MS-RCPSP

Optimization using Ant colony and NSGA-II

JUNE 24, 2024

UNIVERSITY OF SCIENCE AND TECHNOLOGY
Kamil Krawiec 260330, Maciej Sieroń 261704

Github link to project

Table of Contents

Introduction

In recent years, addressing complex scheduling challenges has become essential for optimizing project management across various industries. We tackled the Multi-Skill Resource-Constrained Project Scheduling Problem (MS-RCPSP), a complex extension of the classical RCPSP, which adds the dimension of matching diverse skill sets to tasks along with precedence and resource constraints. Inspired by the hybrid ant colony optimization approach described in Myszkowski[1], our goal was to develop a comprehensive solution that not only balances cost and duration but also integrates multi-objective optimization. We implemented two advanced algorithms: NSGA-II to address multiple objectives and Ant Colony Optimization to improve task scheduling through intelligent pheromone-based updates.

RCPSP definition

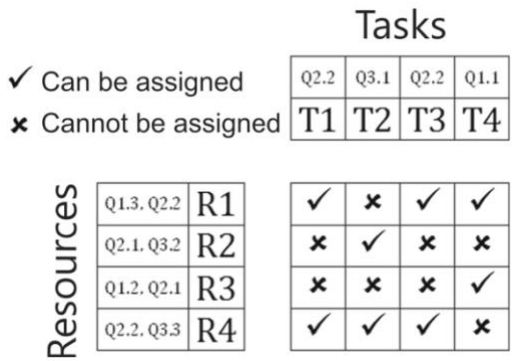
Resource Constrained Project Scheduling Problem is one of scheduling problems. It aims to find such resource-to-task assignments, which makes project plan the cheapest or the shortest.

Classical RCPSP

In RCPSP, you have a bunch of tasks with how long they take, when they start, and when they finish. These tasks can't be stopped once they start. Some tasks need to be done before others can start. The ones that have to be done first are called "predecessors".
In the classic version, you also have limited resources. Each resource has a certain number of units, and tasks need some of these units to get done. You can't assign more units to tasks than the resource has. Also, each task can only be assigned to one resource at a time.
The goal is to figure out the best way to assign tasks to resources so that the schedule works (tasks are done in the right order and resources aren't overused) and is as short as possible.

Multi Skill RCPSP

MS-RCPSP extension adds a skill requirement to the classic RCPSP. Now, each task in a project needs specific skills at certain familiarity levels to be done. Similarly, each resource (like a person or a tool) has a set of skills, each with a familiarity level. Task can be assigned to specific resource only if it has required familiarity level. Below there is an example of skills matrix.



[1] Hybrid ant colony optimization in solving multi-skill resource-constrained project scheduling problem

Example instance

```
============================================================
General characteristics:
Tasks: 10
Resources: 3
Precedence relations: 4
Number of skill types: 3
============================================================
ResourceID      Salary   Skills
1                        56.0            Q1: 0    Q2: 1
2                        53.6            Q2: 2    Q0: 1
3                        28.9            Q0: 1    Q1: 0
============================================================
TaskID   Duration        Skill     Predecessor IDs
1                        37        Q2: 1
2                        36        Q2: 2
3                        21        Q0: 1
4                        23        Q1: 0
5                        36        Q0: 1
6                        13        Q2: 1
7                        13        Q1: 0           4        5
8                        37        Q0: 1
9                        36        Q2: 1           7
10                       19        Q1: 0           3
============================================================
```

This is example of instance for MS-RCPSP problem in format, which our reader can work with.
Instances mentioned in introduction, which are used for experiments, are in this format. They also have additional information in the header, but they are not necessary for experiments, so they are not included here.

Algorithms

Ant Colony

Ant colony is general metaheuristic, which may be adapted to many different optimization problems, for example Travelling Salesman Problem. This section describes version of algorithm, which is already adapted to MS-RCPSP problem.
Ant Colony algorithm is inspired by real behaviour of ants colonies. More specifically it is using ant's way of finding the food, which is strongly connected with pheromones. In simple words, when ant finds the food, it leaves pheromone trail while returning to the rest. Shorter paths accumulate more pheromones over time as more ants travel back and forth. This stronger pheromone trail attracts more ants, leading them to the food source efficiently.
Algorithm aims to construct solutions with virtual ants, which leave pheromone as in real life. There is pheromones matrix, where for every pair (task, resource) we keep level of pheromone. Initially it is 1.0 for every pair and it is being updated when new solutions are being created basing on their fitness level.
Virtual ants are constructing solutions basing on pheromone levels and some heuristic. Influence of these values is controlled using parameters alpha and beta, which determine to which power a given value should be raised when deciding on the next task. Algorithm calculates probabilities basing on these values and selects next task using them.
Ants are generating new solutions iteratively, all the time updating pheromone levels. After each iteration pheromone level is also decreased basing on evaporation rate, which range is [0,1]. For every pair it is multiplied by (1-evaporation_rate). General approach of algorithm is presented in the run() method below.
Number of ants is also a parameter of algorithm and it determines how many solutions are constructed in every iteration.

```
def run(self, version):
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(self.num_iterations):
        solutions = self.construct_solutions(version)
        self.update_pheromones(solutions, version)
        for solution in solutions:
```

Two different versions of construct_solution() are implemented for this project. First one is selecting next tasks basing on probabilities calculated as mentioned before and then selecting suitable resource with lowest availability for this task. This version is very good in terms of duration, but not in terms of cost. Second approach is good in terms of cost, but not duration.
It is selecting next task and resource at once. It also skips part with probabilities and just take task and resource with best score, which is calculated as product of heuristic and pheromone level. It also includes alpha and beta factors.

Multi Dimensional – Non-dominated Sorting Genetic Algorithm

The foundation of many evolutionary algorithms, including NSGA-II, is the creation of an initial population or set of candidate solutions. To enhance diversity and ensure a broad exploration of the solution space, a random solution generator is often employed. This generator serves as a crucial step in forming the initial population for our optimization algorithms, enabling effective exploration and exploitation of potential schedules.

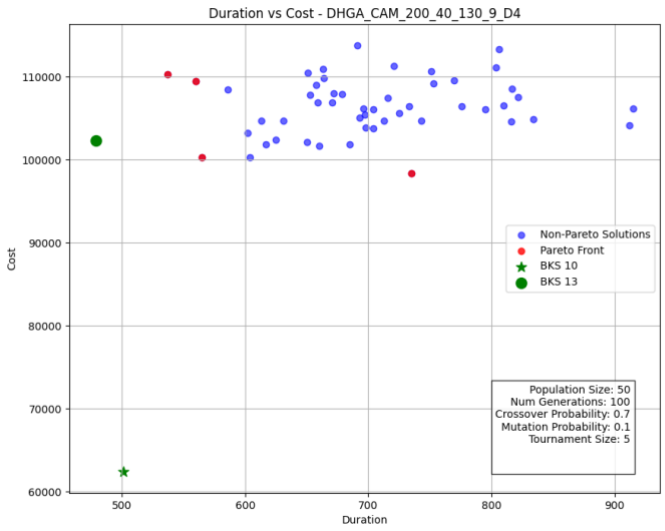
The goal of our random solution generator is to produce feasible initial schedules by introducing randomness in task sequencing and resource allocation. This approach aims to avoid premature convergence and provide a varied starting point for subsequent optimization processes. By shuffling tasks and resources, the generator increases the likelihood of finding diverse initial solutions, which is essential for robust performance in multi-objective optimization.

To compare our results, we are using best known solutions found on web.
Best known solutions from this[2] article are noted as *BKS 10*, and from article[3] – *BKS 13.*

```
def random_solution(self):
    create new solution

    shuffle tasks to introduce randomness

    create dictionary to store the end times for tasks

    create dictionary to store resource availability times

    while tasks:
        task = tasks.pop(0)

        if there are no predescessors:
            earliest_start = 0
        else:
            check if all predecessors have been scheduled
            if predesscors have not been scheduled:
                Push the current task back at index 0,schedule predecessors first

                continue - Skip scheduling the current task now

            Calculate the earliest start time considering predecessors


        find all resources that can perform the task

        shuffle the valid resources to introduce randomness


        if selected_resource is not None:
            Assign the task to the selected resource at the earliest available time

            Update the end time for the task and the availability time for the resource


    sort schedule by hour
    return solution
```

Random pseudocode

[2] Myszkowski P.B., Olech L.P., Laszczyk M. and Skowroński M.E. "Hybrid Differential Evolution and Greedy (DEGR) for Solving Multi–Skill Resource–Constrained Project Scheduling Problem"
[3] Myszkowski P.B., Siemienski J.J., "GRASP applied to Multi-Skill Resource-Constrained Project Scheduling Problem"



Generated population with random initialization

The *MultiobjectiveOptimizer* extends the *Optimizer* class and is tailored for multiobjective optimization. It leverages different mutation and crossover methods to evolve a population of solutions towards Pareto optimality.

Optimizer class holds important parameters and abstract functions:

```
class Optimizer(ABC):
    POPULATION_SIZE = 50
    NUM_GENERATIONS = 100
    CROSSOVER_PROBABILITY = 0.7
    MUTATION_PROBABILITY = 0.1
    TOURNAMENT_SIZE = 5
    OTHER_MUTATION_PROBABILITY = 0.3
```

POPULATION_SIZE: Defines the size of the population used in the optimization algorithm.
NUM_GENERATIONS: Specifies the number of generations for which the algorithm will run.
CROSSOVER_PROBABILITY: The probability of performing crossover between two parents during reproduction.
MUTATION_PROBABILITY: The probability of applying a mutation to an offspring.
TOURNAMENT_SIZE: The number of solutions participating in the tournament selection process for choosing parents.
OTHER_MUTATION_PROBABILITY: The probability of selecting an alternative mutation method.

