Assignment 2 - Computational Intelligence
Group 26

David Sachelarie, Sorana Stan, Iarina Tudor, Jason Miao

March 2022

2.1. Ant Preparation

The main code for this part is in AntColonyOptimization.py which calls the Ant.py class to compute the path of an ant. This part uses the data of 3 mazes : easy coordinates.py and easy maze.py (the same applies for medium and hard mazes) and computes the solution in easy_solution.py (same for hard and medium) which is provided in Group_26_code_ACO.zip. We also added the solution in different document they are called Group_26_easy.txt (the same for medium and hard)

1. Difficult features

Features that increase difficulty are loops, open areas, dead ends and maybe no exit at all.

2. Pheromones

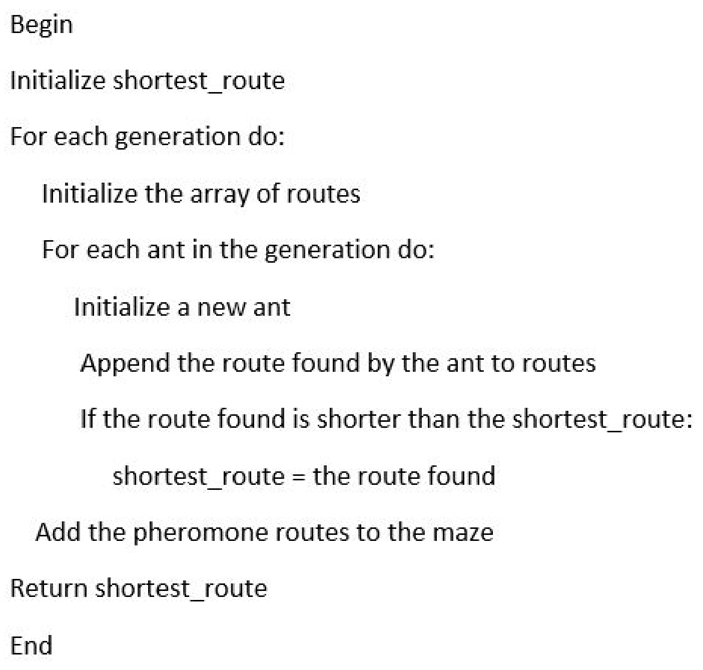
In every generation, after finding a path from the starting point to the end, an ant will drop the same amount of pheromone on the whole path found. This will be calculated depending on the length of the path found. So, every ant will drop Q/length of the path amount of pheromone, where Q is a constant used to optimize the amount of pheromones released by the ants. By dropping pheromones at the end of each generation, the next generation of ants will take the pheromones into consideration when choosing the paths to walk on. Since the shorter paths will have more pheromones than the longer ones, this will mean that further generations of ants will continue to find shorter paths, until at some point, the result will converge to the shortest path possible. The purpose of the algorithm is to find the shortest path from one starting point to another specified end point. In order to do this, we try to simulate the behaviour of real ants by releasing pheromones and upgrading the probabilities of taking certain paths as opposed to others.

3. Evaporation

After every generation of ants, a certain amount of pheromone will evaporate. We do this in order to simulate the real behaviour of ants and to normalize the amount of pheromones left on all the trails, which will make it easier to calculate the probabilities of taking a specific route in the later generations. If we define the amount of pheromones currently on a path as: pheromones, after every generation of ants the amount of pheromones on each path after the evaporation, will be equal to (1-rho)*pheromones, where rho is a constant that can be changed in order to optimize the amount of pheromones evaporated.

2.2. Implementation

4. Pseudocode



```
Begin
Initialize shortest_route
For each generation do:
    Initialize the array of routes
    For each ant in the generation do:
        Initialize a new ant
        Append the route found by the ant to routes
        If the route found is shorter than the shortest_route:
            shortest_route = the route found
    Add the pheromone routes to the maze
Return shortest_route
End
```

Figure 1: pseudocode of finding shortest path

There is also the Ant find route algorithm which basically start from a point and tries to move until it reaches the end. We tried to compute the possible moves and their probabilities(by summing all possible pheromone values from valid positions and dividing the value over the sum), in case there is no move, we hit a dead end and we go one step back. A valid move is to a position that has not been visited yet, is still in the maze and it is not a wall. We created a separate method for that in Ant.py called is_valid. We only stop our algorithm when we hit the end of the maze.

2.3. Intelligent Ants

5. Improved Algorithm

Our algorithm is able to tackle multiple "hard problems". Firstly, we want to explain a bit how the pheromone level is actually updated. We observed that 0's are walls and 1's are travelable paths. This can be translated into the level of pheromones. We decided that if we keep all walls encoded to 0 in the maze and all other positions with a level of pheromone bigger than 0, we can combine the pheromone and walls into a single map.

Dead Ends: When we are not able to move in any direction (this is checked in a method by looking if an element is in the map, it is not a wall and it has not been visited yet), we are going one step back. This will happen until we get back to the intersection where we took the wrong path. We decided to not keep those steps (from intersection to dead end and back) in the final route. That will mean that even if an ant discovered a corridor who was not good in the end, its route will ignore that and the next generations of ants will have a lower chance to discover it. On those corridors, at the end of the generation when we update the pheromone level, we will not add any more pheromone.

