# Evolutionary Algorithm & Hybrid Approaches

Maximilian Kleinegger*
Vienna University of Technology

Martin Wustinger*
Vienna University of Technology

*All authors contributed equally to this research.

## 1 Introduction

This report aims to provide insight into the second programming assignment. It includes all the necessary information about the assignment, the algorithms implemented, the parameters tuned, and the results obtained. Furthermore, it outlines the challenges that had to be overcome during the development process.

## 2 Overview

In this programming exercise, we are tasked with developing two algorithms for the *MWCPP*. We have chosen the *Evolutionary/Genetic Algorithm* and a *hybrid algorithm.* Additionally, we will perform hyperparameter tuning and conduct statistical significance testing before final evaluation on the provided instances.

To begin, let us briefly revisit the *Minimum Weighted Crossing with Constraints Problem (MWCCP)*: This problem involves working with a bipartite graph $G = (U \cup V, E)$, where the goal is to minimize the sum of weights of the crossed edges while maintaining constraints such that that vertex $v$ must appear before vertex $v'$. To achieve this, we aim to find a permutation $\pi$ of the vertices $V$ such that the objective function is minimized:

$$f(\pi) = \sum_{(u,v)\in E} \sum_{(u',v')\in E \ u<u'} (w_{u,v} + w_{u',v'}) \cdot \delta_\pi((u,v),(u',v'))$$

Here, $\delta_\pi(\cdot,\cdot)$ is an indicator function that returns 1 if the two edges cross in the solution defined by the permutation $\pi$, and 0 otherwise. Formally, it is defined as:

$$\delta_\pi((u,v),(u',v')) = \begin{cases} 1, & \text{if } \mathrm{pos}_\pi(v) > \mathrm{pos}_\pi(v'), \\ 0, & \text{otherwise.} \end{cases}$$

## 3 Methods

This section describes the implemented algorithms.

### 3.1 Evolutionary/Genetic Algorithm

Since we wanted to implement an Evolutionary/Genetic Algorithm to solve our problem, we encountered one major challenge: the constraints. Handling these constraints proved to be the most cumbersome aspect, especially since evaluating them has a complexity of $O(|V|^2)$. To address this, we opted for a strategy where the solutions generated in each generation are inherently valid, thereby avoiding the additional step of checking for validity.

For this reason, we implemented the genetic algorithm as a **Biased Random Key Genetic Algorithm (BRKGA)**, which inherently avoids constraint violations. The approach is based on an initially valid solution, which we split into Elitist and Non-Elitist groups. We then always select one parent from each group to generate new valid solutions. These solutions are diversified by introducing randomly mutated solutions to prevent premature convergence to the best Elitist solution. To achieve this, we implemented the following functions:

- **Encoding:** We encoded all the nodes in the solution with random numbers, as is typical for such algorithms.
- **Decoding:** Solutions were sorted by their random numbers and used as a base to generate valid solutions. To ensure correctness, especially at the beginning, we adopted a similar approach to our construction heuristic. Preferred positions were stored based on the sorted random keys, all constraints were mapped out, and based on open constraints (incoming edges) and preferred positions, we decoded the solution into a valid one.
- **Recombination:** We followed the recombination approach proposed by *BRKGA*. Specifically, one parent was selected from the Elitist group and one from the Non-Elitist group to perform a biased crossover that favored the Elitist. The degree of bias was controlled by the parameter **rho_br**, and the fraction of Elitists was controlled by **elite_fraction**.
- **Mutation:** Mutated solutions were generated for each new generation and replaced the worst solutions from the current population. The fraction of mutated solutions was controlled by **mutation_fraction**.
- **Fitness Function:** The fitness function was implemented with $C_{\max}$ representing the current worst solution and $g(I)$ representing the objective value:

$$f(I) = \begin{cases} C_{\max} - g(I), & \text{if } g(I) < C_{\max}, \\ 0.01, & \text{otherwise.} \end{cases}$$

### 3.1.1 Problems & Possible Optimizations

The main problems of this algorithm for the *MWCPP* are twofold. First, we must calculate the objective value, which has a complexity of $O(|V|^2)$, and perform the respective correctness check, which is also in $O(|V|^2)$. This is not computationally cheap, especially as $|V|$ increases and the number of instances per generation grows. To avoid at least the second check, we opted for *BRKGA*, where fairness is only verified for mutated solutions.

