GROUP 44 REPORT

1. The TSP problem is usually defined as finding the shortest possible route that a traveling salesman can take to visit a set of cities exactly once and then return to the starting city (in our example just visit the end node). The problem is usually formulated as a graph with cities as nodes and edges representing the distances between cities.

2. In our problem, the nodes represent the locations of products in a supermarket and the edges represent the distance between two products. Our problem is different from the classic TSP in two ways: first, the graph is not complete, i.e., not all products are connected to each other directly, but instead connected through the supermarket aisles; second, the graph is asymmetric, i.e., the distance between two products may not be the same in both directions due to the layout of the supermarket. Therefore, the problem of finding the shortest path or the optimal route between two products is different than the classic Traveling Salesman Problem (TSP), where the graph is complete and symmetric.

3. Computational intelligence techniques are appropriate to solve the TSP because they can efficiently search through the vast solution space of possible routes and find a near-optimal solution. These techniques are particularly suited for tackling typically intractable problems because they can handle large and complex search spaces, and can learn from past experiences to improve their performance. Characteristics that make the TSP intractable include the exponential growth in the number of possible solutions as the number of products increases, the combinatorial nature of the problem, and the presence of multiple local optima. Computational intelligence techniques can mitigate these challenges and find good solutions within a reasonable amount of time.

4. The genes in our genetic algorithm represent the order in which products have been visited. For example, if we have five products labeled as 1, 2, 3, 4, and 5, one possible gene sequence could be [2, 5, 4, 1, 3]. This means that the robot will visit product 2 first, followed by product 5, then 4, 1, and finally 3.

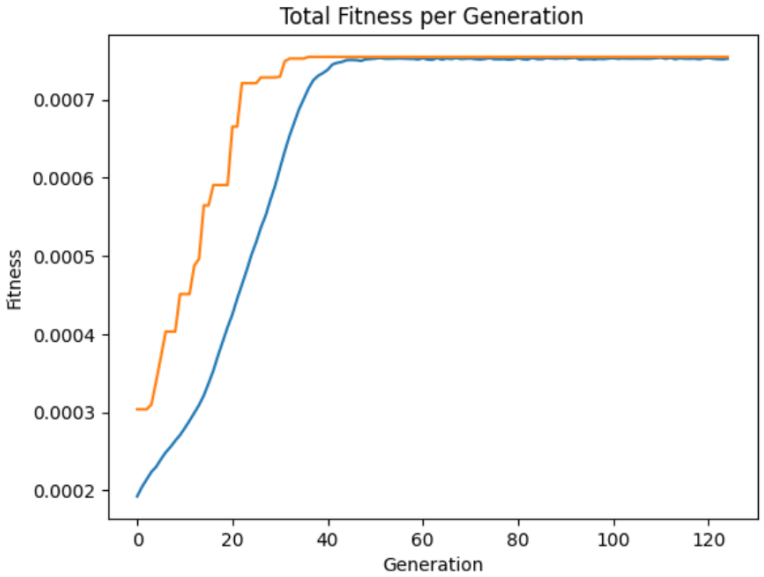
5. The fitness function used in this case is the inverse of the total distance traveled by the robot. The reason for taking the inverse is to favor shorter distances, as larger fitness values indicate better solutions. The fitness function can be represented as:

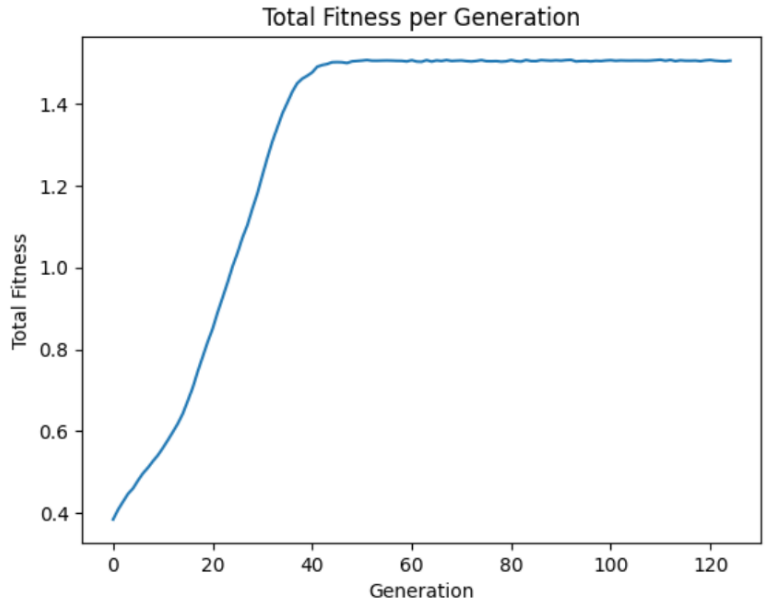
$$fitness = 1 / total\_distance$$

   where $total\_distance$ is the sum of the distances traveled by the robot between all the products.
   This is a suitable choice because it encourages the algorithm to converge towards shorter and more efficient routes. By selecting fitter individuals in the population, the algorithm can improve the overall quality of solutions.

6. We tried three crossover techniques - Partially Matched Crossover (PMX), Order Crossover (OX), and Caterpillar Crossover - in our genetic algorithm. PMX was found to be the best and is currently used, while the others are commented out in the code. We also used two selection methods - roulette wheel selection and tournament selection. Tournament selection with 2 candidates and choosing the better one was found to be more effective than roulette wheel selection for our problem.

After testing the different combinations of crossover and selection techniques, we found that the combination of partially matched crossover with tournament selection was the most effective in terms of both speed of convergence to the optimal solution and general time performance. However, all techniques are still available in the code and can be tested for other problems. Best (orange) and average (blue) vs generation and total fitness (blue) vs generations graphs are available below:





7. In the implemented $partially\_matched\_crossover$ function, a repair function is used to prevent points from being visited twice in the offspring chromosomes. The repair function ensures that each city is visited exactly once, which is a crucial constraint of the traveling salesman problem.
   The repair function works by scanning the offspring chromosome and checking if any city is repeated. If a city is repeated, the function replaces it with a valid city that has not yet been visited. This process continues until the entire chromosome has been scanned and all duplicate cities have been replaced with valid cities.
   There are different strategies to select a valid replacement city. One of them is to choose the city closest to the current city in the chromosome. This strategy can be effective when the cities are clustered closely

together. Another strategy is to randomly select a city that has not been visited yet. This strategy can be effective when the cities are distributed more evenly throughout the solution space.

By implementing a repair function that prevents points from being visited twice, the genetic algorithm generates valid solutions to the traveling salesman problem. This is a critical aspect of the algorithm since violating the constraint of visiting each city exactly once can lead to invalid and suboptimal solutions.

8. To prevent the algorithm from getting stuck in local minima, we use a diverse population and introduce randomness through mutation. By using a diverse initial population and allowing for random mutations, we encourage the algorithm to explore new regions of the solution space and avoid getting stuck in local optima.
9. Elitism is a technique that involves preserving the best individuals from one generation to the next. In our implementation, we have applied elitism by selecting the fittest individual in the population and preserving it for the next generation without modification. This ensures that the best solution found so far is not lost and can potentially improve in future generations.
10. Ant Colony Optimization (ACO) is an algorithm inspired by the behavior of ants. The purpose of ACO is to find optimal solutions to optimization problems, such as the traveling salesman problem (example from the slides), by mimicking the way ants search for the shortest path between their colony and a food source.

    ACO can be used in various optimization problems that involve finding the shortest or most efficient path between multiple points, such as routing problems, scheduling problems, and network design problems. It has been successfully applied in transportation and logistics planning, telecommunications, and manufacturing, among other fields.
11. Features like obstacles on the path, unreachable states (no links incoming and outcoming to the node), and dead ends can make finding the finish line in a maze more difficult for ants. These challenges require creative solutions and advanced algorithms to optimize the path to the finish line.
12. Based on slide information:

    The amount of pheromone currently on path ij corresponds to the amount of pheromone existing on it, decreased by a quantity proportional to the evaporation constant, plus the sum of all the pheromone released on that link by all the ants that passed by it this round.

    $$\sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

    => This summation represents the sum of all pheromones released on that link by all ants that when through it.

    $$\Delta\tau^{k} = Q \times \frac{1}{L^{k}}$$

    where Q is constant and L^k is the cost of the path taken by K^{th} and this equation represent the amount of pheromone which Kth ant deposited on the link it explored.

    Ants drop pheromones in the maze to communicate with other ants in their colony. When an ant finds a food source, it will lay down a trail of pheromones on the path back to the colony. Other ants in the colony can then follow this pheromone trail to the food source, reinforcing the trail by depositing more pheromones as they go

By dropping and following pheromones, ants are able to coordinate their behavior and find the most efficient path to food sources, even in complex and changing environments like a maze.