- CAM-> CHEAPEST
- SWAP->CAM
- CHEAPEST->CAM
- CPO->SWAP

MultiobjectiveOptimizer:

Main Methods

1. **optimize()**

- **Description:** Main method that orchestrates the optimization process over multiple generations. It evaluates the population, performs non-dominated sorting, and iteratively creates new generations through selection, crossover, and mutation.

- **Key Steps:**

    1. Evaluate the initial population.
    2. Perform non-dominated sorting to organize the population into Pareto fronts.
    3. Create a new population using selection, crossover, and mutation.
    4. Update the population and repeat for the defined number of generations.

2. **non_dominated_sort()**

    - **Description:** Sorts the population into different Pareto fronts based on their rank and crowding distance.

    - **Key Steps:**

        1. Identify non-dominated solutions (those that are not outperformed in all objectives by any other solution).
        2. Assign ranks to these solutions and group them into fronts.
        3. Calculate crowding distances within each front to preserve diversity.

3. **calculate_crowding_distance(pareto_front)**

    - **Description:** Calculates the crowding distance for each solution in a Pareto front. This helps in maintaining diversity by measuring how crowded the neighborhood of each solution is.

Mutation Methods

These methods introduce variations in individual solutions to explore the solution space and avoid local optima.

1. **mutationCAM(solution)**[4]

    - **Description:** Implements Conflict Avoidance Mutation (CAM). This method attempts to reassign tasks to avoid conflicts based on resource availability and capability.

    - **Key Steps:**

[4] SPECIALIZED GENETIC OPERATORS
FOR MULTI–SKILL RESOURCE–CONSTRAINED PROJECT SCHEDULING PROBLEM

1. Identify conflicts in the task schedule.
2. Reassign tasks to different resources to minimize conflicts.

2. **mutationSWAP(solution)**
    - **Description:** Randomly swaps tasks between resources to explore new potential schedules.
    - **Key Steps:**
        1. Randomly select tasks (10% of all genes).
        2. Swap these tasks between different resources.
3. **mutationCHEAPEST(solution)**
    - **Description:** Reassigns tasks to the least expensive resources to minimize the cost.
    - **Key Steps:**
        1. Identify capable resources.
        2. Select the cheapest among them and reassign tasks accordingly.
4. **mutationCPO(solution)**
    - **Description:** Focuses on optimizing the schedule based on critical path duration.
    - **Key Steps:**
        1. Identify critical tasks in the current schedule.
        2. Attempt to reassign these tasks to reduce the overall duration.

Crossover Methods

1. **DHGAcrossover(parent1, parent2)**[5]
    - **Description:** Implements a crossover inspired by the DHGA approach. Combines parts of the schedules of two parents by splitting at a randomly chosen hour range and merging the parts.
    - **Key Steps:**
        1. Select a start and end hour for crossover.
        2. Combine schedules from both parents based on these hours.

[5] A New Efficient Genetic Algorithm for Project Scheduling under Resource Constraints

2. **PMXCrossover(parent1, parent2)**

    - **Description:** Partially Matched Crossover (PMX) swaps segments between two parents based on randomly selected crossover points.

    - **Key Steps:**

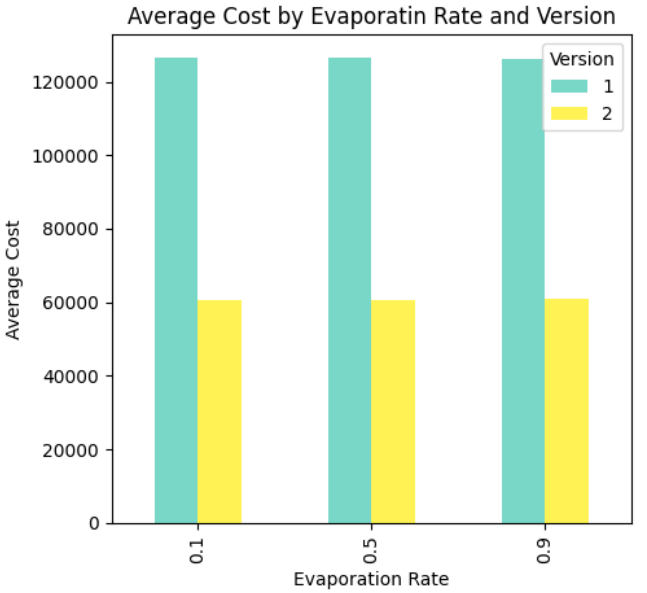
        1. Select two crossover points.

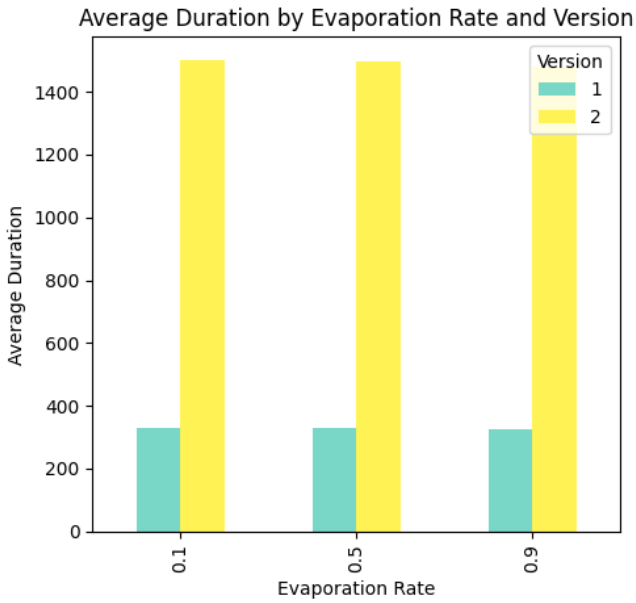
        2. Create a child by combining segments from both parents between these points.

Experiments

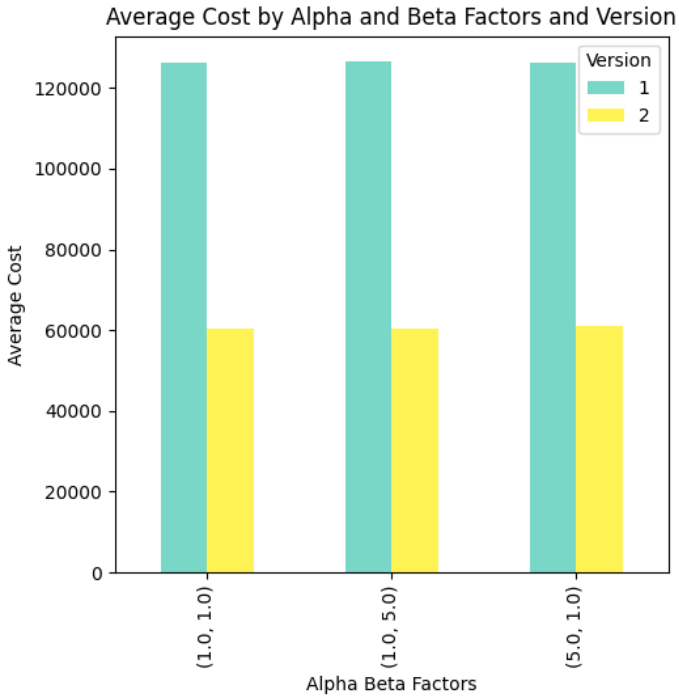
Ant Colony

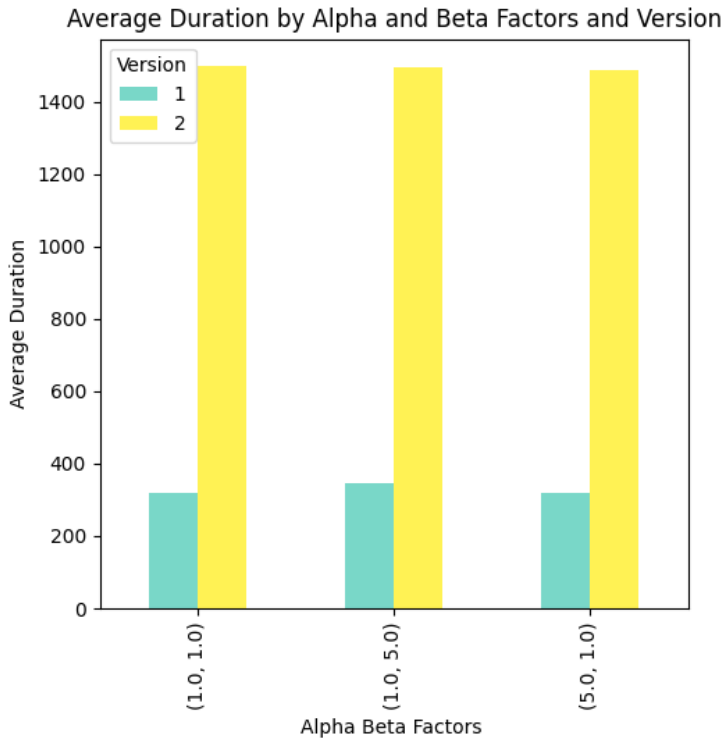
For both versions of algorithm various combinations of parameters were tested – small and big number of ants (1 and 30), three different evaporation rates (0.1, 0.5, 0.9) and three combinations of alpha and beta factors ({1,1}, {1,5}, {5,1}). Number of iterations was always 10. Algorithms were tested on many different instances.



