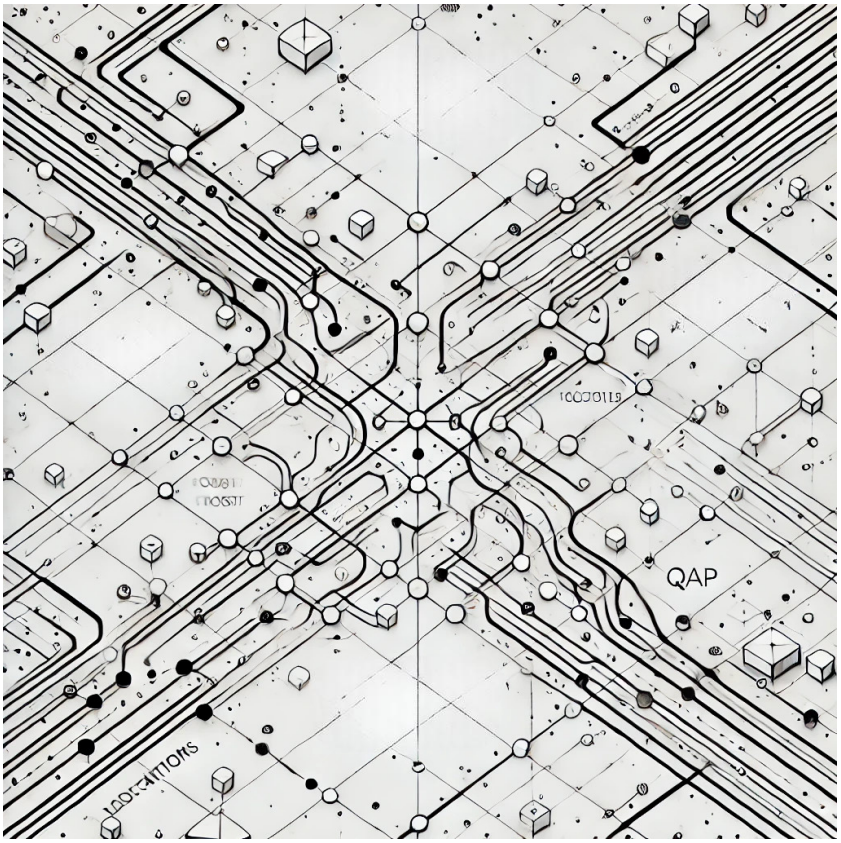
UNIVERSIDAD
DE GRANADA

Project assignment

Solving the Quadratic Assignment Problem using Evolutionary Algorithms

January 26th, 2025
Granada



Name:	Raoul Luqué
University:	Universidad de Granada
Study program:	Computer Science M.Sc.
Teachers:	Prof. Berzal,
	David Criado

Contents

1 Introduction

Evolutionary algorithms (EAs) are a class of metaheuristic optimization techniques inspired by the process of natural evolution. These algorithms leverage mechanisms such as selection, crossover, and mutation to iteratively improve solutions to complex optimization problems. Known for their flexibility and robustness, EAs have been successfully applied to a wide range of combinatorial and continuous optimization challenges, see [Bar+14].

One such challenge is the Quadratic Assignment Problem (QAP), a well-known combinatorial optimization problem that models the assignment of facilities to locations in a way that minimizes cost. The QAP is classified as NP-hard, meaning that solving large instances of the problem within a reasonable timeframe is computationally challenging. Due to its complexity, heuristic and metaheuristic approaches, including EAs, have become popular tools for tackling the QAP, see [Loi+07].

This work investigates the application of evolutionary algorithms to solving the QAP. It is written as a homework assignment in the course of computational intelligence as part of the masters on computer science at the Universidad de Granada. More precisely, the goal of this work is to develop three different variants of evolutionary algorithms, a standard, Baldwinian and Lamarckian variant. The goal being to obtain the best possible solution on the `tai256c` instance from the QAPLIB, see [BKR97]. Therefore, this work serves both a presentation of our results, as well as an introduction to the field of evolutionary algorithms.

We will start by discussing the problem at hand, the QAP and presenting its mathematical problem statement. Subsequently, we will discuss the encoding used in this work, both for the problem parameters, as well as the solutions to the problem and also discuss the up- and downsides of a possible alternative encoding. Next, we will give an introduction into evolutionary algorithms going through the different phases of a basic evolution loop. We will also present the two alternative variants we will be using: Baldwinian and Lamarckian. While doing so, we will also point out the specifics of our implementation and the variants we used to obtain our best result. At last, we will evaluate the different variants of the evolutionary algorithm and compare them with respect to the quality of their obtained results and their running time. Followingly, we will present the best result we were able to obtain and conclude this work. [1]

[1] The title image was created with the assistance of DALL·E 2.

2 Quadratic Assignment Problem

The quadratic assignment problem (QAP) as originally suggested by Koopmans and Beckmann, see [KB57], is a fundamental combinatorial optimization problem. More specifically, it is a part of the branch of optimization of mathematics and forms part of the category of facility location problems. The latter being a branch of research which is concerned with the optimal placement of facilities to minimize transportation and other costs. However, it also finds application in parallel and distributed computing, combinatorial data analysis, finding of maximal clique and graph partitioning, see [Loi+07].

In our case, the QAP may be described as follows:

Definition 2.1 (Quadratic assignment problem [Wik24b])**.** There are a set of n facilities and a set of n locations. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified (e.g., the amount of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.

That is, we have n facilities and our goal is to find a permutation of these facilities (a bijective mapping of the facilities to the different locations) such that the cost is minimized. The cost being, for each pair of facilities, their distance with respect to their determined locations multiplied by the flow between the two facilities. In mathematical/symbol terms we can describe the problem as follows:

Definition 2.2 (Quadratic assignment problem)**.** Given two sets F and L of same size together with a distance function $d : L \times L \to \mathbb{R}$ and a flow function $f : F \times F \to \mathbb{R}$, find a bijection $\sigma : P \to L$ such that the cost function:

$$\sum_{a,b \in F} f(a,b) \cdot d(\sigma(a), \sigma(b))$$

is minimized.

