gpickle
 |   ├─ plot_output
 |   |   ├─ exhaustive_n5_k2.png
 |   |   ├─ exhaustive_n5_k3.png
 |   |   ├─ greedy_n5_k2.png
 |   |   ├─ greedy_n5_k3.png
 |   |   ├─ randomized_n5_k2.png
 |   |   ├─ randomized_n5_k3.png
 |   |   └─ n5.png
 |   └─ executions
 |       ├─ exhaustive.csv
 |       ├─ greedy.csv
 |       └─ randomized.csv
 └─ Warehouse_Location.ipynb
```

Fig. 1 - Project Directory Structure

- **graphs**
  This folder hosts two (2) sub-directories as detailed below;
  1. *pickles* :- This directory holds all the graphs generated and saved in $gpickle$ format for the $networkx$ library. In total, eighteen (18) graphs were gener-

ated, with number of vertices ranging from $(2-20)$ as shown in the output of Fig (1).

2. *plot_output* :- This directory also holds the output of the graphs generated using each of the implemented algorithms for varying *k-sets* considered. The output of Fig (1) shows output of the search algorithms implemented using the graph with five (5) vertices.

3. *executions* :- The repository contains csv register of *counter values*, *output of optimal locations* and *time* it took to find a solution for each *k-set* considered for a given *vertex*. As seen from the tree representation, there are only three (3) registers/files in this sub-folder.

- **Warehouse_Location.ipynb**
  This is the python notebook file generating the *graphs* and *plot_output* files detailed above.

  All the implementations were done in Python. The packages used were; numpy, shutil, os, time,itertools, networkx and matplotlib. The functions for which these packages were used are detailed in the $warehouse_location.ipynb$ file.
  This file implements two (2) classes as detailed below;

  – **IOBASE Class** (Input/Output Operations)
  We employed six (6) functions to help with the implementation of this project. They are;

    1) CreateDirectory
       ∗ create directories to hold graphs and plots.
       ∗ deletes directories as requested.
    2) GenerateNodeCoordinates
       ∗ After successful generation og *Erdos Renyi Graphs*, for consistency with the preamble, this method generates integer valued coordinates to each vertex in the XOY plane between 1 and 1000.
    3) ComputeEdgeDistances
       ∗ the euclidean distance between the source and sink of each edge of the graph is computed in this method. The Euclidean distance is expressed as;

$$w((u,v)) = \sqrt{\sum_{i=1}^{2}(u_i - v_i)^2}$$

       ∗ for a cleaner representation, the computed distances were rounded to the nearest whole number.
    4) GenerateGraphs
       ∗ implements the generation of random *Erdos Renyi Graph* for specified number of vertices.
       ∗ fetch integer valued coordinates (between 1-1000) for all vertices (from *GenerateNodeCoordinates*) to update graph.
       ∗ fetch computed euclidean distances (from *ComputeEdgeDistances*) for all edges to update the graph.
       ∗ plot graphs after generation.
       ∗ save the graph to current graphs directory in $gpickle$ format.
    5) LoadGraph
       ∗ load a graph from file.
       ∗ plot the graph if requested.
    6) PlotGraph
       ∗ main implementation of graph plotting.
       ∗ weighted or not weighted.
       ∗ partial coloring of given vertices.

  – **WarehouseLocation Class** (main implementation)
  This is the main class that implements the search algorithms, namely; Exhaustive Search, Greedy and randomized Algorithms. A method for generating the prime numbers used for k was also implemented.

III. Exhaustive-Search Implementation

Exhaustive search means we try every possible solution to the problem and choose the best one that satisfies the objective. The main strength of the Exhaustive Search is that it is guaranteed to find the best optimal solution from all possible solutions, in this case, a *k-set* subset of all cities which minimizes the largest distance of any other city to its closest city in the *k-set* subset selected [8].

The solution to this problem is a subset of the vertices set. So in this case, it means we figure out all *k-set* subsets of the vertices set which are potential solutions to our problem. The *k-set* subset which $minimizes$ the $largest$ distance of any other city to its $closest$ warehouse in the *k-set* is our optimal solution to the warehouse locations problem.

The deficiency of this algorithm is that it takes a long time to complete execution for graphs with larger vertices and/or edges [8].

