The Warehouse Location Problem

Pedro Figueiredo 97487

Abstract - **In the Warehouse Location problem (WLP), a company considers opening warehouses at some candidate locations in order to supply its existing stores. In graph theory this means finding a set of k vertices for which the largest distance of any other vertex to its closest vertex in the k-set is minimum.**

This article's purpose is to present a comparative analysis of different types of solutions for the warehouse location problem.

I. INTRODUCTION

This article comes within the scope of the exam season advanced algorithms project, where the solving of the warehouse location problem will be demonstrated.

Along with the report it's possible to find the file **graphgenerator.py**, which uses a constant seed to generate graphs with n vertices. Also, it's possible to find the files **exhaustivesearch.py, greedysearch.py** and **randomsearch.py**, where the code of the algorithms can be found.

II. GRAPH GENERATION

To begin with, graphs are generated, as previously mentioned, through the function graphgenerator(n), where n is the number of vertices intended for the graph. This function returns vertices placed at random positions in the xOy plane with coordinates between [1,1000].

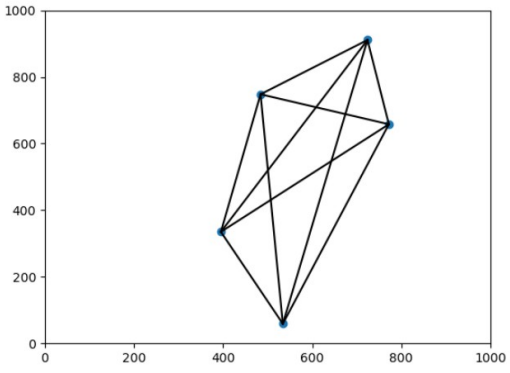

Fig. 1 - Graph example for n = 5 vertices

In the example of figure 1, the following vertices were used, resulting from the generating function of the graphs: (395,335), (534, 59), (772, 658), (724, 911), (484, 748).

After generating the graphs and saving them in separate files, the warehouse location problem was solved through an exhaustive search algorithm, a greedy heuristics algorithm and a randomized algorithm.

III. FORMAL COMPUTATIONAL ANALYSIS

A. *Exhaustive Search*

This algorithm tries to find the set of k vertices from a given set of n vertices, such that the largest distance of any other vertex to its closest vertex in the k-set is minimized.

The time complexity of the algorithm is

$$O(n!/(n-k)!) * O(nk) \qquad (1)$$

, where *O(n!/(n − k)!)* is the number of combinations of k vertices from a set of n vertices and *O(nk)* is the time to compute the closest distance for each vertex to a k-set.

The first term grows rapidly as n increases, leading to an exponential increase in the number of combinations that need to be checked. The second term depends on the number of vertices and the k-set size, but for large values of n andk, the algorithm becomes computationally expensive.

In conclusion, while the algorithm is guaranteed to find the optimal solution, it may not be feasible for large values of n and k due to its high time and space complexity.

B. *Greedy Search*

The computational complexity of the algorithm can be analyzed as follows:

The distance calculation between every pair of vertices has a time complexity of $O(n^2)$, where n is the number of vertices. This is because for each vertex, the code has to compare it with all other n-1 vertices.

The removal of a vertex from the list of vertices has a time complexity of O(n), as the code has to search through the list to find the correct vertex and then remove it.

The loop continues until the length of the final solution is equal to k. The maximum number of iterations is k/2, when k is even, and (k+1)/2 when k is odd.

Putting everything together, the overall time complexity of the algorithm is

$$O(k * n^2) \quad (2)$$

C. *Randomized algorithm*

This algorithm iterates for a fixed number of iterations, let's call it m.

In each iteration the algorithm selects k vertices at random from the set of vertices, this step has a time complexity of O(k), as it is a simple sampling process.

It then calculates the largest distance of any other vertex to its closest vertex in the k-set, this step has a computational complexity of O(n * k) where n is the number of vertices and k is the number of selected vertices. This is because it needs to iterate through the set of vertices and for each vertex find the closest vertex in the k-set and calculate the distance.

Finally, the algorithm checks if the largest distance is lower than the lowest largest distance found so far and updates the final set of k vertices and the lowest largest distance, this step has a computational complexity of O(1).

