Computational Intelligence 2022-2023

Assignment 2

Genetic Algorithm and Ant Colony Optimization

Group 47

Question 1.1 - Answer 1.1

The TSP problem is usually defined as follows: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" ~Wikipedia

It means that we start at some point and have to end at the same point, visiting each city exactly once and travelling the smallest distance.

Question 1.2 - Answer 1.2

- In our instance of the problem the products (nodes) are not singularly connected i.e. there are multiple routes that can be taken from product A to product B.
- The relative distances are also not known in this scenario, so we are going to use the Ant Colony Optimization to find them.

Question 1.3 - Answer 1.3

Computational Intelligence methods are appropriate for solving TSP since the computational space of the problem is way too large to approach it using standard methods. The characteristics that allow us to tackle intractable problems are:

- Parallelism - GAs allow for evaluating multiple candidates at the same time, speeding up the whole process
- Stochastic nature - randomness helps in not getting stuck in local minima and exploring the search space more effectively
- Diversity - GAs keep a diverse set of candidate solutions which helps in avoiding local extrema
- Adaptability - ACO uses pheromones for searching the problem space to promote the best solution found so far but can easily adapt in case a better solution is found. This also helps in avoiding local minima.

Question 1.4 - Answer 1.4

The genes are going to represent the respective products. The chromosome contains multiple genes, setting the order in which we are going to visit and collect those products. They are going to be represented as lists with lengths equal to the number of products we have to pick up. We are going to start in the *start coordinate*, go through the list from the chromosome and finish in the *end coordinate*. This encoding enables us to store the order and easily calculate the length of the path, based on the products order, which will be used in the fitness function.

Question 1.5 - Answer 1.5

Our fitness function is defined as 1*/length* where *length* is the distance of the route described by the genes in the chromosome. It is suitable since the shorter the path the higher output of our fitness function and vice versa. The ratio of this function value to the sum of fitness values across the population is going to reflect well the “size of the pie’s slice” for the respective chromosome, which will be useful during the selection of chromosomes for reproduction. Our aim is going to be to maximise the fitness score as it will correspond to the shortest path.

Question 1.6 - Answer 1.6

Parents are the chromosomes that have been selected through the roulette wheel, to avoid getting stuck in local minima. With this approach, every chromosome has at least a slight chance of being selected for reproduction, to keep the variety in our population. It works as follows: each chromosome has a probability to be selected based on its fitness score. The exact probability for a chromosome to be selected is the ratio between the fitness function score and the total sum of fitness scores in the whole population. Chromosomes with the highest fitness score, according to our function, have the highest chance of being selected for reproduction. This assures that we don't fall into local minima by just picking the shortest path that is in our population.

Question 1.7 - Answer 1.7

For our genetic operations, we use **OX crossover**, **swap mutation** and **roulette selection**. In the OX crossover, we select a random substring of genes from one parent and give it to the newly created child. We fill out the remaining genes by taking the second parent's genes that have not been a part of the substring and inserting them into the child from left to right. Swap mutation is done by simply selecting two random genes from the chromosome and swapping their places. (We have also implemented and tried the inversion mutation but it didn't improve the performance.) Lastly, roulette selection uses fitness ratio (equal to the fitness score divided by the sum of all fitness scores in the population) to assign each chromosome the probability of being chosen.

In order to avoid points being visited twice we instantiate the points with permutations of product numbers and use OX crossover as well as swap mutation. The swap mutation impacts only the order of the genes so it won't introduce duplicates. The OX crossover creates an offspring by taking a substring from one parent and then using such genes from the second parent that weren't in the substring thus avoiding duplicates. Thanks to these operations each gene can only appear once in our successor chromosome.

Question 1.8 - Answer 1.8

We prevent local minima by using crossover reproduction, mutations and roulette selection. Each of these operations introduces randomness in its own way that should be able to get out or avoid getting into local minima. We have decided to use 0.085 mutation probability and 0.7 probability for a crossover (otherwise one of the parents gives all of his genes), as after running the algorithm multiple times they resulted in the most promising results.

