COMPARATIVE ANALYSIS OF GDE3 AND NSGA-II ALGORITHMS ON MULTI-OBJECTIVE CAPACITATED FACILITY LOCATION PROBLEM

Kemal Bayik

Abstract

In this report, GDE3 and NSGA-II algorithms are compared by applying them to the Multi Objective Capacitated Facility Location Problem (MOCFLP). In MOCFLP, there are stores and facilities that supply goods to these stores, and these facilities have a certain goods capacity. The purpose of the problem is to determine which facilities will serve which stores in the most optimal way. The parameters tried to be optimized in this report are to minimize the costs of this logistics operation between stores and facilities and the CO2 emissions released during the transportation of goods. The purpose of this report is to compare the performance of GDE3 and NSGA-II algorithms on MOCFLP and find out which algorithm gives better results in terms of hypervolume, runtime and spacing metrics. According to the results obtained, it was determined that the GDE3 algorithm works faster on this problem. However, in most experiments it has been observed that the NSGA-II algorithm achieves more diverse and successful results. The GDE3 algorithm has never been applied to MOCFLP before in the literature, and the findings are important in this respect. At the same time, comparing GDE3 and NSGA-II algorithms together in a complex problem such as MOCFLP provides important information about the performances of the algorithms.

1. Introduction

Nowadays, logistics costs of stores are one of their biggest expense items. It is very important to optimize and minimize these costs. However, considering that these logistics operations are provided by vehicles that use oil, a huge carbon dioxide emission occurs. Minimizing this carbon dioxide emission plays a very important role for the future of the world. The classic capacitated facility location problem addresses this problem. However, a single objective (total cost) is tried to be minimized [7]. The Multi Objective Capacitated Facility Location problem (MOCFLP), discussed in this report, also addresses this problem and attempts to optimize more than one objective.

The aim of this project is to produce a solution to the Multi Objective Capacitated Facility Location problem using GDE3[12] and NSGA-II[13] algorithms and to compare the performance of the algorithms on this problem using the results obtained.

The explanation of MOCFLP is as follows. In the MOCFLP, there are certain amount of stores and facilites that supply products to these stores. Each of these stores has product demands. Facilities also have a limit of stores they can serve, product capacity they can supply and fixed expenses. At the same time, facilities have transportation costs when supplying products to stores and the carbon dioxide values emitted by vehicles during this transportation. The targeted solution to this problem is to determine which facilities will serve which store in order to meet the demands of the stores with minimum cost and minimum carbon dioxide emissions. The constraints of the problem are as follows. Not all facilities have to be used, the demands of all stores must be met and the capacity limit of the facilities must not be exceeded. An example image of the desired supply chain structure is shown in Figure 1.

MOCFLP addresses a current problem today, and finding an optimal solution is important both to reduce the costs of stores and facilities and to cause less harm to the world by reducing carbon dioxide emissions. This study offers an alternative solution to this problem by using the GDE3 algorithm, which has not been applied to this problem before in the literature. At the same time, the NSGA-II algorithm, which has been applied to this problem and many other multi-objective optimization problems in the literature, will be compared with the GDE3 algorithm and the performance of the GDE3 algorithm will be evaluated in this way.

This project focuses on these fundamental questions.

1. How do different evolutionary algorithms such as NSGA-II and GDE3 perform in solving this problem?
2. Which algorithm is more effective based on the results obtained and the metrics used?

3 different success metrics were used in the project and the performances of the algorithms will be evaluated using these metrics. These are hypervolume[15], spacing [11] and runtime.

The rest of the report is organized as follows. In section 2, papers related to this project found in the literature review are introduced. In the third section, GDE3 and NSGA-II algorithms are introduced and the mathematical formulation of MOCFLP is shown. At the same time, the performance metrics used to measure the performance of the algorithms are introduced. In the fourth section, the experiments performed are explained in detail. In section 5, the results obtained from the experiments are shared in different tables. In the sixth section, the results obtained are discussed and the algorithms are compared according to these results.

2. Literature Review

In the literature, there are many papers written to find solutions to MOCFLP. Many different algorithms were used in these papers, and evolutionary algorithms were frequently used to solve this NP-Hard problem. For example, the NSGA-II (Non-dominated Sorting Genetic Algorithm) algorithm [1] [2] [5], which are also used in this report, the SPEA-II algorithm (Strength Pareto Evolutionary Algorithm) [6] and SEAMO2 (Simple Evolutionary Algorithm for Multi-objective Optimization) [3] [8] [9] have been used on this problem in the literature.

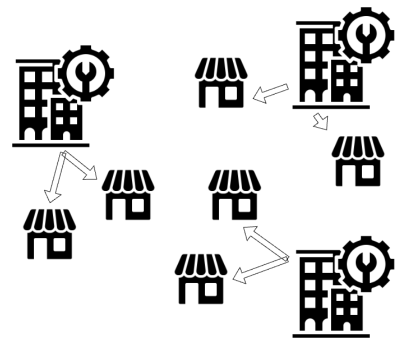

Figure 1: Supply chain structure

However, there are not many papers that include minimizing environmental impact among these objectives. A study in which environmental impact was considered as an additional objective to cost was presented by Harris et al. [4] [1]. In this article, Harris studied the multi-objective uncapacitated facility location problem. She later worked on MOCFLP in her published papers [2] [3].

There is no study in the literature where the GDE3 algorithm is applied to MOCFLP but it has been used on many other large multi-objective optimization problems [10].

3. Method

In this section of the report, the algorithms, the mathematical formulation of MOCFLP and the performance metrics used in the report are introduced.

3.1. Algorithms

In this section of the report, I will introduce the algorithms I used in the project.

3.1.1. DE

Before introducing the GDE3 algorithm, I would like to introduce the Differential Evolution (DE) [14] algorithm because the GDE3 algorithm is an extended version of the DE algorithm.

