Task 0: Implementation and Study of Evolutionary Algorithms Compared to Non-evolutionary Methods

March 11, 2025

Contact

michal.przewozniczek@pwr.edu.pl
pawel.myszkowski@pwr.edu.pl
konrad.gmyrek@pwr.edu.pl
marcin.komarnicki@pwr.edu.pl

Exercise Goal

The aim of the exercise is to familiarize oneself in a practical manner through the independent implementation of the evolutionary algorithm metaheuristic. Despite its name suggesting otherwise – it is not an algorithm – it is a metaheuristic that mimics natural evolution through selective pressure and natural selection. To apply it, one must define a potential solution (individual), ways of its modification (mutation), combination (crossover), and evaluation of the solution quality (evaluation function).

Evolutionary Algorithm (EA)

A small clarification regarding the terminology of the methods used.

An algorithm, by definition, is a finite sequence of operations that allows solving a problem in a finite amount of time. The performance time of the algorithm depends on the size of the input data (i.e., the computational complexity of the algorithm) and guarantees finding the optimal solution. In the case of heuristics, the computational complexity can vary (depending on the creativity of the programmer) and does not guarantee finding the optimal solution. In this context, a metaheuristic is referred to as a higher-level heuristic, i.e., a heuristic that determines how to apply lower-level heuristics. An example of a lower-level heuristic is initialization, selection, crossover, etc.

When do we use (meta)heuristics? Primarily, when we do not have an algorithm.

An evolutionary algorithm (EA) is based on the pseudocode 1. EA initially creates (usually randomly) a population of solutions. Then it evaluates the quality of each individual. The next step checks the stopping conditions: whether an acceptable solution has been achieved and/or the generation limit has been exceeded and/or the maximum number of evaluations has been exceeded. The selection of individuals (suggested method of tournament or roulette) for the next population occurs before the action of the crossover and mutation operators, which build individuals of the new generation. Then, the new solutions are evaluated, and the EA cycle closes upon checking the stopping conditions.

```
begin
    t := 0;
    initialise(population[t]);
    evaluate(population[t]);
    while (not stopping_condition) do
        begin
            population[t + 1] := select(population[t]);
            population[t + 1] := crossover(population[t + 1]);
            population[t + 1] := mutate(population[t + 1]);
            evaluate(population[t + 1]);
            t := t + 1;
        end
    return best_solution(population[t]);
end
```

Listing 1: Pseudocode for evolutionary algorithm

In Pseudocode 2, another method of constructing the flow of individuals in the Evolutionary Algorithm is presented. Operators creating a single individual were used there; however, without significant obstacles, this pseudocode can be extended for operators returning two individuals.

```
begin
    t := 0;
    initialise(pop(t));
    evaluate(pop(t));
    while (not stop_condition) do
        begin
            while (pop(t+1).size() != pop_size)
                begin
                    P1 := selection(pop(t));
                    P2 := selection(pop(t));
                    if (rand(0,1) < Px)
                        O1 := crossover(P1, P2);
                    else
                        O1 := P1;
                    O1 := mutation(O1, Pm);
                    evaluate(O1);
                    pop(t+1).add(O1);
                    if (the_best_solution < O1)
                        the_best_solution := O1;
                end
            t := t + 1;
        end
    return the_best_solution
end
```

Listing 2: Pseudocode as individual flow in the Evolutionary Algorithm

In Pseudocode 2, the current population, Pop(t), serves as the basis for selection operations – individuals P1 and P2 are chosen, and then the probability of crossover is checked. If it occurs, the operation is executed, and the offspring individual O1 is mutated. To avoid "wasting" the progress of the EA, if no crossover operation occurs, individual O1 is copied from parent P1 and then mutated. Subsequently, it undergoes evaluation and is added to the next population, Pop(t+1). The procedure can be easily expanded to include a crossover operation that returns two offspring individuals.

