Bacterial Memetic Algorithm for Asymmetric Capacitated Vehicle-Routing Problem

Ákos Holló-Szabó and János Botzheim *

Department of Artificial Intelligence, Faculty of Informatics, ELTE Eötvös Loránd University, Pázmány Péter Sétány 1/A, 1117 Budapest, Hungary
* Correspondence: botzheim@inf.elte.hu or dr.janos.botzheim@ieee.org

Abstract: The vehicle-routing problem (VRP) has many variants, including the most accurate models of real-life transportation tasks, making it one of the most important mathematical problems in the field of logistics. Our goal was to design an algorithm that can race against the most recent solutions for VRP and capacitated VRP (CVRP), while also being applicable to real-life models with simulations of real transports. Our algorithm is a variant of the bacterial memetic algorithm (BMA), which we improve upon with novel operators and better methods for manual parameter optimization. The key to our performance is a balanced mixture of the global search of evolutionary algorithms, local search of 2-OPT variants, and the pseudo-global search of probabilistic construction algorithms. Our algorithm benefits from the advantages of all three methods resulting in fast convergence and avoidance of global minima. This is the first time BMA is applied for VRP, meaning that we had to adapt the method for the new problem. We compare our method with some of the most-used methods for VRP on the ABEFMP 1995 dataset. We provide comparison results with the coronavirus herd immunity optimizer, genetic algorithm, hybridization of genetic algorithm with neighborhood search, firefly algorithm, enhanced firefly algorithm, ant colony optimization, and variable neighborhood search. Our algorithm performed better on all data instances, yielding at least a 30% improvement. We present our best result on the Belgium 2017 dataset for future reference. Finally, we show that our algorithm is capable of handling real-life models. Here we are also illustrating the significance of the different parameters.

Keywords: vehicle-routing problem; evolutionary computation; genetic algorithm; bacterial memetic algorithm

1. Motivation and Introduction

The cost of logistics is a burden on the entire economy. It is present in the production and distribution of any physical product (see [1]). Every percentage that can be cut leaves more profit for distributors and makes products cheaper. Furthermore, shorter travel time can result in lower CO₂ gas emissions, of which the logistics sector is one of the biggest contributors. The task of real-life transport optimization is an extended case of the vehicle-routing problem (VRP) [2]. Researchers have been motivated to design solutions for VRP since the 1950s, but VRP is still one of the most active fields of research. The problem is NP-complete; most solutions are hard to apply to more specific cases of VRP and problem sizes of more than 50 transport targets are a big challenge even for the most recent algorithms.

In the case of real-life transportation, vehicles are limited by multiple factors: their carrying capacity, the work time of the driver, the capacity for fuel, etc. The optimal route between two customers varies based on the time of start, weight, and size of the vehicle. The cost of a route is not the same between two customers in the two directions. There can also be multiple depots from which products can be distributed. Algorithms designed for...
the traveling salesman problem (TSP) and simple VRP are rarely efficient for asymmetric capacitated VRPs, especially for multi-depot ones.

One of the most promising algorithms for the problem is the evolutionary algorithm. We keep track of a set of transport plans and iterate on them mixing their properties in the hope of improving their cost. This implements a global search of the solution set, where the tracked set will slowly converge to a local optimum. The main challenges of more traditional evolutionary algorithms are to increase the convergence speed and avoid local minima. To speed up the convergence of evolutionary algorithms, memetic algorithms were introduced that mix the methodology with local search. This change solved the first challenge but made local minima avoidance even harder.

A newer branch of evolutionary algorithms is the bacterial memetic algorithm (BMA). No solutions are eliminated from the set making the risk of crucial property extinction lower. The space is searched based on each solution separately and information is transferred between solutions only in the gene transfer step. In this paper, we present an improved BMA, in which lower-cost properties are more likely to be generated and transferred. This makes the algorithm efficient even on the biggest problem sizes, which previously would have needed clustering. Because the higher-cost properties still have probability, this makes convergence to the local minima less likely than pure local search-based solutions. When local search gets stuck, global search and pseudo-global search help the algorithm break free.

The structure of the paper is as follows. In Section 2 the formulation of the VRP is explained. Section 3 reviews the related literature. In Section 4, the typology of the heuristics algorithms is discussed. Section 5 presents the proposed algorithm for solving VRP. Experimental results are presented in Section 6. Here, we also compare our method with some of the most used methods for VRP, like the genetic algorithm, hybridization of genetic algorithm with neighborhood search, ant colony optimization, and variable neighborhood search. We also present comparison with some of the most recent designs like the coronavirus herd immunity optimizer, firefly algorithm, and enhanced firefly algorithm. Observations are summarized in Section 7. Section 8 concludes the paper.

2. Problem Statement

VRP can be simplified to a well-known problem, the traveling salesman problem [3,4]. Because the TSP is NP-complete, the VRP is also NP-complete [3], and therefore cannot be computed over a given input length with the resources available to humanity due to its excessive runtime.

TSP can mathematically be described as the following: Given a complete weighted graph \( G(V, E) \) with \( V \) set of vertices and \( E \) set of edges, and a weight function \( f(e) \rightarrow c \) (where \( c \in \mathbb{R} \) and \( c > 0 \)). We are looking for a minimal weight Hamiltonian cycle of \( G \) with \( H \) edges (where \( \forall h \in H : h \in E \) and weight is calculated as \( \sum_{h \in H} f(h) \)). Here, \( G \) is also called a distance graph, where elements of \( V \) are locations and elements of \( E \) are the length of routes between them. The Hamiltonian cycle is also referred to as the trip. In the allegory of the problem, a salesman wants to start a trip to visit all his customers. Each customer is at a different location, and the trip ends by returning to the start position. The salesman wants to minimize the length of the trip, but the trip can be completed \( n! \) ways in case of \( n \) customers. The order of service of the customers can be represented as a permutation. The possible permutations can be bijectively matched with the possible directed Hamiltonian cycles. This means that there are twice as many permutations as non-directed Hamiltonian cycles (except at \( n = 1 \) and \( n = 0 \)). However, if we found an optimal permutation, its reverse is also optimal.

TSP and VRP have many variants. Here we present a broad overview of the most common directions. We are not going to describe all in detail and with precise mathematical models because most variants have too many interpretations for the scope of this publication. We list the variants to show the number of possibilities, and to use as a reference
in the description of the used datasets. These variants can be mixed with more complex problems including our real-life model, which includes all listed features.

The distance graph can be directional, meaning the problem is asymmetric (ATSP). The solution set to be examined is twice as large because the reverse of the serving order can result in a different cost.

Moreover, there can be multiple salesmen (the problem is multi-agent, MTSP). This means that instead of a Hamiltonian cycle, we are looking for as many cycles as there are agents. Each customer’s location is touched by exactly one cycle, and the starting location is touched by all cycles. Here, the weight to minimize can be the sum of the weights of the cycles or the maximum of the weights of the cycles. By default, we use the first interpretation. Here a bijective match between solutions and permutations is no longer possible. This makes the application of optimizations originally designed for TSP nontrivial. There are methods by which to use permutations as a representation of solutions, but a! permutations will refer to the same solution, where a is the number of agents.

Furthermore, there can be multiple start locations, where each agent starts its trip from a different one. These variants are referred to as multi-depot multi-agent variants (MDMTSP). Here, each cycle must cover exactly one depot. There are variants, in which agents can finish trips in different depots than those from which they started, but this is an unusual twist on the MDMTSP. In this paper, we work only with single-depot variants.

The main difference between TSP and VRP is the allegory behind the problems. In VRP, we have vehicles instead of salesman and sites instead of start locations. Most VRP variants are dealing with real-life problems like the capacity of vehicles or the cost of fuel. The weight of edges rarely represents distance; they are more likely to represent time or direct money. All VRP variants are multi-agent and they are usually asymmetric.

The vehicles or transport units can have many kinds of capacities (the problem is capacitated, CVRP). In the most common CVRP variant, we have a demand function \( g(v) \rightarrow d \) (where \( v \in V \) of \( G(V,E) \) and \( d \in \mathbb{R} \)) and a capacity function \( h(i) \rightarrow ca \) (where \( i \) is the index of the vehicle and \( ca \in \mathbb{R} \)). For each vehicle \( tu_i \) in \( TU \) (as transport unit, where \( i \) is the index of the vehicle) and their trip \( T_i \), \( \sum_{v \in V \cap T_i} g(v) < h(i) \). If a trip does not satisfy this restriction, its weight (or cost) is infinite or punished by additional cost. However, in the case of possible trips, the cost is still the sum of the weight of the edges of the trips, while demand is matched with the vertices.