However, we must acknowledge that this algorithm is not the most suitable approach for this problem, particularly when comparing its runtime to, for example, a *Local Search* using the first improvement heuristic, which can explore much more of the search space. To address these limitations, we attempted optimizations such as parallelizing and caching solutions.

Unfortunately, since we are using Python and are thus restricted by the Global Interpreter Lock (GIL), the first approach did not yield any improvements. Additionally, the second approach did not produce significant gains, as its benefits were too marginal. Considering the memory effort required, caching was not feasible for larger instances.

## 3.2 Construct Solve Merge Adapt

The Construct, Merge, Solve, and Adapt (CMSA) algorithm is a metaheuristic optimization approach designed to solve complex combinatorial problems efficiently. *CMSA* operates by iteratively constructing partial solutions, merging these into a restricted solution pool, and solving the resulting subproblems using an exact method, in our case an Integer Linear Program. The adaptative component dynamically adjusts the restricted solution pool based on feedback from prior iterations, promoting convergence to high-quality solutions.

### 3.2.1 Integer Linear Program

In our implementation we use an Integer Linear Program (ILP) in the solve step of our *CMSA* algorithm. We use the following formulation:

**Given:**

- $w_{ij}, i, j \in V$, sum of weights of edges that cross if $i$ would come before $j$ in a solution $\pi$.
- graph $G = (V, E)$ and set of constraints $C$

**Variables:**

- binary $x_{ij}, i, j \in V, i < j$, states whether the nodes $i$ and $j$ are in their numerical order in the solution. Hence, the variable $x_{ij} = 1$, if $pos_\pi(i) < pos_\pi(j)$ for solution $\pi$.

**Model:**

$$\min \sum_{i \in V} \sum_{j \in V, i<j} w_{ij} x_{ij} + w_{ji}(1 - x_{ij}) \tag{1}$$

$$x_{ij} = 1 \qquad \forall (i,j) \in C \tag{2}$$

$$x_{ij} + x_{jk} - x_{ik} \leq 1 \qquad \forall i, j, k \in V \tag{3}$$

$$x_{ij} + x_{jk} - x_{ik} \geq 0 \qquad \forall i, j, k \in V \tag{4}$$

**Description:**

(1) Objective: Minimize the total sum of corssing edge weights.

(2) Given Constraints: each of the given constraints must be satisfied.

(3)/(4) Transitivity constraints: if $i$ and $j$ are in the same order as $j$ and $k$, then $i$ and $k$ have to be in that order as well.

### 3.2.2 Algorithm and Optimization

The algorithm operates on so called solution components. In our case a solution component was, similar to the variables of the ILP, simply the decision wheter two nodes are in their numerical order in the solution or not. In our implemented the algorithm executes the following three tasks in a repeated manor. At first we create randomized greedy solutions, with the greedy algorithm from the previous exercise. after each greedy we update the set of solution components based on the new solution and add all components that aren't already included. Secondly, we update the ILP Model, by fixing the variables based on the solution components contained in the set. Hence, if a solution component only appears once, so either wit hthe value 0 or 1, we can also set the variable of the ILP to that value. Lastly, we increase the age of each component by one and reset the age of each component contained in the ILP solution back to 0.

**Optimization** We could have omitted the given constraints from the ILP or don't track the solution components related to these constraints, since they have allways a fixed value and are tracked twice in our solution. However, Gurobi should be intelligent enough to omit duplicate constraints.