An evolutionary approach for optimization called Differential Evolution (DE) is very useful for continuous function optimization. Its strength, simplicity of use, and ease of construction set it apart. An initial population of potential solutions is created at random for DE. By using crossover, mutation, and selection techniques, iteratively improves this population. In the process of mutation, two randomly chosen population vectors weighted differences are added to a third vector to generate a new one. A trial vector is created during crossover by combining elements of the mutant and target vectors. In order to ensure that the population progresses towards the optimum, selection entails selecting the better vector (target or trial) based on the objective function value. The pseudo code of the DE algorithm is shown in Algorithm 1.

Algorithm 1 Differential Evolution Algorithm

Initialize population with P random solutions.
Evaluate each individual in the population.
while stopping criterion is not met **do**
  for each target vector x_i, $i = 1, 2, \ldots, P$ **do**
    Randomly select three vectors x_{r1}, x_{r2}, x_{r3} from the population, where $r1 \neq r2 \neq r3 \neq i$.
    Generate a trial vector u_i by combining x_{r1}, x_{r2}, and x_{r3}.
    Perform crossover between x_i and u_i to generate a trial
    Evaluate the trial vector v_i.
    if trial vector v_i is better than target vector x_i **then**
      Replace x_i with v_i in the population.
    end if
  end for
  Update population with the new individuals.
end while
return the best solution found.

While DE is generally used in single objective optimization problems, GDE3 is more commonly used in multi objective optimization problems. Therefore, although mutation and crossover processes have the same logic, the main difference between them is in the selection stage.

3.1.2. GDE3

GDE3 algorithm is an extended and customized version of the DE algorithm for multi objective optimization problems. GDE3 uses the mutation and crossover operators that the DE algorithm also uses. Furthermore, introduces a new mechanism for multi-objective optimization problems. This mechanism is the pareto dominance approach. In the Pareto dominance approach, solutions are evaluated based on their dominance over others in the objective space. GDE3 is widely used in academic research and real-life projects

The GDE3 algorithm works as follows. First, the population is generated randomly. In the second stage, mutation is applied. In the mutation step, for each target vector x_i, three distinct vectors x_{r1}, x_{r2}, x_{r3} are randomly selected from the current population. The mutant vector v_i is then generated as equation (1). Afterwards, crossover is applied. The crossover step combines the target vector x_i and the mutant vector v_i to create the trial vector u_i. For each dimension j, the trial vector is formed as equation (2). In the selection phase, offspring and parents are compared and selected using non-dominated sorting and crowding distance methods. Non-dominated sorting and crowding distance are explained in detail in the NSGA-II algorithm. The mutation method used in this algorithm is differential mutation and the crossover method is binomial crossover. The pseudo code of GDE3 is shown in Algorithm 2 and an example graph of the Pareto front results obtained with the GDE3 algorithm is shown in Figure 2.

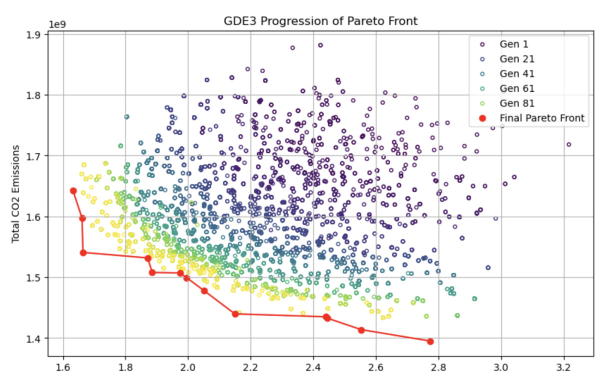

Figure 2: GDE3 Pareto Fronts

$$v_i = x_{r1} + F \cdot (x_{r2} - x_{r3}) \tag{1}$$

- x_{r1}, x_{r2}, x_{r3}: Three distinct vectors randomly selected from the current population
- F: Differential weight, a factor that controls the amplification of the differential variation.
- v_i: Mutation vector.

$$u_{i,j} = \begin{cases} v_{i,j} & \text{if } \mathrm{rand}_j \leq CR \text{ or } j = \mathrm{rand}_i \\ x_{i,j} & \text{otherwise} \end{cases} \tag{2}$$

- rand_j: Uniformly distributed random number between 0 and 1
- CR: Crossover rate
- rand_i: Randomly chosen index that ensures that u_i gets at least one component from v_i

Algorithm 2 GDE3 Algorithm

```
Initialize a random population
Evaluate the fitness of each individual in the population
while termination criteria not met do
    for each individual i in the population do
        Randomly select three distinct individuals x_r1, x_r2, x_r3 from the population
        Compute the mutation vector: v = x_r1 + F · (x_r2 − x_r3)
        Randomly select an index rand_index
        for each gene j of individual i do
            if rand(0, 1) ≤ CR or j = rand_index then
                u_ij = v_j
            else
                u_ij = x_ij
            end if
        end for
        Evaluate the fitness of the offspring
    end for
    Combine the parent and offspring populations
    Apply non-dominated sorting to the combined population
    Calculate crowding distance for each individual
    Select individuals for the next generation based on non-domination rank and crowding distance
end while
```

3.1.3. NSGA-II

NSGA-II (Non-dominated Sorting Genetic Algorithm II) algorithm efficiently addresses issues like computational complexity and maintenance of a diverse set of solutions. Fundamentally, NSGA-II finds Pareto-optimal solutions quickly using a non-dominated sorting strategy, crowding distance calculation, and a unique selection method. Because of these characteristics, NSGA-II is able to efficiently strike a balance between exploring and exploiting the search area, which helps it solve complicated optimization problems with several competing goals. Researchers and practitioners in a variety of domains like NSGA-II because to its computational efficiency and capacity to offer a wide range of optimal solutions.