As a result, the total computational complexity of the algorithm is

$$O(m) * (O(k) + O(nk) + O(1)) = O(mn * k) \quad (3)$$

where n is the number of vertices, k is the number of selected vertices, and m is the number of iterations.

IV. EXPERIMENTS

A. *Basic Operations*

Table 1 shows the number of basic operations related to exhaustive search, table 2 shows the number of basic operations related to greedy search and table 3 shows the values for randomized search.

Exhaustive Search					
n	k				
-	5	7	11	13	17
6	450				
8	5880	1176			
10	34020	22680			
12	130680	182952	4356		
14	390390	936936	156156	7098	
16	982800	3603600	2162160	327600	
18	2184840	11361168	17853264	5680584	15606

Table 1 - Basic operations of exhaustive search for different values of n and k

Greedy Search					
n	k				
-	5	7	11	13	17
6	66				
8	147	150			
10	264	282			
12	417	462	483		
14	606	690	753	756	
16	831	966	1095	1113	
18	1092	1290	1509	1554	1575

Table 2 - Basic operations of greedy search for different values of n and k

Random Search					
n	k				
-	5	7	11	13	17
6	900				
8	1200	1680			
10	1500	2100			
12	1800	2520	3960		
14	2100	2940	4620	5460	
16	2400	3360	5280	6240	
18	2700	3780	5940	7020	9180

Table 3 - Basic operations of random search for different values of n and k

Through the tables it's possible to see that the number of basic operations increases in all algorithms, however it is worth noting the fact that in the exhaustive algorithm the values grow much more, reaching values that are too large, and for n = 18 the number of basic operations is already in the order of 10^7, as the algorithms iterates all possible k-size combinations of the existing vertices (n).

It's also worth noting that the number of basic operations for this approach follows a gaussian distribution for varying numbers of k with equal value n.

For the greedy algorithm, we do not require iterating through multiple combinations and computing all distances between vertices to reach the final result, instead only computing the distance between 2 vertices to find those furthest apart at every iteration.

The randomized approach uses a different method to restrict operations, only selecting m k-sized sets of vertices to test. Assuming a consistent m throughout all the tests (as is the case), the starting values may be higher than the greedy approach, as we may test sets repeatedly, but the growth will be much slower.

B. *Execution Times*

Table 4 shows the execution times related to exhaustive search, table 5 shows the number of execution times related to greedy search and table 6 shows the values for randomized search.

Exhaustive Search (s)					
n	k				
-	5	7	11	13	17
6	0.001				
8	0.015	0.003			
10	0.086	0.057			
12	0.034	0.45	0.01		
14	0.98	2.74	0.56	0.02	
16	2.49	9.42	5.51	1.00	
18	6.08	29.12	47.43	18.66	0.05

Table 4 – Execution times of exhaustive search for different values of n and k

Greedy Search (s)					
n	k				
-	5	7	11	13	17
6	0.0002				
8	0.0004	0.0004			
10	0.0006	0.0008			
12	0.001	0.0011	0.0011		
14	0.0015	0.0018	0.002	0.0021	
16	0.002	0.0024	0.0028	0.0034	
18	0.0032	0.0033	0.0041	0.0045	0.0049

Table 5 – Execution times of greedy search for different values of n and k

Random Search (s)					
n	k				
-	5	7	11	13	17
6	0.002				
8	0.003	0.004			
10	0.004	0.006			
12	0.005	0.006			
14	0.005	0.008	0.011	0.014	
16	0.006	0.009	0.014	0.02	
18	0.008	0.01	0.015	0.02	0.03

Table 6 – Execution times of random search for different values of n and k

In a similar way to the number of operations, and as can be seen in the tables above, the execution times for the exhaustive algorithm are much larger compared to the other algorithms, and also follow a pattern of being smaller for more extreme k sizes (due to the fact that more extreme k-sizes create smaller sets of possible combinations).

Because the other 2 algorithms are not influenced by the number of possible combinations in their execution, they do not show the same pattern. Also because of the same reason, the execution times are much more reduced. We also see some slightly improved performance from the greedy search compared to the random search. This difference is only in the tenths of second though, and as such they are deemed comparable in performance.

C. *Configurations tested*

In the following tables the number of tested configurations can be seen, for graphs with different numbers of vertices and k-sizes.