Question 1.9 - Answer 1.9

Elitism is a strategy where a few chromosomes with the best fitness, called the elites, are directly inserted into the next generation. It is done without undergoing crossover and mutation. It allows for faster convergence of the algorithm.

However, as elitism propagates already known individuals to the next generations, the possibility of finding new individuals better fitted to solve the problem reduces, which leads to a higher chance of getting stuck in local minima. To avoid this, we have decided not to use elitism in our algorithm. Our convergence was already happening fast, so we do not need additional speedup in it.

Question 2.10 - Answer 2.10

Ant Colony Optimization is an algorithm inspired by the behaviour of ants. The main purpose of ACO is to find optimal solutions to combinatorial optimization problems. It is used to solve problems where the goal is to find the best combination of elements from a set of possibilities.

ACO can be applied to different optimization problems, including for example the travelling salesman problem, vehicle routing problem, and job shop scheduling problem. It works very well in situations where there are many possible solutions and it is difficult to evaluate them all in a short time. It is often used in fields such as transportation, logistics or telecommunications.

Question 2.11 - Answer 2.11

Features that might increase the difficulty of the maze are for instance (apart from loops): **dead ends**, **open areas**, and **wide paths**. The first one is problematic as there might be a long path with a lot of forks that all lead to a dead end and therefore takes a lot of computation just to evaluate a wrong path. If we do not do anything about it the ant can still fall again into a dead end. The second one is a problem because an ant will not necessarily go straight to the exit from the open area but it might end up taking some quirky path. And the last one is a difficulty as it is a bit similar to the open area problem. If we have a path of width two an ant could at some point make a U-turn and move away from the correct solution or traverse the path in a snake-like movement.

Question 2.12 - Answer 2.12

Equation for the amount of pheromone dropped by the ant k on the path ij, where $\Delta\tau$ is the amount of added pheromone, Q is a constant used to scale the pheromones and L_{ij} is a length of the path ij:

$$\Delta\tau_{ij}^{k} = \frac{Q}{L_{ij}}$$

The ants need to drop the pheromone to indicate which paths they took so the rest of ants can use it while deciding on their route. The paths with more pheromone are more likely to be chosen as the amount of pheromone is inversely proportional to the length of the link so short paths are promoted. This means that the shorter the path to the finish the more pheromone it will receive.

Question 2.13 - Answer 2.13

After each generation the pheromones should be evaporated from the maze to allow the ants to explore other paths. By doing that we make sure that if ants go through a longer path at the beginning we let them pick new paths as well.

Here is the equation for the evaporation of pheromone dropped by the k-th ant on path ij.

ρ is an evaporation constant which can be tuned and optimized (at the beginning we have set it between 0.1 and 0.4. Values above 0.4 have resulted in ants not staying on the best path when they found it. And values below 0.1 did not allow ants to escape local minima. Therefore we have tuned our values to be between these values), Q is a constant used to scale the pheromones, L_i is a length of the i-th link, m is the number of ants which can be tuned and optimized but it is not strictly related to the context of evaporation and $\Delta\tau_{ij}^{k}$ is the amount of pheromone dropped by ant k on the i-th link:

$$\tau_{ij}^{k} = (1 - \rho) \cdot \tau_{ij} \frac{Q}{L_i} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

In each iteration the ρ * 100% of pheromone will evaporate during each iteration. The evaporation allows the ACO to correct wrong decisions taken in the past and not get stuck in local minima. If at first mostly long paths were chosen and got a lot of pheromone, evaporation will correct this over time by removing a proportion of it and allowing shorter paths to be prioritized.