The NSGA-II algorithm works as follows. First of all, the initial population is generated randomly. In the second stage, fast-non-dominated sorting is applied. At this stage, each individual is compared with each other. If the conditions in the equation (3) are met, that is, if an individual dominates the other individual, that individual is assigned to non-dominated fronts. Then, for individuals in each front, crowding distance is calculated with equation (4). In the selection process, individuals in lower-ranked fronts and individuals with high crowding distance in the same front are selected. Afterwards, crossover and mutation are applied. Simulated binary crossover (SBX) is generally used in the crossover process. In mutation, polynomial mutation is generally preferred. SBX and polynomial mutation were used in this project too. The pseudo code of NSGA-II is shown in Algorithm 3 and an example graph of the Pareto front results obtained with the NSGA-II algorithm is shown in Figure 3.

$$\forall k \in K, f_k(i) \leq f_k(j) \quad \text{and} \quad \exists k \in K, f_k(i) < f_k(j) \tag{3}$$

$$cd(i) = \sum_{k=1}^{K} \left(f_k(i+1) - f_k(i-1)\right) \tag{4}$$

- K: Number of objective functions.
- f_k: *kth* objective function.

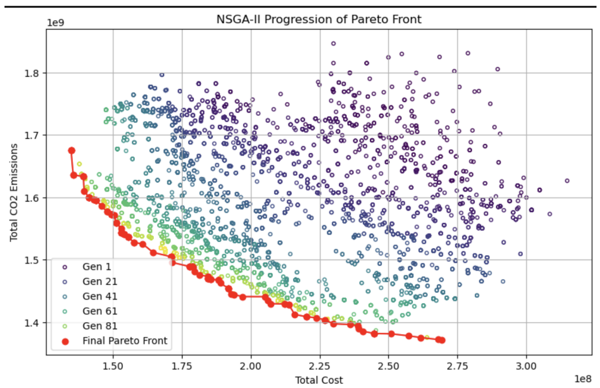

Figure 3: NSGA-II Pareto Fronts

Algorithm 3 NSGA-II Algorithm

```
Initialize population with random solutions
Evaluate the fitness of each individual
while termination criteria not met do
    Apply fast-non-dominated sorting to the population
    for each front F until the parent population is filled do
        Calculate crowding distance for individuals in F
    end for
    Perform selection based on rank and crowding distance
    Apply crossover and mutation to create new offspring
    Combine parent and offspring populations
    Select the best individuals for the next generation
end while
```

3.2. Problem Formulation

Decision Variables:

- x_i: The index of the facility serving store i, $i = 1, \ldots, N$.

Parameters:

- N: Number of stores.
- M: Number of facilities.
- C_i: Fixed cost of facility i, $i = 1, \ldots, M$.
- S_j: Store capacity of facility j, $j = 1, \ldots, M$.
- K_j: Case capacity of facility j, $j = 1, \ldots, M$.
- D_i: Demand of store i, $i = 1, \ldots, N$.
- TC_{ij}: Transportation cost between facility i and store j.
- $CO2_{ij}$: CO2 emissions between facility i and store j.

Objectives:

1. Minimize total cost:

$$\text{Minimize} \sum_{i=1}^{N} TC_{x_i,i} \cdot D_i + \sum_{\text{facility} \in \text{facilities\_used}} C_{\text{facility}}$$

2. Minimize CO2 emissions:

$$\text{Minimize} \sum_{i=1}^{N} CO2_{x_i,i} \cdot D_i$$

Constraints:

1. Store capacity constraint for each facility:

$$\sum_{i=1}^{N} (x_i == j) \leq S_j \quad \forall j = 1, \ldots, M$$

2. Case capacity constraint for each facility:

$$\sum_{i=1}^{N} D_i \cdot (x_i == j) \leq K_j \quad \forall j = 1, \ldots, M$$

3.3. Success Metrics

The algorithms used in the project were evaluated using three different success metrics. In this section of the report, these used success metrics are introduced.

3.3.1. Hypervolume

The area covered by the Pareto fronts generated by the algorithm is measured by hypervolume. Higher hypervolume values show that the algorithm's results are more widely distributed and produce more purposeful solutions.

3.3.2. Spacing

Spacing metric evaluates how uniformly spread the algorithm's generated solution set is. A lower spacing value suggests a more evenly and regular distribution of the solution set.

3.3.3. Runtime

Runtime metric is the comparison of the running times of algorithms on the same parameters.

4. Experimental Setup

In this section, the experiments I conducted to compare GDE3 and NSGA-II algorithms are introduced.

In this project, 4 different experiments and three different data sets were prepared to compare GDE3 and NSGA-II algorithms. Transportation cost, store demands and CO2 emissions values in these data sets were determined randomly among a certain value. The experiments will be carried out as follows.

4.1. Experiment - 1

In the first experiment, the population size was fixed as 100, the crossover rate was fixed as 0.7 and the number of generations was fixed at 100, and the data sets were changed.

4.2. Experiment - 2

In the experiment 2, the first data set was used, the crossover rate was fixed at 0.7, the number of generations was fixed at 100, and the population size was changed. Population size was determined as 100, 200 and 500.

4.3. Experiment - 3

In the third experiment, the first data set was used, the population size was fixed as 100, the number of generations was fixed as 100, and the crossover rate was changed. Crossover rate was determined as 0.3, 0.7 and 0.9.

4.4. Experiment - 4

In the fourth and the last experiment, the first data set was used, the population size was fixed as 100, the crossover rate was fixed as 0.7, and the number of generations was changed. Number of generations was determined as 100, 200 and 500.

In this way, comparisons of GDE3 and NSGA-II algorithms were made in different data sets, different number of generations, different population sizes and different crossover rates. However, during the experiments, each algorithm run 100 times and the values obtained are the average of the values obtained in each iteration.