A VRP can have multiple capacities (multi-capacitated, MCVRP) with their own demand values and restrictions. These capacities can also have other sources than the edges. For example, the transport unit might be limited by the work time of the driver. Time can be spent by travel, but also by unloading the goods or filling the fuel tank of the vehicle at a nearby gas station. The vehicle might be limited by its gas tank if it must complete the trip without refilling. This means the demand can be derived from the whole trip not only from the vertices. If the fleet is not homogeneous (it contains multiple types of vehicles), then the optimal route between two locations may vary based on the type of vehicle. They might have different fuel consumption profiles, they might be forbidden from some of the roads, etc. Having different optimal routes means not just different costs, but different demands.

A VRP may be also time-dependent, meaning the cost and demand on a route for the same vehicle can vary based on the time of start. This is usually solved by slicing the day of transport into multiple time slots and having a parallel edge between two customers for each slot. This means the change in cost caused by local changes on a plan is harder to calculate, usually, and that the whole trip must be recalculated.

In most sophisticated models of real-life transport tasks, the edges of the graph represent all statistics of the route between two customers for each vehicle, time slots,
with or without refilling the tank. The cost function becomes a simulation, which can contain polynomial time optimizations in itself. The simulation we use in the real-life example is relatively simple, so it runs in \( O(c) \) time for a delivery plan, where \( c \) is the number of customers. However, most realistic simulations have a complexity of \( O(c^2) \), which increases the complexity of all algorithms by using it as a cost function.

3. Related Works

There are two groups of heuristics for TSP and VRP: solution construction algorithms and iterative optimizations. Solution construction algorithms select edges of the graph to the solution until a complete solution is built. Iterative optimizations take an initial complete solution and use iterative methods to improve upon it. The initial trip might be generated randomly, a result of a trip construction algorithm or some other methods specific to that type of optimization. In most cases, we have a complete solution after each iteration, even if in the case of CVRP the solution might be impossible to execute.

Most heuristics we present are designed for TSP. There are multiple solutions to adapt for MTSP, VRP, and CVRP. We present multiple approaches for solving the issue; however, detailed analyses of each is beyond the scope of this publication.

a. The solution for TSP might be cut into multiple trips, meaning edges of the Hamiltonian cycle are replaced by an edge to the start location and another from the start location. In the case of MTSP or regular VRP, a cut should be made only if it improves the cost of the solution. In the case of regular CVRP, we want to make cuts that add less cost and create possible trips. These are tasks that require their own optimization algorithms and in the case of CVRP the task is NP-complete.

b. Another solution would be to cut the start location into multiple instances with edges with zero weight between them. This means in the case of \( n \) agents \( n \) representations of the starting location are added. This way the task is still to build a minimal-weight Hamiltonian cycle in which the representations of the start location break the trip into multiple pieces. The problem with this solution is that almost all greedy algorithms will use only one agent because after selecting the first start location the next cheapest edge will lead back to the starting location. We could solve this problem by having edges with a weight of infinity, but that would result in the usage of all agents, which is rarely an optimal solution. In the case of CVRP, it is important to keep track of the consumed capacity separately for each slice of the trip.

c. A common solution is to apply clusters separating the customer locations into regions. There is a cluster for each agent and the start location belongs to each cluster. Each agent runs the heuristic in its own cluster building a Hamiltonian cycle on the subgraph. Finding the optimal clustering is also NP-hard, meaning the optimal solution is almost impossible.

d. In most advanced solutions, we add complex multi-agent negotiation logic to the algorithms. For example, the agents might mark their next target and resolve conflicts of common targets before selection, or there might be a global logic that selects the next agent that can make a decision.

Clusterization is the most popular because it cuts the problem into many smaller sizes. In real-life applications, speed is one of the most crucial limitations, making clustering attractive. However, finding the optimal division of locations would be as hard as solving the problem without clusters making the solution suboptimal. One of the most important tasks of modern research is to decrease algorithm complexity to make heuristics applicable for problem sizes of a thousand clusters. Our solution has a complexity of \( O(n^{2.5}) \) per iteration, eliminating the need for clustering.

In the following sections, \( n \) refers to the number of vertices in the distance graph.

3.1. Solution Construction Algorithms

- **Nearest neighbor**: The nearest neighbor heuristic is a greedy trip construction algorithm. It starts with the nearest location to the start location and always selects the
nearest to the last location that is not used yet. The result is usually far from optimal because the optimal trip might require the use of more costly edges near $o$ at the beginning of the trip. The algorithm is very fast, having an $O(n^2)$ runtime.

- **Cost-based probability:** This is the same as the nearest neighbor, but some probability for selection is distributed between others not selected as neighbors too. First, a weight function $f(d) \rightarrow w$ (where $d$ is the distance of the neighbor and $w$ is the weight) is applied for each neighbor. This function’s first determinant is negative for all $d$ values (where $d \in \mathbb{R}$ and $d \geq 0$). The choice is random, and each location gains probability proportional to its weight. The heuristic may run multiple times taking the best result. The algorithm has an $O(n^2)$ runtime but is slower than the nearest neighbor due to weight calculation.

- **Greedy edge selection:** This sorts the edges by cost to reduce complexity. It always takes the next edge of the array unless it would result in a Hamiltonian cycle. The sorting has an $O(n^2 \cdot \log(c))$ runtime, and the rest can be completed in $O(n^2)$, because checking for the Hamiltonian cycle has an $O(1)$ runtime implementation.

- **Insertion heuristic:** This starts with the shortest edge as a cycle on two vertices (in the case of ATSP, it would select both edges). In each iteration, a vertex will be added by breaking an edge in the cycle and connecting the new vertex to the vertices of the original edge. It checks each vertex not in the cycle and each edge in the cycle for the added cost. It adds the vertex and edge that results in the least-added cost completing the procedure. It continues until all vertices are added to the cycle. The generation takes $O(n^3)$ runtime, because in case of a cycle with $m$ edges and vertices $m \cdot (n - m)$ checks must be made and the limit of $m$ is $n$.

- **Advanced insertion heuristic:** This starts with the convex hull of the locations as a tour. Calculating the convex hull takes $O(n^3)$ runtime. It runs the insertion heuristic.

- **Christofides:** It finds a minimal spanning tree on the weighted complete graph. This takes $O(n^2 \cdot \log(n))$, because a greedy process finds the optimum in $O(n^2)$, but sorting of the edges is necessary. It takes the vertices of the tree with an odd degree, $O$. It also takes the complete subgraph of the original complete graph defined by the vertices in $O, GO$. It finds a minimum weight perfect matching on $GO$ by adding the edges of the matching to the tree. After merging the two graphs all vertices have an even degree. It finds an Euler cycle on this graph and removes duplicate vertices from the Euler cycle until a Hamiltonian cycle is generated. Generation takes $O(n^2 \cdot \log(n))$ runtime. A better result can be achieved by applying heuristics for the selection of vertices to eliminate. For example, we can calculate the additional cost of removal for each duplicate and select the one with the least added cost. Each iteration of this heuristic takes $O(n)$ time because the Euler cycle has only $O(n)$ vertices, making the reduction an $O(n^2)$ runtime algorithm.

### 3.2. Iterative Optimizations

The 2-Opt algorithm is a neighborhood search (or local search) algorithm. It takes a set of elementary modifications as operators over a solution and considers two solutions neighbors if they can be generated from each other by using any of the operators. It starts from a solution and applies all operators, reverting all changes that result in solutions with worse costs. If it finds a better solution, it tries each operator again; otherwise, it stops. A local search is defined by the operators it uses, which is usually the same operator with different parameters. We are presenting variants for the 2-Opt algorithm, but these variants can be generated for all local searches by using the same logic. The limitation of local searches is the same.

- The smaller the complexity of the operator, the smaller improvement can be made in the iteration.
- It rushes into a local minimum, almost never finding the optimal solution.
• On real-life models, the recalculation of the cost of a solution is necessary for increasing complexity compared to usage on TSP.
   Let us introduce some of the most known iterative optimizations.
• 2-Opt [5]: It takes a Hamiltonian cycle on the weighted complete graph. It iterates through all two edges of the cycle and inverts the order of vertices between the two edges. If the new cycle is better, it is kept; if not, it reverts to the last modification. It continues the iteration and repeats until no two edges result in improvement. One iteration takes \( O(n^2) \) runtime because all two edges are checked. The cost recalculation is not necessary because the cost difference comes only from the two removed and two added edges. The required runtime is greater in case of asymmetric problems: \( O(n^3) \) because cost must be recalculated for the whole section reverted.
• Simplified 2-Opt: As in 2-Opt, both vertices are checked in an iteration and stopped if the iteration results in no improvement. However, instead of inverting the section, only the two vertices are swapped. This variant requires less runtime if the edge direction matters or an array data structure is used.
• 3-Opt: This is the same as 2-Opt, but three edges are selected and the inversion of all three cycle segments is tested. It takes at least \( O(n^3) \) and \( O(n^4) \) in case of asymmetric problems. A simplified version can be also used, such as the simplified 2-Opt, in which all iterations are tested.
• k-Opt [6]: This is the same as 2-Opt, but \( k \) edges are selected and the inversion of all \( k \) cycle segments are tested (see Figure 1). It takes at least \( O(n^k) \) runtime. It can be improved by using a low \( k \) (like \( k = 2 \) value), and only increasing it if no better solutions are found in the iteration. If any improvement is found by higher than \( k = 2 \), it decrements \( k \). A simplified version can be also used, like the simplified 2-Opt and 3-Opt.