As part of the task, you are required to write your own method of Genetic Algorithm (GA). GA is used to search for the best solutions to optimization problems. This task is preliminary and is not intended to demonstrate the full spectrum of possible GA applications. However, it can provide intellectual entertain-

ment for those interested in this subject matter.

This task is preliminary and serves to familiarize oneself with the process of conducting research in the field of optimization.

CVRP

The Capacitated Vehicle Routing Problem (CVRP) is an NP-hard problem, and due to its difficulty, metaheuristic methods are commonly used to solve it. First, we will describe the Traveling Salesman Problem as it is closely related to CVRP.

The Traveling Salesman Problem (TSP) consists of n locations and a matrix of distances between these locations. In the case under consideration, the matrix is symmetrical, which means the distance from location i to location j is the same as from location j to location i. The goal is to find the shortest possible route that visits each location exactly once:

$$\min f(\langle x_0, \ldots, x_i \rangle) = \sum_{i=1}^{n-1} d_{x_i, x_{i+1}}, \tag{1}$$

where x represents the proposed solution route:

$$x = \langle x_i, \ldots, x_n \rangle,$$

and d_{x_i,x_k} is the distance between two specified locations: the i-th and the k-th location.

In CVRP we have a special location — the depot. Additionally, each location has a demand for "resources" which must be delivered at once (i.e., cannot be split into multiple trips). For such a defined problem, we have two constraints: (1) each location must be visited exactly once (excluding the depot where we resupply our vehicle), and (2) the size (capacity) of the vehicle.

The solution is a set of routes from the depot to locations (one or more) and back, which fulfills the demand of each location for the required "goods."

The full problem description and input/output file formats together with instances can be found at the following link:
http://dimacs.rutgers.edu/programs/challenge/vrp/cvrp/

You should consider 7 different instances from Set A (http://vrp.galgos.inf.puc-rio.br/index.php/en/).
Basic instances (5): A-n32-k5, A-n37-k6, A-n39-k5, A-n45-k6, and A-n48-k7.
Hard instances (2): A-n54-k7 and A-n60-k9.

Suggested problem representation, specialized operators and hints

To solve cVRP, one can use a representation similar to the classical TSP, where the solution is a sequence of locations representing the order of visiting them. Typically, this involves transforming the TSP solution into cVRP through considering the vehicle capacity and moving back to the depot when the capacity could be exceeded.

An individual in the TSP problem can be **represented** as a sequence of locations, e.g. 35124. We start from location 3, then visit location 5, and so on. Finally, from location 4, we return to the location 3. In this representation, each location must appear exactly once.

The **mutation** operator for the above vector representation can be implemented by randomly swapping two locations (a *swap*). For example, we select two genes in a chromosome and swap their values. If we swap the second position (location 6) and the penultimate position (location 3), a new individual is created.

Another mutation type is **inversion**, where two random positions in the sequence are selected, and the order of locations within this selected segment is reversed.

Below (see Table 1) is a structured representation that illustrates these mutation operations clearly.

Mutation Type	Original Sequence	Mutated Sequence
Swap	5671234	5371264
Inversion	5671234	5321764

Table 1: Swap and inversion mutation examples

The **crossover** operator combines two individuals to form a descendant (or two, depending on the operator). In the simplest form, crossover finds a random intersection point and for the descendant, the first part is taken from one parent, the second part from the other. However, if we were to use such a simple crossover we would have to repair solutions afterwards. Below we present specialized crossover operators that operate on permutations.

The OX (Ordered Crossover) operator selects two points within one parent individual. The subsequence between these points is directly copied into the offspring without changes. The remaining positions of the offspring are filled sequentially with locations copied from the second parent individual. An example illustrating the OX crossover operator is presented below (see Figure 1).

P1 |123456789| x P2 |574913628| → O: |793456128|

Figure 1: OX example: subsequence from P1, completed by the order in P2.