There is no mutation rate in the GDE3 algorithm. For this reason, no experiments have been conducted to compare mutation rates. In all experiments, the mutation rate value of the NSGA-II algorithm was fixed at 0.7. Likewise, the F value, which is found in the GDE3 algorithm and not in the NSGA-II algorithm, was not included in the experiment and was fixed at 0.7.

The 3 different datasets created are as follows. There are 5 facilities and 30 stores in the first dataset. The fixed costs of these facilities are 10000, their case capacity is 20000 and the maximum number of stores they can serve is 10. The demands of the stores were chosen randomly between 500 and 1000. Transportation costs between the facility and the store were randomly determined between 2000 and 20000, and CO2 emission values were randomly determined between 50000 and 100000. The following changes were made in the second and third datasets. Randomly determined values were determined randomly among the same values. In the second dataset, the number of facilities is 20, the number of stores is 500, and the case capacity of each facility is 25000. The fixed costs of the facilities are randomly selected between 5000 and 10000, and the maximum number of stores they can serve is 30. In the third dataset, the number of facilities is determined as 50, the number of stores is 1000 and the case capacity is 50000. The fixed costs of the facilities are randomly selected between 5000 and 10000, and the maximum number of stores they can serve is 30. The purpose of preparing different datasets in this way is to measure and compare the performance of the algorithms when the problem becomes more complex.

The coding of these experiments was done with the Python programming language. Compiled in JupyterLab and run on Macbook Pro with M1 chip, 32GB ram configuration. The libraries used are shown below.

1. pymoo: Used for implementation of NSGA-II and the MOCLFP
2. pymoode: Used for implementation of GDE3
3. sklearn: Used for MaxMinScaler
4. networkX: Used to visualize the results obtained.
5. numpy
6. matplotlib: Used to visualize the results obtained.
7. time: Used to calculate running times of algorithms

5. Results

In this section, the results obtained from the experiments are shown. Three different tables are introduced. These are the average hypervolume table, average runtime table and average spacing table.

It can be seen from the results that the GDE3 algorithm runs faster in the experiments, the NSGA-II algorithm often provides more diverse and more optimal results and the NSGA-II algorithm generally produces more evenly distributed results. The data presented in this section was analyzed in more detail in the discussions section.

6. Discussion

In this section, the results in the tables presented in the results section are discussed.

Average Hypervolumes					
Algorithm	Dataset	Gen.	Pop. Size	CR	Hypervolume
GDE3	1	100	100	0.7	0.970992
NSGA-II	1	100	100	0.7	0.946984
GDE3	2	100	100	0.7	0.591069
NSGA-II	2	100	100	0.7	0.945674
GDE3	3	100	100	0.7	0.691146
NSGA-II	3	100	100	0.7	0.673394
GDE3	1	100	200	0.7	0.667144
NSGA-II	1	100	200	0.7	0.933447
GDE3	1	100	500	0.7	0.903466
NSGA-II	1	100	500	0.7	0.931701
GDE3	1	100	100	0.3	0.919988
NSGA-II	1	100	100	0.3	1.004374
GDE3	1	100	100	0.9	0.995151
NSGA-II	1	100	100	0.9	1.001298
GDE3	1	200	100	0.7	0.778103
NSGA-II	1	200	100	0.7	0.956063
GDE3	1	500	100	0.7	0.889545
NSGA-II	1	500	100	0.7	0.936189

Table 1: Average hypervolume results obtained in different experiments

Average Runtime (second)					
Algorithms	Dataset	Gen.	Pop. Size	CR	Runtime
GDE3	1	100	100	0.7	0.983829s
NSGA-II	1	100	100	0.7	1.867406s
GDE3	2	100	100	0.7	4.672496s
NSGA-II	2	100	100	0.7	6.911369s
GDE3	3	100	100	0.7	12.578952s
NSGA-II	3	100	100	0.7	17.105656s
GDE3	1	100	200	0.7	1.930498s
NSGA-II	1	100	200	0.7	3.986220s
GDE3	1	100	500	0.7	4.883579s
NSGA-II	1	100	500	0.7	12.416475s
GDE3	1	100	100	0.3	1.022286s
NSGA-II	1	100	100	0.3	1.934537s
GDE3	1	100	100	0.9	0.996106s
NSGA-II	1	100	100	0.9	1.903178s
GDE3	1	200	100	0.7	12.219404s
NSGA-II	1	200	100	0.7	3.851362s
GDE3	1	500	100	0.7	4.907074s
NSGA-II	1	500	100	0.7	9.298624s

Table 2: Average runtime results obtained in different experiments

Average Spacing					
Algorithm	Dataset	Gen.	Pop. Size	CR	Spacing
GDE3	1	100	100	0.7	11385406.76
NSGA-II	1	100	100	0.7	1867296.26
GDE3	2	100	100	0.7	129020064.14
NSGA-II	2	100	100	0.7	18817115.31
GDE3	3	100	100	0.7	40591970.28
NSGA-II	3	100	100	0.7	10545676.01
GDE3	1	100	200	0.7	9158189.63
NSGA-II	1	100	200	0.7	2499265.96
GDE3	1	100	500	0.7	14440172.37
NSGA-II	1	100	500	0.7	481832.84
GDE3	1	100	100	0.3	2258802.00
NSGA-II	1	100	100	0.3	1613473.07
GDE3	1	100	100	0.9	10877439.54
NSGA-II	1	100	100	0.9	4533633.18
GDE3	1	200	100	0.7	18992539.12
NSGA-II	1	200	100	0.7	2327126.58
GDE3	1	500	100	0.7	6848474.98
NSGA-II	1	500	100	0.7	1289247.79

Table 3: Average spacing results obtained in different experiments

6.1. Comparison of algorithms according to average hypervolumes

When looking at the hypervolume values, it is seen that the GDE3 algorithm is more successful than the NSGA-II algorithm in the conditions where the crossover rate, number of generations and population size parameters are in the initial state in the first dataset. In the second dataset it can be seen that there is a huge decrease in the hypervolume value of the GDE3 algorithm. In the third dataset, it seems that the hypervolume values are very close to each other and the GDE3 algorithm is more successful with a small difference.