![K-Opt Algorithm Family](image_url)

**Figure 1.** K-Opt Algorithm Family. Here, the dashed lines represent the new edges of the cycle created by the different operators.

• Lin-Kernighan [7]: This is the same as adaptive k-Opt, but a complex calculation is added to determine the correct \( k \) necessary to improve in each iteration. The calculation is quite complex, but in the case of TSP, the found local optimum will be only two percent worse than the optimum itself. However, the necessary iteration count is unknown, and one iteration takes \( O(n^{2k}) \) runtime. The algorithm also has worse results on ATSP and CVRP, because the \( k \) value selection logic is hard to adapt.
• Tabu-Search [8]: Most neighborhood searches stop when they do not find more improvement. The Tabu-search improves on this by allowing changes with negative gain in these situations. The changes continue randomly until a better position is found. Because some modifications invert each other, the algorithm lists the modifications that invert earlier ones. For example, because switching two values twice results in their original order, the algorithm bans switching of already switched values. The usage of the list could increase the time required to \( O(n^3) \) per iteration for simplified 2-Opt moves, but by using clever implementation with \( O(n^2) \) memory requirement this can be reduced to \( O(n^2) \).
• Simulated Annealing [9]: Simulated annealing is an improvement on neighborhood searches like Tabu-search. However, instead of using a list, it uses probabilistic deci-
sions. It allows moves with a negative gain with a low probability and by adjusting this probability over time it can avoid local minima. Because it does not have a stop condition, it can run at any time. The results of more advanced versions are comparable to those of Lin–Kernighan; however, convergence is very slow due to the negative gain steps.

- **Branch & Bound [10]:** It uses a solution construction algorithm in each iteration. It iterates all possible solutions searching for the optimum, but in an almost optimal order. It constructs a solution by adding vertices to the end of the trip. During construction, a decision tree is built containing the sum weight of the subtours. This decision tree helps eliminate all routes that would result in a longer solution than the best solution so far. If there is no subtour left in consideration, the algorithm stops and the optimum is guaranteed. The choice of one subtour should be interpreted as one iteration of the algorithm. This iteration takes $O(c^3)$ time because of the graph manipulation. The construction of the first tour takes $O(c^4)$ time; however, that time requirement decreases because their subtour is already considered. By eliminating branches strategically, the memory usage can be $O(n^2)$; however, it still can take exponential time to find the best solution.

- **Ant Colony Optimization (ACO) [11,12]:** Multiple agents are walking the cost graph after each other. Each leaves a pheromone in each visited node, meaning a constant value marking the quality of the node. Later, agents choose vertices with higher pheromone values with a bigger probability while increasing the values themselves. The pheromones are also decreased in each iteration, aging old assumptions. This process continues, and ants are finding better and better solutions in time. One iteration takes $O(n^3)$, meaning it is comparable to genetic algorithms. It can be applied to MTSP and VRP by starting multiple agents in one iteration, always moving the one with the lowest subtour. These take $O(n^4)$ runtime per iteration since the ant with the lowest time must be found.

- **Improved Intelligent Water Drops Algorithm (IIWD) [13]:** This is a multi-agent algorithm built for the behavior of water droplets. Each agent is called a water droplet, which runs through the nodes of the graph. First, a solution is generated, then knowledge is stored about the solution to generate a better solution in the next iteration. The knowledge is stored in so-called dynamic parameters representing the potential of already walked paths. Velocity is a parameter of the droplets representing their remaining capacity, while soil represents the difficulty of each path. The main difference between IIWD and ACO is that IIWD stores information in the edges of the graph, while ACO stores data in nodes. They also have a different mechanism for information updates.

- **Advanced Cuckoo Search [13]:** This is based on the cuckoo bird’s egg-laying behavior. Each nest is a solution set, and eggs are the solution. A nest is randomly selected to be the cuckoo nest. From the nest, a solution is generated by the random discovery of solution space. The new solution is added to another nest. The selection is done to drive convergence. New nests are generated and the process is repeated.

4. Heuristics Typology

The idea of evolutionary algorithms (see [14]) is to treat the solution set as if it were a species. Then the solutions tracked simultaneously are treated as a population of that species. In each iteration, we perform a selection on the population, in which a part of it is eliminated. Here we give a higher survival rate to the lower-cost solutions. The second step of the iteration is a mutation, where new instances are formed and existing ones are randomly modified. The selection step ensures that the process converges to an optimum. If the population does not die out of the elementary traits that make up the optimal instance, it has a good chance of finding it. It is just a matter of time.

Genetic algorithms (see Figure 2) (see [15,16]) are the most common class of evolutionary algorithms. During selection, we halve the population. In the crossover step (see
Figure 3), new specimens are generated from the retained instances by combining their properties, and the selected instances are called parents, while the generated ones are called their children. The new instances are likely to have a cost similar to their parents because they are created from their properties. In the standard case, two new instances are generated for every two instances, doubling the population. If only one child is generated, we may call the process a half-crossover. After crossover, the mutation is applied to only some of the children as the last step of the iteration. The biggest problem with genetic algorithms is their speed. Because, like selection, crossover and mutation are random processes, improvement is less and less likely to occur. This is why evolutionary algorithms are usually supplemented with local search, which ensures constant convergence. Such augmented evolutionary algorithms are called memetic algorithms.

Figure 2. Genetic algorithm.

Figure 3. Crossover step of genetic algorithm. We marked the elements inherited from the primary parent by underlining the values. We only demonstrate the basic idea, there are hundreds of other known crossover operators.

A new branch of memetic algorithms is the bacterial memetic algorithm (see Figure 4 and [17–19]), which are the memetic version of bacterial evolutionary algorithms (see [20]). This algorithm type removes the selection step and overrides mutation and crossover with its own variants. During mutation, the mutated specimen is modified separately. In GAs, this modification is small, and its goal is to bring back extinct properties and find unexpected solutions. In a BMA, the bacterial mutation’s goal is to make a pseudo-global search on the specimen. The mutation step consists of cycles. In each mutation cycle, a segment of the mutant is selected (not necessarily a continuous segment) and cloned. The mutation operator is used on each clone on the given segment. After mutation, the original specimen is overwritten with one of the clones or left alone. Usually, the specimen is overwritten by the best clone if it is better than the original. In this step, we can find a specimen very far from the original one. The bigger the segment, the further the search reaches and the more clone is created, the bigger is the chance to find a better solution in a cycle. The crossover step is also replaced by the gene transfer step, in which segments are injected from one specimen to another. Usually, the genes are transferred from a better specimen to a worse one. The main problem with the original BMA design is that the mutation step can underperform. The gene transfer task is to signal better search positions for the left-behind
specimen, whereas mutation and local search are responsible for convergence. If a mutation is underperforming, we are basically left with local search and a lot of overhead.

Figure 4. Bacterial memetic algorithm.

5. Proposed Algorithm

The algorithm we use is also a bacterial memetic algorithm (see Figure 4). Bacterial evolutionary algorithms (see [21]) are more modern than genetic algorithms but not nearly as widespread. Their emergence represents not only a further development of genetics but also a change of approach. As mentioned (see Section 4), in bacterial evolutionary algorithms the selection step disappears, the crossover is replaced by gene transfer, and the mutation step is replaced by bacterial mutation.