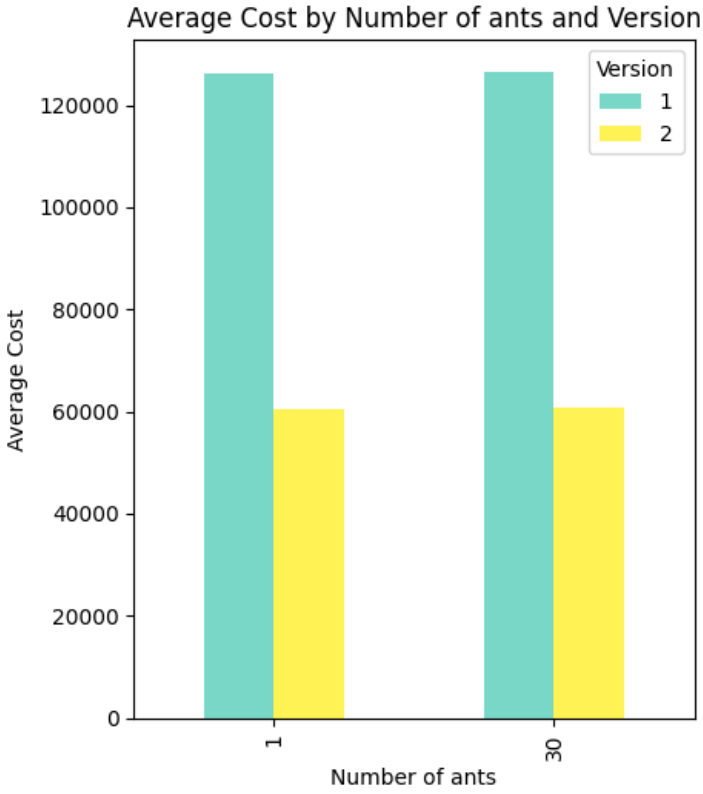


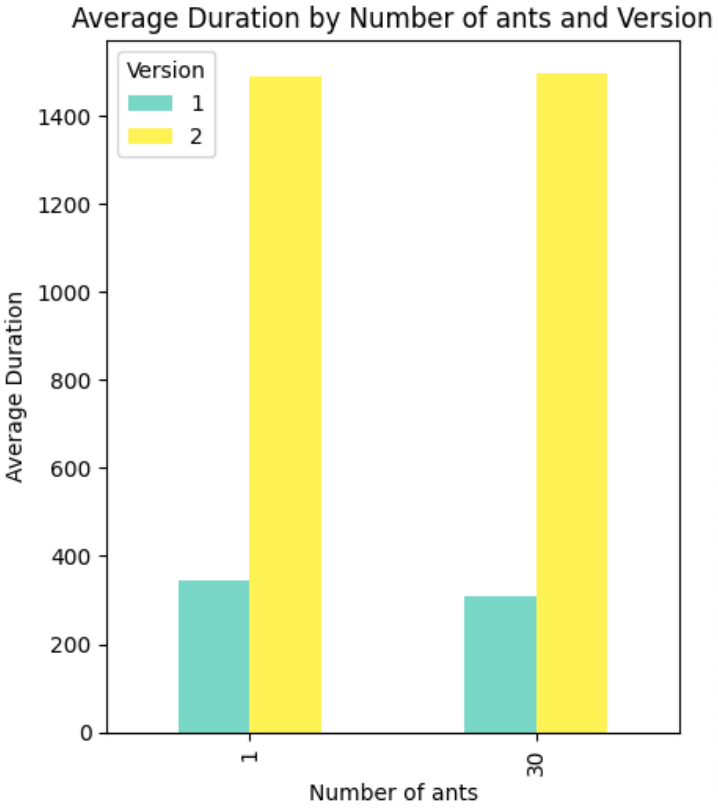
Above we can see average results of evaporation rates for two different versions, which are described in previous section. As we can see first version's solution has much better duration, but worse cost. Differences based on evaporation rate are negligible on these plots. These differences may be more noticeable on plots for single instances.





Above we can see average results of alpha and beta factors, again for two different versions. As we can see results are similar as for evaporation rate. Influence of these factors is also small, but we can notice that for first version of algorithm, duration is little bit worse for factors (1.0, 5.0). These differences may be more noticeable on plots for single instances.





Above we can see average results of number of ants. Again results are similar. Duration for first algorithm is little bit better for bigger number of ants. These results are unexpected, bigger number of ants should have bigger influence on results, but probably it means that all of the ants are finding very similar solutions or these plots are disturbed, because there are average results for different instances.

Now let's look at best runs for every tested instance. First, let's see best runs in terms of duration. Results for all combinations are available in separate file ant_colony_results.csv. They were too big to put them in the report.

instance	num_ants	alpha	beta	evaporation_rate	version	cost	duration
10_3_5_3.def	30	1.0	5.0	0.1	1	12457	58
10_5_8_5.def	30	5.0	1.0	0.1	1	11618	51
200_20_150_9_D5.def	30	1.0	1.0	0.5	1	95278	907
100_20_23_9_D1.def	30	1.0	1.0	0.5	1	52319	148
100_10_27_9_D2.def	30	1.0	1.0	0.9	1	45230	234
100_20_65_9.def	30	1.0	1.0	0.9	1	128042	154
200_10_50_9.def	30	5.0	1.0	0.9	1	256080	473
100_20_47_9.def	30	5.0	1.0	0.1	1	130143	128
200_40_45_9.def	30	5.0	1.0	0.5	1	279420	136
200_40_90_9.def	30	1.0	1.0	0.9	1	298726	157
100_5_48_9.def	30	1.0	1.0	0.5	1	195143	480
100_10_47_9.def	30	5.0	1.0	0.1	1	142962	252
15_9_12_9.def	30	1.0	1.0	0.9	1	18761	68
200_40_130_9_D4.def	30	1.0	1.0	0.9	1	103305	497
100_10_64_9.def	30	1.0	1.0	0.5	1	116566	257
200_10_84_9.def	30	1.0	1.0	0.9	1	228406	514
100_5_20_9_D3.def	30	1.0	1.0	0.9	1	41436	386
200_20_55_9.def	30	1.0	1.0	0.9	1	235765	231
200_20_97_9.def	30	1.0	1.0	0.9	1	280562	249
200_10_135_9_D6.def	30	1.0	1.0	0.5	1	106532	646
15_3_5_3.def	30	1.0	1.0	0.1	1	10703	197
15_6_10_6.def	30	1.0	1.0	0.9	1	15610	76

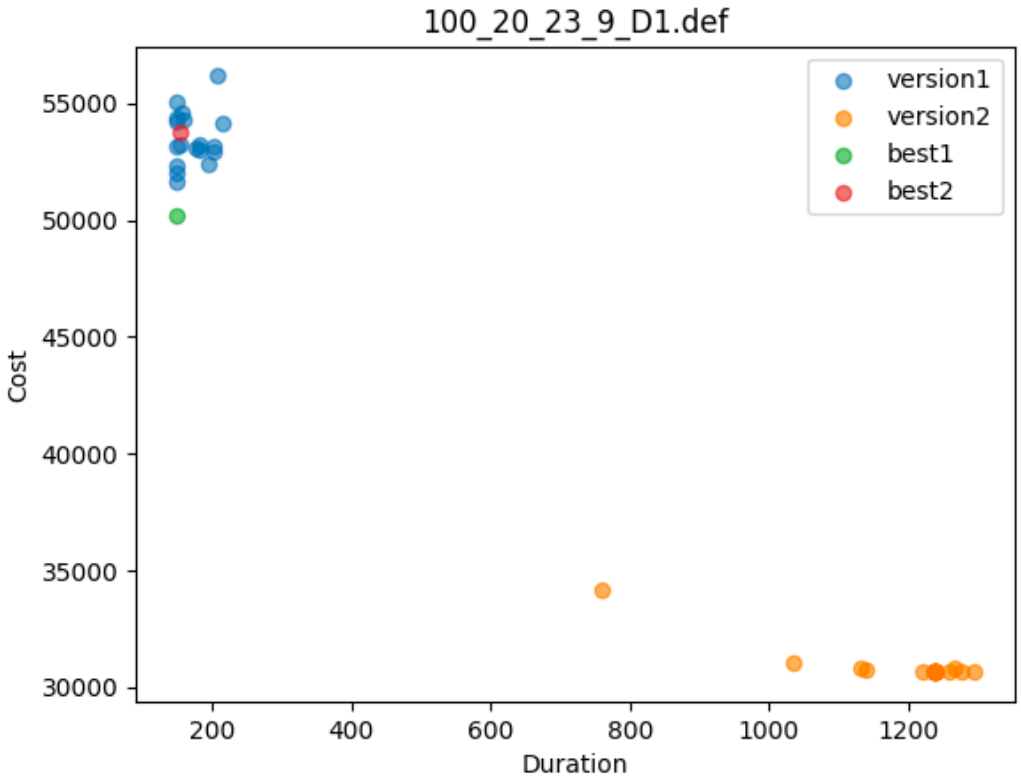
As we can see, all best results have number of ants equal 30 and uses first version of algorithm. Evaporation rate is rather bigger than smaller, but we can't see clear dependency. Alpha and beta factors are usually (1.0, 1.0), so we can assume it works best for this combination.