In the experiment where the population size changed, there was a significant decrease in the hypervolume value of the GDE3 algorithm when the population size was 200. In the case where the population size is 500, it seems that the hypervolume value has increased, but it is still worse than the NSGA-II algorithm. Looking at these results, it can be said that the NSGA-II algorithm is more successful at higher population size values.

In the experiment where the crossover rate changed, it is possible to say that the NSGA-II algorithm gave a more successful result compared to the GDE3 algorithm when the crossover rate was 0.3. When the crossover rate is 0.9, we see that the results are almost the same, but the NSGA-II algorithm is more successful by a very small margin. However, when the crossover rate is 0.7, it seems that the GDE3 algorithm is more successful than the NSGA-II algorithm. Although it is difficult to make a general inference with these results, it is possible to say that the NSGA-II algorithm is more successful at low crossover rate values.

In the experiment where the number of generations was changed, it was seen that the NSGA-II algorithm performed better than the GDE3 algorithm at higher generation number values. Likewise, it is seen that the GDE3 algorithm gives better performance when the number of generations is 100, and it can be concluded that the GDE3 algorithm gives better performance than the NSGA-II algorithm at low generation number values.

6.2. Comparison of algorithms according to average runtimes

When a comparison is made according to the running times of the algorithms, it is observed that the GDE3 algorithm gives better performance in almost every experiment. Especially in the experiment where the population size is variable, the difference between working times increases more obviously as the population size increases. I think this may make the GDE3 algorithm preferred in more complex problems.

It seems that the running time of the GDE3 algorithm increases significantly only when the number of generations is 200. Since this is very inconsistent compared to all other runtime results, I think this may be due to a system bug.

6.3. Comparison of algorithms according to average spacing

When the algorithms are compared according to spacing values, it is seen that the spacing values of the NSGA-II algorithm are less in each experiment. Accordingly, it can be said that the Pareto fronts of the NSGA-II algorithm are in a more uniform distribution. At the same time, it can be said that the NSGA-II algorithm explores the solution space better.

Considering the results in general, I think that the results obtained by the NSGA-II algorithm are more successful than the results obtained by the GDE3 algorithm. The fact that the spacing values of the NSGA-II results are much lower than the spacing values of the GDE3 algorithm clearly shows that NSGA-II explores the solution space much better than GDE3. At the same time, this finding is clearly visible when looking at Figure 2 and Figure 3. However, the fact that the hypervolume values of NSGA-II are bigger than the hypervolume values of GDE3 in many cases shows that the results of NSGA-II spread over a wider area and produce more optimal results.

However, I think it is a great advantage that the GDE3 algorithm runs approximately 1.5 times faster. Considering that the maximum number of facilities was 1000 and the maximum number of stores was 50 in my experiments, when this problem is adapted to real cities, there will be much larger numbers of facilities and stores and the problem will become much more complex. In this case, the speed of the GDE3 algorithm can be a great advantage.

In conclusion, I think that the comparison presented in this paper provides important information about the performance of the algorithms. The part that I think could be better is that I could compare these algorithms with more performance metrics, but most of the performance metrics used in the literature, except the ones I use, compare the optimal solution with the solutions of the algorithms. Since the data sets I prepared were created randomly and it was almost impossible to find the most optimal solution manually, I could not use performance metrics that worked in this way.

References

[1] Tang, Xifeng, and Ji Zhang. ”The multi-objective capacitated facility location problem for green logistics.” 2015 4th International Conference on Advanced Logistics and Transport (ICALT). IEEE, 2015.

[2] Irina Harris, Christine L. Mumford, and Mohamed M. Naim. 2011. An evolutionary bi- objective approach to the capacitated facility location problem with cost and CO2 emis- sions. In Proceedings of the 13th annual conference on Genetic and evolutionary computa- tion (GECCO ’11). Association for Computing Machinery, New York, NY, USA, 697–704.

[3] Harris, Irina, Christine L. Mumford, and Mohamed M. Naim. ”A hybrid multi-objective approach to capacitated facility location with flexible store allocation for green logistics modeling.” Transportation Research Part E: Logistics and Transportation Review 66 (2014): 1-22.

[4] Harris, Irina, Christine Mumford, and Mohamed Naim. ”The multi-objective uncapacitated facility location problem for green logistics.” 2009 IEEE Congress on Evolutionary Computation. IEEE, 2009.

[5] Bhattacharya, Ranjan, and Susmita Bandyopadhyay. ”Solving conflicting bi-objective facility location problem by NSGA II evolutionary algorithm.” The International Journal of Advanced Manufacturing Technology 51 (2010): 397-414.

[6] Silav, Ahmet. Bi-objective facility location problems in the presence of partial coverage. MS thesis. Middle East Technical University, 2009.

[7] Klose, A., Drexl, A., 2005. Facility location models for distribution system design. Eur. J. Oper. Res. 162 (1), 4–29.

[8] Mumford, Christine L. ”Simple population replacement strategies for a steady-state multi-objective evolutionary algorithm.” Genetic and Evolutionary Computation Conference. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[9] Valenzuela, Christine L. ”A simple evolutionary algorithm for multi-objective optimization (SEAMO).” Proceedings of the 2002 Congress on Evolutionary Computation. CEC’02 (Cat. No. 02TH8600). Vol. 1. IEEE, 2002.