Exhaustive Search					
n	k				
-	5	7	11	13	17
6	150				
8	1960	392			
10	11340	7560			
12	43560	60984	1452		
14	130130	312312	52052	2366	
16	327600	1201200	720720	109200	
18	728280	3787056	5951088	1893528	5202

Table 7 – Tested configurations of exhaustive search for different values of n and k

Greedy Search					
n	k				
-	5	7	11	13	17
6	22				
8	49	50			
10	88	94			
12	139	154	161		
14	202	230	251	252	
16	277	322	365	371	
18	364	430	503	518	525

Table 8 – Tested configurations of greedy search for different values of n and k

Random Search					
n	k				
-	5	7	11	13	17
6	300				
8	400	560			
10	500	700			
12	600	840	1320		
14	700	980	1540	1820	
16	800	1120	1760	2080	
18	900	1260	1980	2340	3060

Table 9 – Tested configurations of random search for different values of n and k

Table 7 above shows that the number of configurations for the exhaustive search has a very fast growth, being at the order of 10^7 at n = 18. This is linked to the exponential growth of number of possible combinations, that we have seen in the last 2 subsections. The fact that the number of combinations follows a gaussian distribution for instances of same n and increasing k also explains the tendency for values in a line to first increase and then decrease.

The number of configurations tested will be equal to the number of combinations tested. That's why random search, although it has a higher value for smaller instances (due to the set number of m iterations meaning some combinations were calculated more than once), actually has a very slow increase.

With the greedy algorithm, since we only check one distance in every iteration, the number of configurations tested is fairly smaller.

As tables 8 and 9 show, for the random and greedy approaches, with the increase in the number of vertices, it is expected that the number of configurations tested will also increase, since more edges are needed to connect them.

V. EXPERIMENTAL AND FORMAL ANALYSIS COMPARISON

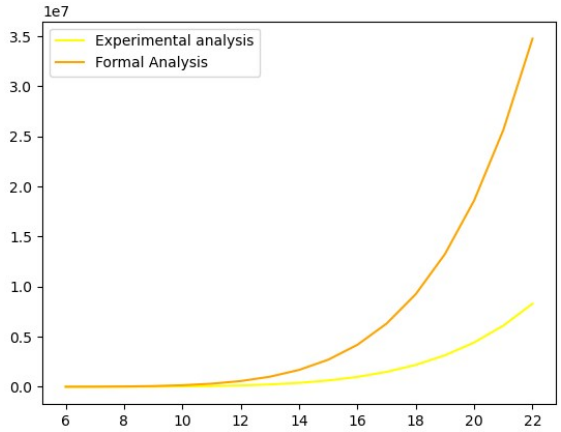

Figure 2 – Exhaustive Search Comparison of Number of Basic Operations with Formal Analysis

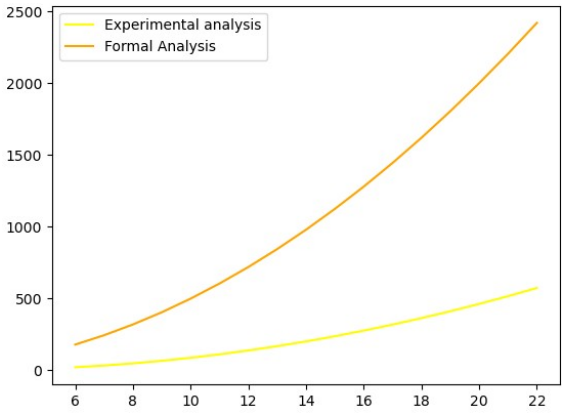

Figure 3 – Greedy Search Comparison of Number of Basic Operations with Formal Analysis

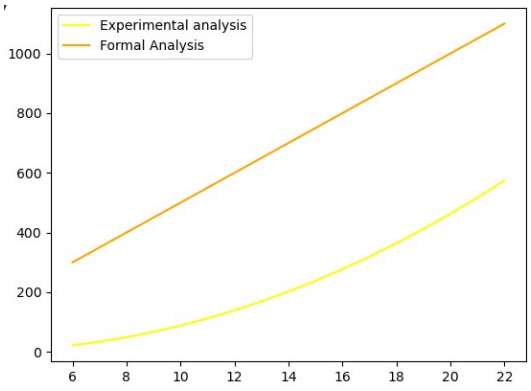

Figure 4 – Random Search Comparison of Number of Basic Operations with Formal Analysis

Looking through the graphs that compare the number of basic operations with the formal analysis is possible to see that, for all the algorithms, the 2 analysis have similar progressions, meaning the formal analysis correctly describe their complexity.