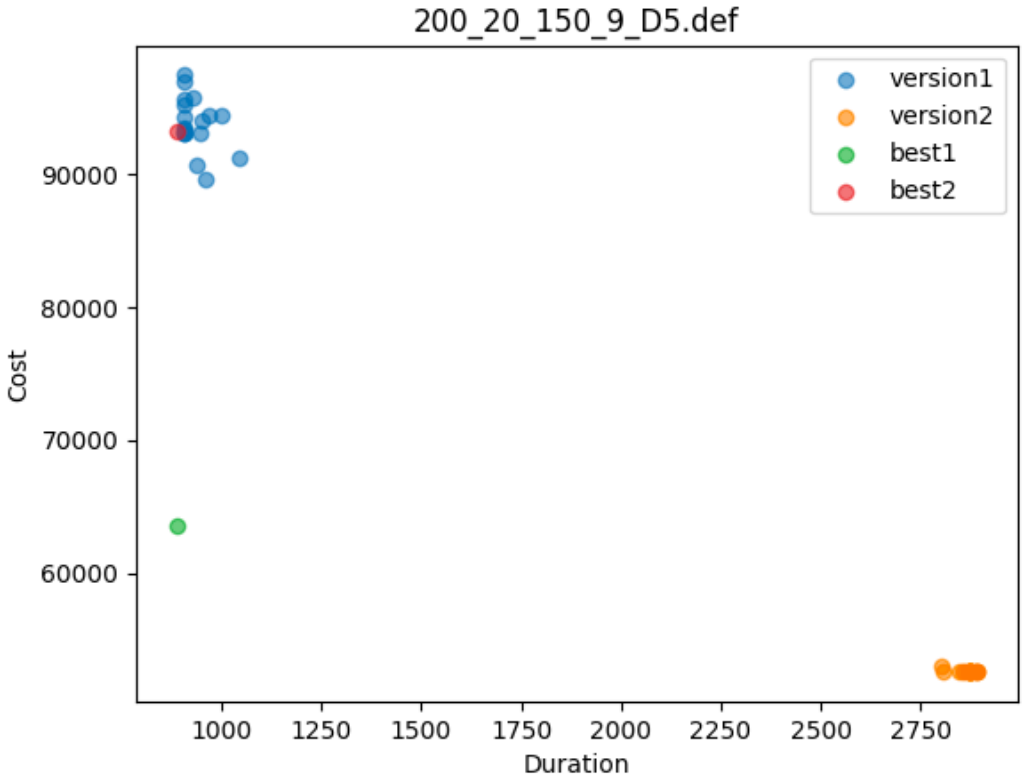
Below there are similar results, for best runs in terms of cost. Important fact is, that a lot of runs here has the same cost, then one of them was chosen randomly.

instance	num_ants	alpha	beta	evaporation_rate	version	cost	duration
10_3_5_3.def	30	1.0	1.0	0.1	2	10845	189
10_5_8_5.def	30	1.0	1.0	0.1	2	9013	127
200_20_150_9_D5.def	30	1.0	1.0	0.1	2	52542	2847
100_20_23_9_D1.def	30	1.0	1.0	0.5	2	30643	1260
100_10_27_9_D2.def	30	1.0	1.0	0.5	2	26771	982
100_20_65_9.def	1	1.0	1.0	0.1	2	60954	924
100_5_64_9.def	30	1.0	1.0	0.1	2	74199	1486
200_10_50_9.def	30	1.0	5.0	0.5	2	106986	2881
100_20_47_9.def	30	1.0	1.0	0.5	2	56190	1190
200_40_45_9.def	30	1.0	5.0	0.1	2	79979	2663
200_40_90_9.def	30	1.0	1.0	0.5	2	82177	2300
100_5_48_9.def	30	1.0	1.0	0.5	2	178346	1323
100_10_47_9.def	30	1.0	5.0	0.9	2	92771	1267
15_9_12_9.def	1	1.0	1.0	0.9	2	9841	120
200_40_130_9_D4.def	1	1.0	1.0	0.5	2	47050	1833
100_10_64_9.def	30	1.0	1.0	0.5	2	63279	1030
200_10_84_9.def	30	1.0	1.0	0.5	2	119500	2928
100_5_20_9_D3.def	30	1.0	1.0	0.9	2	30728	1344
200_20_55_9.def	30	1.0	1.0	0.5	2	71262	2991
200_20_97_9.def	30	1.0	5.0	0.9	2	100421	1665
200_10_135_9_D6.def	30	5.0	1.0	0.5	2	73207	2526
15_3_5_3.def	30	1.0	1.0	0.9	2	6289	284

As we can see, most best results have number of ants equal 30 and all of them uses second version of algorithm. Evaporation rate is rather bigger than smaller, but we can’t see clear dependency. Alpha and beta factors are usually (1.0, 1.0), so we can assume it works best for this combination.

Now let's compare our results with best known solution, which are taken from two sources mentioned in references section. Below there are plots generated for a two exemplary instances. There are presented all runs for each instance, not only these from tables above.





We can see, that solutions are pretty good. Let's start with discussing solutions generated with first version of Ant Colony algorithm. They are really near optimal solution from second source. For instance 100_20_23_9_D1 some of them are even better in terms of duration and a lot of them is better in terms of cost. For instance 200_20_150_9_D5 there is no better solution in terms of duration, but they are really close and few of them is better in terms of cost. Optimal solution from first source is definitely better than our solutions in terms of cost, but in terms on duration it is similar to best of ours.

Solutions generated by second version of Ant Colony algorithm are far away from best known solutions from both sources. They are much worse in terms of duration, but also much better in terms of cost.

So, summarizing:

- First version of Ant Colony algorithm is very nice in terms of duration and second version in terms of cost. The difference comes from way of constructing solutions.

- On first plots number of ants didn't seem to have much of an effect on quality of solutions, but in the tables we could see it has pretty big influence. The more ants implicates better solutions, especially for first version of algorithm.

- Evaporation rate and alpha, beta factors also didn't seem to have much of an effect on quality of solutions in first plots, but in the tables we could see some dependencies. Not as clear as with number of ants, but generally bigger evaporation rate is better and best solutions usually had (1.0, 1.0) factors.

- Compared to reference solutions, ours perform well. The first version of the algorithm finds similar solutions to the reference ones, and the second one is much worse in terms of duration, but much better in terms of cost.

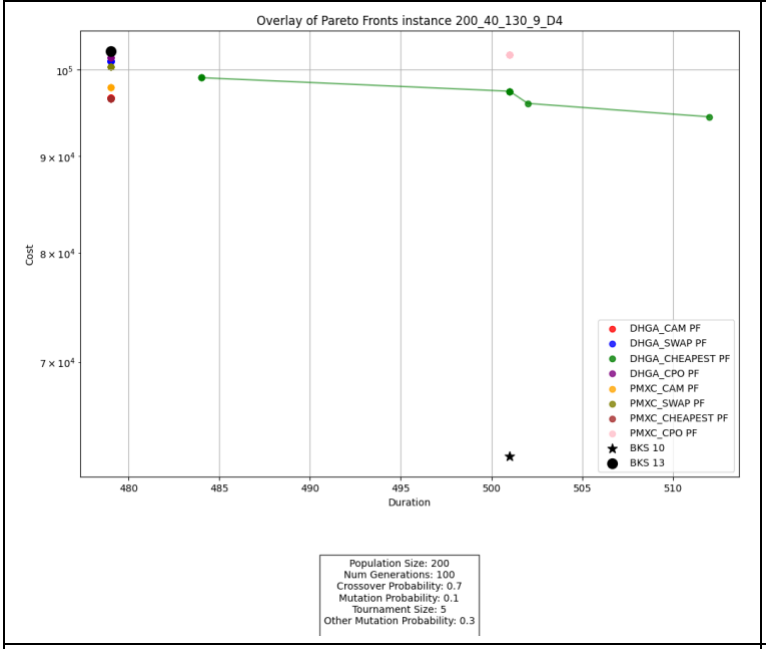
Multi Dimensional – Non-dominated Sorting Genetic Algorithm

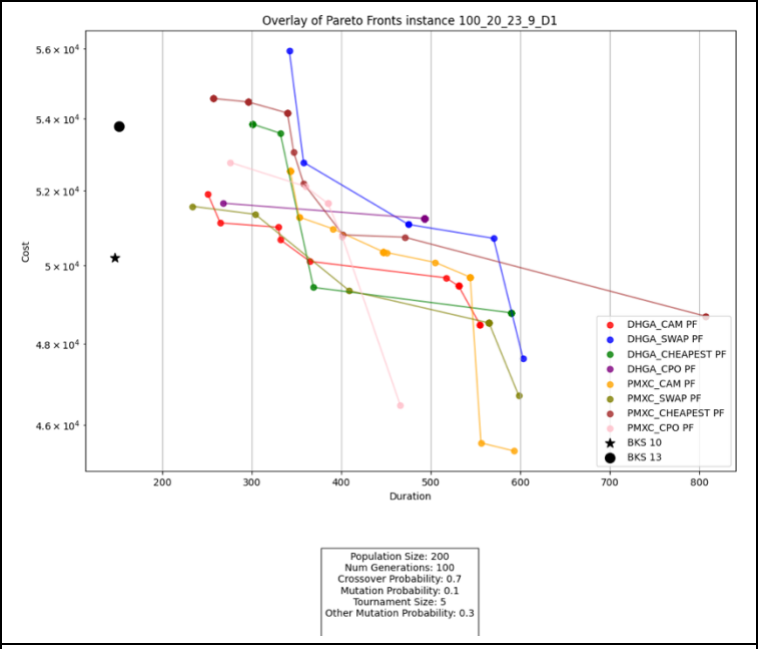
1. Overall results

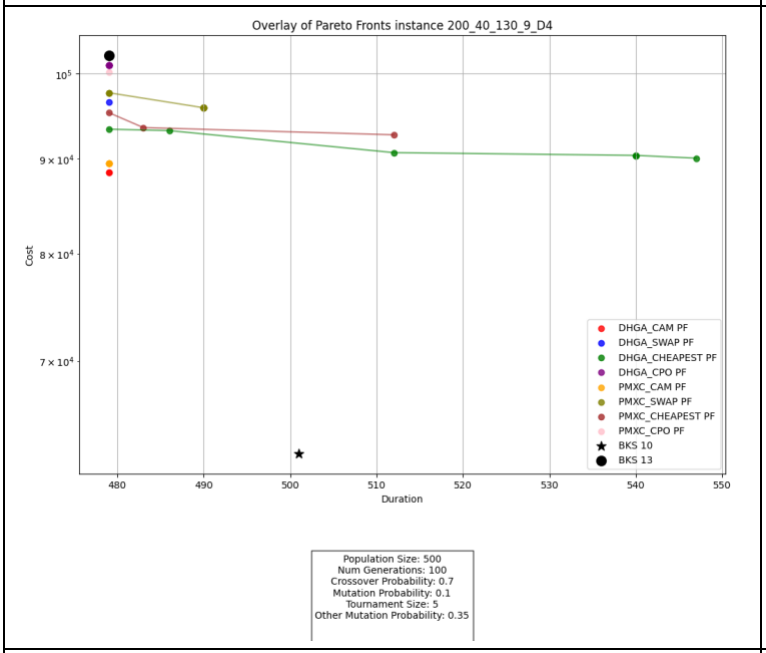
In this charts there are only non-dominated pareto fronts from all generations. They are connected to better clarity. In this section we are running our optimizer with different parameters and comparing it with best known solutions for the problem.