Question 2.14 - Answer 2.14

Here's a short pseudo-code of the ant algorithm:

1. Initialize the maze with pheromone values at the beginning all fields have 1 and all walls have 0 pheromones.
2. Repeat for a number of generations
    a. Spawn a number of ants at the start of the maze
    b. For each ant, repeat until it reaches the end of the maze or a maximum number of steps:
        i. Choose the next cell to move to based on a probability function that takes into account the pheromone level of the neighbouring cells.
        ii. Add the direction in which the ant moved to the path.
    c. If the ant has reached the destination
        i. Update the pheromone values on each cell based on the path the ant has taken.
    d. Update the shortest path from the start to the end of the maze based on all the paths the ants have taken and store it as the best path.
3. Return the best path

The amount of pheromone left by the ants on each cell is also based on the distance travelled, which ensures that shorter paths receive more pheromone and are more likely to be chosen by future ants.

Question 2.15 - Answer 2.15

The Ant colony algorithm suffers from several limitations. We have identified the following problems: **dead ends**, **open spaces**, **wide paths** and **loops** (which were suggested in the question above).

1. The first one is a problem as without any optimizations the ant might fall into the same dead end many times. This causes the algorithm to converge much slower.
2. The second one is a problem as they might not know what to do in an open space and therefore can for instance take curvy paths.
3. The third one is obviously a problem as we usually want the ants to go straight when we have a path of width 2 for instance.
4. The last one is self-explanatory as we don’t want the ants to fall into a cycle and never reach the end.

To improve the ant algorithm, we have introduced three **meaningful** improvements.

Firstly, we have given each ant some memory. Each ant has a map in which we store whether we have visited a particular field. This helps to prevent the ants from visiting the same field multiple times. It helps to solve the dead-end problem as well as the loop problem as it simply does not allow an ant to go along a path it already went.

Secondly, when encountering a dead end, we use a form of backtracking to get out of it. Each ant stores the route it has already taken. Once we see that we cannot move further anymore we use the route we have walked to go back to the nearest field where we can change direction. We then take this other direction. This resolves the problem of a dead end and makes it impossible to get stuck at a dead end and waste time.

Lastly, we have introduced a straight factor to tackle the problem of open spaces and wide paths. This factor increases the probability of going in the same direction by adjusting the probability of the field that lies in the same direction as the previous step. We have found that usually, our ants would do these snake-like movements on a wide path and when we introduced this factor the problem was resolved. This also improved the open area problem as the ant would go straight to the opposite wall of the open space and look for the exit there. As an example, if an ant went north and we have a straight factor of 1.2, then the ant will have a 1.2 higher probability of going north again. This helps to guide the ants towards the destination while avoiding unnecessary detours. After a bit of tuning, we decided to use **1.2** as our straight factor.

There is one little improvement we also tried. It does not however relate to any particular problem per se. We wanted to change the Q value (the total amount of pheromones per path) in each generation. We start with a small value and in each generation, we increase it by a constant amount. This allows the ants to explore the path well at the beginning and as the generations go by the better paths will receive more and more pheromones. This did not improve our algorithm greatly so we decided to not incorporate it.

Overall, these improvements enhance the efficiency and effectiveness of the ant algorithm in finding the shortest path between two points.

Question 2.16 - Answer 2.16

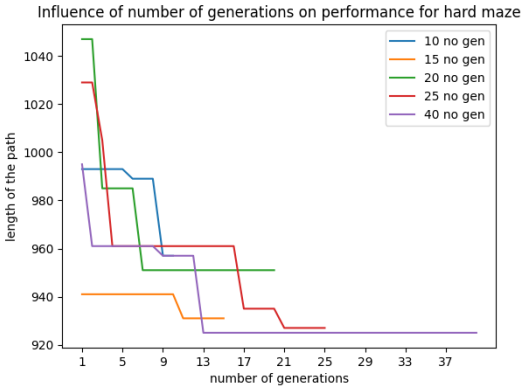
To tune the parameters we decided to plot the graph showing the length of the best-found path over generations. It allowed us to see which parameters led to the best performance while achieving it in a reasonable time. By “reasonable time” we mean roughly 1 minute for running one instance of the ACO algorithm on the hard maze (obviously for smaller mazes it gives us a result faster). That way each graph generated took us roughly 5 minutes as we were testing 5 different values per parameter.