The **Algorithm 1** is a helper function for the main implementation of the Exhaustive Search as presented in **Algorithm 2**. In its implementation, the goal is to generate all non-empty, *k-set* subsets of the given vertices set $\mathcal{V}$. We obtain such sets by considering the powerset (ρ) of $\mathcal{V}$. The function returns a list of tuples, each comprising of a subset of length $k < |\mathcal{V}|$. Each of these subsets are potential solutions to be investigated in the main function as described in **Algorithm 2**.

First, we get all the various subsets of $\mathcal{V}$ that have equal chance of being the optimal solution of interest $(\mathcal{V}^*)$ using the $PotentialVStars$ function in **Algorithm 1**. We find

Algorithm 1: Generate Potential $\mathcal{V}^*$s

```
Input: $\mathcal{V}$, $k$    // $\mathcal{V}$:vertices set, $k$:# locations
Output: $p\_sets$    // potential locations list
1 Function PotentialVStars:
2     Set $p\_sets \leftarrow$ [ ]
3     Set $\rho \leftarrow$ Powerset($\mathcal{V}$)    // length $2^{|\mathcal{V}|}$
4     foreach $subset \in \rho$ do
          // non-empty subsets each of size $k$
5         if ($subset \neq \emptyset$) or ($subset \neq \mathcal{V}$) then
6             if len($subset$) == $k$ then
7                 $p\_sets$·insert($subset$)
8     return $p\_sets$
```

Algorithm 2: Exhaustive Search

```
Input: $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{W})$, $k$ /* G:graph, k:locations */
Output: $\mathcal{V}^*$    // optimal k-set locations
1 Function ExhaustiveSearch:
      /* $\mathcal{V}^*$ - optimal warehouse locations.
      $p\_vstar$ - potential $\mathcal{V}^*$s.
      $p\_sets$ - set of $p\_vstar$.
      $ol\_max\_dist$ - list of $p\_vstar$ with largest
         distance from cities.                        */
2     Set $p\_sets \leftarrow$ PotentialVStars($\mathcal{V}$, $k$)
3     Set $pv\_max\_d \leftarrow \{p : -\infty \ni \forall p \in p\_sets\}$
4     foreach $p\_vstar \in p\_sets$ do
5         $c\_dists \leftarrow$ [ ] // closest dist from $p\_vstar$
6         foreach $vertex \in \mathcal{V}$ do
              /* find closest distance from
                 $p\_vstar$, for every city
              update $c\_dists$                        */
7             $d \leftarrow \text{Adj}_{\mathcal{G}}[vertex]$  // neighbour dists
8             $ds\_from\_pvstar \leftarrow d$ from $p\_vstar$
9             $c\_dists$·add(min($ds\_from\_pvstar$))
10        $largest\_dist \leftarrow$max($c\_dists$)
11        if $largest\_dist > pv\_max\_d[p\_vstar]$ then
12            $pv\_max\_d[p\_vstar] \leftarrow largest\_dist$
13        $V\_star \leftarrow$ min($pv\_max\_d$)
14    return $V\_star$
```

the largest distance of the closest city/vertex in the graph to these potential optimal solutions. The optimal solution ($\mathcal{V}^*$) is the set corresponding to the minimum distance.

For each of the element of the array or potential optimal location set, we then compute the largest distance of the closest city to vertices making up the set of the element. The minimum is then returned as the optimal location for the warehouses.

IV. Greedy-Heuristics Implementation

A Greedy algorithm is any algorithm that follows the problem-solving heuristic of making a locally optimal choice at each stage [1]. In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time, where as an exhaustive search will look at all possible solutions and pick the most optimal one. Thus, the Greedy heuristics runs faster as compared to the Exhaustive search.

A solution to this warehouse location problem, was given by [5] in 1985 using a greedy heuristics. In this algorithm, we build the set $\mathcal{V}^*$ by iteratively adding a vertex to which is farthest from the set of vertices we have already chosen. We keep doing this until we have chosen k vertices as implemented in Algorithm (3). The algorithm implements a random initialization of the first member of $\mathcal{V}^*$. This algorithm is a modified version of the vanilla randomized approach described under the Randomized implementation V (random sampling k vertices from $\mathcal{V}$), constraint on the fact that, new vertices added to $\mathcal{V}^*$ are farthest away from members of $\mathcal{V}^*$. This thereby increases the chance of realizing if not the true warehouse locations set, but an estimate very close to the true set. This is further confirmed in the following sections.