PMX (Partially Matched Crossover) operator is a bit more complicated in its operation. Two random points are chosen that determine the crossover subsequence. In the parent individuals, the section to be mapped is identified, and the corresponding locations (genes) are matched. In the offspring individuals, this mapping section is copied from one parent, and the missing locations are copied in order from the other parent. An example of how the PMX operator works is shown below (see Figure 2).

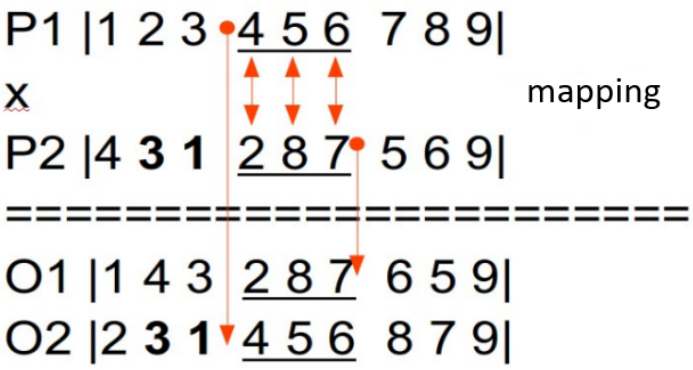

Figure 2: PMX example

An interesting example of a crossover operator is CX (Cycle Crossover). The principle of its operation is to look for cycles in which certain locations (genes) form a "cycle" from which subsequent genes should be copied. As a result, an offspring is created in which these cycles are taken alternately from different parents. An example of how the CX crossover operator works is shown below (see Figure 3).

The **selection** operator is a very important element of EA – it decides on the selection pressure.

If the pressure is too high, the population quickly becomes uniform and gets stuck in a local optimum. If the pressure is too low (or absent), the EA has difficulty convergence. The most commonly used selection methods are tournament and roulette.

The **tournament** selection method involves randomly choosing N individuals from the population and selecting the individual with the best fitness as a parent. If N is very large, there is practically no selection

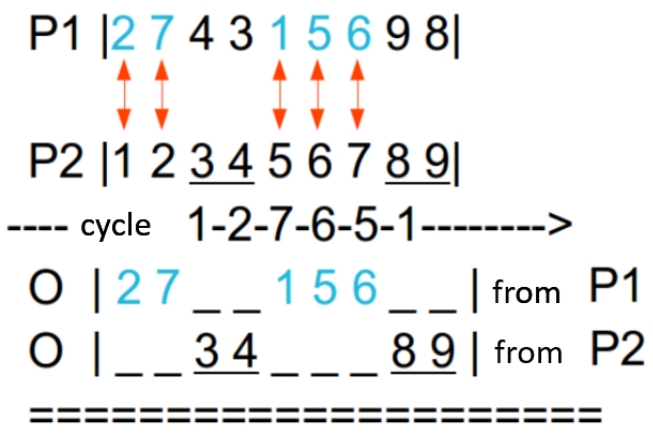

Figure 3: CX example

pressure. When N is equal to the size of the entire population, the tournament method becomes more similar to the elitist method (described below), meaning the best individual is chosen for the population.

Another selection method, called **roulette**, allows individuals to be selected with a probability proportional to their fitness value. The fitness values of all individuals are summed, and then sectors proportional to those values are placed on the roulette wheel. Consequently, selection more strongly favors individuals with higher fitness, yet every individual still has some chance—whether small or large—of advancing to the next population.

An additional parameter is **elitism**, it determines how many best individuals pass on to the next generation without changes. The elitism parameter can be very useful for large populations (we do not lose the best ones), but for small ones, it is very disastrous because it can ”anchor” the population in local optima.

Greedy Algorithm for CVRP

The Greedy Algorithms solve problems by making the locally optimal choice at each stage of solution construction. For CVRP we propose to create a solution by starting at the depot and then visiting the closest unvisited location, returning to the depot each time when the vehicle can not supply all of the demand of the next city.