## 4 Hyperparameter Tuning with SMAC3

Sequential Model-Based Algorithm Configuration (SMAC) is an efficient approach to hyperparameter tuning that leverages a probabilistic model to explore the hyperparameter space. SMAC iteratively refines its search by selecting the most promising configurations to evaluate, focusing computational resources on regions of the search space that are likely to yield the best results. Since the task was to tune hyperparameters not for a single instance, but for a group of instances, we decided to create a more sophisticated objective function for *SMAC* to optimize, rather than tuning each instance individually and then aggregating the results. Different instances within the same size group often have drastically different objective functions, so simply summing the objective values would not have been beneficial. Such an approach would favor instances with larger weights or more edges. To address this, we generated a fixed greedy baseline for each instance, and the objective value for each instance was calculated by dividing the value returned by either *BRKGA* or *CMSA* by this baseline. By summing these normalized scores, we obtained a more robust metric for comparing multiple runs of the algorithms.

| | | BRKGA | | | |
|---|---|---|---|---|---|
| Instance_Size | elite_frac | max_time | mutation_frac | pop_size | rho_br |
| small | 0.29 | 900 | 0.09 | 242 | 0.61 |
| medium | 0.29 | 900 | 0.12 | 157 | 0.61 |
| medium_large | 0.12 | 900 | 0.21 | 76 | 0.58 |
| large | 0.10 | 900 | 0.21 | 76 | 0.58 |

Table 1: Tuned *BRKGA* Hyperparameters

| | | CMSA | | | |
|---|---|---|---|---|---|
| Instance_Size | construct_time | ilp_time | k | max_age | max_time |
| small | 2 | 4 | 6 | 3 | 10 |
| medium | 16 | 12 | 35 | 6 | 50 |
| medium_large | 14 | 32 | 6 | 3 | 100 |
| large | 25 | 77 | 5 | 2 | 300 |

Table 2: Tuned *CMSA* Hyperparameters

For the final run of the competition instances, we set the `max_time` for both algorithms to 900 seconds for each instance size. Initially, we chose to set a lower `max_time` during the hyperparameter tuning process to speed up the individual runs. However, in retrospect, considering that both algorithms, especially *CMSA*, took significant time for precomputing and initializing the ILP, we could have easily set the `max_time` to 900 seconds during tuning for each instance size as well.

Notably, for *BRKGA*, we observe an inverse trend in the hyperparameters, except for *rho_br*. This could be attributed to the fact that, for larger instances, a more diverse exploration is required, as the restriction on running time has a much greater influence compared to smaller instances. This trend is also evident in the *pop_size*, where the algorithm prioritizes more generations for larger instances than for smaller ones. Therefore, the *max_time*, depending on the instance size, clearly has a significant impact. The results could potentially differ if we used the number of generations as a stopping criterion instead of a fixed maximum runtime.

## 5 Significance Testing

We proceed with statistical testing to determine whether one of our tuned algorithms significantly outperforms the other. For this, our null hypothesis is:

$$H_0 : \theta_{\text{BRKGA}} = \theta_{\text{CSMA}}$$

We test this against the following alternative hypothesis:

$$H_1 : \theta_{\text{BRKGA}} \neq \theta_{\text{CSMA}}$$

Additionally, we aim to assess whether *BRKGA* consistently performs better than *CSMA*. For this purpose, we propose a second alternative hypothesis:

$$H_1' : \theta_{\text{BRKGA}} < \theta_{\text{CSMA}}$$

To test these hypotheses, we run both tuned algorithms on all test instances 30 times to collect results. Since we have an equal number of results for both algorithms on each instance, we decided to use paired tests as our primary approach. Finally, when determining whether to use a *t-test* or a *Wilcoxon test*, we will consider the distribution of the differences for each instance.

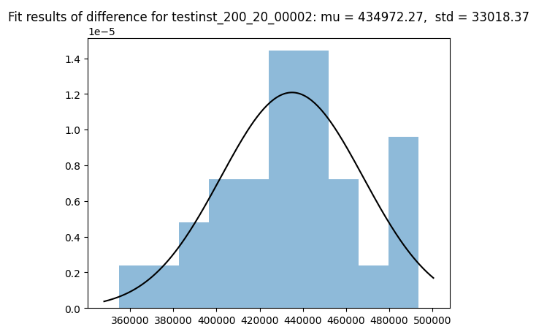

Figure 1: Histogram of $\theta_{\text{BRKGA}} - \theta_{\text{CSMA}}$ for test instance 200_20_00002