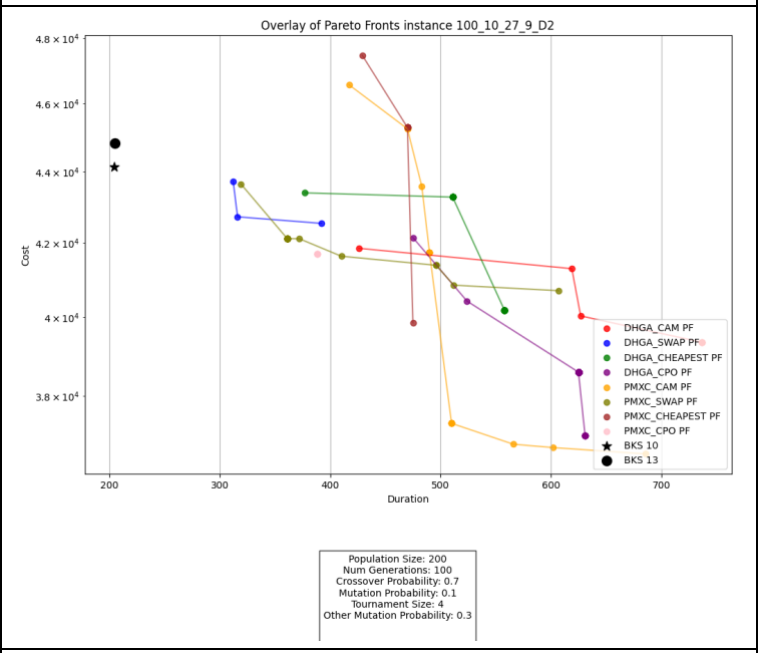
Example runs for different instances with parameters listed on charts.

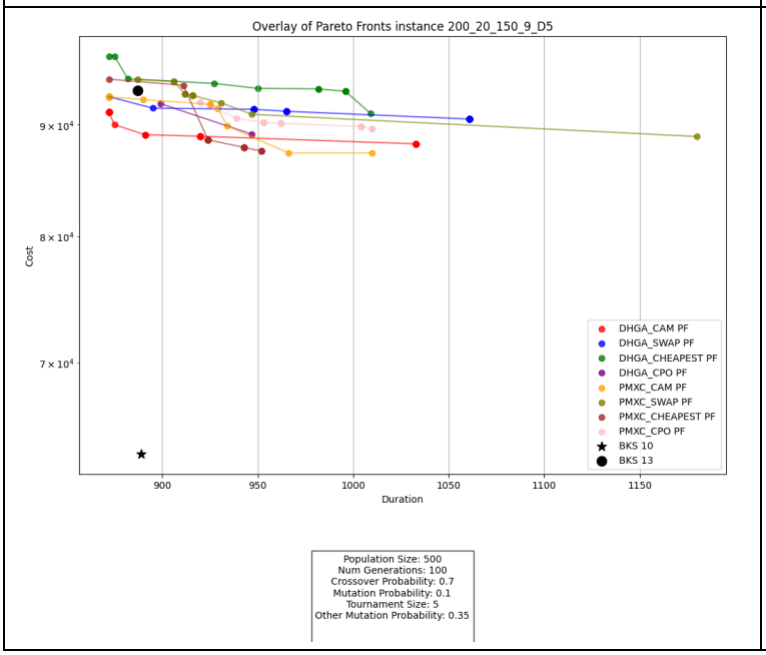
When it comes to searching for parameters. We found the best parameters by checking whether they give better or worse solutions. We did this empirically.

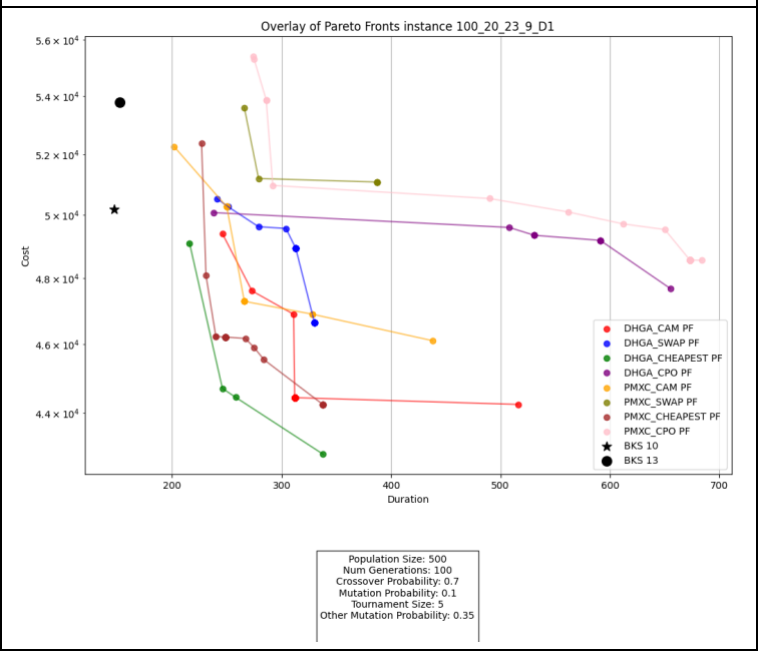

Overlay of Pareto Fronts instance 200_40_130_9_D4
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 200
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.3

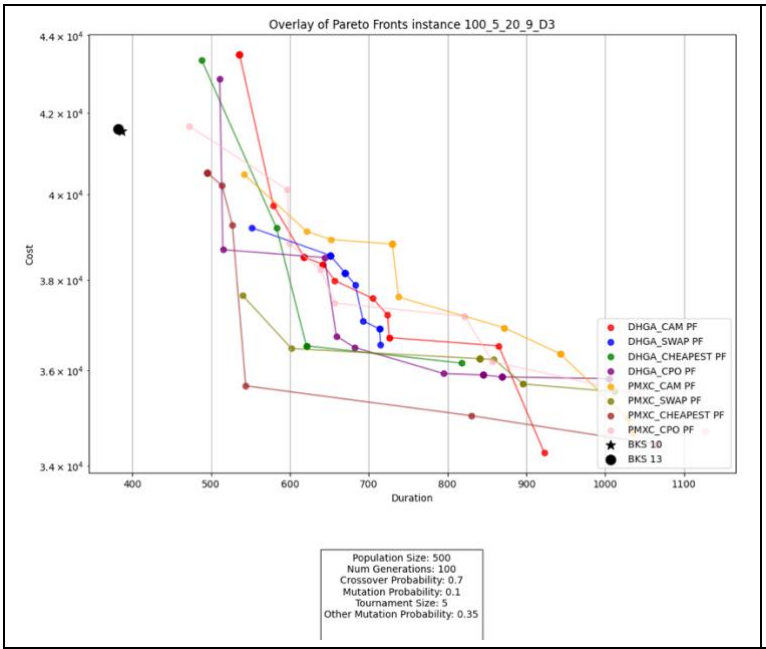

Overlay of Pareto Fronts instance 100_20_23_9_D1
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 200
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.3

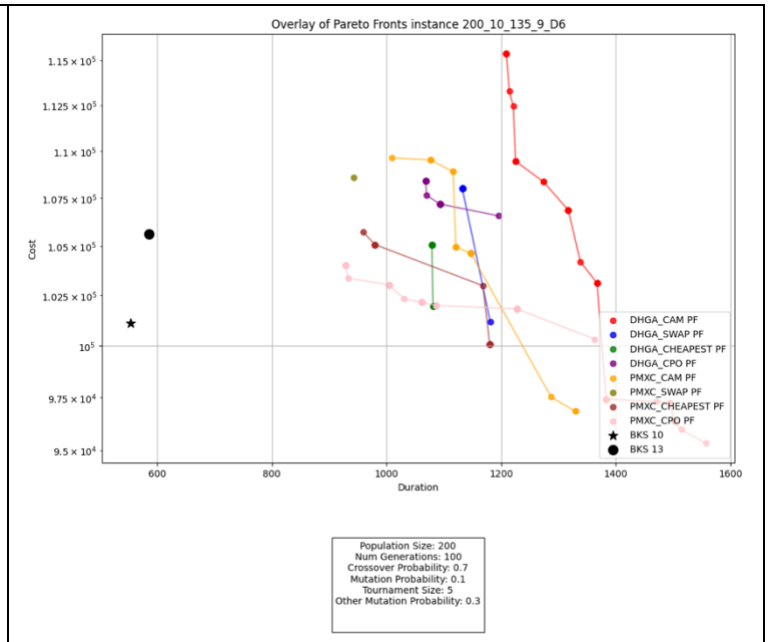

Overlay of Pareto Fronts instance 200_40_130_9_D4
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 500
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.35