[10] L. M. Antonio and C. A. C. Coello, ”Use of cooperative coevolution for solving large scale

multiobjective optimization problems," 2013 IEEE Congress on Evolutionary Computation, Cancun, Mexico, 2013, pp. 2758-2765, doi: 10.1109/CEC.2013.6557903.

[11] J. Schott, Fault Tolerant Design Using single and Multicriteria Genetic Algorithm Optimization, Master thesis, Department of Aeronautics and Astronautics, Institute of Technology, Cam- bridge, Massachusetts ,1995.

[12] Kukkonen, Saku, and Jouni Lampinen. "GDE3: The third evolution step of generalized differential evolution." 2005 IEEE congress on evolutionary computation. Vol. 1. IEEE, 2005.

[13] Deb, Kalyanmoy, et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." IEEE transactions on evolutionary computation 6.2 (2002): 182-197.

[14] Storn, Rainer, and Kenneth Price. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces." Journal of global optimization 11 (1997): 341-359.

[15] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," IEEE Transactions on Evolutionary Computation, vol. 3, no. 4, pp. 257–271, November 1999.

Appendix A. Project Codes

Dataset.py

```
class Dataset:

  def __init__(self, number_of_facilities, number_of_stores, fixed_costs_for_facilities, number_of_cases_capacity, number_of_stores_capacity, store_demands, transportation_cost, total_transport_co2_emissions):
    """
          Constructor of Dataset class
        :param number_of_facilities: Number of facilities
        :param number_of_stores: Number of stores
        :param fixed_costs_for_facilities: Fixed costs of facilities
        :param number_of_cases_capacity: Case capacities of facilities
        :param number_of_stores_capacity: Store capacities of facilities
        :param store_demands: Demands of stores
        :param transportation_cost: Transportation costs
        :param total_transport_co2_emissions: Total CO2 emissions of transportations
    """
    self.number_of_facilities = number_of_facilities
    self.number_of_stores = number_of_stores
    self.fixed_costs_for_facilities = fixed_costs_for_facilities
    self.number_of_cases_capacity = number_of_cases_capacity
    self.number_of_stores_capacity = number_of_stores_capacity
    self.store_demands = store_demands
    self.transportation_cost = transportation_cost # Transportation cost to satisfy all demand from each facility to each store
    self.total_transport_co2_emissions = total_transport_co2_emissions  # Total transport CO2 emissions to satisfy all demand from each facility to each store
```

MOCFLP.py

```
import numpy as np
from pymoo.core.problem import Problem
from pymoo.core.problem import ElementwiseProblem
class MOCFLP(ElementwiseProblem):
    def __init__(self, number_of_facilities, number_of_stores, fixed_costs_for_facilities, number_of_stores_capacity, number_of_cases_capacity, store_demands, transportation_cost, total_transport_co2_emissions):
        """
            Constructor of MOCFLP problem implementation class
        :param number_of_facilities: Number of facilities
        :param number_of_stores: Number of stores
        :param fixed_costs_for_facilities:
```

```
Fixed costs of facilities
        :param number_of_cases_capacity: Case capacities of facilities
        :param number_of_stores_capacity: Store capacities of facilities
        :param store_demands: Demands of stores
        :param transportation_cost: Transportation costs
        :param total_transport_co2_emissions: Total CO2 emissions of transportations
        """
        super().__init__(n_var=number_of_stores,
                         n_obj=2,
                         n_constr=2 * number_of_facilities,
                         xl=0,
                         xu=number_of_facilities - 1)
        self.number_of_facilities = number_of_facilities
        self.number_of_stores = number_of_stores
        self.fixed_costs_for_facilities = fixed_costs_for_facilities
        self.transportation_cost = transportation_cost
        self.total_transport_co2_emissions = total_transport_co2_emissions
        self.number_of_stores_capacity = number_of_stores_capacity
        self.number_of_cases_capacity = number_of_cases_capacity
        self.store_demands = store_demands

    def _evaluate(self, x, out, *args, **kwargs):
        x = np.round(x).astype(int)
        total_cost = 0
        total_co2 = 0
        facilities_used = set(x)

        #Evaluating transportation cost and co2 emissions
        for i in range(self.number_of_stores):
            facility_idx = x[i]
            total_cost += self.transportation_cost[i][facility_idx] * self.store_demands[i]
            total_co2 += self.total_transport_co2_emissions[i][facility_idx] * self.store_demands[i]

        #Adding used facilities fixed costs to find the total cost
        for facility in facilities_used:
            total_cost += self.fixed_costs_for_facilities[facility]

        G_store = np.array([np.sum(x == j) - self.number_of_stores_capacity[j] for j in range(self.number_of_facilities)])
        G_case = np.array([np.sum(self.store_demands[x == j]) - self.number_of_cases_capacity[j] for j in range(self.number_of_facilities)])

        out["F"] = np.array([total_cost, total_co2])
        out["G"] = np.concatenate([G_store, G_case])
```