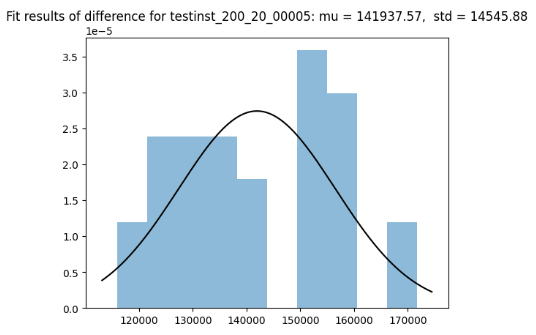

Figure 2: Histogram of $\theta_{\text{BRKGA}} - \theta_{\text{CSMA}}$ for test instance 200_20_00005

As shown in Figure 2 and Figure 1, the data does not closely follow a normal distribution. Therefore, we decided to proceed with the *Wilcoxon test*.
Referring to Table **??**, we observe that for all instances, there is a significant difference when considering $H_1$. Additionally, not a single instance shows *BRKGA* performing significantly worse than *CSMA*, as the p-value for $H_1'$ is 1 for all instances.
Furthermore, we conducted the same test for the best solutions found for each instance, using only those in the Wilcoxon test. The results, presented in Table 3, reveal the same pattern as observed previously.

| p-value $H_1$ | p-value $H_1'$ |
|---|---|
| 0.00 | 1.00 |

Table 3: p-values for only best found solutions over all instances

Therefore, we can conclude that CSMA significantly performs better than *BRKGA*.

### 5.1 Results

We present the results of all runs on the test instances in Table **??**. Notably, we did not apply any normalization to ensure comparability with the tables from Programming Assignment 1.

Furthermore, Figure 3 and Table 7 display the results obtained by applying both algorithms to the competition instances. Since we initially had sufficient time, we decided to use both algorithms instead of selecting only the better one, as suggested in the exercise description.
While *CSMA* clearly outperforms *BRKGA*, it requires significantly more preprocessing time. This trade-off makes it difficult to declare a clear winner in some cases. For this reason, we wanted to highlight the contrast between a faster but worse-performing algorithm and a slower but better-performing one.

## 6 Conclusion

In conclusion, we can state that *CSMA* using an ILP clearly outperforms *BRKGA* due to the nature of the *MWCCP* problem. This superiority is primarily attributed to the optimality of ILP compared to Evolutionary Algorithms and the fact that dealing with invalid solutions is significantly more time-consuming when using *BRKGA*. Calculating the objective value and verifying the correctness of each solution requires substantial computational effort, which becomes increasingly challenging as the population size grows. Consequently, *CSMA* outperforms *BRKGA* in terms of solution quality.
However, while the ILP approach is highly effective, it has one major downside: the construction of the ILP formulation takes an exceptionally long time compared to *BRKGA*. As a result, even though we achieve optimality for many instances, the overall time required often exceeds the original time limit proposed. Thus, while very effective, the ILP approach is also very time-consuming, making it less suitable for certain instances.
Overall, both techniques performed well, but the clear winner is *CSMA*.
Finally, comparing these results to the first assignment, we can observe that a *Local Search* or *Variable Neighborhood Descent (VND)* with the respective neighborhood structures and the first improvement heuristic achieve comparable or even better results in significantly shorter times. This is not only impressive but also highlights the efficiency of these simpler tools, demonstrating their value despite their relative simplicity.

