Assignment 2 - Evolutionary Computing

Michał Jóźwik, Lukas Ostrovskis, Martynas Krupskis, Denis Tsvetkov

March 2022

1 Running the code

Open **Group_75_code/src** directory with PyCharm, inside you will find main.py file, which you can run through PyCharm. You can also run this code via command line using *python main.py* command (tested with Python 3.9.5) in the provided directory. You need to install numpy and matplotlib before running the code. The code in main.py will generate 3 maze solutions and a TSP solution for data specified in data folder. Output of this script will be written to the data folder in the format "*<maze difficulty>_solution.txt*" and "*TSP solution.txt*". Be aware that running main.py will overwrite current solution files in the data folder.

2 Ant Colony Optimization

1. The features that we encountered:
   - Loops
   - Dead ends
   - Open rooms
2. Pheromone dropped by a single ant is: $\nabla\tau_k = \frac{Q}{L_k}$

   Pheromone dropped by k ants: $\sum_{i=1}^{k} \nabla\tau_i$

   We drop pheromone to attract more ants to follow a certain path. The purpose of the algorithm is to find the shortest path between two points A and B.
3. $\tau_{ij} = (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m} \nabla\tau_{ij}^{k}$

   Here $\boldsymbol{\rho}$ ($0 \leq \rho \leq 1$) is the evaporation constant. With each iteration $\boldsymbol{\rho}$ (or equivalently $100 \cdot \boldsymbol{\rho}\%$) of the paths' pheromone evaporates. Pheromone evaporation helps to eliminate pheromone trails that might misguide ants without destroying any knowledge gained from previous environments.
4. ```
   function ACO(maze, start, end, n_ants, n_gen, q, evap):
       min_route ← None
       for i from 0 to n_gen do
           routes ← List()
           for j from 0 to n_ants do
               ant ← new Ant(maze, start, end)
               r ← ant.find_route(q)
   ```
   ```

```
8:          routes.add(r)
9:          if min_route is None then
10:             min_route ← r
11:         if r.shorter_than(min_route) then
12:             min_route ← r
13:     sort(routes)
14:     maze.update_pheromone(take_a_third(routes))
15:     maze.evaporate_pheromone(evap)
16: return min_route
```

Note: take_a_third only considers the shortest 33% of the routes. This way the ACO converges to an optimal solution way quicker than the original algorithm. However, one drawback is that sometimes more ants are required so that it doesn't get stuck in some local minimum.

5. Our 2 main concerns were how to deal with dead ends and loops and it turned out that the solution to both of these problems was relatively simple - in the ant class we introduced a matrix with the same shape as the maze that was going to keep track of the coordinates visited by an ant(initially the matrix was filled with 0s and whenever an ant visited a coordinate we updated the entry at that position to 1). Then whenever an ant would move in a new direction we first check whether that coordinate has already been visited and if that is the case we set the probability of moving to that direction to 0. If the probability for each of the 4 directions is 0 we remove the last taken direction from the route and go back to the previous coordinate and continue looking for a path. Otherwise, we take a random a number between 0 and 1 and if it is above p (in our case 0.875) we take the direction with the highest probability. If it is smaller than 0.875 then we pick a random direction given a probability computed as follows:

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}(t)^\alpha \cdot \eta_{ij}^\beta}{\sum_{m \in C_i^k} \tau_{im}(t)^\alpha \cdot \eta_{im}^\beta} & \text{if j is not yet visited} \\ 0 & \text{Otherwise} \end{cases}$$

where $\tau_{ij}(t)$ is the pheromone in that direction, η_{ij} is the distance (in our case 1 so it and β don't matter), and C_i^k is the set of all coordinates that we can visit from i. We also decided that since the distance between the coordinates is the same, the probability of choosing a direction would be proportionally equal to pheromone in that direction over the sum of the pheromones in all directions (that is why we set α to 1). And finally, for dealing with open rooms we adapted the algorithm as follows: instead of initializing the pheromone equally we take into consideration the Manhattan distance to the closest wall - the closer the position is to a wall the higher the pheromone. This way the ants will prioritize paths closer to the walls and thus will escape open rooms quicker and via a shorter path.

6. All of the parameters were tuned using the experiment approach, where we tested multiple different combinations of parameters and looked at the average path length, shortest path length and the heatmap of the pheromone matrix. Using this method we found a set of decent parameters, which are represented by the 4-tuple $(n, num\_gen, Q, evap\%)$, where n is the number of ants in a single generation, num_gen is the number of generations, Q is the pheromone

strength and *evap*% is the percentage of pheromones that gets evaporated after each generation. These are the results:

- easy maze (25, 10, 100, 0.8)
- medium maze (100, 50, 200, 0.8)
- hard maze (120, 60, 400, 0.8)