In the above definition, F would be the set of the Facilities, L the set of locations and σ the assignment of facilities to locations. The QAP can also be interpreted as a more general form of the traveling salesperson problem. This becomes apparent when defining the flow function such that it connects all facilities in a single cycle with a constant value and zeros everywhere else. Meanwhile the distance function has the same values as the traveling salesperson instance. It was shown that the QAP is $\mathcal{NP}-$hard to solve and that there cannot exist an approximation algorithm running in polynomial time for any constant factor, unless $\mathcal{NP} = \mathcal{P}$, see [SG76]. This motivates the search of algorithms which find approximate solutions of the problem in shorter time than exponential. One such approach could be evolutionary algorithms as presented in this work.

3 Encoding

Before being able to discuss the precise evolutionary algorithm, we must define the encoding used for the quadratic assignment problem. As presented in definition 2.2, an instance of the QAP consists of a set of facilities, locations, a flow and distance function and the solution which is an assignment of the facilities to the locations. To simplify our encoding, we may interpret the set of facilities and locations as sets of numbers from 0 to n. Hence, using this method, a solution of the QAP would be a permutation of the numbers from 0 to n. Now, two possible encodings of permutations may come to mind. First, a list of numbers from 0 to n, that is, the 'one-line notation' for permutations, or as a binary matrix, that is, the permutation matrix corresponding to the matrix. We will be using the former encoding and discuss some aspects on why this encoding is more suitable for the problem in section 3.1. The one-line notation is also known as path code in the context of the traveling salesperson problem, see [YG10, p. 283]. For the encoding of the flow and distance function, we will use the usual and provided encoding of real valued matrices of size $n \times n$. Since the instance of the QAP we will be looking at has 256 facilities, this means that we will have two matrices of size 256×256, that is $n = 256$.

3.1 Encoding as Permutation Matrices

Although there may be some benefits using permutation matrices as the encoding of solutions, it turns out that they are way less space efficient and also slow down several computations in our evolutionary algorithm. One benefit of using permutation matrices is the speedup of calculating the fitness of a given individual. By this, we are referring to calculating the cost of a given solution as seen in definition 2.2. Instead of calculating a sum, one can instead calculate the cost as the trace of matrix product. Given the flow and distance function in matrix forms F' and D, respectively, one may compute the cost of a permutation given in matrix form P as follows:

$$c(P) = \text{trace}(F'XD^TX^T)$$

Although this may speed up the computation of the costs of solutions and can be parallelized very well for computing the cost of multiple solutions at the same time, there are also downsides of using a matrix encoding for the permutations. As we will see, in many steps of the evolutionary algorithm, we need to determine where a certain facility gets mapped to. This would translate to having to find a row in a matrix. Since we are considering a QAP instance with $n = 256$, this would mean searching a $256 \times 256 = 65536$ bit (8192 byte) matrix. If we instead use the list method, in the optimal encoding case (depends on the specific implementation), we could store each number in 8 bits, that is one byte, and would therefore need only 2048 bits (256 bytes) to store a solution to our QAP instance. This equates to a space reduction of 32-fold. In this case we would of course not necessarily be concerned with the needed space but instead the implicit increase in running time that is needed if our solutions are stored 'so inefficiently'.

4 Evolutionary Algorithm

Similar to artificial neural networks that try to apply the structure of neurons in brains and apply that to learning algorithms, evolutionary algorithms (EAs) try to simulate a process similar to evolution to solve a variety of problems.

At their core, EAs embrace the concept of 'survival of the fittest', where a population of potential solutions, represented as individuals, undergoes a process of selection and variation to produce progressively improved offspring over successive generations, see [YG10, p. 6]. The effectiveness of EAs lies in their ability to effectively explore and exploit the solution landscape, even in the absence of gradient information. This characteristic makes them particularly suitable for tackling challenging problems where gradient information may not be available or not make sense in the context of the problems. In such cases, EAs can help finding near-optimal solutions within a reasonable time frame.

So far, with respect to our QAP problem at hand, we referred to permutations as solutions of our problem. Since we are taking an evolutionary approach and are therefore considering populations instead of solutions, we will from now on refer to solutions as individuals or chromosomes in our population. Furthermore, instead of using the word cost to describe the quality of a solution, we will use the term fitness, since this attribute will be directly tied to an individuals chance of surviving or being able to produce offspring. The computation of the fitness/cost will stay the same however. At last, we will call the entries of our permutation (in one-line notation) as genes.

We want to use this chapter to understand evolutionary algorithms and while doing so explain the exact method used in our work to solve the QAP instance at hand.

4.1 Basic Evolution Loop

The foundation of every evolutionary algorithm (EA) is its evolution loop. This method is looped until a result of the desired quality or a certain number of iterations is achieved. Usually, the loop consists of four key steps: selecting the mating partners, recombining to generate a new population, possibly mutating individuals in the population and evaluating the new population in order to then restart at the selection. Each iteration of the loop is referred to as a generation. Figure 4.1 shows a visualization of the basic evolution loop.

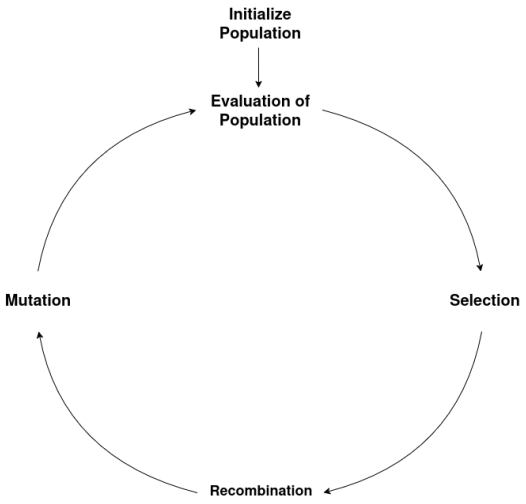


Figure 4.1: Evolution loop

Each of the steps in the evolution loop has various ways that it could be implemented each of which comes with its own hyperparameters that can be tweaked, respectively. We want to discuss each individual step of the evolution loop in the sections to follow

while also specifying the specific methods we used for our implementation.