## 7 Appendix

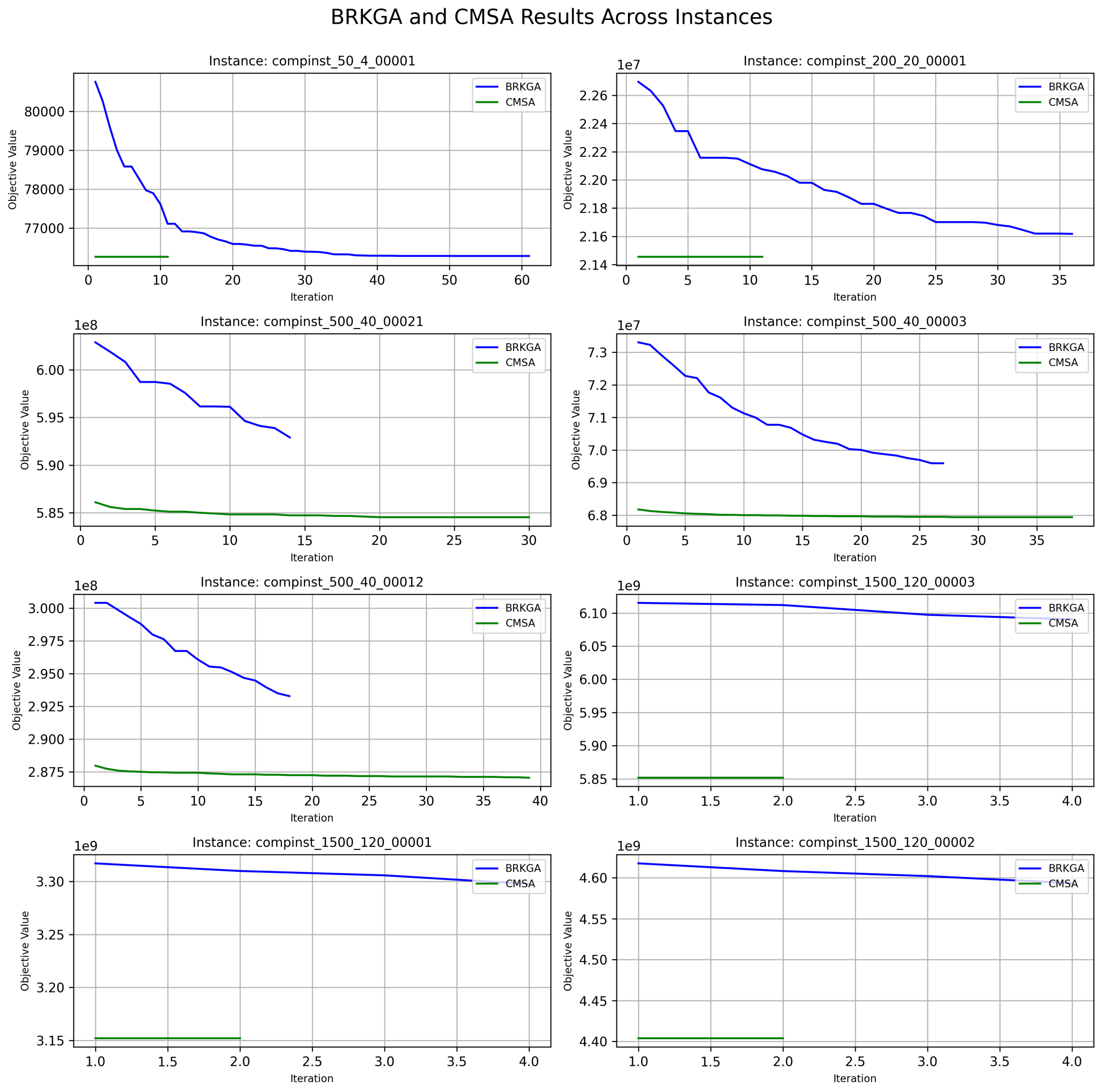

Figure 3: Comparison of *BRKGA* and *CMSA* on Competition Instances

| Instance | BRKGA | | | CMSA | | |
|---|---|---|---|---|---|---|
| | obj | init_time | algo_time | obj | init_time | algo_time |
| inst_50_4_00001 | 76287 | 1 | 204 | 76269 | 0 | 26 |
| inst_200_20_00001 | 21617809 | 37 | 912 | 21456107 | 39 | 210 |
| inst_500_40_00003 | 69593380 | 36 | 912 | 67939372 | 517 | 884 |
| inst_500_40_00012 | 293280893 | 56 | 921 | 287052995 | 496 | 901 |
| inst_500_40_00021 | 592906278 | 65 | 943 | 584549471 | 492 | 680 |
| inst_1500_120_00001 | 3297468098 | 223 | 952 | 3152139305 | 6953 | 1288 |
| inst_1500_120_00002 | 4593496613 | 257 | 1040 | 4403862046 | 7665 | 913 |
| inst_1500_120_00003 | 6090599476 | 271 | 1135 | 5851597299 | 6567 | 938 |

Table 4: Comparison of *BRKGA* and *CMSA* on Competition Instances