First, the algorithm parameters are gathered and the initial population is initialized (see Algorithm 1). Here we used a modulo-stepper approach with a bigger population size, resulting in great diversity. Then we applied bacterial mutation to each specimen and selected the ones that improved the most, taking them as the ones with the most potential. After that, the cost of each specimen is calculated and the population is ordered by cost. Because the population is sorted, statistics gathering is easy, which is done before each iteration of the optimization phase. In one iteration of the optimization first, the local search is applied to one or more specimens, but only if no improvement was achieved in the previous iteration. This is a simplified version of the 2-Opt (see [5]) algorithm. It is followed by gene transfer, mutation, and sorting of the population. The cost of a new specimen is calculated lazily when it is necessary. Each step of the iteration gathers its own statistics, but they are logged only at the end of the iteration. The algorithm runs until the number of cost calculation exceeds the budget limit. This approach is selected because cost calculation can be an entire simulation of the transport trip, making it very resource-consuming. This makes algorithms with different complexity levels comparable without being restricted to the exact environment specification.

The steps are described in more detail below. For describing them, the introduction of the data format is necessary. In our implementation, we used the so-called one chromosome representation of the permutation and assignment as illustrated in Figure 5. Basically, instead of cutting a permutation of length $c$ into $s$ segments (representing the routs of the $s$ vehicles), we inserted breakpoints into the permutation, numbered from $c$ increasing. This results in a large $c + s - 1$ long permutation. However, in this case multiple representations (up to $s!$) are corresponding to a single transport plan, which makes data handling far easier. The individual breakpoints could also correspond to the vehicle the customers of the trip are served by. The question arises as to what is considered an elementary trait and what is considered a gene. By default, we would use the elements of the permutation, but it is also a valid observation that it is more important to know which elements directly follow which than their position in the serving order. This is an important difference in approach since it redefines which specimens are similar to each other. If the order of trips within the permutation of the specimen is randomized, we still handle it as almost the same specimen.
Algorithm 1 Improved BMA Algorithm

Parameters:
- \( \text{count}_{\text{vehicle}} \) is the number of vehicles
- \( \text{count}_{\text{customer}} \) is the number of customers
- \( \text{Graph} \) is the graph of demands and costs
- \( \text{size}_{\text{population}} \) is the size of the population
- \( \text{budget}_{\text{max}} \) is the upper limit in number of cost calculations
- \( \text{step}_{\text{max}} \) is the limit of 2-opt in steps

Program:

Initiate \( \text{population}_{\text{big}} \) with size \( \text{count}_{\text{vehicle}} + \text{count}_{\text{customer}} - 1 \) using modulo stepper method

Initiate \( \text{budget}_{\text{spent}} \) to 0

Mutate each specimen in \( \text{population}_{\text{big}} \) and increase \( \text{budget}_{\text{spent}} \)

Order \( \text{population}_{\text{big}} \) descending by improvement percentage during mutation

Initiate \( \text{population} \) by first \( \text{size}_{\text{population}} \) specimen of \( \text{population}_{\text{big}} \)

Initiate \( \text{cost}_{\text{best}} \) to infinity

For Until \( \text{budget}_{\text{spent}} \) is less than \( \text{budget}_{\text{max}} \) do

- If cost of \( \text{population}[0] \) is \( \text{cost}_{\text{best}} \) then
  - Apply local search to \( \text{population}[0] \) and increase \( \text{budget}_{\text{spent}} \)
  - Set \( \text{cost}_{\text{best}} \) to cost of \( \text{population}[0] \)

- Apply gene transfer to \( \text{population} \) and increase \( \text{budget}_{\text{spent}} \)

- Apply mutation to selected specimen of \( \text{population} \) and increase \( \text{budget}_{\text{spent}} \)

Order \( \text{population} \) by cost

Figure 5. Chromosome representation. In the first representation, the end of segments are represented by wider separator lines. In the second representation, star symbols represent the inserted break points. Break points are replaced by numbers in the final design.

During population initialization, the modulo-stepper method is used (see Algorithm 2). This means that we take a random \( c + s - 1 \) long permutation as a base permutation. The first specimen is generated by taking the first element of the permutation and then walking the array in one long step. The second specimen is generated by taking the first position and then making steps of length two, etc. Every gene touched is gathered into the current permutation’s next empty position and every position that was already touched is stepped over by one. This results in a \( c + s - 2 \) specimen with maximal distances and an almost even distribution of genes. If more than a \( c + s - 2 \) specimen is needed, a new base permutation can be generated and the process can be repeated again. Usually, the used population for BMA is much smaller than \( c + s - 2 \) which means the population contains only a fraction of possible genes. That is why we decided to generate a \( c + s - 2 \) population and select the best ones for the final population. The specimen could be chosen based on cost, but the potential for cost reduction is a better indicator of long-term fast convergence. To measure the potential of cost reduction on each specimen, we ran the bacterial mutation on each and selected the ones with the highest reduction percentage.
Algorithm 2 Modulo stepper initialization

Parameters:
- \(\text{size}_{\text{population}}\) is the size of the population
- \(\text{size}_{\text{permutation}}\) is the size of the permutation of specimen
- \(\text{population}\) array of specimen with size of \(\text{size}_{\text{population}}\)

Program:

\[
\text{Init } \text{population}_{\text{sliced}} \text{ with } \text{population} \text{ cut into } \text{size}_{\text{permutation}} - 1 \text{ sized slices}
\]

\[
\text{for each slice of } \text{population}_{\text{sliced}} \text{ do}
\]

\[
\text{for index}_{\text{permutation}} \text{ in 0 until size of slice do}
\]

\[
\text{Init size}_{\text{step}} \text{ to index}_{\text{permutation}} + 1
\]

\[
\text{Init permutation}_{\text{base}} \text{ to random permutation with size of size}_{\text{permutation}}
\]

\[
\text{Init permutation}_{\text{new}} \text{ array of negatives with size of size}_{\text{permutation}}
\]

\[
\text{Init index}_{\text{read}} \text{ to size}_{\text{step}}
\]

\[
\text{for index}_{\text{gene}} \text{ in 0 until sizes}_{\text{permutation}} \text{ do}
\]

\[
\text{if permutation}_{\text{new}} \text{ already contains permutation}_{\text{base}}[\text{index}_{\text{read}}] \text{ then}
\]

\[
\text{Increment index}_{\text{read}} \text{ modulo size}_{\text{permutation}}
\]

\[
\text{Set permutation}_{\text{new}}[\text{index}_{\text{gene}}] \text{ to permutation}_{\text{base}}[\text{index}_{\text{read}}]
\]

\[
\text{Increase index}_{\text{read}} \text{ by size}_{\text{step}} \text{ modulo size}_{\text{permutation}}
\]

\[
\text{Set permutation of slice}_{\text{index}_{\text{permutation}}} \text{ to permutation}_{\text{new}}
\]

In the local search (also called boost) step, we iterate over all two elements of the selected permutations. We swap the contents of the two positions, evaluate the instance, and if the cost is better, overwrite the original instance. In the original 2-Opt (see Section 3.2) (see citation [5]) algorithm, the section between the two positions is inverted to generate a neighboring solution, but the cost of this was too high, so we opted for a simpler solution. In addition, the original algorithm iterates over and over every two positions until it finds a better solution. This takes \((c + s - 1) \cdot (c + s - 2)\) budget per iteration, making it too expensive. Instead, we limit the budget to \((c + s - 1)\) or another custom value and continue the iteration from the last position pair on the next call. If the local search found the first improvement, it exists, and it will not rush to the local minimum. The iteration order of positions is also randomized, so the optimization is more distributed instead of optimizing the start of the permutation first. If the local search touched all two positions but failed to improve on the permutation, it will skip running until other steps improved on the specimen, meaning it left the local minima. A lot of specimen selection logic was tested, but in the end, we settled on the lazy on the best approach. This means the specimen with the lowest cost is optimized, but only if the specimen did not improve since the last optimization. This means boost is run only if the other methods for optimization failed.

In gene transfer, we also need a selection and operator logic (see Algorithm 3). Selection needs to make as many pairs of specimen and as many transfers between pairs as required. One specimen can take part in multiple transfers, but it cannot be both the donor and acceptor of the segment in a transfer. During operation, the donor must not change, and the acceptor’s modification is also optional. We chose the tournament selection method, whereby pairs are generated randomly but the genes are transferred from better specimen to worse. To make parallelization possible, we randomized by shuffling the population and cutting it into two long segments. This minimizes the possibility of conflicting locks on specimens. We also removed the original operator and replaced it with a heuristic crossover (see Algorithm 4) (see citation [22]) operator. Here, we generate a child from the two specimens and overwrite the worse specimen with the child. This novel operator can be viewed as a feedbacked half crossover. The overwrite only happens if the child has a better cost than its parent. During heuristic crossover, the child starts with a random element. After this, the next position is always filled by using the following logic. First, the neighboring values of the value are gathered from the parents. Next, the values already present in the children are removed from the list. If the list is empty, then a random value
not present in the children is selected. If the list is not empty, a weight is calculated for all neighbors. The weight in our case is the multiplicative inverse of an estimation for the additional cost of the selection of the value as the next value. If the child is complete, its cost is calculated. Other crossovers might be also used, but during our research, we found that heuristic crossover has the best convergence speed.