For now, note that the basic idea of the evolutionary algorithm and therefore evolution loop is to generate a population of individuals/chromosomes, each of which is a solution to our QAP. By then applying the evolution loop, we simulate an evolution process of generating children for each generation which have possible mutations in order to successively obtain better solutions, that is, individuals with better fitness. In our case, lower fitness values are better since the fitness is equal to the cost of an individual/solution.

4.2 Initialization

Before properly running the evolution loop, we need to initialize our population. For this, different approaches can be used. Of course, one does not want to have a completely heterogeneous population from the start since in that case, evolution will probably not achieve much. Instead, we want a diverse initial population to search the complete solution space, see [YG10, p. 17]. One way to achieve this is to initialize it randomly. We do this with the following method from `src/evolutionary_tools/chromosome.py`.

```
def generate_random_chromosomes(number_of_chromosomes: int,
    number_of_facilities: int) -> ndarray:
    """
    Generate random chromosomes. Individual chromosomes are represented
    as a one-dimensional numpy array (permutation list)
    and generated using np.random.permutation.
    :param number_of_chromosomes: The number of chromosomes to generate
    :param number_of_facilities: The number of facilities (length of the
                                 permutation)
    :return: A two-dimensional numpy array representing the list of chromosomes
    """
    return np.array([np.random.permutation(np.arange(number_of_facilities))
                     for i in range(number_of_chromosomes)])
```

Listing 4.1: Code for the initialization of the population. Taken from the code repository

It is basically a wrapper around the `np.random.permutation` function which generates a random permutation given a size.

4.3 Selection

Next up, we want to discuss the concept of selection functions. These are usually applied at the start of a generational loop to select the parents which are used to generate new children for the next generation. Therefore, the selection function has major impact on the direction a population takes genetically and whether it evolves in the direction of fitter individuals or not.

There are many possible selection functions of which we will be presenting two popular choices. One of which is the roulette wheel selection, also known as (fitness) proportionate selection, see [YG10, p. 67] and the other is the tournament selection. Each selection function can be thought of as given the population and their fitness values, returning one individual. Then, the selection function can be applied as often as desired. In our

case, we select as many individuals as the previous generation had since two parents will generate two children with our recombination functions.

Roulette wheel selection is based on the idea that each individual should have a chance of 'passing on its genes', that is, being selected, proportionate to its fitness. This is then translated to a roulette wheel, where a ball in the wheel decides which individual is selected. Each individual gets a slice of the roulette wheel in proportion to its fitness value (relative to the sum of all fitness values in the population). In symbols this translates to the following. If f_i is the fitness of individual i in the population, its probability of being selected is:

$$p_i = \frac{f_i}{\sum_{j=1}^{popsize} f_j}.$$

Where $popsize$ is the size of the population.

Tournament selection on the other hand has various possible implementations. We implement two kinds of $k-$tournament selection. One of which is usually referred to as unbiased, see [YG10, p. 75]. The basic idea of $k-$tournament selection is to pick k individuals randomly (without duplicates) from the population. Then, the fittest individual amongst these k is selected. The unbiased version would be passing a probability p to the method and choosing the fittest individual with chance p. In the case the fittest individual is not chosen, the second fittest individual is chosen with a chance of p. If not, the third fittest is chosen with a chance of p and so on.

In our tests, roulette wheel selection performed best and was the easiest to implement with the fastest running time. There is however an improved version of roulette wheel selection, stochastic universal sampling, suggested in [Bak87], which tries to address the problems that roulette wheel selection might face. Among other things, roulette wheel selection can favour very fit individuals too much which leads to premature convergence. Nonetheless, we were able to obtain great results with roulette wheel selection which is why we opted against implementing further selection algorithms. For future works, it could be tried replacing the roulette wheel selection with stochastic universal sampling to possibly reduce the number of needed generations to find the same quality solutions or find even better solutions in general.

The implementations of the selection functions can be found in the code repository at `src/evolutionary_tools/selection.py`.

4.3.1 Making sure the fittest individual is selected

We slightly tweaked the selection algorithm by making sure that the fittest individual in a given population is selected. Not only that, but the individual will be transferred to the next generation as is without recombination or mutation (see section 4.5). We did this by applying the selection methods as discussed above and after evaluating the new population, replacing the worst individual with the fittest individual from the previous generation. Doing so, we made sure that the population would be guaranteed to not worsen over time and instead evolve towards fitter individuals.

4.4 Recombination

Given that we have chosen the individuals to use as parents for the next generation, we need to combine their genes somehow to generate new children. Again, we want to inject some randomness into this process, to explore as much of the solution space as possible.

Since the individuals in our algorithm are permutations, we can not arbitrarily cross/recombine chromosomes/individuals. When recombining parents, we have to guarantee that the resulting lists (potential permutations) are indeed permutations. Therefore, often times in literature about evolutionary algorithms, there are recombination methods specifically for traveling salesperson-like problems or permutations. We will be discussing two of such methods, namely order crossover, see [YG10, p. 289], and partially mapped crossover (PMX), see [YG10, p. 288].

As a first and easier recombination function, we present order crossover. The original idea dates back to 1991, see [Dav91]. A benefit of order crossover is its conservation of the relative order of genes in the parent chromosomes, see [YG10, p. 290]. Additionally, it is relatively simple in comparison with other methods such as the following PMX. This can also be seen in the following pseudocode of order crossover.

```
Input: Parent1, Parent2 (two parent permutations)
Output: Child1, Child2 (two offspring permutations)

1.  Select two random crossover points, C1 and C2,
   such that 0 <= C1 < C2 < length of Parent1.

2.  Copy the segment between C1 and C2 from Parent1 to Child1,
    and from Parent1 to Child2.

3.  For Child1:
    a.  Starting from position C2 in Parent2, iterate through
        its genes in circular order.
    b.  For each gene m in Parent2:
        i.  If m is not already present in Child1, place it
            in the first available position in Child1.

4. Repeat step 3 for Child2 and Parent1

5. Return Child1 and Child2.
```

Listing 4.2: Pseudocode of order crossover

Partially mapped crossover was first proposed in 1985 as a recombination method for traveling salesperson like problems, see [GL85], and has since then gained a lot of traction. Similarly to the roulette wheel based selection, there are various improvements upon the existing PMX algorithm depending on the specific application such as the popular edge crossover, see [ES15, p. 72]. Again, since we were able to achieve great results with the easier PMX, we sticked with this recombination algorithm and also used it for obtaining our best result.