Algorithm 3: Greedy Heuristics

```
Input: $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{W})$   /* G:graph, k:locations */
Output: $\mathcal{V}^*$    // warehouse locations $\mathcal{V}^* \subseteq \mathcal{V}$
1 Function GREEDY:
2     Set $\mathcal{V}^* \leftarrow \emptyset$
3     $\mathcal{V}^*$·add(random($v \in \mathcal{V}$))
4     for $iter$ in $range(k\text{-}1)$ do
5         Set $max\_dist \leftarrow -\infty$    // maximum trick
6         Set $max\_dist\_node \leftarrow None$
7         foreach $node \in \mathcal{V}$ do
8             foreach $sel\_node \in \mathcal{V}^*$ do
                  // distance from adj. matrix
9                 $dist \leftarrow \text{Adj}_{\mathcal{G}}[node][sel\_node]$
10                if $dist < max\_dist$ then
11                    $max\_dist\_node \leftarrow node$
12        $\mathcal{V}^*$·add($max\_dist\_node$)
13    return $\mathcal{V}^*$
```

V. Randomized Implementation

The Randomized approach as described in **Algorithm 4** provides a naive solution to the warehouse location problem. The algorithm selects k random unique vertices one (1) at a time. In probability theory, this sampling technique is known as Sampling Without Replacement. As a result, the distances between vertices pair, which are of high essence to satisfying the objective function is disregarded.

In order to improve the output/result of the randomized approach presented, we can decide to run the algorithm multiple times and report the location set which minimizes the largest distance of any other city as the optimal solution set. Another approach as presented in the Greedy Search implementation 3 partially fulfills the objective function could be considered.

Algorithm 4: Randomized Algorithm

```
Input: $\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{W}), k$ /* G:graph, k:locations */
Output: $\mathcal{V}^*$   // optimal locations set estimate
1 Function ProbabilisticRandomized:
2     Set $\mathcal{V}^* \leftarrow \emptyset$
3     for $iter$ in $range(k)$ do
          // choose random vertex from $\mathcal{V}$
4         $selected \leftarrow$ rand($\mathcal{V} \backslash \mathcal{V}^*$)
5         $\mathcal{V}^*$.add($selected$)
6     return $\mathcal{V}^*$
```

VI. Formal Analysis

This section presents a formal analysis of the complexities and running times of the exhaustive-search algorithm described in **Algorithm 2**, the greedy implementation in **Algorithm 3** and the randomized approach as described in **Algorithm 4**.

A. Exhaustive Search

Considering a connected graph with n vertices. Thus, $|\mathcal{V}| = n$. If p_sets is the set of all potential optimal location sets of $\mathcal{V}^*$. An expression for the cardinality of p_sets is given as

$$|p\_sets| = \binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$$

.

Given a set with n elements, the total number of different subsets that can be obtained from the set is expressed as $|\rho| = 2^n$ [2]. From the implementation of the helper function as described in **Algorithm 1**, we generated the power-set of the vertices set (ρ) and filtered it to obtain our desired list of potential optimal location set. Since our interest is to obtain non-empty subsets of size ($k < |\mathcal{V}|$), we eliminate the empty set ($\emptyset$) and the set with the full vertices. This results in $(2^n - 2)$ subsets returned by this function. The complexity of this algorithm runs in exponential time given as $C = \mathcal{O}(\mathbf{2^n})$.

The main implementation finds the largest distance of the closest city for each potential optimal location set and stores the result in an associative array. This is achieved by going over each of the n vertices. The key with the smallest/minimum distance is our desired optimal location set for our warehouse location problem.

Summing it all up, the Exhaustive Search implementation of the warehouse facility problem runs in complexity

$$\begin{aligned} C &= \mathcal{O}\left(2^n \cdot n\right) \\ &= \mathcal{O}\left(\mathbf{2^n}\right) \end{aligned}$$

Exponential running time algorithms like this is one of the worse algorithms to have for larger graph inputs if we want to obtain result within the fastest time which is impossible but will guarantee optimal solution.