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

so the total amount of pheromone deposited by the ants on the specific link can be described using this equation.

13. The equation for the evaporation of pheromone in ant colony optimization algorithms can be represented as follows:

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

Where:

- p is a constant evaporation rate (evaporation rate is the rate at which pheromone evaporates over time)
- τ_{ij} is the current amount of pheromone on the link (i, j)
- the sum of all pheromones released on the link (i, j)

The amount of pheromone that evaporates in each iteration is determined by the evaporation rate (rho) and the amount of pheromone deposited by the ants at each edge.

A high evaporation rate means that the pheromone deposited earlier will decrease faster.

The purpose of pheromone evaporation is to prevent the ants from getting stuck in a suboptimal solution by allowing the exploration of alternative solutions. Without evaporation, the ants would follow the same path repeatedly, resulting in the reinforcement of a suboptimal solution

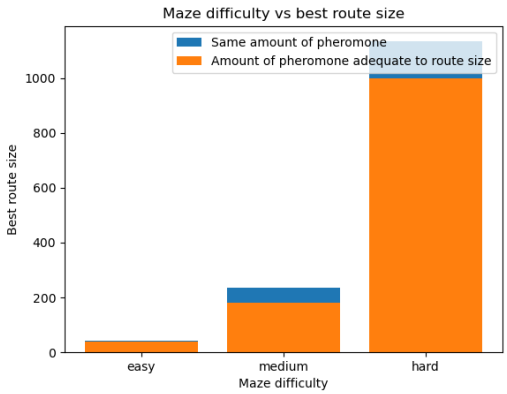
14. Initialize:
    - Number of $ants\ per\ generation, size$
    - Number of $generations, num\_of\_generations$
    - Pheromone constant, q
    - $Evaporation\ rate, evaporation\_rate$
    - Set $pheromone\ level$ on each edge
    - Set $convergence\ criteria$

```
For each generation in num_of_generations:
    For each ant in size:
        Chose starting node randomly
        While current position isn't the end of the maze:
            Calculate possible directions
            Randomly choose a direction from allowed directions and move ant
            Update the route
        End while
    End for
    Update the pheromone level on each edge
End for
```

If a certain number of ants fail to improve the route (convergence criterion), we return from the function.

15. The ant colony optimization algorithm has been improved in three significant ways. Firstly, we introduced route-dependent addition of pheromone. This ensures that the amount of pheromone is adequate to the length of the route. More precisely, the longer the route, the smaller the amount of

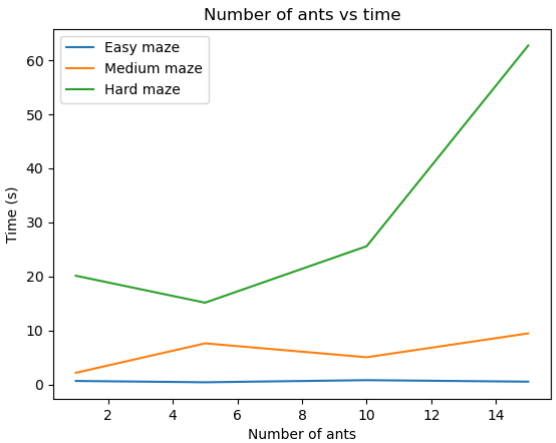
pheromone left by a single ant. As you can see below on the graph the effect of this change gradually increased with the complexity of the maze. This occurrence is due to the fact, that ants are more influenced by pheromone left by their priors on the higher level of the maze.

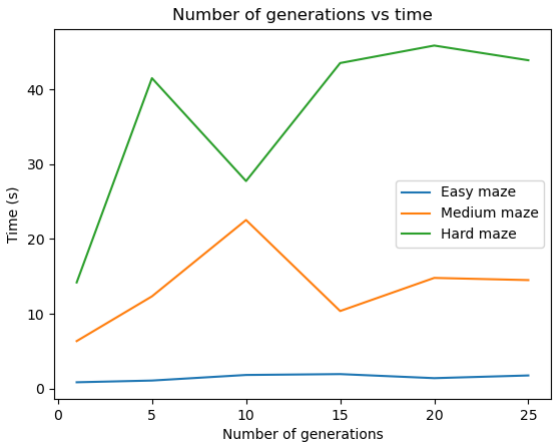


Moreover, we introduces the possibility of backtracking, allowing them to back off when they encounter a dead end. This increased their chances of finding a more optimal route, without having to start their "journey" from the beginning of the maze. Otherwise, we were just hoping that any ant will find the proper way to the end of the maze, without a single encounter with a dead end. While in the easy level this was possible with a significant amount of ants, in the hard level it was impossible to achieve and definitely too time consuming. It goes without saying that, if ant encounter the dead end it does not leave the pheromone.