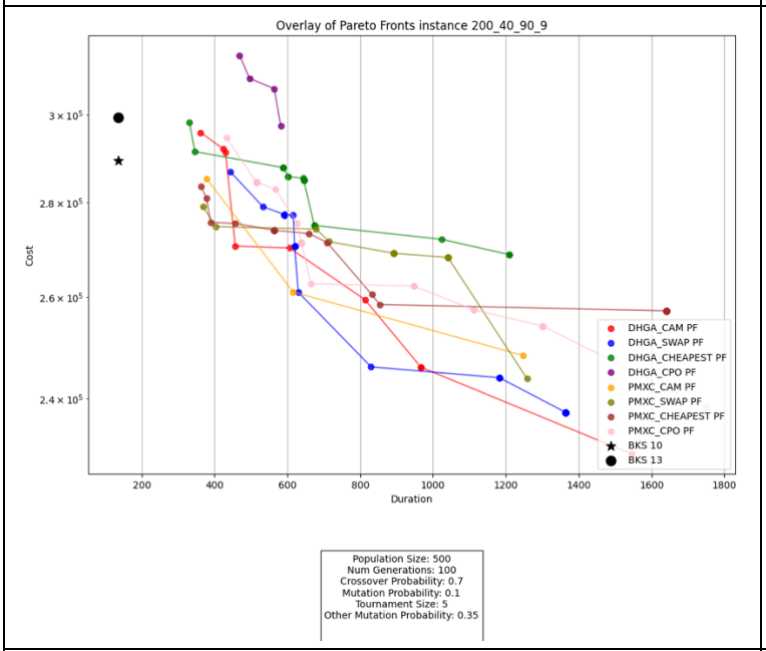

Overlay of Pareto Fronts instance 100_10_27_9_D2
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 200
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 4
Other Mutation Probability: 0.3

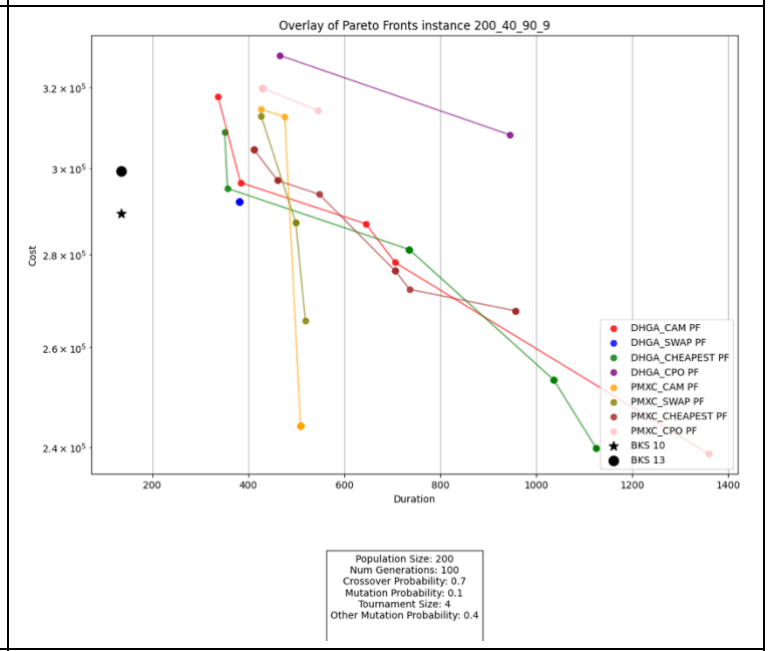

Overlay of Pareto Fronts instance 200_20_150_9_D5
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 500
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.35

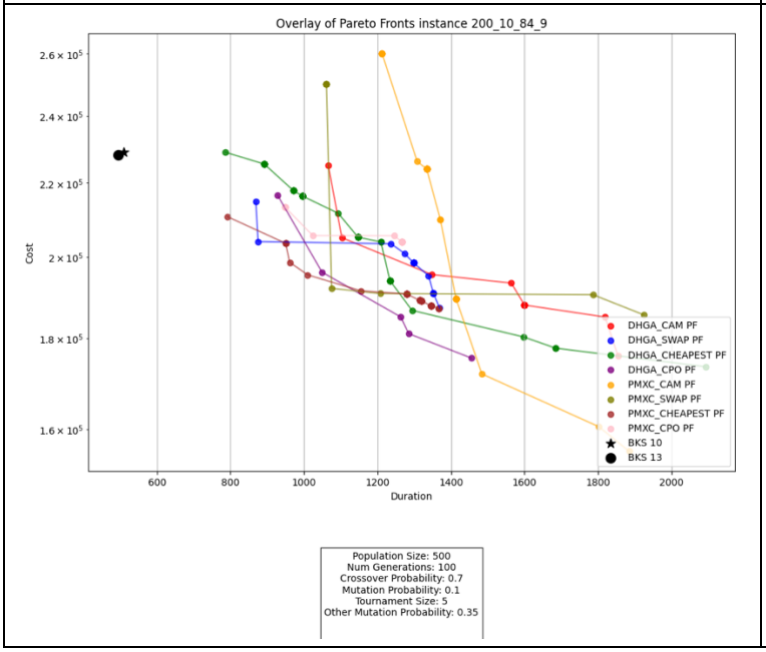

Overlay of Pareto Fronts instance 100_20_23_9_D1
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 500
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.35

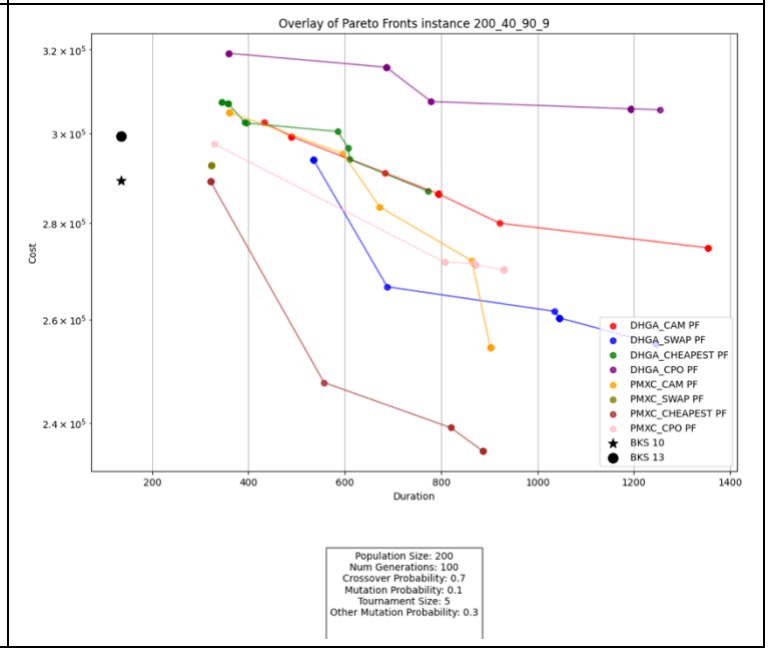

Overlay of Pareto Fronts instance 100_5_20_9_D3
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 500
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 200_10_135_9_D6
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 200
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.3


Overlay of Pareto Fronts instance 200_40_90_9
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 500
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 200_40_90_9
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 200
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 4
Other Mutation Probability: 0.4


Overlay of Pareto Fronts instance 200_10_84_9
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 500
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 200_40_90_9
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Population Size: 200
Num Generations: 100
Crossover Probability: 0.7
Mutation Probability: 0.1
Tournament Size: 5
Other Mutation Probability: 0.3

1. Overview

The performance evaluation of our Multi-Dimensional Non-dominated Sorting Genetic Algorithm (MD-NSGA) across various instances reveals a noteworthy pattern. The algorithm demonstrates robust capabilities in handling cost optimization, while also adhering to the Best Known Solutions (BKS) concerning duration. This report details the observed outcomes, emphasizing the balance between duration constraints and cost efficiency.

2. Duration Performance

Throughout multiple runs across different problem instances, the optimizer consistently respects the Best Known Solutions (BKS) regarding duration. The algorithm maintains adherence to these BKSs, indicating its effectiveness in managing time constraints. Notably, there was only one instance where the optimizer succeeded in surpassing the BKS:

Instance: 200_20_150_9_D5.def
Result: The optimizer broke the duration BKS, demonstrating its potential to exceed standard benchmarks under specific conditions.

3. Cost Optimization Performance

When evaluated for cost optimization, the MD-NSGA consistently outperforms expectations, showcasing superior capabilities in identifying and optimizing cost-efficient solutions. The optimizer excels in both exploration and exploitation phases, striking a balance between discovering new solutions and refining existing ones to achieve cost reduction.

Exploration: The algorithm effectively navigates the solution space, discovering local optima that contribute to cost savings.
Exploitation: It refines these solutions through iterative improvements, often reaching or approximating global optima in terms of cost.