B. Greedy Search

As studied in the Randomized implementation, the probability of any of the members of p_sets being the optimal location set the becomes

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

The Greedy Heuristics improves this probability constraint on the basis that new vertices selected have their distances from the selected vertices largest compared to the other not selected vertices. This ensures an increased chance of obtaining the optimal warehouse location set as compared to the naive randomized implementation.

According to [9], this algorithm runs in polynomial-time and provides a solution that is never worse than twice the optimal solution, which always work only if the distances between cities follow Triangular Inequality (the distance between two points is always smaller than the sum of distances through a third point. This condition is satisfied by the euclidean distance [6].

The Greedy approach in **Algorithm (3)** finds k vertices set $\mathcal{V}^*$ from $\mathcal{V}$. After randomly choosing a vertex from the vertices set, for each of the remaining *(k-1)* vertices to be chosen, they are chosen in a way that, their distance from the selected vertices is maximum.

In a nutshell, the greedy algorithm runs in complexity

$$\begin{aligned} C &= \mathcal{O}\left((k-1)\cdot n \cdot k\right) \\ &= \mathcal{O}\left(k \cdot n \cdot k\right) \\ &= \mathcal{O}\left(k^2 \cdot n\right) = \mathcal{O}\left(n^3\right) \\ \therefore C &= \mathcal{O}\left(n^3\right) \end{aligned}$$

Thus, the Greedy algorithm as discussed above, runs in polynomial time.

C. Randomized Algorithm

In this module, we describe the analysis of the vanilla Randomized approach which results in an estimate of the true solution of the warehouse location problem. In a more strict sense, we say we perform a uniform random sampling of k vertices from $\mathcal{V}$ without replacement. The order of picking the vertices is not relevant to us. As described in [7], the probability of realizing the true location set $\mathcal{V}^*$ is given as

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

Here, no optimization is done in order to obtain a more realistic estimate of the true locations set. Result from this implementation is likely not to be the true optimal locations set.

This algorithm runs in complexity $\mathcal{O}(k.n) = \mathcal{O}(n^2)$. On multiple runs, the algorithm may return very different estimates of the optimal warehouse locations compared to the degree of randomness of the Greedy Heuristics.

This is because, the Greedy approach partially satisfies the objective function.

VII. Practical Analysis

This section presents analysis of results for the individual algorithms. Table of results (counter values, running time (secs) and ratio of the counter values) were detailed for each implementation. Various summary graph outputs were also presented and equally analyzed.

A. *Exhaustive Search*

A snapshot of the results obtained by implementing the exhaustive search algorithm on all graph inputs for the various possible k values is presented in **Table I**. The **executions** directory holds the complete and detailed register of these summaries.

Fig. 2 is a line plot of counter value recorded for all eighteen (18) graph inputs considered in this experiment for all *k-set* subsets of vertices analyzed for each graph. As it can be observed, the counter values rise exponentially and peaks at median value of k and decays exponentially after this point. This phenomenon is observed across all graphs, assuming a bell-shaped distribution of the counter values as shown in the outputs. A summary display is shown in the line plot in **Fig. 3**. Here, we clearly see an exponential rise in the counter values for larger graphs.

Just as analyzed for the counter values, the time taken to complete each *k-set* execution is also recorded. A summary of the execution time is given in **Fig. 4**. It took a maximum of roughly thirty (30) seconds to complete search for a graph with twenty (20) vertices.

The time taken for larger graphs to complete appears to be exponential as well as depicted in **Fig. 4**. My laptop was able to find solution for up to twenty-four (24) graph inputs in a reasonable amount of time.