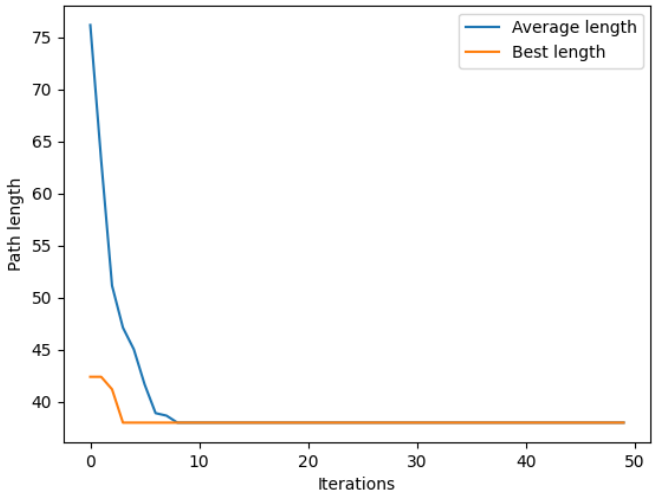

Figure 1: Convergence on the easy maze with 25 ants

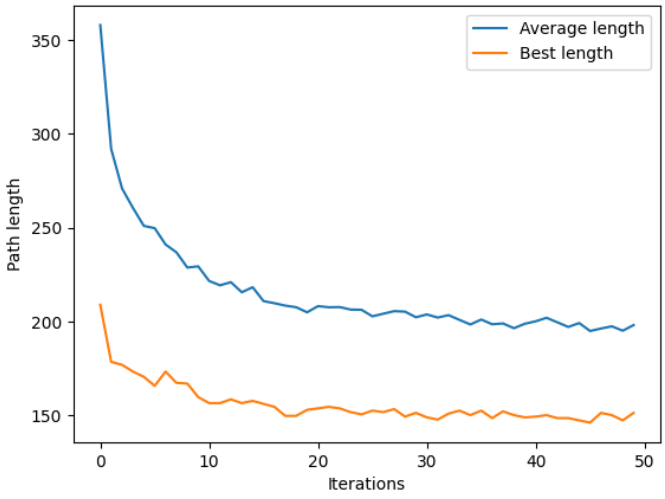

Figure 2: Convergence on the medium maze with 100 ants

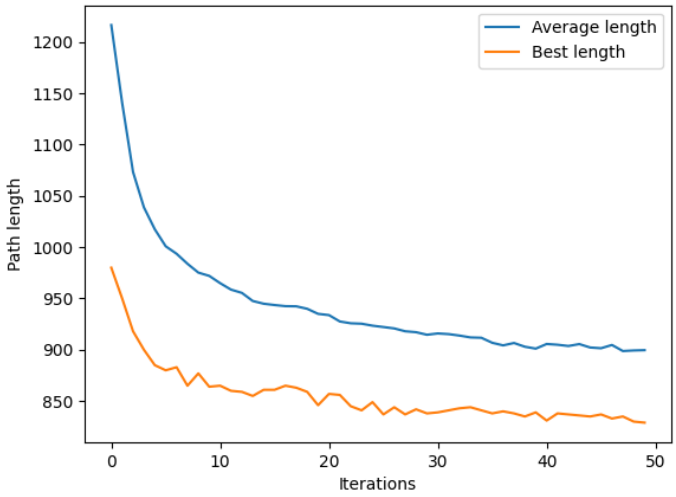

Figure 3: Convergence on the hard maze with 125 ants

The overall speed of convergence is very dependant on the first two parameters as they have a significant impact on the running time, which could be described by this time complexity $O(n \cdot num\_gen \cdot A)$, where A is the running time of the single ant finding the route from start to finish.

7. The size of the maze has the biggest influence on the time that a single ant spends looking for an end node. Furthermore the complexity of the maze requires a lot more ant to traverse the different paths and find the optimal route. Those ants also have a higher probability of creating more sub-optimal routes and thus influence other onto those paths. That again makes the simulation require a lot more generations as a lot of ants need more time to settle on a single optimal path. In general we can conclude that a maze with a higher complexity and bigger size makes the time required to find a near-optimal path significantly longer and also more error prone as there is much more room to settle on the local optimum and miss the global optimum, which is the path we generally want.

3 The Travelling Robot Problem

8. **The Travelling Salesman Problem**: given the list of the cities and the distances between each pair of cities find a path that routes through all the cities exactly once and **ends up in the start city**.

9. Differences from the classical problem in our case:

    - Due to the stochastic nature of the Ant Colony Optimization algorithm which we use to calculate the distance between different nodes, the distances between node A and B, and node B and A can end up being different, thus not symmetric, unlike the original problem definition.

- We have defined both a starting and an ending node / point, which are not the same and shouldn't change. For this reason our TSP solution has fixed start and end point, therefore the whole route will not be a cycle, given that end point is not the same as the start point.