Algorithms.py

```
from pymoo.core.problem import Problem
from pymoo.core.problem import ElementwiseProblem
import matplotlib.pyplot as plt
from pymoo.core.problem import Problem
from pymoo.algorithms.moo.nsga2 import NSGA2
from pymoo.operators.crossover.pcx import PCX
from pymoo.operators.mutation.pm import PM
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.optimize import minimize
from pymoode.gde3 import GDE3
from pymoo.factory import get_performance_indicator
from sklearn.preprocessing import MinMaxScaler
import time
import numpy as np

class Algorithms:

    def __init__(self):
        pass

    def nsga2_algorithm(self, problem, number_of_generations, population_size, crossover_rate, show_plot):

        """
            NSGA-II algorithm implementation
        :param problem: MOCFLP class object
        :param number_of_generations: Number of generations
        :param population_size: Population size
        :param crossover_rate: Crossover rate
        :param show_plot: Prints pareto
```

```
front plot to the screen if True
        :return: Sorted pareto front results
        :return: Best facility indices
        :return: Runtime of the algorithm
        """
        # Starting time to find the runtime

        start_time = time.time()

        # Algorithm setup
        algorithm = NSGA2(
            pop_size=population_size,
            sampling=FloatRandomSampling(),
            crossover=PCX(prob=crossover_rate),
            mutation=PM(eta=0.7),
            eliminate_duplicates=True
        )

        # Solve the problem
        res = minimize(
            problem,
            algorithm,
            ('n_gen', number_of_generations),  # Termination criterion
            seed=1,
            save_history=True,
            verbose=False
        )

        best_solution = res.X[np.argmin(res.F[:, 0])]
        best_facility_indices = np.round(best_solution).astype(int)

        # Finding open facilities
        open_facilities = np.unique(best_facility_indices)

        # print("Open Facilities:", open_facilities)
        # for customer in range(problem.number_of_stores):
        #    print(f"Customer {customer} is served by Facility {best_facility_indices[customer]}")

        # Sorting pareto front
        sorted_final_front_nsga2 = res.F[np.argsort(res.F[:, 0])]

        # If show_plot is True prints pareto front plot
        if show_plot:

            plt.figure(figsize=(10, 6))
            colors = plt.cm.viridis(np.linspace(0, 1, len(res.history)))

            for i, entry in enumerate(res.history):
                plt.scatter(entry.pop.get("F")[:, 0], entry.pop.get("F")[:, 1], s=12, facecolors='none',
                            edgecolors=colors[i], label=f'Gen {i + 1}' if i % 20 == 0 else "")
            # Pareto front
            plt.scatter(sorted_final_front_nsga2[:, 0], sorted_final_front_nsga2[:, 1], c='red', label='Final Pareto Front')

            plt.plot(sorted_final_front_nsga2[:, 0], sorted_final_front_nsga2[:, 1], c='red')

            plt.title('NSGA-II Progression of Pareto Front')
            plt.xlabel('Total Cost')
            plt.ylabel('Total CO2 Emissions')
            plt.grid(True)
            plt.legend()
            plt.show()

        # Finding the runtime
        runtime = time.time() - start_time

        return sorted_final_front_nsga2, best_facility_indices, runtime


    def gde3_algorithm(self, problem, number_of_generations, population_size, crossover_rate, show_plot):
        """
            GDE3 algorithm implementation
        :param problem: MOCFLP class object
        :param number_of_generations: Number of generations
        :param population_size: Population size
        :param crossover_rate: Crossover rate
        :param show_plot: Prints pareto front plot to the screen if True
        :return: Sorted pareto front results
        :return: Best facility indices
        :return: Runtime of the algorithm
        """

        # Starting time to find the runtime
        start_time = time.time()
```

```python

    # Algorithm setup
    gde3 = GDE3(pop_size=population_size, variant="DE/rand/1/bin", CR=crossover_rate, F=0.7)

    res = minimize(problem, gde3, ('n_gen', number_of_generations), save_history=True, verbose=False)

    best_solution = res.X[np.argmin(res.F[:, 0])]
    best_facility_indices = np.round(best_solution).astype(int)

    # Finding open facilities
    open_facilities = np.unique(best_facility_indices)

    # Sorting pareto front
    sorted_final_front_gde3 = res.F[np.argsort(res.F[:, 0])]

    if show_plot:

        # Plotting the Pareto front progression
        plt.figure(figsize=(10, 6))
        colors = plt.cm.viridis(np.linspace(0, 1, len(res.history)))

        for i, entry in enumerate(res.history):
            plt.scatter(entry.pop.get("F")[:, 0], entry.pop.get("F")[:, 1], s=12, facecolors='none', edgecolors=colors[i],
                        label=f'Gen {i + 1}' if i % 20 == 0 else "")
        # Pareto front
        plt.scatter(sorted_final_front_gde3[:, 0], sorted_final_front_gde3[:, 1], c='red', label='Final Pareto Front')

        plt.plot(sorted_final_front_gde3[:, 0], sorted_final_front_gde3[:, 1], c='red')

        plt.title('GDE3 Progression of Pareto Front')
        plt.xlabel('Total Cost')
        plt.ylabel('Total CO2 Emissions')
        plt.grid(True)
        plt.legend()
        plt.show()

    # Finding the runtime
    runtime = time.time() - start_time

    return sorted_final_front_gde3, best_facility_indices, runtime


def calculate_hypervolume(self, pareto_front_gde3, pareto_front_nsga2):
    """
        Calculates hypervolume of GDE3's and NSGA-II's pareto front results
    :param pareto_front_gde3: Pareto front results of GDE3
    :param pareto_front_nsga2: Pareto front results of NSGA-II
    :return: Hypervolume of GDE3's pareto front results
    :return: Hypervolume of NSGA-II's pareto front results
    """


    reference_point = np.array([1.1, 1.1])  # Reference point is 1.1 because I scale the data
    hv = get_performance_indicator("hv", ref_point=reference_point)

    scaler = MinMaxScaler()
    scaled_pareto_front_gde3 = scaler.fit_transform(pareto_front_gde3)
    scaled_pareto_front_nsga2 = scaler.fit_transform(pareto_front_nsga2)

    hv_gde3 = hv.do(scaled_pareto_front_gde3)
    hv_nsga2 = hv.do(scaled_pareto_front_nsga2)

    return hv_gde3, hv_nsga2


def spacing(self, pareto_front):
    """
        Calculates spacing value of given pareto front result
    :param pareto_front: Pareto front result
    :return: Spacing value of given pareto front result
    """
    n = len(pareto_front)

    if n < 2:
        return 0