Pseudocode for PMX could look as follows:

```
Input: Parent1, Parent2 (two parent permutations)
Output: Child1, Child1 (two offspring permutations)

1.  Select two random crossover points, C1 and C2,
    such that -1 < C1 < C2 < length of Parent1 - 1.

2.  Copy the segment between C1 and C2 from Parent1 to Child1,
    and from Parent2 to Child2.

3.  For each gene m in the segment of Parent2 between positions C1 and C2:
    a.  If m is not already in the Child1:
        i.  Find the position n in Parent2 where m is located.
        ii. Trace the mapping from Parent1 to Parent2 until an
            empty position is found in the offspring.
        iii.Place m in the empty position in the offspring.

4.  Fill the remaining empty positions in Child1 by copying the genes
    from Parent1 to the offspring in their original order,
    skipping any genes already present.

5. Repeat step 3 and 4 for Child2 switching the parents' roles.

6. Return Child1 and Child1.
```

Listing 4.3: Pseudocode of partially mapped crossover (PMX)

For the specific implementation please refer to `src/evolutionary_tools/recombine.py` in the code repository.

4.5 Mutation

As a last step in our basic evolution loop, we apply mutation to the children, that is, the new population. This mutation is however applied with a certain chance `MUTATION_PROB`. There are several possible mutation operators. In general, mutation has the goal of introducing new chromosome configurations into the population. These mutations might then lead to better fitness than previous populations, similar to evolution in the real world.

When applying mutation one has to find the balance of introducing enough variance into the population such that it does not get stuck or converge to seemingly good individuals to fast but to also not disrupt the step-wise improvement of the population by introducing too much randomness.

Again, similar to what we noted in the previous section 4.4 about recombination operators, one has to be careful that the mutation operator produces valid permutations. We opted for the most simple mutation, the swap or exchange mutation, see [YG10, p. 286]. Iterating over each individual in the (new) population, one applies a swap of two random entries in the permutation list with a chance of `MUTATION_PROB`.

Although the implementation should be clear given the above explanation, the following listing shows the code of the swap mutation from our implementation.

```
def swap_mutation(chromosome: ndarray) -> ndarray:
    """
    Perform a swap mutation on the chromosome.
    :param chromosome: The chromosome to mutate
    :return: The mutated chromosome
    """
    index1, index2 = np.random.choice(chromosome.shape[0], 2, replace=False)
    chromosome_entry_one = chromosome[index1]
    chromosome[index1] = chromosome[index2]
    chromosome[index2] = chromosome_entry_one
    return chromosome
```

Listing 4.4: Code of swap mutation operator. Taken from the code repository

4.6 Baldwinian and Lamarckian Variants

To recap, our evolution algorithm so far, creates a population once, selects some individuals to create offspring according to their fitness, creates offspring, mutates the offspring and repeats. This is not completely dissimilar to real evolution, there are some differences however. For one, when looking at humans, the individual might change throughout the course of its life. That is, there might be factors in the individuals life which do not necessarily have something to do with its genes, which however determine its chances of reproducing. That is, if a pre-civilization human was to find an undiscovered region with a lot of of food, it would probably increase the chances of that human surviving and reproducing, that is, 'being selected'.

We want to try to incorporate this aspect into our evolutionary algorithm. This is where Baldwinian and Lamarckian variants of evolutionary algorithms come in, see [YG10, p. 118]. We will be applying these variants in the following way: Before the selection step in our evolution loop (at the stage where the individuals of the current generation are 'grown'), we apply a greedy, local optimization algorithm to each individual to try to find local changes to the chromosome/genes which lower the fitness. That is, we assume, with respect to the optimization algorithm, that all individuals have great lifes and are therefore the 'best versions of themselves' and use these versions to select the parents for the new generation.

We need to make a slight distinction between the Baldwinian and Lamarckain method however. The Baldwinian approach is based on the Baldwin effect presented by Baldwin in 1896, see [Bal96]. For our purposes, it states that an individuals' genes do not change in the course of its life even though their fitness might increase. Note that a decrease in fitness is good in our case since we are looking to minimize the cost of the solutions. To translate this into our algorithm, we would when selecting parents for the next generation, take the locally optimized fitness values. We would however, not apply any changes to the genes and recombine the parents as they were before the fitness computation. On the other hand, Lamarckism would believe that these changes in the fitness during the course of the life would be reflected in an individuals' genes. That is, translating to our application, we would try reducing the individuals fitness with our local optimization function and immediately apply the changes that would be needed to the genes to obtain

the optimized fitness value. This idea has no specific starting point and is instead believed to have already existed for centuries before the name giver proposed it, see [Wik24a].

In the following two subsections, we will discuss the optimization we used for the Baldwinian and Lamarckian variant and some caching techniques we applied to speed up the computation which might otherwise take a very long time. Since we now have different variants of our algorithm, we will state which variant we are referring to, that is, standard, Baldwinian or Lamarckian.

Note that one might also vary whether to apply the optimization to all individuals of a population or not, but we want to specify that for our application, the local optimization was applied to all individuals in both the Baldwinian and Lamarckian variant.