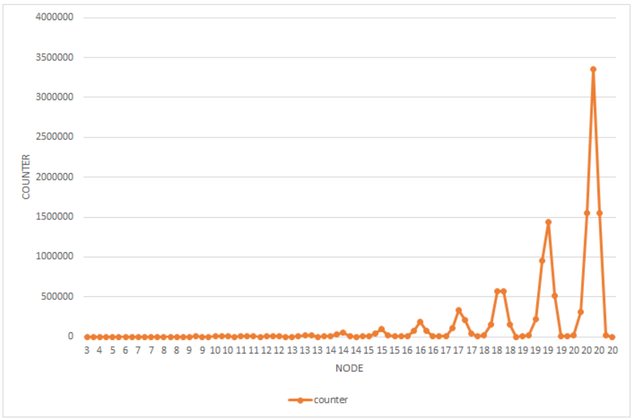

Fig. 2 - Exhaustive - Counter vs Nodes vs k

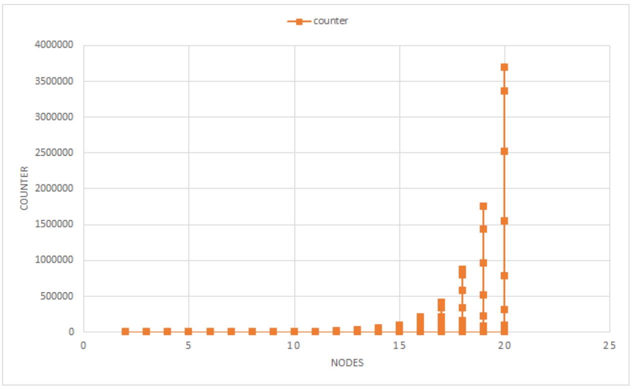

Fig. 3 - Exhaustive (summary) - Counter vs Node vs k

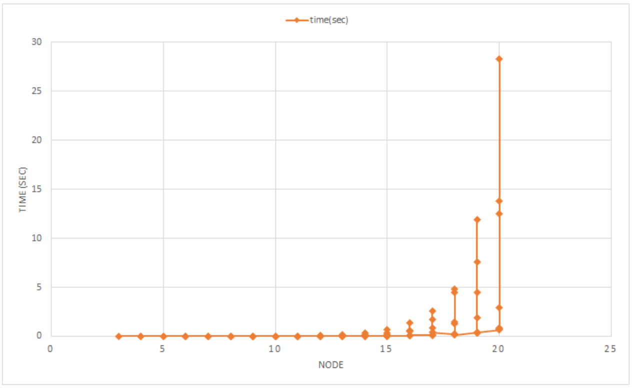

Fig. 4 - Exhaustive - Time vs Node vs k

node	k	counter	ratio	time (secs)
3	2	9	-	0
4	2	24	1.5	0
4	3	16	0.67	0.001
5	2	50	2	0
5	3	50	1	0
6	2	90	2.5	0.001
6	3	120	1.333	0
6	5	36	0.3	0.001
10	2	450	4.5	0.003
10	3	1200	2.67	0.007
10	5	2520	2.1	0.0155
10	7	1200	0.476	0.0078
20	2	3800	9.5	0.674
20	3	22800	6	0.791
20	5	310080	13.6	2.82
20	7	1550400	5	12.118
20	11	3359200	2.167	27.414
20	13	1550400	0.462	13.308
20	17	22800	0.0147	0.817
20	19	400	0.0175	0.632

TABLE I
Exhaustive - Output Snapshot

B. *Greedy Heuristic*

For each prime number k, for a given vertex, **Table II** we presents a record of the number of operations (counter), the ratio of the number of operations (ratio) and the time (secs) the algorithm takes to complete execution. This is a snapshot of the output for the Greedy implementation is presented. A detailed result is given in the register output.

Unlike in the case of the Exhaustive-Search, the time taken for the Greedy-search implementation is insignificant. The maximum time taken spent on the largest graph with twenty (20) nodes is barely three (3) milliseconds. The results output also show very good estimations by the greedy heuristics. An instance of this is described in the next section where out of three (3) locations, the greedy heuristics was able to correctly estimate two (2) of the locations. Out of two (2), it was able to correctly estimate one (1) of the locations.

The time taken for larger graphs to complete looks more to be linear linear as seen in **Fig. 7**.