**Algorithm 3** Gene transfer with feedbacked half heuristic crossover

Parameters:
- `population` is the array of specimen gene transfer is applied to
- `count_injection` is the number of injections per gene transfer

Program:

```plaintext
for index_injection in 0 until count_injection do
    Init specimen_selected with pair of distinct specimen from population at random
    Init specimen_donor with specimen with lower cost in specimen_selected
    Init specimen_acceptor with specimen with higher cost in specimen_selected
    Init specimen_child with heuristic crossover of specimen_selected
    Calculate cost of specimen_child
    if cost of specimen_child is less then cost of specimen_acceptor then
        Write permutation of specimen_child to specimen_acceptor
```

**Algorithm 4** Heuristic crossover with one child produced

Parameters:
- `parents` is a pair of specimens the crossover is applied to
- `size_permutation` is the size of permutation of specimen
- `Graph` is the graph of demands and costs

Program:

```plaintext
Init permutation_child with an array of negatives with size_permutation
Set permutation_child[0] to random between 0 and size_permutation
for index_gene in 1 until size_permutation do
    Init value_previous to last not negative element of permutation_child
    Init value_neighbours to elements preceding and following value_previous in permutations of parents
    Remove values of value_neighbours that permutation_child already contains
    if value_neighbours is empty then
        Set value_missing to array of values between 0 and size_permutation
        Remove elements of value_missing already present in permutation_child
        Set permutation_child[index_gene] to randomly selected value of value_missing
    else
        Init vertex_previous with vertex associated by value_previous
        Init vertex_neighbours with vertices associated by value_neighbours
        Init edges_neighbours with array of edges from vertex_previous to vertex_neighbours
        Init cost_neighbours with cost estimates of edges of edges_neighbours
        Init weight_neighbours with weights calculated from costs of cost_neighbours by weight
        function
        Init weight_sum with sum of weights of weight_neighbours
        Init random with random number between 0 and weight_sum
        Init index_selected to 0
        for index_neighbour in 0 until size of value_neighbours and random > 0 do
            Decrease random by weight_neighbours[index_neighbour]
            if random < 0 then
                Set index_selected to index_neighbour
            Set permutation_child[index_gene] to value_neighbours[index_selected]
        Return permutation_child as specimen
```
During the mutation, all specimens might be mutated (see Algorithm 5). However, because it is the most budget-consuming step of the algorithm we suggest running it only on the best specimen in case of big populations. We use continuous segments because the edges are the elementary properties and not the positions of the values. In each cycle, we select a random position for the start of the segment. On each mutated specimen, a given number of clones are generated. One of the clones contains the same segment in a reverse order to have a similar cost clone with maximal distance.

Algorithm 5 Bacterial mutation on a specimen

Parameters:
specimen is the specimen selected for mutation
length_segment is the length of the segment mutated by mutation operator
count_cycle is the number of mutation rounds per specimen per iteration
count_clone is the number of clones
index_generation is the generation index

Program:
for index_cycle in 0 until count_cycle do
    Init index_selected using index_generation and index_cycle
    Select continuous segment of permutation with size length_segment from index_selected
    Init clones with count_clone + 1 clone of specimen
    Apply mutation operator to each clone in clones except one
    Calc cost of each clone in clones
    Order clones by cost
    Set permutation of specimen to permutation of clones[0]

For the other clones, an edge builder heuristic is used to increase improvement probability (see Algorithm 6, Figure 6). During this heuristic, all elements of the segment are selected with the complete graph of routes between the nodes referred to by the values. We add the edges from the preceding value and the edges to the following value to this complete graph. By merging the two nodes, we receive one complete graph derived from the network graph (distance graph in simple VRP). We rebuild the segment by selecting the edges of the complete graph until a Hamiltonian cycle is built. We ensure that a Hamiltonian cycle is built by denying the selection of edges that would complete a cycle until a Hamiltonian path is built. We weight the selection of allowed edges by an estimate of the cost their selection would add. This process is $O(s^3)$ where $s$ is the length of the segment mutated. Since the optimal size of $s$ is not dependent on the size of the problem, this does not increase the complexity of the algorithm.

Algorithm 6 Mutation operator with edge builder heuristics

Parameters:
specimen is the specimen the operator is applied to
segment is the segment of the permutation the operator is applied to
Graph is the graph of demands and costs

Program:
Init vertices_segment with array of vertices of Graph the elements of segment refer to
Init edges_inner with matrix of edges between elements of vertices_segment
Init cost_inner with matrix of cost estimates calculated from edges of edges_inner
Init weights_inner with matrix of weights calculated from elements of cost_inner using weight function

Find vertex_previous the vertex descending the first vertex of segment in trip
Algorithm 6 Cont.

Init $edges_{previous}$ with array of edges from $vertex_{previous}$ to elements of $vertices_{segment}$
Init $cost_{previous}$ with array of cost estimates calculated from edges of $edges_{previous}$
Init $weights_{previous}$ with array of weights calculated from elements of $cost_{previous}$ using weight function

Find $vertex_{next}$ the vertex following the last vertex of $segment$ in trip
Init $edges_{next}$ with array of edges from elements of $vertices_{segment}$ to $vertex_{next}$
Init $cost_{next}$ with array of cost estimates calculated from edges of $edges_{previous}$
Init $weights_{next}$ with array of weights calculated from elements of $cost_{next}$ using weight function

Init $weights_{united}$ to matrix from union of $weights_{previous}$ vector, $weights_{next}$ vector and $weights_{inner}$ matrix
Consider $vertex_{previous}$ and $vertex_{next}$ the same vertex in the following steps
Elements of $weights_{united}$ are decode-able to edges of $Graph$ using their position

Init $weights_{final}$ with square matrix with same size as $weights_{united}$
for each position in $weights_{united}$ do
  if position is in diagonal of $weights_{united}$ then
    Skip position
  Init $weight$ with element of $weights_{united}$ in position
  Init $weight_{inverse}$ element of $weights_{united}$ in position mirrored to diagonal
  Init $weights_{row}$ with other elements of $weights_{united}$ in same row
  Init $avg_{row}$ with average of $weights_{row}$
  Init $weights_{column}$ with other elements of $weights_{united}$ in same column
  Init $avg_{column}$ with average of $weights_{column}$
  Set $weight_{final}$ in same position to $weight/(weight_{inverse} + avg_{row} + avg_{column})$

Init $positions_{selected}$ the array of selected positions
for $length_{segment} - 1$ do
  Init $weight_{sum}$ to sum of elements of $weights_{final}$
  Init $random$ to random number between 0 and $weight_{sum}$
  for each $weight$ in $weights_{final}$ until $random > 0$ do
    Decrease $random$ by $weight$
    if $random < 0$ then
      Add position of $weight$ to $positions_{selected}$
  for each $weight$ in $weights_{final}$ until $random > 0$ do
    if selecting position of $weights$ would lead to $positions_{selected}$ referring to a cycle then
      Set $weight$ to 0
Find $position$ that would complete $positions_{selected}$ to maximal size cycle
Add $position$ to $positions_{selected}$
Decode $positions_{selected}$ to edges $edges_{selected}$
Decode $edges_{selected}$ to sequential representation of $segment$, $segment_{sequential}$
Decode $segment_{sequential}$ to new order of elements of $segment$
Write elements of $segment$ back to permutation of $specimen$ in new order

We estimate and confirm that the algorithm has a running time of $O((c + s)^{3.5})$. True, this is strongly influenced by parameterization and convergence speed. Because the complexity depends on the number of customers and vehicles, the parameters are chosen accordingly. $O(\sqrt[3]{c + s})$ is the population size. In gene transfer, $O(\sqrt[3]{c + s})$ is the injection count. During mutation, the number of clones is $O(1)$, just like the length of the mutated sequence, $O(\sqrt[3]{c + s})$ is the number of clones selected for mutation, and $c + s$ is the cycle count. Tests were run with $O(c + s)$ iterations. The initialization takes $O((c + s)^2)$ time, because each instance is completed in $O(c + s)$ time and mutated in $O((c + s)^2)$. The gene
transfer takes $O((c + s)^{1.5})$ time because the injection count is proportional to population size, and cost calculation even in the simplest of cases takes $O(c + s)$ time. Bacterial mutation takes more, $O((c + s)^{2.5})$ processor time, as it makes clones for each element of the population, evaluates them, and repeats this $O(c + s)$ times. The local search is limited to $O(c + s)$ iterations, so it takes $O((c + s)^2)$ because specimen evaluation takes $O(c + s)$. This implies a run time of $O((c + s)^{2.5})$, making the runtime of each iteration $O((c + s)^{2.5})$. Finally, this is multiplied by the number of iterations.