10. Computational Intelligence techniques are appropriate for solving a TSP, because in the TSP number of paths grows exponentially with number of nodes, meaning that brute-forcing is infeasible solution for larger test sets. Computational Intelligence techniques allow us to explore the solution space and converge towards the optimal solution intelligently without searching through the whole solution space, thus greatly decreasing the complexity of finding an optimal or a good enough solution.

11. In our solution genes represent the product - each gene is an index of one of the products. To encode the chromosomes, we'll index the products (from 0 to n-1 for n products). Each chromosome will contain exactly 1 of each index - a permutation of the n indices. Thus a chromosome represents the order in which products will be picked. This way the path from the starting point to the first product and the path from the last product to the ending point are implicitly encoded, as those fixed positions are given and always fixed for each and every unique chromosome.

12. Fitness function of a chromosome is **the inverse of the sum of the distances from each consecutive two products in a chromosome including the distance from start to the first product and from the last product to the end**. We chose inverse, because GA aims to find a solution that maximizes fitness value, but doing that would increase the sum of the distances of a path. Taking an inverse of the sum of the paths means that maximizing fitness value will decrease the sum of the distances of the path and will lead to an optimal solution.

13. We used Roulette Wheel Selection with fitness ratio for each chromosome representing the probability of it being selected for the next crossover. The Roulette Wheel Selection is implemented by:

    (a) Generating a random number between 0 and 1

    (b) Checking the slice of the Roulette the generated number falls in

    (c) Selecting the chromosome corresponding to the slice.

14. Both mutation and cross-over algorithms are guarded by mutation probability x and cross-over probability y, meaning that mutation will occur with probability x and cross-over will occur with probability of y. In case they do not occur our operators just return the input chromosomes.

    Mutation:

    (a) Select 2 random numbers from 0 to length of the chromosome and assign them to *left* and *right* variables.

    (b) Swap elements in positions denoted by left and right.

    (c) Return the new chromosome

Example:
Chromosome: 1 7 5 6 4 3 2

left = 2
right = 5
Mutated_Chromosome: 1 7 3 6 4 5 2

Cross-over: For cross-over we used something called Order Crossover or OX[1], which preserves the order of elements from both parents. The OX is describer by the following algorithm:

(a) Select 2 random numbers from 0 to length of the chromosome and assign them to left and right variables.

(b) The range between left and right will be part of the chromosome from Parent 1 that will be preserved, meaning its genes and positions in the chromosome are kept fixed.

(c) Any other missing elements are taken from Parent 2 chromosome in the order of appearance. Important thing here is that we don't include any more genes that were preserved, thus not introducing any duplicates in the offspring.

Example of OX:
Parent 1: 1 7 5 6 4 3 2
Parent 2: 5 6 2 1 4 3 7
left = 2
right = 5
preserved = x x 5 6 4 3 x
X's are placeholders for genes from Parent 2 that do not appear in *preserved*.
In the order of the appearance that's 2 1 7, therefore:
Offspring = 2 1 5 6 4 3 7.

15. Chromosomes will never have non-unique genes is an invariant of our Genetic Algorithm and it is guaranteed by:

    - The initial population is a list of chromosomes, each of which is a permutation of all the product indices without any duplicates.

    These chromosomes are then only modified by two operators Mutation and Crossover:

    - In case mutation happens any two random genes are swapped meaning that no chromosome will contain duplicates after mutation. In case mutation does not happen we return the same chromosome.
    - In case crossover happens, our crossover operator ensures that no duplicate is introduced as we draw only non-duplicate values into the offspring as described in the 3rd step of the Crossover Operator in the last question. In case Crossover does not happen, we return two original parents, who do not have any duplicates.

    Thus we ensure that at no point in the algorithm any chromosome will have duplicate genes.

[1] https://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/Order1CrossoverOperator.aspx#:~:text=Order%201%20Crossover%20is%20a,they%20appear%20in%20parent%202

16. To prevent local optimum we use mutation operator to introduce random mutations in chromosomes. It helps us avoid converging to some local optimum, as it allows to search in some random and possibly unexplored space.

17. Elitism allows the best performing genes to be carried over to the next generation without any modifications (crossover or mutation). We did not perform elitism, as according to our testing it was able to converge to results equal or close to brute-force solution.

18. Genetic Algorithm aims to optimize for fitness function, meaning that we are looking for an individual with the highest fitness value. Not knowing correct fitness function means optimizing for incorrect goal and thus might lead to results that are unwanted.

19. Survival functions allow us to see how long an individual chromosome survives over generations. We don't always want the strongest individual to survive cause that might be leading us to a local and not a global optima, therefore survival function is a good way to mitigate risk of converging to local optima. As for the selection, we believe we could track how long each chromosome has lived and as survival function suggests take into consideration their age when selecting parents, the larger the age value, the less likely it is to survive/be selected. Of course, increasing crossover / mutation probabilities to allow for a wider range of variety within individuals is another great way to mitigate risk of local optima.