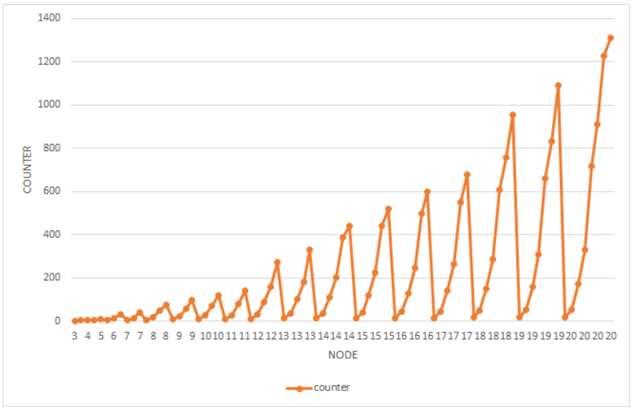

Fig. 5 - Greedy - Counter vs Nodes vs k

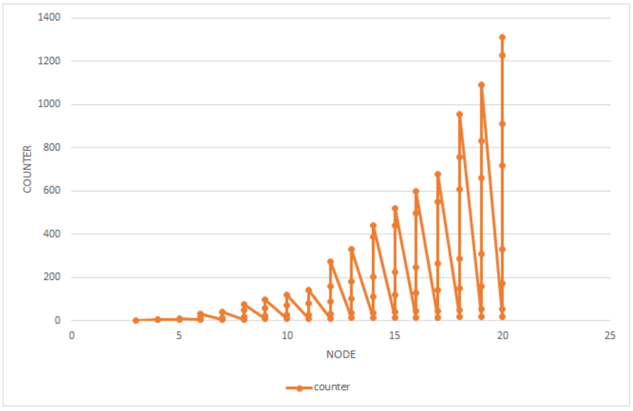

Fig. 6 - Greedy (summary) - Counter vs Node vs k

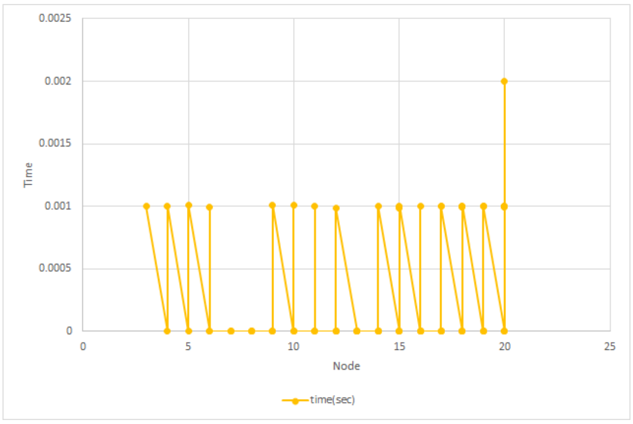

Fig. 7 - Greedy - Time vs Node vs k

node	k	counter	ratio	time (secs)
3	2	3	-	0.0
4	2	2	1	0
4	3	3	1.5	0.0
5	2	2	0.667	0
5	3	3	1.5	0.001
6	2	2	0.667	0
6	3	3	1.5	0
6	5	5	1.667	0
10	2	2	0.286	0
10	3	3	1.5	0.001
10	5	5	1.667	0
10	7	7	1.4	0.001
20	2	2	0.118	0
20	3	3	1.5	0.001
20	5	5	1.667	0.001
20	7	7	1.4	0
20	11	11	1.571	0.001
20	13	13	1.182	0
20	17	17	1.308	0.001
20	19	19	1.118	0.001

TABLE II
GREEDY - OUTPUT SNAPSHOT

C. *Randomized Approach*

A snapshot of result from the output register for this strategy is presented in **Table III** recording the number of operations, and time taken to complete each operation of k

Fig. 8 shows a plot of the number of operations (counter) carried out for each k. As shown, we observe a linear relationship between the number of operations its corresponding graph node.

The trend in time taken to complete an operation seems to be constant at about one (1) milliseconds as observed from **Fig. 9**. This is the most fastest algorithm discussed so far.

As a trade-off, there is no guarantee the obtained result is near optimal.