Task realization

The exercise implementation has several goals, which can be specified as follows:

- Getting acquainted with metaheuristics – evolutionary algorithms
- Defining, implementing, and solving CVRP
- Implementing a genetic algorithm:
  - Individual (solution)
  - Evaluation function (calculating the distance based on the individual)
  - Population management
    - Crossover – appropriate for the type of problem
    - Mutation – appropriate for the type of problem
- Constructing another metaheuristic like Tabu Search or Simulated Annealing for comparison.

- Implementation of methods in any object-oriented language.
- Investigating the influence of values for different parameters:
  - Mutation probability P_m
  - Crossover probability P_x
  - Selection
  - Population size pop_size
  - Number of generations gen

  on the efficiency and effectiveness of the evolutionary algorithm, as well as selecting the best configuration of those parameters for final comparison.
- Preparing a report from the exercise.
- Showing in graphs the change in fitness value (value of the optimized objective function) in successive generations: best individual, average value in the population, and worst individual.
- Compare the operation of the evolutionary algorithm with the non-evolutionary optimization methods selected by yourself (suggested methods are **random search** and **greedy algorithm**).
- Please pay attention to the obtained result, operating time, and the number of evaluations of the optimized objective function. Please show and discuss the most interesting results.

The exercise report should include all the points required to complete the task.

Implementation – practical guidelines

When implementing the exercise, it is suggested to create a computer program that will cover:

- Loading data from a file, saving the problem to the computer's memory (logger, problem),
- Solution evaluation (individual) in the context of the selected problem,
- Managing algorithm parameters (configuration),
- Controlling the course of metaheuristics (EA),
- Storing the solution (individual) and genetic operators (crossover, mutation),
- Managing the population – in particular selection, initialization, and creation of a new one,
- Logging method progress (logger) to an external file (csv format?)

The program code should be designed flexibly enough to allow its use in subsequent main project. For task realization, it is necessary to compare the EA results with other non-evolutionary methods. Two methods are indicated: random and greedy. In the case of the random method, it is sufficient to run the method as many times as there are solutions reviewed by the EA (number of generations x number of individuals) to compare them. For the greedy algorithm, which is deterministic (always providing the same solution), repetitions are not needed.

The results of the methods should be averaged.

From the programming perspective, one should consider whether creating individuals requires a deep copy or if a shallow copy suffices. For code efficiency, also consider how the individual is encoded (which/what collection is used for representation, sequence of locations is only our proposition) – this will have a significant impact on the speed of the code. Similar considerations should be made when implementing the tournament – consider which (if any) collection might be useful in the implementation.

Reporting results and suggestions related to their analysis

The Evolutionary Algorithm (EA) has an element of non-determinism, and the results returned by successive runs may differ. Therefore, the results of studies conducted using the indicated files must be repeated ten times, averaged, and the standard deviation provided.

For each result/chart, the used method configuration should be provided: population size (pop_size), number of generations (gen), crossover probability (P_x), and mutation probability (P_m). Additionally, which type of crossover/mutation/selection was used? If it requires additional parameters, e.g., tournament size (Tour), this value should also be provided in the configuration.

For presenting research results, it is suggested to use a summary table format like Table 4. Suggested format for presenting the results of metaheuristic performance based on the example

	Optimal	**Random Alg. [10k]**				**Greedy Alg.**	**Evol. Alg. [10x]**				**TA or SA [10x]**			
Instance		**best***	**worst**	**avg**	**std**	**best***	**best***	**worst**	**avg**	**std**	**best***	**worst**	**avg**	**std**
Instance 1	90.2	102.5	98.7	138.9	117.8	100	95.2	140.0	118.4	8.3	**93.4**	139.7	116.3	7.5
Instance 2	92.0	100.0	97.0	139.0	118.0	102	94.0	141.0	119.0	9.0	**92.2**	140.2	116.5	8.6