4.6.1 2-opt

2-opt is a simple local search algorithm which was first suggested for the traveling salesperson problem. The idea is to swap two destinations in the route and see if the resulting route is better than the existing one. Applying this to our QAP, we swap the position of two facilities and see if the resulting cost is lower than the previous configuration. The idea is so simple, we will immediately present the code.

```
def two_opt(flow_matrix: ndarray, distance_matrix: ndarray,
    chromosome: ndarray) -> ndarray:
    """
    Perform a 2-opt optimization on a given chromosome.
    :param chromosome: One-dimensional numpy array representing a chromosome.
    :param flow_matrix: NumPy array representing the flow matrix.
    :param distance_matrix: NumPy array representing the distance matrix.
    :return: Optimized chromosome (chromosome) as a numpy array.
    """
    best_chromosome = chromosome.copy()
    n = len(chromosome)
    improved = True
    number_of_iteration = 0

    while improved and number_of_iteration < NUMBER_OF_ITERATIONS_FOR_OPT
        and not cache_hit:
        number_of_iteration += 1
        improved = False
        for i in range(1, n - 1):
            for j in range(i + 1, n):
                new_chromosome = best_chromosome.copy()
                new_chromosome[i], new_chromosome[j] = best_chromosome[j],
                best_chromosome[i]
                current_cost = basic_fitness_function(flow_matrix,
                    distance_matrix, best_chromosome)
                new_cost = basic_fitness_function(flow_matrix,
                    distance_matrix, new_chromosome)
                cost_delta = new_cost - current_cost
                if cost_delta < 0:
                    # Perform the 2-opt swap
                    tmp = best_chromosome[j]
                    best_chromosome[j] = best_chromosome[i]
                    best_chromosome[i] = tmp
                    improved = True
```

Listing 4.5: Code of two-opt algorithm. Taken from the code repository

Looking at the two inner for loops, we can see that we are iterating over all possible swapping positions and checking if the new chromosome is cheaper than the existing one, and if so swapping the facilities in the existing chromosome. The swapping seems a bit unnecessary since we could simply `best_chromosome = new_chromosome`, however it will make sense in the next iteration of the algorithm.

Furthermore, we have an outer while loop which checks if there has been an improvement. This enables us to possibly iterate over all the possible swapping positions multiple times, since a swap in the first iteration might enable a swap in the second iteration. We however, did not use this option for any runs and instead always defined `NUMBER_OF_ITERATIONS_FOR_OPT = 1`.

Now, let us address the seemingly unnecessary swapping of the facilities in the best chromosome. We do this, because we can speed up the computation of the cost delta. See, when we swap two facilities, we do not need to recalculate the cost of the entire new chromosome. Instead we could just calculate the cost change this swap would induce. This is what the following function does:

```python
def calculate_delta_cost(flow_matrix: ndarray, distance_matrix: ndarray,
    chromosome: ndarray, i: int, j: int
) -> float:
    """
    Calculate the delta cost of a 2-opt swap. That is, the resulting
    change in cost if the i-th and j-th elements in the chromosome are swapped.
    :param flow_matrix: NumPy array representing the flow matrix.
    :param distance_matrix: NumPy array representing the distance matrix.
    :param chromosome: One-dimensional numpy array representing the chromosome.
    :param i: Index of the first element to swap.
    :param j: Index of the second element to swap.
    :return: The delta cost of the swap.
    """
    n = len(chromosome)
    if i == j:
        return 0

    # Get the indices of the elements to be swapped
    a, b = chromosome[i], chromosome[j]

    # Calculate the delta cost
    delta = 0
    for k in range(n):
        if k != i and k != j:
            c = chromosome[k]
            delta += (flow_matrix[i, k] *
                        (distance_matrix[b, c] - distance_matrix[a, c]) +
                      flow_matrix[j, k] *
                        (distance_matrix[a, c] - distance_matrix[b, c]) +
                      flow_matrix[k, i] *
                        (distance_matrix[c, b] - distance_matrix[c, a]) +
                      flow_matrix[k, j] *
                        (distance_matrix[c, a] - distance_matrix[c, b])
                     )
    return delta
```

Listing 4.6: Code for computing the delta cost in two-opt. Taken from the code repository

This speeds up the 2-opt algorithm drastically which already presents a bottleneck for our evolutionary algorithm which is otherwise very fast. To further speedup the computation of the delta cost, we have implemented a vectorized version of the algorithm using NumPy.

```python
def calculate_delta_cost_numpy(flow_matrix: ndarray, distance_matrix: ndarray,
    chromosome: ndarray, i: int, j: int
) -> float:
    """
    Calculate the delta cost of a 2-opt swap using NumPy to vectorize
    the operation. That is, the resulting change in cost if the i-th
    and j-th elements in the chromosome are swapped.
    :param flow_matrix: Numpy array representing the flow matrix.
    :param distance_matrix: Numpy array representing the distance matrix.
    :param chromosome: One-dimensional numpy array representing the chromosome.
    :param i: Index of the first element to swap.
    :param j: Index of the second element to swap.
    :return: The delta cost of the swap.
    """
    if i == j:
        return 0

    # Get the indices of the elements to be swapped
    a, b = chromosome[i], chromosome[j]

    # Create masks to exclude the i-th and j-th elements
    mask = np.ones(len(chromosome), dtype=bool)
    mask[[i, j]] = False

    # Get the relevant rows and columns
    flow_i = flow_matrix[i, mask]
    flow_j = flow_matrix[j, mask]
    flow_k_i = flow_matrix[mask, i]
    flow_k_j = flow_matrix[mask, j]

    distance_b = distance_matrix[b, chromosome[mask]]
    distance_a = distance_matrix[a, chromosome[mask]]
    distance_c_b = distance_matrix[chromosome[mask], b]
    distance_c_a = distance_matrix[chromosome[mask], a]

    # Calculate the delta cost
    delta = np.sum(flow_i * (distance_b - distance_a) +
                   flow_j * (distance_a - distance_b) +
                   flow_k_i * (distance_c_b - distance_c_a) +
                   flow_k_j * (distance_c_a - distance_c_b))

    return delta
```

Listing 4.7: Code for computing the delta cost in two-opt more efficiently. Taken from the code repository

For the entire implementation please refer to `src/evolutionary_tools/greedy_optimizations.py` in the code repository.