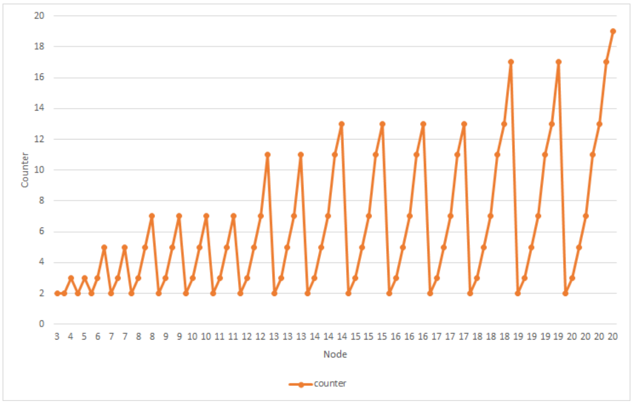

Fig. 8 - Randomized - Counter vs Nodes vs k

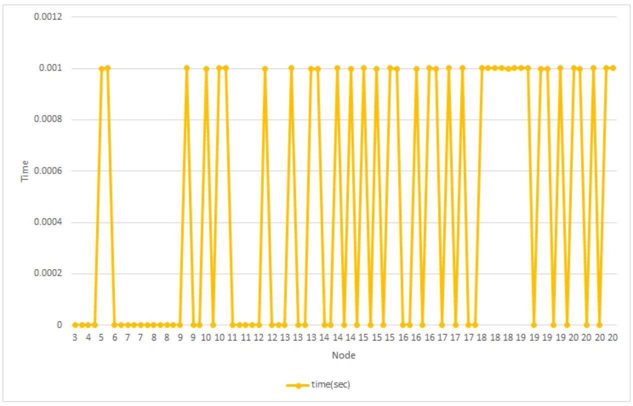

Fig. 9 - Randomized - Time vs Node vs k

node	k	counter	ratio	time (secs)
3	2	3	-	0.001
4	2	4	1.333	0
4	3	8	2	0.001
5	2	5	0.625	0
5	3	11	2.2	0.001
6	2	6	0.545	0
6	3	14	2.333	0.001
6	5	31	2.214	0
10	2	10	0.101	0
10	3	26	2.6	0
10	5	71	2.731	0.0015
10	7	120	1.69	0
20	2	20	0.0184	0
20	3	56	2.8	0.001
20	5	171	3.054	0
20	7	330	1.93	0
20	11	716	2.17	0.001
20	13	911	1.272	0.001
20	17	1225	1.345	0.002
20	19	1312	1.071	0.001

TABLE III
RANDOMIZED - OUTPUT SNAPSHOT

VIII. EXAMPLE CASE ($n = 5$)

In this section, we consider instance problem of a graph with $|\mathcal{V}| = 5$ vertices and $|\mathcal{E}| = 10$ edges, as presented in **Fig. 10**. Hence, the k values considered for this example are $[2, 3]$ which are the expected prime numbers as described in the preamble. For this considered example, the Exhaustive-Search, Greedy and Random algorithms were implemented and their corresponding results analyzed.

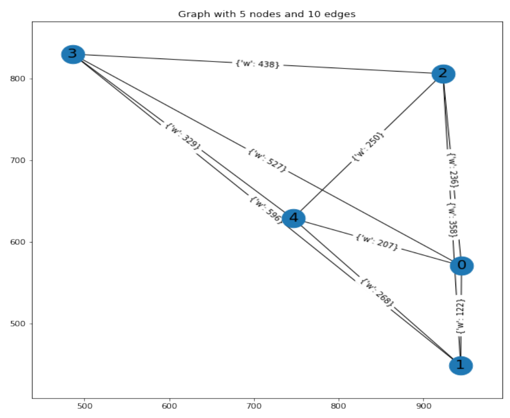

Fig. 10 - Graph With 5 vertices and 10 edges

Fig. 11 shows a graphical output of a solution to the above problem using the Exhaustive-Search, **Fig. 12** also represents a graphical optimal warehouse locations set using the Greedy Heuristics and **Fig. 13** shows similar result/solution using a Randomized algorithm.

The Exhaustive-Search implementation presents the optimal locations solution. As it can be observed, we have the Greedy heuristics able to provide a near optimal solution set. For *k=2*, the Greedy implementation was able to correctly predict one (1) warehouse location and for *k=3*, it correctly predicted two (3) warehouse locations. We can say these near-accurate estimation of the optimal solution set is as a result of the partial optimization implemented by the algorithm. If there is a coincidence of the Randomized algorithm correctly estimating a warehouse location, we can say the coincidence is due to chance and not based on some carefully chosen heuristics as can be seen for in the outputs.