Furthermore, we introduced a notion of memory for our ants. Namely, they posses the knowledge of previously visited nodes. This improvement allows ants to recall already explored nodes, making them reluctant to visit them again. This enables them to explore new paths and eventually finding more optimal solution. It also significantly speeds up the algorithm.

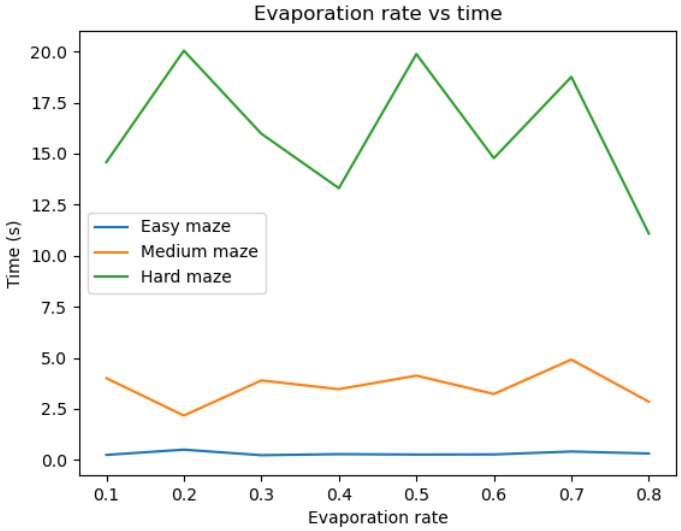
16. Hyperparameters of ant colony have significant impact on the speed of the algorithm. While the changes are barely visible on the easiest level of difficulty of the maze, the hard level is greatly affected by even the slightest change in them. As you can see on the graphs below number of ants per generation and the number of generations itself, can change the speed of the algorithm dramatically. It is due to the fact that the loop that is inside the $find\_route$ method in our implementation perform a number of iterations equal to the overall number of ants.



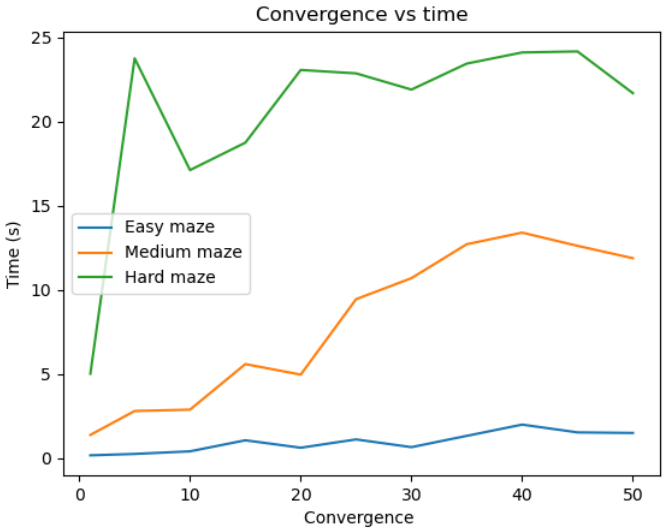


Secondly, we would like to point the effects of pheromone constant on the speed. Pheromone constant specifies how much are the ant influenced the pheromone left by their priors. In general, increasing this value will lead to faster convergence of the algorithm, as ants will be more willing to follow already existing pheromone trails. However, too high level may cause getting stuck in local optima of the algorithm.

Furthermore, effects of evaporation rate. This parameter focuses on the time of vanishing of the pheromone left on the trails. While increasing its value will help ants escape from local optima, too high level can also prevent them from creating strong trails leading to the best solution.



Lastly, for the convergence ratio for our algorithm, we had to choose a fairly small value in comparison with the size of the generation and the number of generations itself. Of course, to see the impact of this parameter we had to choose a value which was smaller than the overall sum of the ants. For example for size equal to 3 and 4 generations for the hard maze, value between 1 and 5 caused the fastest convergence.