4.6.2 Caching

In addition to the just discussed optimization measures, we wanted to speed up the computations even further after having noticed that after a certain number of generation, the same individual(s) seem to have the lowest fitness for many generations in a core. This of course implies that those individuals cannot be optimized further by 2-opt (at least this implication holds in the Lamarckian case). Hence, we implemented a caching system which stores those chromosomes/individuals which have gone through the 2-opt algorithm without being improved. This way, we can skip individuals if they cannot be improved anyways. The following code listing shows the resulting final code of 2-opt. Note that we cleared the cache after 125 generation, in order to avoid a memory error.

```
def two_opt(flow_matrix: ndarray, distance_matrix: ndarray,
    chromosome: ndarray
) -> ndarray:
    """
    Perform a 2-opt optimization on a given chromosome using delta costs
    to determine if a swap is beneficial and caching unoptimizable chromosomes
    for future function calls.
    :param chromosome: One-dimensional numpy array representing a chromosome.
    :param flow_matrix: Numpy array representing the flow matrix.
    :param distance_matrix: Numpy array representing the distance matrix.
    :return: Optimized chromosome as a numpy array.
    """
    global NUM_CACHE_HITS
    best_chromosome = chromosome.copy()
    n = len(chromosome)
    improved = True
    improved_changed = False
    cache_hit = False
    number_of_iteration = 0

    # Check for cache hit
    for chromosome in CACHE_FOR_OPTIMIZATION:
        if np.array_equal(chromosome, best_chromosome):
            NUM_CACHE_HITS += 1
            improved_changed = True
            cache_hit = True

    while improved and number_of_iteration < NUMBER_OF_ITERATIONS_FOR_OPT
          and not cache_hit:
        number_of_iteration += 1
        improved = False
        for i in range(1, n - 1):
            for j in range(i + 1, n):
                cost_delta = calculate_delta_cost_numpy(flow_matrix,
                    distance_matrix, best_chromosome, i, j)
                if cost_delta < 0:
                    # Perform the 2-opt swap
                    tmp = best_chromosome[j]
                    best_chromosome[j] = best_chromosome[i]
                    best_chromosome[i] = tmp
                    improved = True
                    improved_changed = True

    if not improved_changed:
        # Cache the chromosome if opt makes no sense
        CACHE_FOR_OPTIMIZATION.append(best_chromosome.copy())
    return best_chromosome
```

Listing 4.8: Final code of two-opt. Taken from the code repository

5 Evaluation

At last, we arrive at the evaluation of the algorithm and its different variants. As mentioned in the introduction, the goal of this work is to come as close as possible to the best known solution of an instance called `tai256c.dat` proposed by E. D. Taillard which is also part of the QAPLIB, see [BKR97]. The goal being a cost of 44759294. We will first discuss the standard and Baldwinian variants of the algorithm, both of which did not produce the best results we were able to achieve and at last present the Lamarckian variant with which we were able to obtain the best results, see section 5.2. Before presenting the best solution however, we will present a comparison of the three presented variants with regards to the quality of the results, their running times and the fitness values of the fittest individuals over the generations.

5.1 Comparison

The setup of the benchmarks and therefore evaluation the variants was as follows. We used four additional instances from the QAPLIB to evaluate the variants. Namely, `bur26a, chr18b, nug16a, tai60a`, see [BKR97]. Furthermore, we let the standard variant run with `POPULATION_SIZE = 100`, `NUMBER_OF_GENERATIONS = 1000`, `MUTATION_PROB = 0.3`, whereas the Baldwinian and Lamarckian variants were configured to run with `POPULATION_SIZE = 20`, `NUMBER_OF_GENERATIONS = 250`, `MUTATION_PROB = 0.1`. As we will see, these differences, especially in the population size and the number of generations, are mostly due to the running time difference in the standard and the Baldwinian and Lamarckian variants. The latter two being way more cost intensive due to the discussed 2-opt function, see section 4.6.1. The value of `MUTATION_PROB = 0.3` for standard was chosen after testing several different values, whereas the difference in the value for the non-standard variants was chosen with the same reasoning.

Table 5.1 shows the computed results of the different variants on the 5 problems from the QAPLIB. As can be seen, the Lamarckian variant clearly outperforms the other two variants. Not only that, but the standard variant produces significantly worse results than the two versions using the 2-opt method. Note that the Baldwinian fitness is the fitness after applying the two-opt function.

Problem	bur26a	chr18b	nug16a	tai60a	tai256c
Standard	5470965	1898	1710	8046794	48754696
Baldwinian	5432123	1548	1638	7555788	44907676
Lamarckian	5426670	1538	1622	7403044	44820112
Best	5426670	1534	1610	7205962	44759294

Table 5.1: Comparison of computed results of the different variants

One might think that applying the local optimization will bias the evolution too much, however it seems like the variants based on two-opt do not get stuck on local optima. Instead the Lamarckian variant was able to achieve the best known result on the `bur26a` instance. Similarly good results were able to be obtained by the Baldwinian and Lamarckain variants on all other instances except for the `tai60a` instance, where the results of the Baldwinian and Lamarckian variants are not as good, relatively speaking. Next, we want to compare the fitnesses of the fittest individuals in the population over the generations during algorithm runtime. This is shown in figure 5.1. Note that the y-axis of the different subplots might be slightly different and that it is scaled down by 10.000.000. Furthermore, the plot of the Baldwinian variant contains the fitness values with respect to two-opt, that is, the fitness values the individuals 'could have' if the swaps from two-opt would be applied. Also, since the Baldwinian and Lamarckian variant only ran 250 generations, we just continued the last best fitness value for the rest of the generations.