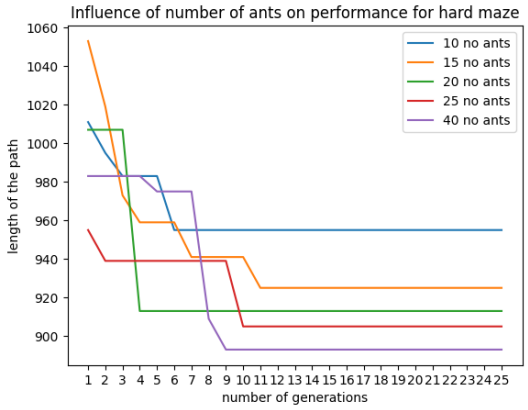
For each parameter, we decided to take the value of the parameter that gave the best solutions (if however, some other values of parameters gave a result within a 5% difference we assumed to take the fastest one, that is the one that converged faster).
We decided to only show the graphs for the hard maze since the limit of 1 A4 doesn’t allow us to show the others, but we performed the same search for each maze type.
For the easy maze, we chose the following parameters:
ants: 10:, generations: 10 , Q: 200, evaporation: 0.1, straight factor: 1.2, convergence steps: 4

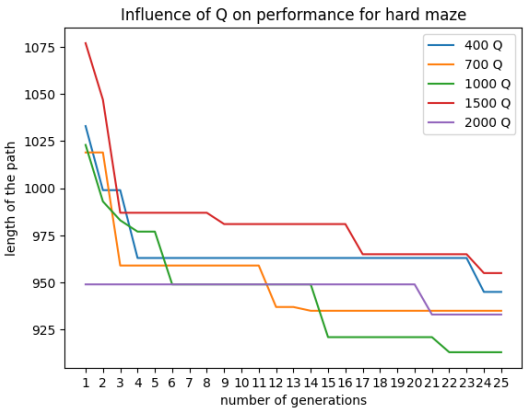
path length: 38, time: 0.30s

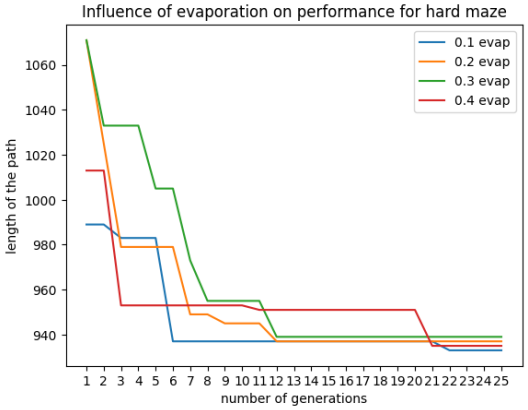
For the medium maze we chose the following parameters:
ants: 25, generations: 25, Q: 400, evaporation: 0.1, straight factor: 2, convergence steps: 6
path length: 151, time: 14.12s

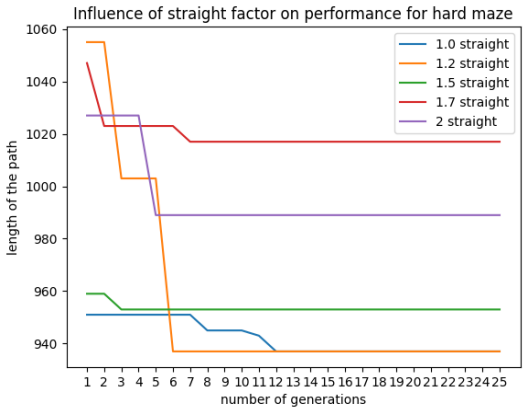
For the hard maze we chose following parameters:
ants: 40, generations: 40, Q: 1000, evaporation: 0.1, straight factor: 1.2, convergence steps: 6
path length: 923, time: 51.24s