**Figure 6.** Bacterial mutation. The values of the selected segment in each specimen are underlined. The white arrow represents that the original specimen is overwritten by the best clone.

### 6. Experiments and Results

In this section, we propose results on three datasets to provide multiple important perspectives. In Section 6.2, we even compare our method with some of the most-used methods for VRP. The comparisons include more classical solutions like a genetic algorithm or ant colony optimization, but also some of the most recent methods like the coronavirus herd immunity optimizer or the enhanced firefly algorithm. The complete version of our algorithm produces 20% better results at its worst while producing 60% better scores at its best. In most cases, the improvement is between 30% and 50%, meaning that our algorithm reliably produces better results than any other compared alternative.

#### 6.1. Datasets

Our datasets had to meet multiple criteria to make optimizations over them proper proof of efficiency. The dataset had to be public so experiments can be repeated by others. They had to be used in recent publications, making comparisons with those results possible. They had to provide tasks with over 1000 customers, making resource consumption tests possible. They had to be detailed real-life data, making more complex cost functions possible. We did not succeed in finding any dataset matching all these criteria. Because the search for the specific dataset failed, we decided to run our experiments on three distinct datasets with different purposes. One was public for comparison with the most recent results, one was public with bigger tasks for future comparisons, and one was with detailed enough data.

For comparison with the most recent results, we used the ABEFMP (see [23]) 1995 dataset. This dataset was used in a publication publishing a new evolutionary algorithm in 2022 (see [24]). This is a symmetric CVRP dataset with Euclidean distances between the coordinates of the locations. For further experiments, we used the Belgium 2017 dataset (see [25]), which contains tasks with sizes right to 2750 customers. Because the task is an asymmetric VRP with bigger problem size, future comparisons will be more relevant to this dataset. Finally, we used a custom dataset with multiple capacities of multiple kinds, asymmetric distances, and detailed vehicle data for performance tests. Unfortunately, we did not get permission to publish the dataset, but we are going to use it in future publications for comparison with our past results.

#### 6.2. Comparison with Coronavirus Herd Immunity Optimizer and Other Algorithms

The new algorithm was proposed in 2022 (see [24]). It uses a new design with per-gene mutations. The solution is compared with many other algorithms on two datasets, in
which ABEFMP (see [23]) 1995 had the bigger problem size. They limited their runs to 500 iterations with a population of 100 specimens. Because all specimen has to be evaluated in each iteration, the whole run is limited to 50,000 evaluations. Because our algorithm is more complex, an iteration-limited comparison would not be fair. We chose instead to limit our algorithm to 50,000 cost calculations.

To better represent the significance of the different steps of our algorithm, we decided to set up four different scenarios. In the first scenario, we only use bacterial mutation on one specimen. The mutation had 5 cycles, 5 clones, and a segment length of 8. This is far from optimal, but with such limited resources, we cannot afford to burn all our budget on one mutation step. In the second scenario, we add the simplified 2-Opt (see [5]) boost on best with a 100-step limit. In the third scenario, we increase the population to 4, adding gene transfer with an injection count of 2. Finally, we increase the population size to 10 and the injection count to 5 in the last scenario. We made 10 runs with each solution, taking the best value for representation. Later, we present more information about the distribution of results.

Our results are shown in Table 1, where we marked the best costs with bold characters. CHIO is the column of the compared algorithm and the other columns are algorithms they compared the solution with. It is important to note that we do not know what budget the other algorithms consume per iteration. The table shows the massive improvement compared to the CHIO, including also the other algorithms. The comparative algorithms from the article are the genetic algorithm (GA) (see [26]), hybridization of GA with variable neighborhood search (HGA) (see [26]), the firefly algorithm (FFA) (see [27]), enhanced FFA (EFFA) (see [27]), the ant colony optimization (ACO) (see [28]), and the variable neighborhood search (VNS) [29]. The biggest improvement can be seen between the first and the second scenario, as local search is added, but gene transfer also means a stable improvement. The significance of gene transfer will be bigger in case of bigger problem sizes.

### Table 1. Comparison of the four proposed scenarios and data from publication [24].

<table>
<thead>
<tr>
<th>Instance</th>
<th>Scen 1</th>
<th>Scen 2</th>
<th>Scen 3</th>
<th>Scen 4</th>
<th>CHIO</th>
<th>GA</th>
<th>HGA</th>
<th>FFA</th>
<th>EFFA</th>
<th>ACO</th>
<th>VNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-n32-k5</td>
<td>649.92</td>
<td>555.4</td>
<td>467.09</td>
<td>467.09</td>
<td>662.68</td>
<td>784</td>
<td>784</td>
<td>NA</td>
<td>NA</td>
<td>784</td>
<td>787.08</td>
</tr>
<tr>
<td>A-n34-k5</td>
<td>722.67</td>
<td>541.07</td>
<td>490.26</td>
<td>490.26</td>
<td>709.95</td>
<td>784</td>
<td>784</td>
<td>779</td>
<td>781.31</td>
<td>780.95</td>
<td>778</td>
</tr>
<tr>
<td>A-n39-k6</td>
<td>793.27</td>
<td>646.66</td>
<td>548.26</td>
<td>550.62</td>
<td>871.7</td>
<td>831</td>
<td>831</td>
<td>NA</td>
<td>NA</td>
<td>831</td>
<td>833.2</td>
</tr>
<tr>
<td>A-n45-k7</td>
<td>754.92</td>
<td>691.18</td>
<td>547.29</td>
<td>548.22</td>
<td>918.81</td>
<td>1152</td>
<td>1146</td>
<td>1149.6</td>
<td>1147.25</td>
<td>1146</td>
<td>1146.91</td>
</tr>
<tr>
<td>A-n48-k7</td>
<td>960.06</td>
<td>755.95</td>
<td>585</td>
<td>594.59</td>
<td>1022.68</td>
<td>1152</td>
<td>1146</td>
<td>1149.6</td>
<td>1147.25</td>
<td>1146</td>
<td>1146.91</td>
</tr>
<tr>
<td>A-n55-k9</td>
<td>1002.33</td>
<td>769.02</td>
<td>598.64</td>
<td>579.92</td>
<td>1318.68</td>
<td>1073</td>
<td>1073</td>
<td>1103.96</td>
<td>1074.33</td>
<td>1073</td>
<td>1074.34</td>
</tr>
<tr>
<td>A-n60-k9</td>
<td>1100.21</td>
<td>841.65</td>
<td>637.33</td>
<td>635.69</td>
<td>1467.78</td>
<td>1354</td>
<td>1354</td>
<td>1379.79</td>
<td>1355.8</td>
<td>1354</td>
<td>1355.8</td>
</tr>
<tr>
<td>A-n69-k9</td>
<td>1375.43</td>
<td>935.3</td>
<td>800.78</td>
<td>783.79</td>
<td>1648.48</td>
<td>1154</td>
<td>1149</td>
<td>1191.37</td>
<td>1166</td>
<td>1159</td>
<td>1170.54</td>
</tr>
<tr>
<td>A-n80-k10</td>
<td>1784.18</td>
<td>1142.14</td>
<td>882.25</td>
<td>910.14</td>
<td>1966.88</td>
<td>1768</td>
<td>1763</td>
<td>1786.25</td>
<td>1766.74</td>
<td>1763</td>
<td>1790.45</td>
</tr>
</tbody>
</table>