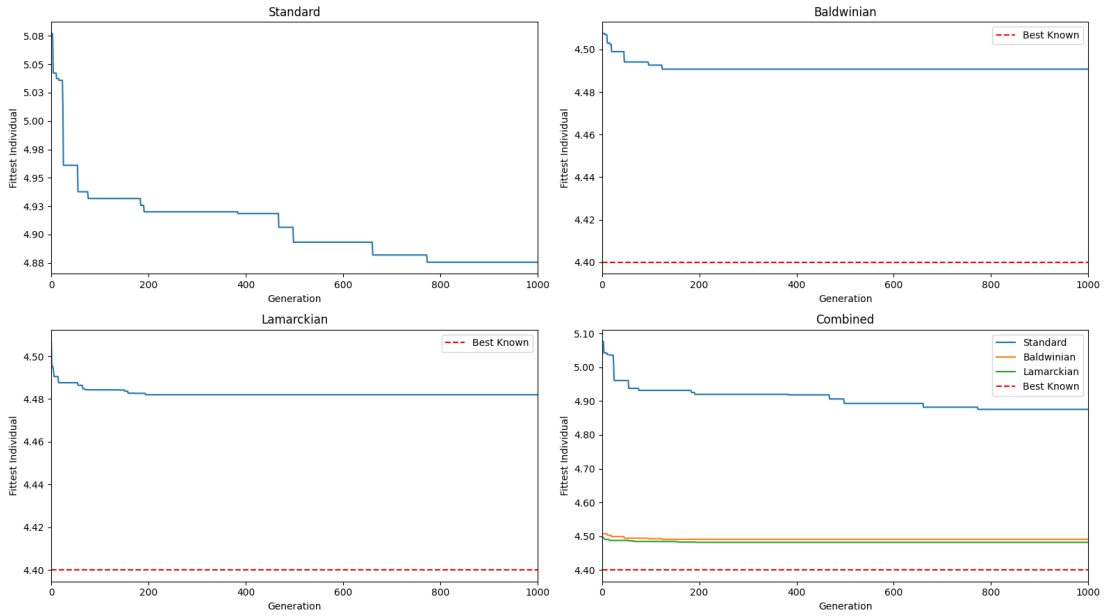


Figure 5.1: Comparison of the variants regarding the fittest individual over the generations. Values for Baldwinian and Lamarckian after generation 250 are the same value as in generation 250. Y-axes are scaled down by 10.000.000

We can immediately see, that the Baldwinian and Lamarckian methods have a big head-start. From the first generation on, their fittest individuals are way fitter than the fittest individual of the standard variant will ever be. For this, note that the individuals from the initial population are optimized using 2-opt when they are evaluated. This shows the power of the two-opt local optimization method. One can also see that after this dramatic drop off in the fitness values for the Lamarckian and Baldwinian variant, their fitness values slightly stagnate and seem to only increase when some lucky recombinations are found that enable for better optimizations. Note that the swap mutation does not do much for the Baldwinian and Lamarckian variants, since if there was a swap that could improve an individual, it would be found by two-opt. Instead it could merely help, if there was a swap which would not strictly decrease the fitness but lead to other swaps with two-opt which in turn result in a better final fitness.

As a last comparison, we want to discuss the downside of the Baldwinian and Lamarckian variants: their running times. The comparison of the running times of all variants on all the instances can be seen in table 5.2. Because the two-opt algorithm is applied to all chromosomes (which have not been cached), the running time increases drastically. For the `tai256c` instance, there is an increase from 1 minute to 50 minutes, that is, almost an hour. This shows the downside of using the Baldwinian and Lamarckian variants. Of course, if time is not a constraint, one can keep the algorithm running and make use of the benefits of the Baldwinian and Lamarckian variants, as we have done for obtaining the best result, see section 5.2.

Problem	**bur26a**	**chr18b**	**nug16a**	**tai60a**	**tai256c**
Standard	9.06	4.52	4.25	12.5	76.81
Baldwinian	42.18	17.8	13.72	227.15	5426.28
Lamarckian	34.06	18.14	12.11	238.33	5419.15

Table 5.2: Comparison of running times in seconds of the different variants

5.2 Best Result

Finally, we want to present the best result we obtained using evolutionary algorithms. Since the Lamarckian variant produced the best results across the boards in various tests, some of which are not presented in this work, we opted for choosing the Lamarckian variant for trying to obtain the best result.

We used the following parameters and functions to obtain our best result:

```
Functions used                          Hyperparameters:
Fitness function:       bulk_basic      Population size:         20
Selection function:     roulette_wheel  Number of generations:   1500
Recombination function: partially_mapped Number of facilities:  256
Mutation function:      swap            Mutation probability:    0.1
```

Listing 5.1: Configuration for best result

The choices of the fitness, selection, recombination and mutation function were already discussed in their respective sections. In the previous section 5.1 and figure 5.1 we saw that the running time of the Lamarckian variant is very high in comparison to the standard variant. Nevertheless we wanted to have an at least somewhat reasonably sized population, which is why we opted for a population size of 20. This is of course a balance since the Lamarckian variant seems to stagnate after some point, that is, decrease in speed of improvement and a higher population size might mitigate this effect. This might of course also be due to the fact that the value of the solution is getting quite close to the optimal solution and there is not 'much to optimize' anymore. Regarding the number of generations, we just chose a sufficiently high value which would not run forever since there is no downside in having many generations. That is, there is no overfitting like there would be with neural networks. The number of facilities is given by the problem instance `tai256c` and the mutation probability was chosen relatively low. We have already discussed that the swap mutation does not make a lot of sense with 2-opt, were however happy with our final result, which is why we did not try another mutation function.

Figure 5.2 shows the fittest individuals' fitness over the course of the generation for our achieved result. The final achieved value was 44792836 which is only 0.07% bigger than the best known result of 44759294.