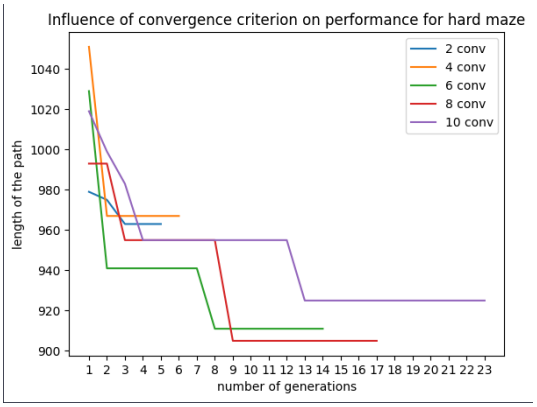












We could observe that the evaporation parameter (in range from 0.1 to 0.4) didn't have a large impact on neither the performance, since all the results converged to similar path length, nor on timing since the differences in the convergence time were not that significant. It was the case for all three mazes.

In the easy maze the straight factor didn't have any influence on the performance nor time of the algorithm, probably, because there are no open spaces in this maze. Also any number of generations higher or equal to 10 did not have an influence on the performance.

Question 2.17 - Answer 2.17
Thanks to the tuning we ran we have drawn the following conclusions about these 6 parameters:

- Evaporation coefficient - we have observed that for bigger and more complex mazes we do not need a higher coefficient. At first we thought that by increasing the evaporation, we allow the algorithm to have a higher chance of escaping from local minima. But this did not changed much. We decided to stick with the 0.1 on all mazes as the algorithm was achieving good results.
- Convergence criterion (for how many generations should the best result not change in order to stop the algorithm) - here we did not have very concrete results but we have observed a tendency that the higher the number the more time the algorithm has to find a better result as indicated by the graph above
- Q normalization factor - here we observed that for larger mazes we needed higher Q value, however, it didn't have that much impact. This tendency might be caused by the fact that in the bigger mazes, the pheromones are more spread out at the beginning so a higher Q compensates for that.
- Number of ants per generation - We believe that for harder mazes we need a higher number of ants. This is quite intuitive as bigger mazes have more routes and therefore need more ants to explore. Although the bigger number of ants for a simple maze will still yield a good result we can still have fewer ants in a shorter time.
- Number of generations - again with the bigger mazes the number of generations needed to converge to the best result. It may come from the fact that ants need more time (number of generations) to explore larger areas.
- Straightness factor - we have discovered that the positive straightness factor is very beneficent for the medium and the hard maze as it allows for a faster convergence. This is most likely caused by the fact that these mazes have a lot of open spaces. We did not see any improvements in the easy maze.

Question 2.18 - Answer 2.18

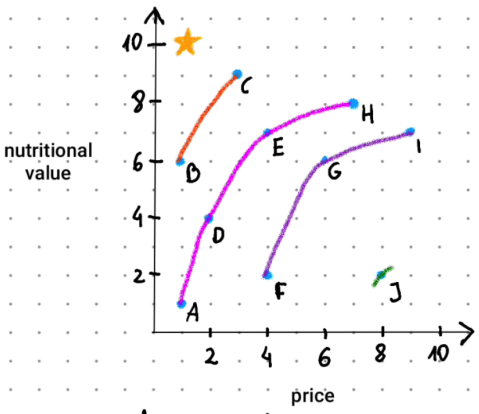
In the synthesis part the path had length of 1415 while in part 1, we had 1343.
The difference in the paths is quite small, but in favour of the algorithm from the 1st part.