Loops: We consider loops somehow like a dead end because it will reach a point we have been to before and no other direction will be available. Our algorithm will just go back one step at a time until we find the "start of the loop" or "intersection" and we can continue from there. The loop will thus be ignored in depositing pheromones the same way a dead end is. We explained this process in the paragraph above.

2.4. Parameter Optimisation

6. Parameters and Convergence

We saw that the more ants and generations we have, the slower the convergence will be. Furthermore, the evaporation coefficient decides if a path will have a bigger probability or not so the ants will either follow it or try to discover something new.

7. Parameters and Maze Complexity

A more complex maze will require more ants and generations to discover it since there are more possible paths to try. This will mean that the convergence will be slower on the correct solution and we will need more time to find the optimal path.

3.1. Travelling Salesman Problem

The code for this part of the assignment is in GeneticAlgorithm.py, which uses data provided by TSPData.py, a path length generator which in turn uses calls to AntColony-Optimization.py, which is provided in Group_26_code_ACO.zip. All of the TSP code is provided in Group_26_code_TSP.zip. In order for TSPData and GeneticAlgorithm to work, they have to be included in a src folder. Then, TSPData will write the distances in tmp/productMatrixDist, data which will be used by the genetic algorithm, which will in turn write to data/TSP solution.txt.

8. Definition

The Traveling Salesman problem is an algorithmic problem which finds the most efficient trip between a set of points and locations (in this case locations in the supermarket), that visits each place exactly once and returns to the starting point.

9. Differences from Our Problem

The only difference between our problem and the Traveling Salesman problem is the existence of the start and end points, which are connected to the products, but don't count as products themselves. To a combination of products found as solution we must also add the distances between the start and end points to the closest products. Consequently, to the weighted complete graph a "source" and a "sink" should be added, just like in a network flow problem, with only outgoing and ongoing edges, respectively.

10. Computational Intelligence Techniques

These techniques are often related to optimization problems. It forms a premise of utilizing a large mass that can be considered as data or information to lead towards a conclusion. Techniques such as genetic algorithms and ant colony optimization are appropriate methods for solving TSP where we mimic hordes of wildlife (ants, fishes etc) and figure out their methodology of finding the shortest path through a designated number of places.

3.2 A Genetic Algorithm

11. Genes and Chromosomes

Each gene represents one product, and each chromosome is a sequence of numbers denoting the order in which products are being taken.

12. Fitness Function

The fitness function is the sum of all distances between products in the sequence that come one after the other, raised to the power of -1 (for choosing the shortest, not longest route), together with the distance between the starting point and the first product and the last product and the ending point, also raised to the power of -1.

13. Parent Selection

Parents are selected at random, but fitter individuals have a higher chance of being selected according to their fitness ratios, which are easily calculated as fitness divided by the sum of all fitnesses in the population.

14. Cross-over and Mutation

Cross-over is done in the following way: suppose we have two parent chromosomes, ch1 and ch2. We take a subsection of the first one, from a random position to the end. We copy those values into the offspring chromosome, making sure that we preserve order and position. The remaining positions now have to be filled with elements from ch2. We thus iterate through ch2, copying elements in order into the offspring chromosome, while making sure that we do not add duplicates. If that element was already added by the subsection taken from ch1, we just skip it.

There is a small probability that a mutation occurs (we currently have it as 0.01), in which case two positions of the chromosome will be swapped. We cannot just change the value of one element, since we have to make sure that no products are visited twice and all products are being visited.

15. Points Visited Twice

As explained above, cross-over and mutation are not done in the traditional way. In a regular cross-over, chunks of elements would be swapped between chromosomes, but in our algorithm we have to make sure that all values in the offspring are different. We thus copy a slice of one chromosome and selected elements from the second chromosome, so that we avoid duplicates. During mutation, we cannot just change the value of one element. Instead, we swap it with another random element inside the chromosome.

16. Local Minima

We prevent the occurrence of local minima using mutation. This way, the algorithm can escape an early convergence by random changes in the genome.

17. Elitism

Elitism means that some of the fittest individuals are passed to the next generation unaltered, together with the offspring that are the result of cross-overs and mutations. We did not apply this technique, since we considered that it will artificially limit the gene pool, restricting the opportunity of other individuals to be created and have their phenotypes refined through generations.

3.3 Reflection

18. Fitness Function

The fitness function is probably the core of the genetic algorithm. Without a way to correctly discern between how well are different individuals handling a task, the algorithm becomes no better than pure chance. Even a generally correct, but imperfect fitness function can cause problems, which become bigger and bigger as generations pass, since the fitter individuals don't necessarily have the correct chance of being chosen as parents.

19. Survival Functions

The offspring of two strong individuals is not always stronger than the one of two weak individuals. Because of the way genes combine, the offspring may only get the weaknesses of its parents or some combination of genes that produced a strong phenotype separately, but are otherwise not very compatible with one another. The opposite may also be true: two weak individuals can have genes that combine very well with one another, producing strong offspring. Those problems are being mitigated automatically by the algorithm. Weak individuals are not disregarded, but they rather have a small chance of reproducing. Their offspring will eventually die out through generations if it is outrun by its competitors. This is also the case for the weak offspring of two strong parents. If it proves to be unfit, it will have limited chances to reproduce. In the end, not strong individuals, but rather strong traits are being propagated and survive through generations.