Figure 4: Example of comparison table that your reports should contain

Where: - Random Alg. [10k] indicates 10,000 runs of the random algorithm - Greedy Alg. indicates a single run of the deterministic greedy algorithm - Evol. Algorithm [10x] indicates ten iterations of the evolutionary algorithm and provides statistical values for these runs. - TS or SA [10x] indicates results of another metaheuristic (Tabu Search or Simulated Annealing); ten iterations of the method and provides statistical values for these runs. - best* – indicates the best value found from each 10 runs - worst – indicates the worst value found from each 10 runs - avg – indicates the average value from each 10 runs - std – indicates the standard deviation value calculated at the average (avg)

Note: The best* value for the evolutionary algorithm is taken as the best value out of 10 runs. The average (avg) represents the value calculated from the best solution of each run. Table 4 shows the effectiveness of the examined solutions, i.e., how good (on average) solutions we can achieve by running the method.

Exercise Implementation Assessment

The realization of the task is evaluated based on the components according to the following points scheme:

- 1pt - Implementation and testing of the genetic algorithm.
- 1pt - Implementation and testing of the random algorithm.
- 1pt - Implementation and testing of the greedy algorithm.
- 2pts - Implementation and testing of another metaheuristic (Tabu Search TS or Simulated Annealing SA).
- 3pts - Studying the impact of the configuration of both metaheuristics on their effectiveness/efficiency.
- 2pts - Comparison of the effectiveness of all methods performance on 7 instances - instances provided along with their descriptions. Tests must be conducted on 5 basic instances (A-n32-k5, A-n37-k6, A-n39-k5, A-n45-k6, and A-n48-k7) and 2 hard instances (A-n54-k7 and A-n60-k9).

Hints

Initial parameters useful for tuning EA:

$$\text{pop\_size} = 100, \quad \text{gen} = 100, \quad P_x = 0.7, \quad P_m = 0.1, \quad \text{Tour} = 5$$

Note: Optimal values of parameters will vary depending on the operators used and the case of the chosen problem.

For mutation probability values, the use of mutation probability (P_m) depends on:

- Individuals, for mutations acting on the individual. An example of such mutation is inversion, where only the targeted action on a given individual is evident. Then $P_m = 0.1$ means that statistically 10% of individuals in the population will be mutated. For example, in a population of 100 individuals, about 10 individuals will be mutated.
- Gene, for mutations acting only on a small fragment of the individual. An example of such mutation is the swap operator (in TSP swaps the order of two locations). In this case, since the mutation begins to spread significantly from $P_m = 0.01$, for a problem with TSP of 100 locations and 100 individuals in the population, on average, about 200 genes (100*100*0.01 operations on two genes) will be changed.

Note: In order to make a correct comparison we must give an equal chance to each algorithm, thus the total number of solution evaluations for each method should be **constant**, with the exception of the greedy algorithm. For the genetic algorithm, we can assume that the number of evaluations is equal to the size of the population multiplied by a number of generations.

Auxiliary Questions

1. How does selection (tournament size *Tour*) affect the operation of EA? What happens if *Tour*=1 and what if *Tour=pop_size*?
2. How do the EA's behaviors differ with *Tour*=1 and *Tour*>1?
3. Does the rule apply correctly to the CVRP problem?
4. Can mutation be too small/large?
5. What role does the mutation operator play?
6. Can a crossover be too small/large?
7. What role does the crossover operator play?
8. What happens if we turn off crossover and/or mutation?
9. Is elitist selection necessary and beneficial?
10. How does the size of the population and the number of offspring affect the efficiency/effectiveness of EA?

References

1. Arabas J. Lectures on Evolutionary Algorithms Arabas.
2. Goldberg D. Genetic Algorithms and their Application, WNT.
3. Michalewicz Z. Genetic Algorithms + Data Structures = Evolution Programs, WNT.
4. Krunoslav Puljić, Robert Manger, "Comparison of eight evolutionary crossover operators for the vehicle routing problem", MATHEMATICAL COMMUNICATIONS Math. Commun. 18(2013), 359–375
5. Potvin, Jean-Yves. 1996. Genetic algorithms for the traveling salesman problem, Annals of Operations Research, Volume 63, pages 339-370.