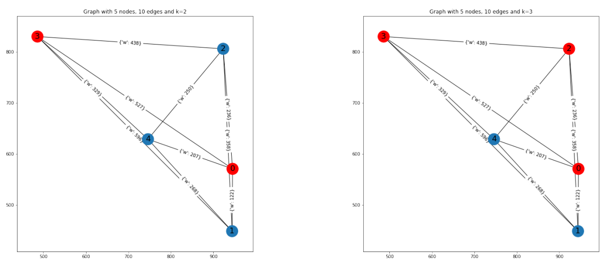
Graph with 5 nodes, 10 edges and k=2
Graph with 5 nodes, 10 edges and k=3

Fig. 11 - Exhaustive (k=2, k=3)

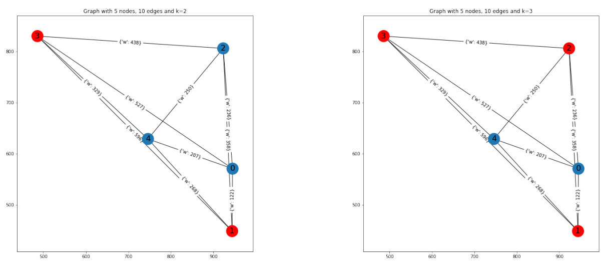
Graph with 5 nodes, 10 edges and k=2
Graph with 5 nodes, 10 edges and k=3

Fig. 12 - Greedy (k=2, k=3)

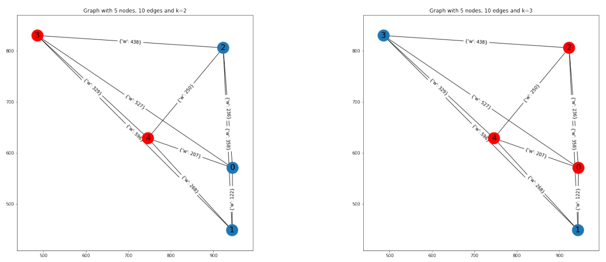
Graph with 5 nodes, 10 edges and k=2
Graph with 5 nodes, 10 edges and k=3

Fig. 13 - Randomized (k=2, k=3)

IX. Conclusion

In conclusion we have seen that a Exhaustive-Search algorithm to solve the warehouse location problem is not feasible for large scale Graphs. The execution time is exponential.

A Greedy Heuristic solution to this problem, is substantially more optimal running in what appears to be linear time for Graphs which would not even be thought possible to run compared to a brute force algorithm. This shows the importance of greedy heuristics, should an approximate solution to a problem be acceptable.

With the Randomized algorithm, we observe a constant time for its execution regardless the size of its graph input. This is a very fast algorithm in obtaining an estimate which may be very far from optimal. The Greedy algorithm which is an extension of the naive randomized approach is our best shot which trades speed for accuracy.

References

[1] Paul E Black. greedy algorithm, dictionary of algorithms and data structures. *US Nat. Inst. Std. & Tech Report*, 88:95, 2012.

[2] Borivoje Djokić, Masahiro Miyakawa, Satoshi Sekiguchi, Ichiro Semba, and Ivan Stojmenović. *Parallel algorithms for generating subsets and set partitions*, volume 450, pages 76–85. 01 2006.

[3] Nykamp DQ. Adjacency matrix definition.

[4] P Erdos and A Renyi. On the random graphs 1, vol. 6. *Institute of Mathematics University of DeBreceniens, Debrecar, Hungary*, 1959.

[5] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[6] Daniel J Greenhoe. Properties of distance spaces with power triangle inequalities. *arXiv preprint arXiv:1610.07594*, 2016.

[7] Daniel G Horvitz and Donovan J Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association*, 47(260):663–685, 1952.

[8] Liu Hui and Cao Yonghui. Study of heuristic search and exhaustive search in search algorithms of the structural learning. In *2010 Second International Conference on Multimedia and Information Technology*, volume 1, pages 169–171. IEEE, 2010.

[9] Kirtan Padh. The k-center problem, 12 2015.