```

```python
        distances = np.zeros(n)
        for i in range(n):
            min_dist = np.inf
            for j in range(n):
                if i != j:
                    dist = np.linalg.norm(pareto_front[i] - pareto_front[j])
                    min_dist = min(min_dist, dist)
            distances[i] = min_dist

        mean_distance = np.mean(distances)
        spacing_value = np.sqrt(np.sum((distances - mean_distance) ** 2) / n)
        return spacing_value


    def experiment(self, number_of_facilities, number_of_stores, fixed_costs_for_facilities,
                   number_of_cases_capacity, number_of_stores_capacity, demand_for_stores,
                   transportation_cost, total_transport_co2_emissions, num_iterations,
                   number_of_generations, population_size, crossover_rate, show_plot):
        """
            Runs the experiment according to given parameters
        :param number_of_facilities: Number of facilities
        :param number_of_stores: Number of stores
        :param fixed_costs_for_facilities: Fixed costs of facilities
        :param number_of_cases_capacity: Case capacities of facilities
        :param number_of_stores_capacity: Store capacities of facilities
        :param demand_for_stores: Demands of stores
        :param transportation_cost: Transportation costs
        :param total_transport_co2_emissions: Total CO2 emissions of transportations
        :param num_iterations: Number of iterations
        :param number_of_generations: Number of generations
        :param population_size: Population size
        :param crossover_rate: Crossover rate
        :param show_plot: Prints pareto front plot to the screen if True
        """

        dataset = Dataset(number_of_facilities, number_of_stores, fixed_costs_for_facilities,
                          number_of_cases_capacity, number_of_stores_capacity, demand_for_stores,
                          transportation_cost, total_transport_co2_emissions)

        problem = MOCFLP(dataset.number_of_facilities, dataset.number_of_stores, dataset.fixed_costs_for_facilities,
                         dataset.number_of_stores_capacity, dataset.number_of_cases_capacity, dataset.store_demands,
                         dataset.transportation_cost, dataset.total_transport_co2_emissions)

        runtimes_gde3 = []
        runtimes_nsga2 = []
        hypervolumes_gde3 = []
        hypervolumes_nsga2 = []
        spacing_values_gde3 = []
        spacing_values_nsga2 = []
        coverage_values_gde3 = []
        coverage_values_nsga2 = []

        for i in range(num_iterations):
            print("Iteration : ", str(i))

            gde3_pareto_fronts, gde3_best_facility_indices, gde3_runtime = self.gde3_algorithm(problem,
                                                                        number_of_generations,
                                                                        population_size,
                                                                        crossover_rate,
                                                                        show_plot)
            nsga2_pareto_fronts, nsga2_best_facility_indices, nsga2_runtime = self.nsga2_algorithm(problem,
```

```
                                                                number_of_generations
        ,
                                                                population_size,
                                                                crossover_rate,
                                                                show_plot)

            runtimes_gde3.append(
gde3_runtime)
            runtimes_nsga2.append(
nsga2_runtime)

            hypervolume_gde3,
hypervolume_nsga2 = self.
calculate_hypervolume(gde3_pareto_fronts,
 nsga2_pareto_fronts)

            hypervolumes_gde3.append(
hypervolume_gde3)
            hypervolumes_nsga2.append(
hypervolume_nsga2)

            spacing_gde3 = self.spacing(
gde3_pareto_fronts)
            spacing_nsga2 = self.spacing(
nsga2_pareto_fronts)

            spacing_values_gde3.append(
spacing_gde3)
            spacing_values_nsga2.append(
spacing_nsga2)

        print("GDE3 Average Runtime ", str(
sum(runtimes_gde3) / num_iterations))
        print("NSGA-II Average Runtime ",
str(sum(runtimes_nsga2) / num_iterations)
)

        print("GDE3 Average Hypervolume ",
str(sum(hypervolumes_gde3) /
num_iterations))
        print("NSGA-II Average Hypervolume "
, str(sum(hypervolumes_nsga2) /
num_iterations))

        print("GDE3 Average Spacing ", str(
sum(spacing_values_gde3) / num_iterations
))
        print("NSGA-II Average Spacing ",
str(sum(spacing_values_nsga2) /
    num_iterations))
```

Example run of experiment - 2

```
number_of_facilities = 20
number_of_stores = 500
fixed_costs_for_warehouses = np.empty(
    number_of_facilities)
fixed_costs_for_warehouses = random.randint(
    low = 5000, high = 10000, size=(
    number_of_stores))
number_of_cases_capacity = np.empty(
    number_of_facilities)
number_of_cases_capacity.fill(25000)
number_of_stores_capacity = np.empty(
    number_of_facilities)
number_of_stores_capacity.fill(30)
demand_for_stores = random.randint(low =
    500, high = 1000, size=(number_of_stores)
    )
transportation_cost = []

for i in range(number_of_stores):
  cost = random.randint(low = 2000, high =
    20000, size=(number_of_facilities))
  transportation_cost.append(cost.tolist())

total_transport_co2_emissions = []

for i in range(number_of_stores):
  transport_co2_emissions = random.randint(
    low = 50000, high = 100000, size=(
    number_of_facilities))
  total_transport_co2_emissions.append(
    transport_co2_emissions.tolist())

algorithms = Algorithms()

num_iterations, number_of_generations,
    population_size, crossover_rate,
    show_plot = 100, 100, 100, 0.7, False

result_2_100_100_100_07 = algorithms.
    experiment(number_of_facilities,
    number_of_stores,
    fixed_costs_for_warehouses,
                  number_of_cases_capacity,
    number_of_stores_capacity,
    demand_for_stores,
                  transportation_cost,
    total_transport_co2_emissions,
    num_iterations,
                  number_of_generations,
    population_size, crossover_rate,
    show_plot)
```