Synthesis order: [1, 2, 7, 5, 14, 16, 4, 9, 8, 18, 10, 15, 12, 13, 6, 11, 3, 17]
GA order form part 1: [1, 2, 7, 5, 14, 16, 4, 9, 18, 8, 10, 15, 12, 13, 6, 11, 3, 17]
The only difference in the order of the products is that the 8th and 18th products are swapped.

The most significant difference was the time it took for the synthesis to run which was much higher that the time to execute the GA algorithm in part 1. That was because the Ant Colony Optimization had to be run first while in part 1 we already knew the distances.

The solution from part 1 is slightly better. We think that the distances generated by our ACO might be a bit worse than those from the *optimal_tsp* file which might have caused worse performance of the GA in this example. The reason for that is probably the small number of ants (5) used in our ACO because otherwise the program took too long to run on our machines.

Question 2.19 - Answer 2.19



The robot should maximize the nutritional value and minimize the price as indicated by the star. Below you can see the first iteration of NSGA-II.

	Points dominated	No. of points that dominate	Pareto rank
A	—	1	2
B	A, D, G, F, J	0	1
C	E, H, F, G, I, J	0	1
D	F, J	1	2
E	F, G, I, J	1	2
F	J	4	3
G	J	3	3
H	I, J	1	2
I	—	3	3
J	—	4	4

The four elements a robot would select are B and C from the first front. And 2 elements from the second front. We have decided to use the crowding distance to select the two from the second front because it is a good way to preserve the diversity of the points. Since A and H have infinite crowding distances, because they have a missing neighbour, we will select them. The best points, in that case, are B, C, A, and H.

Question 2.20 - Answer 2.20

$C(w_1(0), h_1(0)) = 20 \cdot 85 + 70 \cdot 1,9 - 85 \cdot 1,9 + 500$
$= 1700 + 133 - 161,5 + 500$
$= 2171,5$

$C(w_2(0), h_2(0)) = 20 \cdot 60 + 70 \cdot 1,6 - 60 \cdot 1,6 + 500$
$= 1200 + 112 - 96 + 500$
$= 1716$

$C(w_3(0), h_3(0)) = 20 \cdot 75 + 70 \cdot 1,7 - 75 \cdot 1,7 + 500$
$= 1500 + 119 - 127,5 + 500$
$= 1991,5$

$v_1(1) = \begin{bmatrix} 0 + 0 + 0,5 \cdot (85 - 85) \\ 0 + 0 + 0,5 \cdot (1,9 - 1,9) \end{bmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

$v_2(1) = \begin{bmatrix} 0 + 0 + 0,5 (85 - 60) \\ 0 + 0 + 0,5 (1,9 - 1,6) \end{bmatrix} = \begin{pmatrix} 12,5 \\ 0,15 \end{pmatrix}$

$v_3(1) = \begin{bmatrix} 0 + 0 + 0,5 (85 - 75) \\ 0 + 0 + 0,5 (1,9 - 1,7) \end{bmatrix} = \begin{pmatrix} 5 \\ 0,1 \end{pmatrix}$

2.20 continues here

$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = v_1 \qquad \begin{pmatrix} 12,5 \\ 0,15 \end{pmatrix} = v_2 \qquad \begin{pmatrix} 5 \\ 0,1 \end{pmatrix} = v_3$

$w_1(1) = 85$ $\quad h_1(1) = 1,9$

$w_2(1) = 72,5$ $\quad h_2(1) = 1,75$

$w_3(1) = 80$ $\quad h_3(1) = 1,8$

$c(w_1(1), h_1(1)) \quad 2171,5$

$C(w_2(1), h_2(1)) = 1945,625$

$C(w_3(1), h_3(1)) = 2082$

For all 3 particles, caloric intake is higher or equal after first iteration.

$y_1(1) = \begin{pmatrix} 85 \\ 1,9 \end{pmatrix} \qquad y_2(1) = \begin{pmatrix} 72,5 \\ 1,75 \end{pmatrix} \qquad y_3(1) = \begin{pmatrix} 80 \\ 1,8 \end{pmatrix}$