4. Parameter Settings

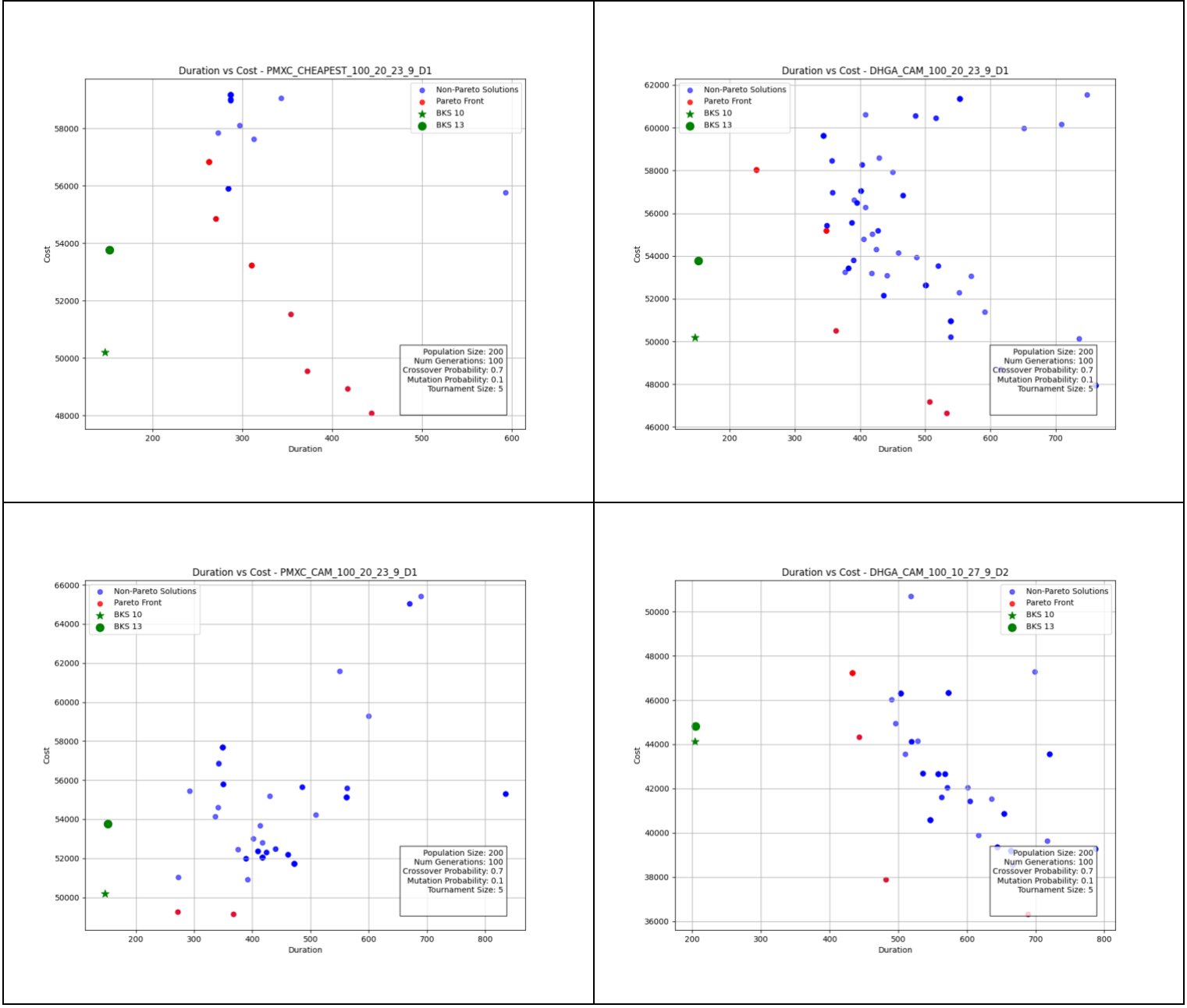
Despite the parameters not being finely tuned to their optimal values, the algorithm demonstrates robust performance. This suggests a level of resilience in the MD-NSGA, allowing it to deliver competitive results even with suboptimal parameter configurations.

Exploration vs. Exploitation: The chosen parameters facilitate a balance between exploration and exploitation, essential for effective multi-objective optimization.
Performance Robustness: The optimizer's ability to adapt and perform well under varying parameter settings underscores its robustness and flexibility.

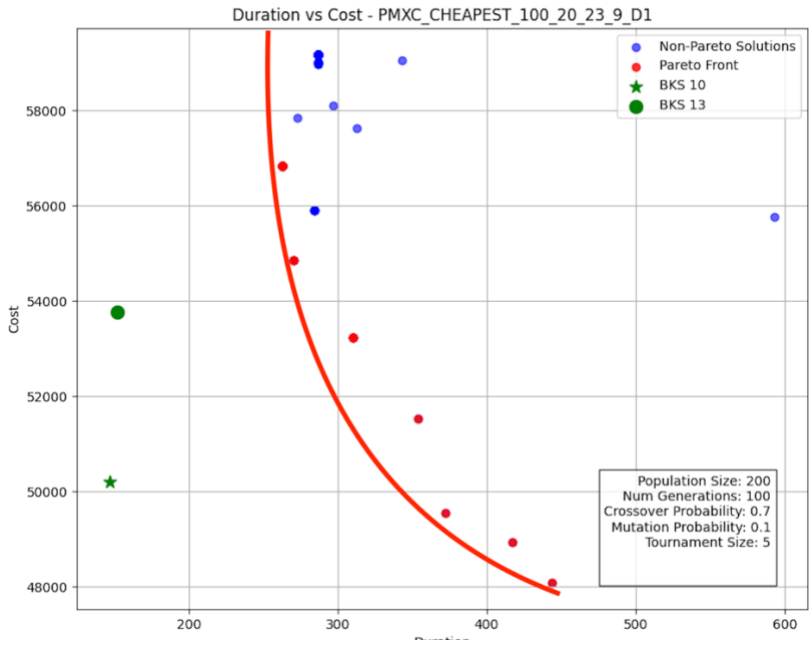
5. Conclusion

The MD-NSGA has proven to be an effective tool for cost optimization while maintaining compliance with duration constraints, as evidenced by its performance across various instances. Its capability to break the BKS for duration in specific instances, coupled with consistent cost savings, highlights its potential for achieving significant optimization results. The algorithm's resilience to parameter settings further enhances its applicability across diverse optimization problems. **But it is hard to predict which one of CX and MUT will be best for all solution instances.**

2. Results within algorithms



As for the standalone optimizer cofigurations, we see that the pareto-front is generated as expected in most cases. It means that we can see pattern similar to 1/x function.



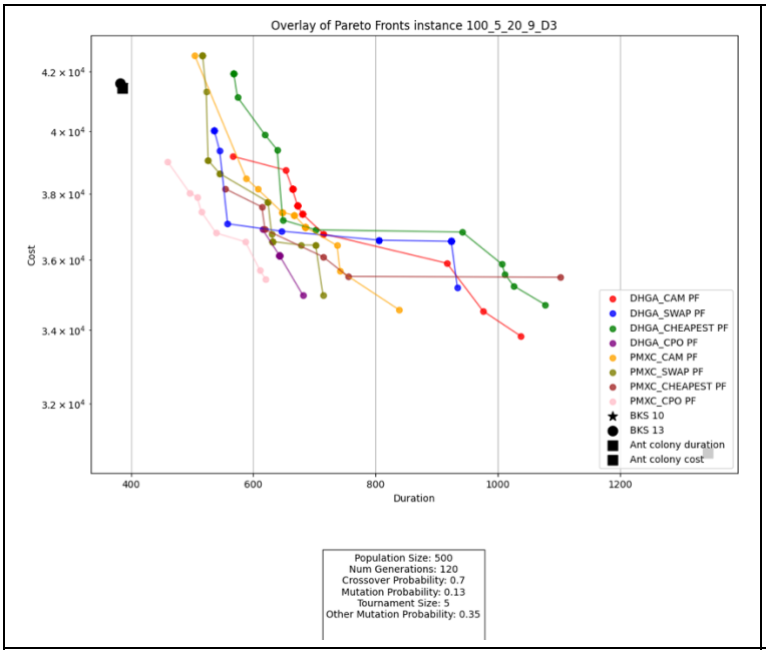
An example of a good paretofront with a sketched "expected" line.