As Table 1 shows, our improvement compared to CHIO ranges from 30% to almost 60%, tending to increase by problem size. As problem size grows GA, HGA, and ACO also produce better results than CHIO, but our solution still generates 30% to 50% better results than any of the compared alternatives. In the following section, we present the distribution of results in separate figures for each task. In Figure 7, we can see that even the worst results of scenarios 3 & 4 are an improvement compared to the 662.68 of CHIO. The median of scenario 2 is also an improvement, whereas scenario 1 is only an improvement by its record. This shows that local search yields major improvements on small problem sizes. Moreover, it is important to note that the spread decreases from scenario 1 to 4, where scenario 1 has 4 outliers, whereas scenario 4 has zero spread except for the two outliers. This suggests that gene transfer is not just improving the results, but increases the consistency of the performance. In Figure 8, scenarios 3 & 4 are also dominating, and even their worst results are better than the best result of the other scenarios or the result of CHIO (709). Scenario 2 has a much bigger spread than scenario 1, suggesting that local search decreases performance consistency. However, in our opinion, it is more due to the
decrease of resources the bacterial mutation receives than due to the effects of local search.
In Figure 9, Scenario 1 is much closer to the 871 of CHIO and 831 of GA, HGA and ACO,
whereas even the worst result of Scenario 3 is better. An interesting improvement is the
similarity of results of Scenarios 3 & 4. They have almost the same distribution, median,
and the best value of scenario 3 is better than the best value of scenario 4, but scenario 4
also has the worst. This lets us assume that big population size and more gene transfer
may result in less consistency improvement on bigger problem sizes. Until this problem
size, the best value of scenarios 3 & 4 was the same, suggesting that they may have found
the optimum. In Figure 10, the comparison between the result of CHIO (918) and the
worst results of the four scenarios are the same. In Figure 11, CHIO improves compared to
scenario 1 with its 1022 result. The order of medians of the four scenarios is the same and
the reverse of the order of spread. The best value of scenario 3 is 30% better than CHIO.
In Figure 12, even Scenario 1’s worst result is better than the 1318 of CHIO, and the worst
result of scenario 4 improves more than 40%. However, both HGA and ACO have a score of
1073, which is better than the median of scenario 1, although scenario 4 is still better by 35%.
The median of scenarios 3 & 4 is almost the same, but scenario 3 has higher quadrants. In
the last three figures (Figures 13–15), we didn’t find any new changes. Scenarios 3 & 4 are
an improvement compared to CHIO, ranging between a 45% and 50% decrease of cost for
the worst values. Even the worst, scenario 1, is better than the result of CHIO. However,
HGA and ACO constantly produce better results than CHIO, and they are comparable
to scenario 1 but produce worse results than scenario 2. The worst result of scenario 4
produces 20% better results in Figure 14 while yielding 45% and 30% better results on the
other two.

Figure 7. Boxplot of scenarios on the A-n32-k5 task of ABEFMP.

Figure 8. Boxplot of scenarios on the A-n34-k5 task of ABEFMP.
Figure 9. Boxplot of scenarios on the A-n39-k6 task of ABEFMP.

Figure 10. Boxplot of scenarios on the A-n45-k7 task of ABEFMP.

Figure 11. Boxplot of scenarios on the A-n48-k7 task of ABEFMP.
Figure 12. Boxplot of scenarios on the A-n55-k9 task of ABEFMP.

Figure 13. Boxplot of scenarios on the A-n60-k9 task of ABEFMP.

Figure 14. Boxplot of scenarios on the A-n69-k9 task of ABEFMP.
6.3. Parametrization and Belgian Dataset

In this section, we list all parameters, explaining their meaning and the experienced effects of different values. The most obvious parameters are the stop conditions, meaning the iteration and budget limits. Usually, the budget limit is used, and during these tests, it will be set to 1,000,000.

The next most important parameter is the size of the population. A bigger population means a bigger cover of the solution set, meaning the population will be less likely to get stuck in local minima. In addition, most optimizations improve on the best-cost specimen meaning other specimens might be useless after the efficiency of gene transfer started dropping. Smaller population sizes can eat less of the budget and make mutation of all specimens possible.

The rest of the parameters correspond to different steps of optimization. The step limit of the 2-Opt (see [5]) variant limits the number of position pairs in the algorithm tests. On a too-big limit, the convergence rushes to a local minimum. On a too-small limit, the local search makes no improvements, slowing down the convergence significantly. Injection count has a similar effect but on a global scale. A too-big injection count means the specimens become too similar to each other, resulting in a local search, effectively. A too-small injection count means that most of the population is left behind and the best specimen takes all the resources, even if others would have more potential.

Bacterial mutation has the most parameters because it has two subcycles. On each mutation, it makes a number of cycles and in each cycle, it generates multiple clones. The key concept to choose optimal parameters is the probability of improvement. Because we have a permutation, an \( n \)-long section can be ordered \( n! \) ways. During clone generations, we cannot check if there were duplicates because that would take significant resources.

Therefore, if the chance of finding a better solution than the original by one clone generation is \( p \), the chance of finding a better alternative with \( c \) clones is \( 1 - (1 - p)^c \). Consequently, even if \( p \) is 50\% to ensure an 80\% chance of improvement, at least three clones are needed. Given that we want to find the best possible clone, our \( p \) is \( 1/n! \) and this means even in the case of segment length of 5 we need 200 clones for an 80\% chance for success. However, in the case of a small segment, every successful improvement will improve less given the smaller set of solutions it can generate from. Because we have a heuristic mutation operator, we have a better chance of finding the best solution. We usually set the segment length to 16 and the clone count to 10. In the case of bigger tasks, we increase the cycle count instead so all positions will be touched by a mutation in fewer iterations of the optimization.

In summary, we ran our optimizations on the 1000 node task of the Belgium dataset (see [25]). We used a population of 50, with a 2-Opt step limit of 1000, injection count of 100, segment length of 16, clone count of 10, and cycle count of 100.

In Figure 16, the best run on the database is shown, in which each line is a specimen of the population. They are not tracking the same specimen, but the specimen in a position...
of the population. The x-axis is the budget, and the y-axis is the cost of the specimen. The specimen with the median cost has a smooth line, whereas the best has jumps in it and the worst has a lot of noise. The worst has a lot of noise because it got into the local minimum a lot, and many specimens changed positions. The best has the jumps because mutation has a fast rate until it gets stuck and Opt-2 helps it escape, causing a jump in steepness. It is easy to see that the population stays relatively together, whereas most of the population closes up to the best specimen.

In Figures 17 and 18 the progress of all 10 runs are displayed. The two axes are the same. The red and the green lines are the two runs with the median results. Each of the ten makes the same kinds of jumps, gaining momentum again and again. The two median runs were actually one of the worst runs until they gained a final momentum. Each finished between 930 and 830 million proving that the algorithm is reliable.

We made 10 runs on the Belgium-road-time-n1000-k20 task of the Belgium 2017 dataset (see Figures 17–19). We calculated a primitive over- and underestimate for comparison. The overestimate was calculated by pretending each customer is served by a separate vehicle, adding up the cost of all routes starting or ending at the depot resulting in $3.62 \times 10^9$. Our underestimate consists of the sum cost of the minimum cost outgoing route for each node of the distance graph, which is $3.218 \times 10^8$. Each optimization started from over $4 \times 10^9$ which was already lower than the overestimate. The optimal start was thanks to the potential measurement of the population initialization that also functioned like a preoptimization. Our best result was $6.653 \times 10^8$, and the worst result was $8.394 \times 10^8$. This means even our worst result was under three times of the primitive underestimate, which is a relatively good result given the limited budget. The data suggests that further optimization still can be made since the convergence progresses even after the last 10,000 budget mark.

![Figure 16. Progress of population of the best run on Belgium 2017 dataset.](image_url)
Figure 17. All 10 runs on Belgium 2017 dataset.

Figure 18. All 10 runs on Belgium 2017 dataset, magnified.

Figure 19. Boxplot of 10 runs on Belgium 2017 dataset.
6.4. Difficult Task and Parameter Changes

The dataset we are using originates from QLM Logistic Solutions, a Hungarian company specializing in logistics software. They are real-life datasets from the partners of the company filtered and processed to contain only the necessary information for the task. Unfortunately, the dataset is owned by the company and we are not allowed to publish it.

In our task, we have only one depot and only one type of vehicle. We pretend the fleet is limitless, meaning we have as many vehicles as we have customers. Because serving more customers with fewer vehicles results in less cost, we expect the convergence to lead to a plan using only a fraction of the fleet. The vehicles are capacitated in weight, volume, and time. The cost is measured in euros and comes from the fuel consumed and the payment of the driver. The length of the routes depends on the time of the day, meaning the cost function is a real-life simulation of the routes starting at eight o’clock. We used the open trip planner to calculate the optimal routes per hour of the day between each location given the vehicle type. At cost calculation, we select the version of the route closest to the time of the simulation. We built the network graph from it, which is used for the subsequent simulations. This makes cost calculation much faster, keeping the time consumption at $O(c)$, where $c$ is the number of customers.