Unfortunately, while choosing the best parameters, we couldn't focus only on the speed on the algorithm, we also had to take accuracy of the solution into account. We had to compromise between speed and accuracy. Our strategy involved trying out the values in the neighbourhood of the fastest one indicated by the above graphs. As a result, we ended up with the following parameters for the mazes:

Easy → evaporation rate: 0.4, pheromone constant: 100, size of the generation: 20, number of the generation: 20, convergence: 150 with the best result as follows: time: 1.98s and route size: 38

Medium→ evaporation rate: 0.4, pheromone constant: 800, size of the generation: 7, number of the generation: 7, convergence: 15 with the best results as follows: time: 10.899s and route size: 199

Hard→ evaporation rate: 0.2, pheromone constant: 1300, size of the generation: 3, number of the generation: 4, convergence: 4 with the best result as follows: time: 7.379s and route size: 1051

17. Maze complexity/size has a strong relationship with evaporation rate and pheromone constant. Former, must be smaller for bigger mazes, as there are more paths for ants to trail. And if that values is too high, pheromone vanishes quicker and ants are not able to search more optimal parameters. Latter, also needs to have a higher value for more difficult mazes, as the multiplicity of paths points to the necessity of one which visibly stands out. This scenario increases the chances of finding more optimal solution.
18. The Genetic Algorithm we used to solve the TSP produced the following array as the best ordering: [0, 1, 6, 4, 13, 15, 3, 8, 7, 17, 9, 14, 11, 12, 5, 10, 2, 16] with the length of 1343. Although we had some runs where the result was better, likely due to better luck with mutations, we found that using the distances calculated by our Ant Colony algorithm resulted in a different solution: [0, 1, 6, 4, 13, 15, 3, 8, 7, 17, 9, 11, 14, 12, 5, 10, 2, 16], with a length of 1585 (to add here, the product enumeration starts from 0 which represent product number 1). This suggests that our distance calculations may not be optimal. While this may only be the case for a small number of pairs of products, it still resulted in a significant difference in the obtained optimum (global or local). The only changes in the ordering is the (11, 14) pair that was swapped in case of ant colony optimization, indicating that the distances calculated for pair of products with this subset of products were not optimal. (Some distances may be larger as well but do not effect the order change, just the overall number of steps).

Of course, we could improve the accuracy of our calculations by changing the parameters of the Ant Colony algorithm, such as increasing the ant set size or number of generations. However, this would significantly increase the computation time, which is currently around 3 hours on average, taking into account the average of 11 seconds per iteration. Changing the parameters from $gen = 3, no\_gen = 4$ $to\ gen = 20, no\_gen = 20$ would increase the iteration time to around 200 seconds, which would drastically increase the computation time.

In summary, we believe that the trade-off between time and accuracy is reasonable, and the result obtained with our current method is close to the one obtained using optimal distances.

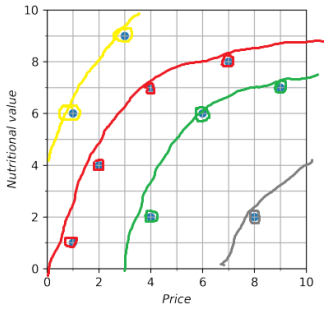
19. The solution:

Firstly, we want to minimize the cost, as in our understanding the lower the cost the better. Obviously, you also want to maximize the nutritional value.

Food	Price (minimize)	Nutritional Value (maximize)	Points Dominated	No. of Points that Dominate	After 1	After 2	Rank
A	1	1	-	1-1	0	-	2
B	4	2	C	4-1-1	2-1-1	0	3
C	8	2	-	7-1-1	5-1-1	3	4
D	2	4	B, C	1-1	0	-	2
E	1	6	F, D, A, B, C	0	-	-	1
F	6	6	C	3-1-1	1-1	0	3
G	4	7	H, F, B, C	1-1	0	-	2
H	9	7	-	3-1	2-1-1	0	3
I	7	8	H, C	1-1	0	-	2
J	3	9	I, G, H F, B, C	0	-	-	1

Pareto Frontiers:

- Pareto rank 1 (front 1): E, J
- Pareto rank 2 (front 2): A, D, G, I
- Pareto rank 3 (front 3): B, F, H
- Pareto rank 4 (front 4): C