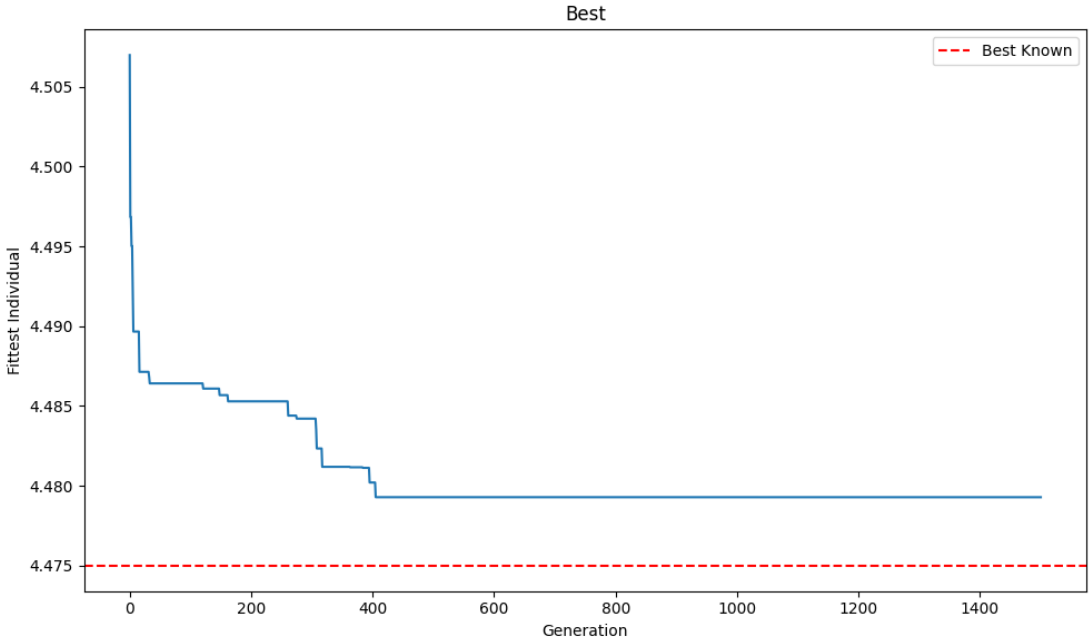


Figure 5.2: Fittest individual over the generations of best result obtained on the `tai256c` instance. Y-axis is scaled down by 10.000.000

From the figure, one can deduce that we could have also stopped the algorithm after about generation 450. Since after that, the solution did not get any better. Of course,

one does not know that beforehand. After completing multiple runs, it seems that there is some luck involved in obtaining such a result because the algorithm at times finds a local optimum and does not seem to be able to improve upon that. We might even be facing the same situation with the above result and there could be a better solution that is relatively easy to obtain with some luck. Note that any solution can be obtained with sufficient luck since the initialization of the population is random.

For the complete logs of the algorithm execution leading to the best result please refer to the `best_result` directory in the code repository.

6 Conclusion

This work demonstrated the application of evolutionary algorithms to solving the Quadratic Assignment Problem, with a focus on analyzing different algorithmic variants and optimization techniques. By using Baldwinian and Lamarckian variants, combined with the 2-opt local optimization strategy, we were able to significantly improve the obtained results.

Several options for further exploration remain. Future work could explore alternative selection algorithms, such as stochastic universal sampling, to potentially enhance the balance between diversity and selection pressure. Another promising direction would be to only apply the optimization (which must not necessarily be 2-opt) to some individuals in the population instead of all. This would reduce the computational load, thus enabling larger population sizes for broader exploration of the solution space. Lastly, choosing alternative mutation methods to swap mutation could further improve the results. The swap mutation is useful in its simplicity however does bring great benefits when used with 2-opt which already takes care of finding the best swaps.

By addressing these problems, the algorithms presented in this work can be further refined to improve both running times and solution quality. Doing so is particularly interesting because the Quadratic Assignment Problem remains one of the most challenging NP-hard problems, with significant theoretical interest and a wide range of practical applications beyond facility location problems, see [Loi+07].

Bibliography

[Bak87] James E. Baker. "Reducing Bias and Inefficienry in the Selection Algorithm". In: *International Conference on Genetic Algorithms*. 1987.

[Bal96] J. Mark Baldwin. "A New Factor in Evolution". In: *The American Naturalist* 30.354 (1896), pp. 441–451. ISSN: 00030147, 15375323.

[Bar+14] Thomas Bartz-Beielstein et al. "Evolutionary Algorithms". In: *WIREs Data Mining and Knowledge Discovery* 4.3 (2014), pp. 178–195. DOI: https://doi.org/10.1002/widm.1124. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1124.

[BKR97] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. "QAPLIB – A Quadratic Assignment Problem Library". In: *Journal of Global Optimization* 10.4 (June 1997). Browsable at https://coral.ise.lehigh.edu/data-sets/qaplib/, pp. 391–403. ISSN: 1573-2916. DOI: 10.1023/A:1008293323270.

[Dav91] Lawrence Davis. *Handbook of genetic algorithms*. English. VNR computer library. New York: Van Nostrand Reinhold, 1991. ISBN: 9780442001735.

[ES15] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. ISBN: 978-3-662-44874-8. DOI: 10.1007/978-3-662-44874-8.

[GL85] David E. Goldberg and Robert Lingle. "AllelesLociand the Traveling Salesman Problem". In: *Proceedings of the 1st International Conference on Genetic Algorithms*. USA: L. Erlbaum Associates Inc., 1985, pp. 154–159. ISBN: 0805804269.

[KB57] Tjalling C. Koopmans and Martin Beckmann. "Assignment Problems and the Location of Economic Activities". In: *Econometrica* 25.1 (1957), pp. 53–76. ISSN: 00129682, 14680262.

[Loi+07] Eliane Maria Loiola et al. "A survey for the quadratic assignment problem". In: *European Journal of Operational Research* 176.2 (2007), pp. 657–690. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2005.09.032.

[SG76] Sartaj Sahni and Teofilo Gonzalez. "P-Complete Approximation Problems". In: 23.3 (July 1976), pp. 555–565. ISSN: 0004-5411. DOI: 10.1145/321958.321975.

[Wik24a] Wikipedia contributors. *Lamarckism — Wikipedia, The Free Encyclopedia*. [Online; accessed 24-January-2025]. 2024.

[Wik24b] Wikipedia contributors. *Quadratic assignment problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-January-2025]. 2024.

[YG10] Xinjie Yu and Mitsuo Gen. *Introduction to Evolutionary Algorithms*. London: Springer London, 2010. ISBN: 978-1-84996-129-5. DOI: 10.1007/978-1-84996-129-5.

List of Figures

Listings