As such, they are a good starting point to estimate execution times and resources for graphs with higher number of vertices and sizes of k.

VI. LARGER PROCESSED GRAPH

For exhaustive search, as the graph size increases, the execution time increases dramatically. The exceptions to this are the smallest and highest k-sizes, that, due to the combinatorial nature of the algorithm previously discussed, are processed quickly when compared to instances the other k-sizes.

For example, a graph with 22 vertices and k = 2 takes 0.08 seconds to process, while the same graph for k = 11 takes 1305 seconds.

As such, we can conclude that the main factor affecting the algorithms performance is it's k-size, with the number of vertices also having an effect.

With the greedy algorithm, unlike brute-force, it was possible to test larger graphs with larger k-sizes, since it is an extremely fast algorithm, the same applying to the randomized algorithm.

For instances with 200 vertices and k = 197, the greedy algorithm takes 4.42 seconds and the random algorithm takes 2.53 seconds.

VII. LARGER PROBLEM INSTANCES

As already mentioned, for the exhaustive search, as the number of vertices and k-size increase , the execution time increases dramatically.

For graphs with very low or very high k-size the algorithm still manages to solve the problem in a reasonably good time. As the average k-sizes become higher, there is an increased difficulty in executing the algorithm.

For example, for a graph with 14 vertices and k = 7, the algorithm takes approximately 2.74 seconds.

With this, it is concluded that in the case of exhaustive search, k-size greatly affects the performance of the algorithm, running well with k-sizes that are close or close to the number of vertices, but being very expensive in other graphs.

It is possible to estimate, approximately, the time that the algorithm needs to run in graphs with larger sizes, through the expression calculated in section II.

For larger graphs the execution time can be approximated by the following expression, where t is the time, in seconds, that the algorithm takes to execute for a graph with 14 vertices and k = 7:

$$ExhTime(n.k) = ((n^2 * C(n,k))/(14^2 * C(14,7))) * t$$

Unlike the previous algorithm, the greedy and randomized algorithms prove to be quite fast and efficient

in solving the problem, with the execution times, with graphs up to 14 vertices and k = 7, in the order of 10^{-3}.

Similarly to the exhaustive search, is possible to approximate the execution times for graphs of larger sizes. For this, a graph with 14 vertices and k = 7 was chosen. For the randomized algorithm, we also chose 10 for m, as it was the value used througout the report.

$$GreedyTime(n.k) = ((n^2 * k)/(14^2 * 7)) * t$$

$$RandomTime(m, n.k) = (m * n * k)/(10 * 14 * 7) * t$$

Through the approximation functions calculated above, it is possible to estimate the time required for the algorithms in relatively larger graphs, as shown in tables 10, 11 and 12.

Exhaustive Search

n	k	Time (s)
20	13	126,31
25	13	13239.01
30	17	439036,84

Table 10 – Exhaustive Search Estimation of Performance for Large Graph Instances

Greedy Search

n	k	Time (s)
300	223	26,33
600	397	187,50
5000	3121	102365,16

Table 11 – Greedy Search Estimation of Performance for Large Graph Instances

Randomized Search

m	n	k	Time (s)
10	300	223	5,59
100	1000	547	457,69
500	5000	3121	65286,22

Table 12 – Random Search Estimation of Performance for Large Graph Instances

VIII. CONCLUSION

To conclude this article, as expected, exhaustive search, also known as brute force, is an algorithm that requires some time to run. It can be very useful for smaller problems or problems with k-sizes that are either very small or very close to the number of vertices, as it is quite simple to implement and allows you to easily achieve a correct solution.

On the other hand, the greedy algorithm is extremely fast, has a small computational cost and can be useful to get an approximate solution quickly. However, this cost does increase with an increase in the number of vertices.

As for the randomized algorithm, it is also extremely fast. It's computational cost is very small too, assuming a small number of m iterations, and is able of achieving good approximations. It is especially useful for larger graph instances, showing the best performance of all the algorithms.

IX. REFERENCES

[1] Warehouse Location Problem https://www.csplib.org/Problems/prob034/

[2] Time Complexity https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

[3] Python Time Complexity https://wiki.python.org/moin/TimeComplexity