When it comes to the actual 4 products to be chosen, we firstly go by fronts (1, 2, 3…). We take two products from front 1, mainly E, J. Now we are left with front 2 containing A, D, G, I. It's obvious that the two outliers are the points to be taken, but to prove our claim we calculate the crowding distance for each of those using two tables:

$$Min - max - diff\ table$$

	Minimum	Maximum	Difference
Price	1	9	8
Nutritional Value	1	9	8

$$CB_D = \frac{4-1}{8} + \frac{7-1}{8} = 1.125$$

$$CB_G = \frac{7-2}{8} + \frac{8-4}{8} = 1.125$$

$Front\ 2\ crowding\ distance$

	Price	Nutritional Value	Crowding distance
A	1	1	Inf
D	2	4	1.125
G	4	7	1.125
I	7	8	Inf

20. Let's firstly calculate the fitness:

$$\boldsymbol{Fitness\ calculation}\ (\boldsymbol{subscript\ for\ particle\ number}):$$

$$C_1\big(w_1(0), h_1(0)\big) = 20 \cdot 85 + 70 \cdot 1.9 - 1.9 \cdot 85 + 500 = 2171.5$$
$$C_2\big(w_2(0), h_2(0)\big) = 20 \cdot 60 + 70 \cdot 1.6 - 1.6 \cdot 70 + 500 = 1716$$
$$C_3\big(w_3(0), h_3(0)\big) = 20 \cdot 75 + 70 \cdot 1.7 - 1.7 \cdot 75 + 500 = 1991.5$$

Clearly particle 1 has the best global optimum with the value of 2171.5.
Let's update the velocity. We assume initial time is 0.

$$\boldsymbol{Particle\ 1}$$
$$v_w(1) = 0 + 0.5 \cdot 1 \cdot (85 - 85) + 0.5 \cdot 1 \cdot (85 - 85) = 0$$
$$v_h(1) = 0 + 0.5 \cdot 1 \cdot (1.9 - 1.9) + 0.5 \cdot 1 \cdot (1.9 - 1.9) = 0$$
$$\boldsymbol{Particle\ 2}$$
$$v_w(1) = 0 + 0.5 \cdot 1 \cdot (60 - 60) + 0.5 \cdot 1 \cdot (85 - 60) = 12.5$$
$$v_h(1) = 0 + 0.5 \cdot 1 \cdot (1.6 - 1.6) + 0.5 \cdot 1 \cdot (1.9 - 1.6) = 0.15$$
$$\boldsymbol{Particle\ 3}$$
$$v_w(1) = 0 + 0.5 \cdot 1 \cdot (75 - 75) + 0.5 \cdot 1 \cdot (85 - 75) = 5$$
$$v_h(1) = 0 + 0.5 \cdot 1 \cdot (1.7 - 1.7) + 0.5 \cdot 1 \cdot (1.9 - 1.7) = 0.1$$

Therefore after one iteration of PSO algorithm, the positions of particles are (subscript represents the particle number, first variable is w, second is h):

$$position_1 = (85, 1.9)$$
$$position_2 = (60 + 12.5, 1.9 + 0.15) = (72.5, 2.05)$$
$$position_3 = (75 + 5, 1.7 + 0.1) = (80, 1.8)$$

And final velocities of each particle (subscript represents the particle number, first value in the vector represents w, second h) are:

$$v_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, v_2 = \begin{pmatrix} 12.50 \\ 0.15 \end{pmatrix}, v_3 = \begin{pmatrix} 5 \\ 0.1 \end{pmatrix}$$

To find local bests, we should calculate new fitness values and compare those to the previous ones.

$$\boldsymbol{Fitness\ calculation}\ (\boldsymbol{subscript\ for\ particle\ number}):$$

$$C_{1\_new}\big(w_1(1), h_1(1)\big) = 20 \cdot 85 + 70 \cdot 1.9 - 1.9 \cdot 85 + 500 = 2171.5$$
$$C_{2\_new}\big(w_2(1), h_2(1)\big) = 20 \cdot 72.5 + 70 \cdot 2.05 - 2.05 \cdot 72.5 + 500 = 1944.88$$
$$C_{3\_new}\big(w_3(1), h_3(1)\big) = 20 \cdot 80 + 70 \cdot 1.8 - 1.8 \cdot 80 + 500 = 2082$$

Looking at new fitness values, first value didn't change so it's the local optima, second and third ones improved compared to the initial state, so they became (new points) new local optimas.