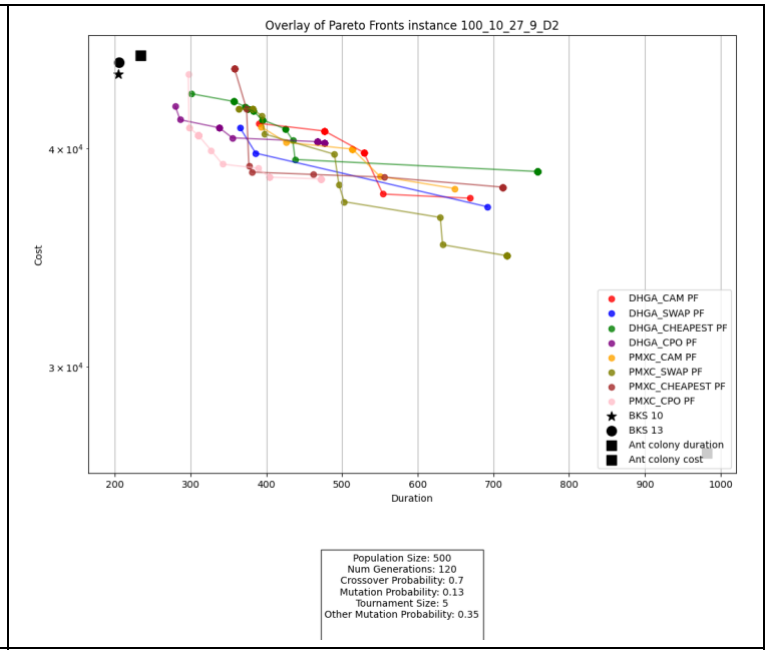
Combined Results

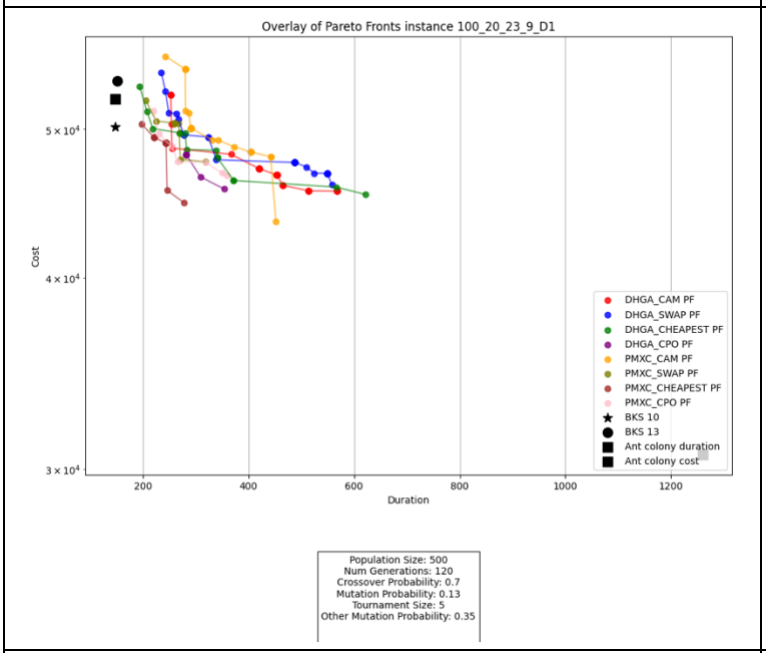
In this section, we conduct a comparative analysis of both models, recognizing that they operate under different principles and mechanisms. Despite these differences, a unified chart allows us to visualize and analyze their performance collectively, providing a comprehensive perspective on their effectiveness.

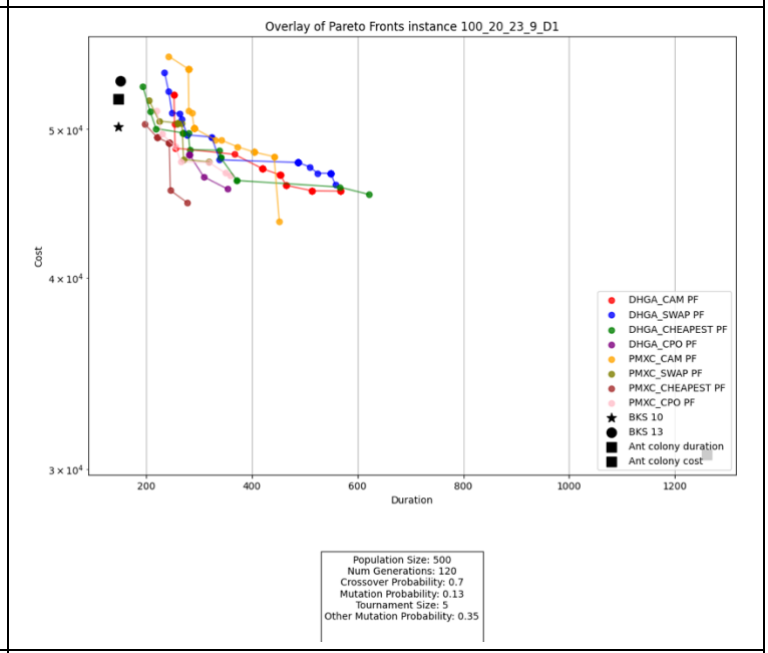
Model NSGA-II: Many paretofronts listed on the chart.
Model Ant Colony: Marked with square in the chart.

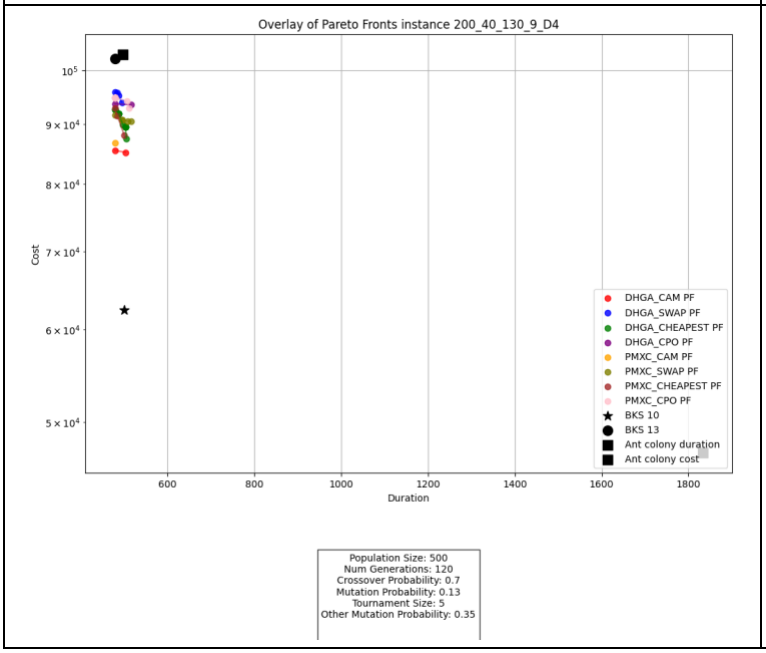
Based on our comparative analysis, it is evident that the Ant Colony model consistently excels in performance, nearly always identifying local or global minima. Particularly in scenarios focused solely on optimizing duration, the Ant Colony model proves to be the superior choice. Despite their differing operational mechanisms, both models demonstrate a robust capability to find optimal or near-optimal solutions across various instances. This convergence in performance underscores their effectiveness in diverse optimization contexts, validating the versatility and reliability of both approaches in achieving high-quality solutions.

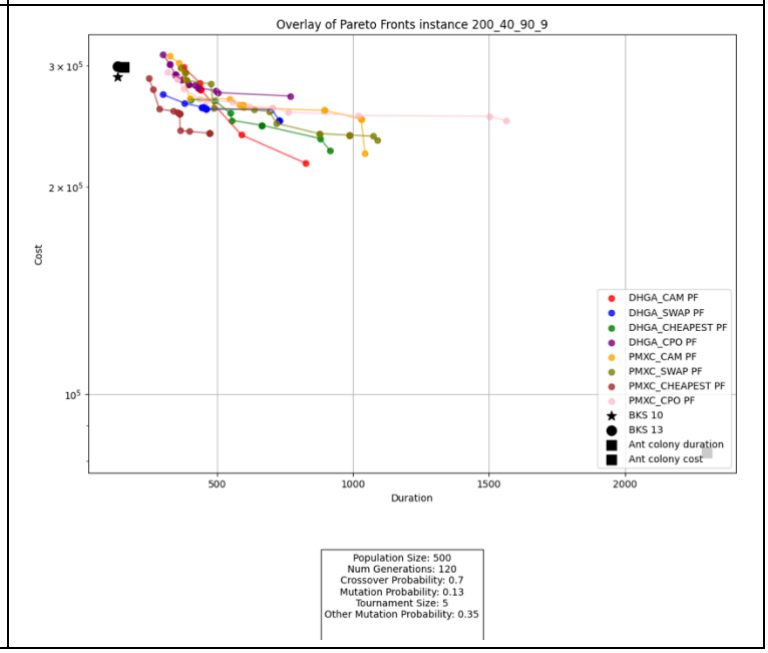

Overlay of Pareto Fronts instance 100_5_20_9_D3
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Ant colony duration
Ant colony cost
Population Size: 500
Num Generations: 120
Crossover Probability: 0.7
Mutation Probability: 0.13
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 100_10_27_9_D2
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Ant colony duration
Ant colony cost
Population Size: 500
Num Generations: 120
Crossover Probability: 0.7
Mutation Probability: 0.13
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 100_20_23_9_D1
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Ant colony duration
Ant colony cost
Population Size: 500
Num Generations: 120
Crossover Probability: 0.7
Mutation Probability: 0.13
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 100_20_23_9_D1
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Ant colony duration
Ant colony cost
Population Size: 500
Num Generations: 120
Crossover Probability: 0.7
Mutation Probability: 0.13
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 200_40_130_9_D4
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Ant colony duration
Ant colony cost
Population Size: 500
Num Generations: 120
Crossover Probability: 0.7
Mutation Probability: 0.13
Tournament Size: 5
Other Mutation Probability: 0.35


Overlay of Pareto Fronts instance 200_40_90_9
Cost
Duration
DHGA_CAM PF
DHGA_SWAP PF
DHGA_CHEAPEST PF
DHGA_CPO PF
PMXC_CAM PF
PMXC_SWAP PF
PMXC_CHEAPEST PF
PMXC_CPO PF
BKS 10
BKS 13
Ant colony duration
Ant colony cost
Population Size: 500
Num Generations: 120
Crossover Probability: 0.7
Mutation Probability: 0.13
Tournament Size: 5
Other Mutation Probability: 0.35

References

http://imopse.ii.pwr.wroc.pl/download.html - test instances

https://link.springer.com/article/10.1007/s00500-014-1455-x - Hybrid ant colony optimization in solving multi-skill resource-constrained project scheduling problem - Paweł B. Myszkowski, Marek E. Skowroński, Łukasz P. Olech, Krzysztof Oślizło

https://tiezhongyu2005.github.io/resources/popularization/ACO_2006.pdf - Ant colony optimization - Marco Dorigo, Mauro Birattari, Thomas Stutzle

Myszkowski P.B., Siemienski J.J., "GRASP applied to Multi-Skill Resource-Constrained Project Scheduling Problem", Computational Collective Intelligence, Volume 9875 of the series Lecture Notes in Computer Science pp.402-411
DOI: 10.1007/978-3-319-45243-2_37 – best known instances 1

Myszkowski P.B., Olech L.P., Laszczyk M. and Skowroński M.E. "Hybrid Differential Evolution and Greedy (DEGR) for Solving Multi–Skill Resource–Constrained Project Scheduling Problem", Applied Soft Computing, 2018 – best known instances 2