We use this data to validate our progress toward real-life transport optimization without clustering and to optimize our parameters. To demonstrate the effects of parameters, we publish five different settings, each running 10 times (see Figures 20 and 21). Even with such a small sample size, multiple important conclusions can be made with certainty. The first setup was our default parametrization on a budget of 5,000,000. The population size is 50, the 2-Opt step limit was 1000, the injection count was 100, the mutation cycle count was 10, the mutation clone count was 20, and the segment length was 8. Here, the population size is very small, but since population initialization selects the ones with the best potential, we have a big convergence speed as with a population size of 210 with a smaller budget per iteration. The mutation has a budget consumption of 200, the gene transfer is 100, and the boost is between 0 and 1000. Because the boost runs only if the best specimen is not improved upon by other steps, it consumes around 360 of the budget on average. The distribution of budget shifts a lot by problem size because the budget of mutation and boost grows linearly whereas the injection count is proportional to the square root of the problem size. The injection count is the same or doubles the population size on bigger tasks. Because the problem is more complex, the heuristic used for the mutation operator performed more weakly, making 20 clones necessary. In return, every improvement had a higher cost-reduction ratio, making fewer cycles necessary. Our overestimate was $8.27 \times 10^6$ and the underestimate was $8.33 \times 10^4$. This is an unusually big range, but given the complexity of the estimation algorithm, it is not surprising. It suggests that the locations are in multiple patches which are expected because the locations are from multiple cities. Our best result was $7.44 \times 10^5$ (see Figures 20 and 21), which is far from an underestimate, but given the range is 10 times compared to the Belgium dataset (see [25]) example, it is very good. Our worst result was $8.3 \times 10^5$ which is a smaller spread compared to the Belgium example.

In the second setup, we halved the mutation segment length, leaving everything else the same. This resulted in more successful mutations, but a smaller cost-reduction ratio. Moreover, since the mutation is always successful, the local search never ran, slowing down the convergence further. The best result was $8.99 \times 10^5$ (see Figures 20 and 21), and the worst was $1.01 \times 10^6$, which is a significant increase.

In the third and fourth setup, we played with the distribution of the budget in the bacterial mutation step. In the third, we doubled the clone count and halved the cycle count, and in the fourth we halved the clone count and doubled the cycle count. Basically, we switch between fewer tries with a higher improvement probability and more tries with a lower improvement probability. The best and worst results of the third scenario are $7.378 \times 10^5$ and $8.784 \times 10^5$ (see Figures 20 and 21). The same for the fourth are $7.586 \times 10^5$ and $8.22 \times 10^6$ (see Figures 20 and 21). The worst result of the fourth is an edge case, so
the results are surprisingly similar. This lets us assume that the distribution of the budget between the clone and cycle count is less important than selecting the correct segment length. However, the more clones we have the less we can increase our probabilities by more clones, so radically decreasing the cycle count, while increasing the clone count can result in poor performance. In addition, the figure shows that the third has a slower start but keeps a better improvement rate, whereas the fourth has a lot better performance in the beginning, but fails to keep a constant curvature.

In the last setup, we doubled the segment length to 16, which is the optimal segment length mentioned earlier. We also doubled the clone count to make sure it is enough and halved the cycle count to stay around the same budget per iteration. This improved our convergence speed greatly, bringing the minimum cost to $6.216 \times 10^5$ whereas the maximum cost was $6.615 \times 10^5$ (see Figures 20 and 21). We also tried to further increase the segment length (to 32), but this made too many clones necessary, meaning the problem grows too big for our heuristic over 16. We assumed this optimum can be dependent on the problem complexity or problem size, but it was the same on the Belgium dataset (see [25]) and ABEFMP dataset (see [23]). The exact probabilistic model is still in progress.

Our main observation from the five scenarios should be that a bigger segment length yields significant improvements. Even if a smaller segment length means the algorithm finds the best solution more often, the improvement amount is a lot smaller per improvement. However, during earlier tests, we found that a significantly bigger segment length also decreases efficiency. With the improvement of the edge builder heuristic, we might be able to further increase segment length. Shifting budget distribution between clone count and cycle count count does not improve performance on longer runs. Both Scenarios 3 & 4 had a median around the median of scenario 1, whereas the spread was inconsistent between the three cases.

![Figure 20. Scenarios on the Hungarian dataset.](image-url)
7. Observations

We conclude that our bacterial mutation with edge builder heuristic is a comparable solution to CHIO as it is. For larger problem sizes, it yielded better results even in the worst cases. Adding a well-balanced local search can be a big improvement. (It is important to limit the run of the local search to use a comparable amount of budget; otherwise, the algorithm will produce similar results to the local search due to its overwhelming limitations.) Having a bigger population and gene transfer increases not only performance, but consistency as well. However, increasing the population from 4 to 100 yielded similar results. A population of 4 often produced better results than a population of 100.

In our opinion, the performance of our bacterial mutation is due to its weighted probabilistic nature. If it were a simple heuristic, it would be limited by the precision of edge cost estimates. It would get stuck easily in local minima and would struggle on more complex tasks like CVRP. On an 8-long segment of a permutation, we can order the elements 40,320 ways. Checking all orders for one segment would use up almost all of the 50,000 budget. We can easily find better solutions at the start of the convergence at random, but it gets exponentially harder closer to local minima. By weighting the probability of neighboring, not just based on the cost estimate of edge, but also the cost estimate of exclusive edges (ones with the same origin, end, or that would create a two-long cycle), we can make not just locally, but globally optimal decisions while building the segment. This allows us to make big improvements like global searches with the continuity of improvement of local searches. That is why we refer to this search as pseudo-global search.

However, our pseudo-global search also has its limitations. It cannot improve on the whole permutation, as the calculation of weights has a high complexity, even if not dependent on the problem size. When its convergence slows down, a local search can pull it to a more advantageous position. This is why even if the bacterial mutation is a lot better than our local search, when it gets stuck we run a local search to find a more advantageous position in the solution space. Adding local search can yield up to 20%, whereas a pure local search produces an outcome 30% worse.

Our gene transfer with feedbacked half heuristic crossover is the real implementation of the genetic algorithm like global search. In genetic algorithms, specimens become similar as the convergence progresses, meaning the results of one optimization will have a smaller spread. In our opinion, this kind of cohesive effect combined with bacterial mutation results in a radical increase in performance consistency. Another factor is the size of the population, but as we demonstrated, its significance decreases by problem size. By using heuristic crossover with similar weighted probabilistic features to our bacterial mutation, gene transfer becomes the third drive of convergence, but on a global scale. It results in a quick convergence on the first iterations, switching to rare but big improvements in later iterations.
At the moment, we already run our algorithm on problem sizes of 1000 or 2500 clients. This is a massive achievement even for the most recent results.

The new algorithm improves on every step of the original setup. In the original setup, the responsibilities of the steps were different. The bacterial mutation was an explorer step, in which the algorithm checked the broader surroundings of the specimen. It was meant more to avoid local minima than to drive the convergence. The only purpose of dependency injection was to share information between parallel independent searches. The only drive of convergence was the local search. This meant that the algorithm got stuck on bigger mutation segment lengths and was slow. In the new design, we could quadruple the segment length. The dependency injection does not just transfer information but became an efficient global search keeping the solutions close to each other in the first segment of the run.

The optimal mutation segment length and clone count are constants, meaning they do not contribute to the complexity of the algorithm. Even if clone count can be compensated by cycle count, it has its limits, so it is better to increase cycle count when necessary. The biggest challenge of the setup is to keep a good ratio between the three main steps of the algorithm. Too much of any of the three steps lead to a local minimum, whereas if balanced they complement each other.

8. Conclusions and Future Work

The VRP is one of the most central problems facing the economy today. Because it is NP-hard, solving it directly is impossible in most cases under realistic time constraints.

The improved bacterial memetic algorithm is an advanced iterative optimization that is an efficient solution. We have adapted the algorithm to the problem and worked out a methodology to find the correct parameters. It can deal with problem sizes that were achievable only by clustering before. We also adapted it for more realistic models of real-life transportation. We compared our method with some of the most-used methods for VRP. Our algorithm yields a 30% to 50% improvement compared to these methods.

We are going to further validate our results with more comparisons and more detailed statistics. We are looking for partners, who are willing to publish real-life data. We are researching transport plans generated with up-to-date solutions. We are already working on better heuristics for the mutation and gene transfer operators. Our code base is more than a research script, and we are going to develop it into an online API. We are working on a logic that keeps the balance between the different steps.

Author Contributions: Conceptualization, Á.H.-S. and J.B.; methodology, Á.H.-S. and J.B.; software, Á.H.-S.; validation, Á.H.-S. and J.B.; formal analysis, Á.H.-S. and J.B.; investigation, Á.H.-S. and J.B.; resources, Á.H.-S.; data curation, Á.H.-S.; writing—original draft preparation, Á.H.-S. and J.B.; writing—review and editing, Á.H.-S. and J.B.; visualization, Á.H.-S.; supervision, J.B.; project administration, Á.H.-S.; funding acquisition, J.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References
2. Dantzig, G.B.; Ramser, J.H. The Truck Dispatching Problem. Manag. Sci. 1959, 6, 80–91. [CrossRef]