Ant Colony Optimization

A brief explanation of its algorithm and its applications in the domain of Machine Learning

Last Updated: June 15, 2023

Ant Colony Optimization (ACO) is a metaheuristic algorithm inspired by the foraging behavior of ants. It mimics the way ants find the shortest path between their colony and food sources. In ACO, artificial ants iteratively build solutions by depositing pheromone trails on the edges of a graph. The pheromone trails serve as communication channels, guiding subsequent ants to prefer paths with higher pheromone levels. Through positive feedback, shorter paths are reinforced over time. ACO has been successfully applied to solve optimization problems, such as the traveling salesman problem as well as selecting the most relevant features in data for machine learning models, by leveraging the collective intelligence of the ant colony to discover near-optimal solutions.

With this in mind a simple question is asked how would using the ant colony optimization algorithm benefit an ML in feature selection for a small dataset? Will it affect the performance of a model positively or negatively with or without it?

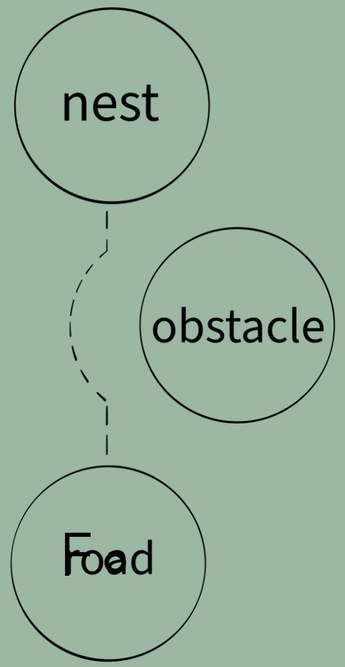
PARTS

OTHER PARTS

Things you should know

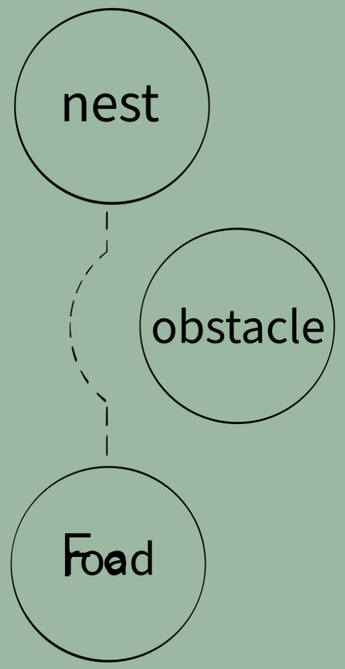
1. An understanding in the Python programming language or any other object oriented programming language
2. An understanding or in-depth understanding of object oriented programming
3. A basic understanding of machine learning
4. Make sure to have a compiler or more preferably an interpreter of Python like python, conda, mini-conda, etc.
5. variables like alpha, beta, Q, rho, epochs, num_ants are all user specific variables that can be edited to the users liking to achieve certain desired results

ANTS IN THE REAL WORLD



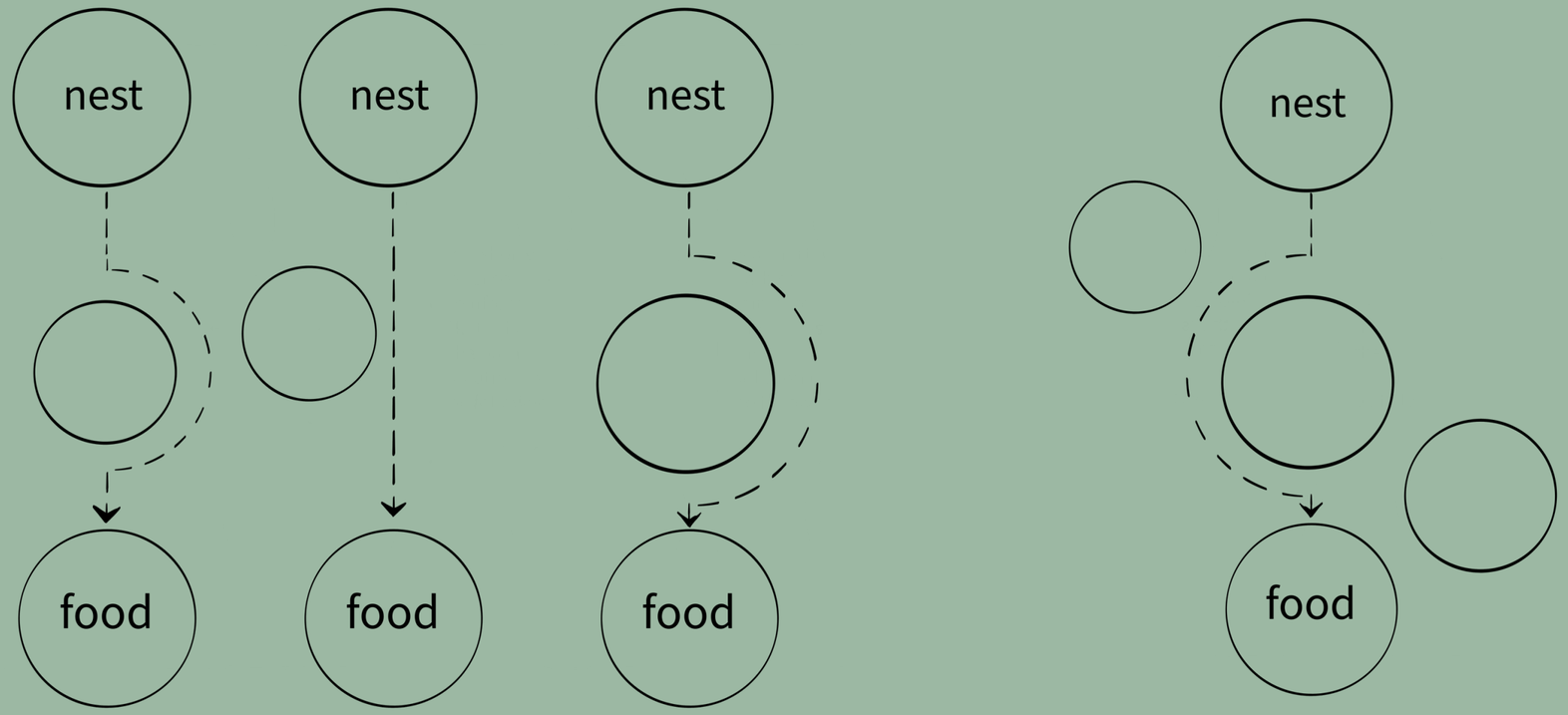
1 The characteristics of Ants:

- Demonstrate collective behaviours such as foraging/seeking food/resources, cooperative support, construction of nests, etc.
- are stimulus-response agents
- each individual ant performs simple and basic actions based on the information of local information
- simple actions that each ant does such as turning where to go appears to have a random component



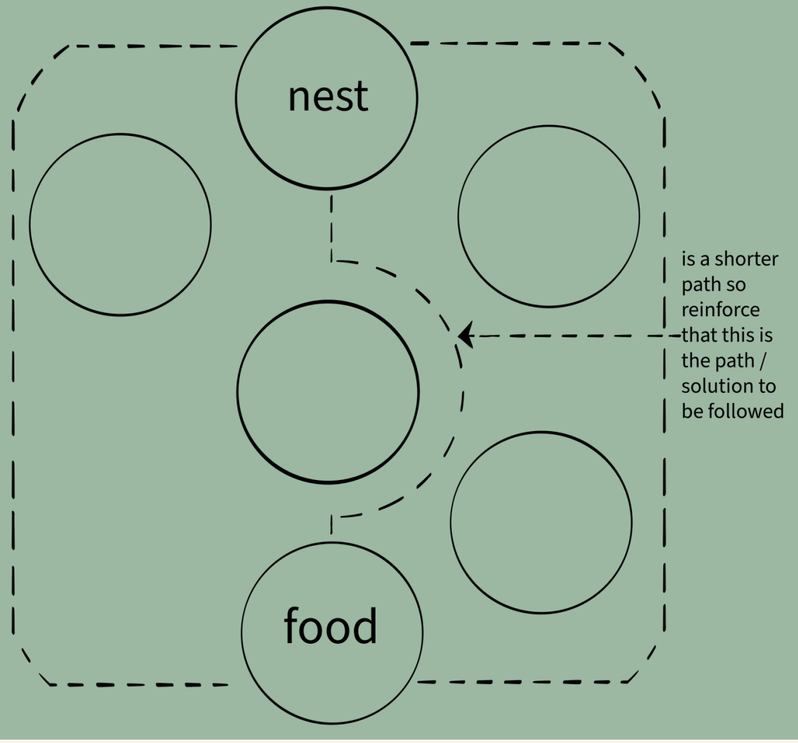
2 Swarm Intelligence

- Like animals that form a swarm like intelligence agents like ants exhibit collective behavior
- In these agents, they interact locally to determine the global solution or shortest path, in the diagram above we see that even when faced with an obstacle ants as a collective work to get to their food source in order to bring back to their nest
- Agents like ants also explore collective problem solving without centralized control
- Ants also work together to find food and haul it back to the nest
- In an ants colony, self-organization of their dynamical mechanisms are where global solutions are found from the interactions of its lower level components which are the ants themselves



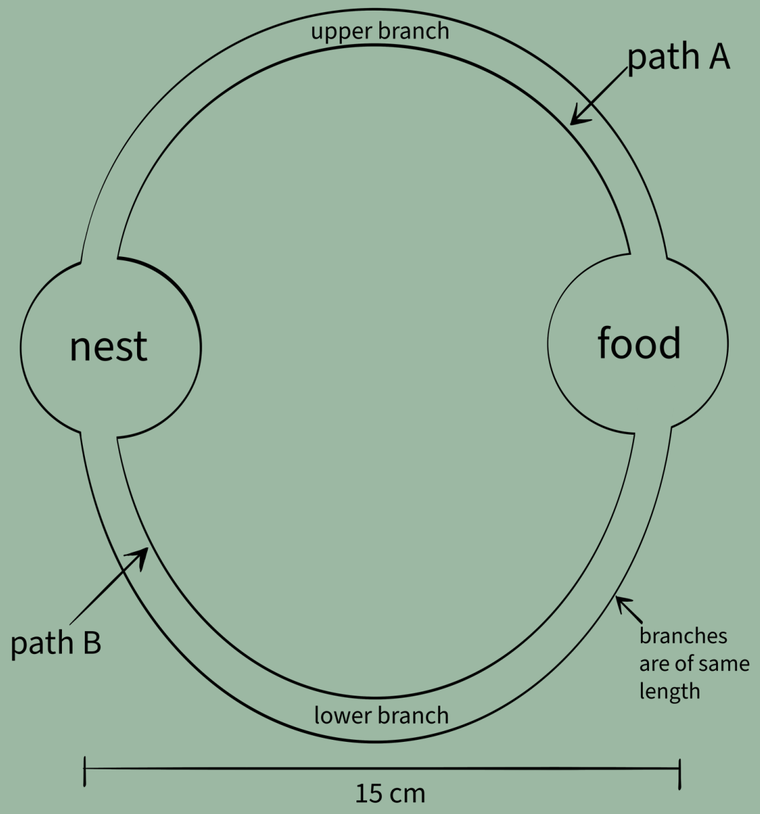
3 Social Colonies

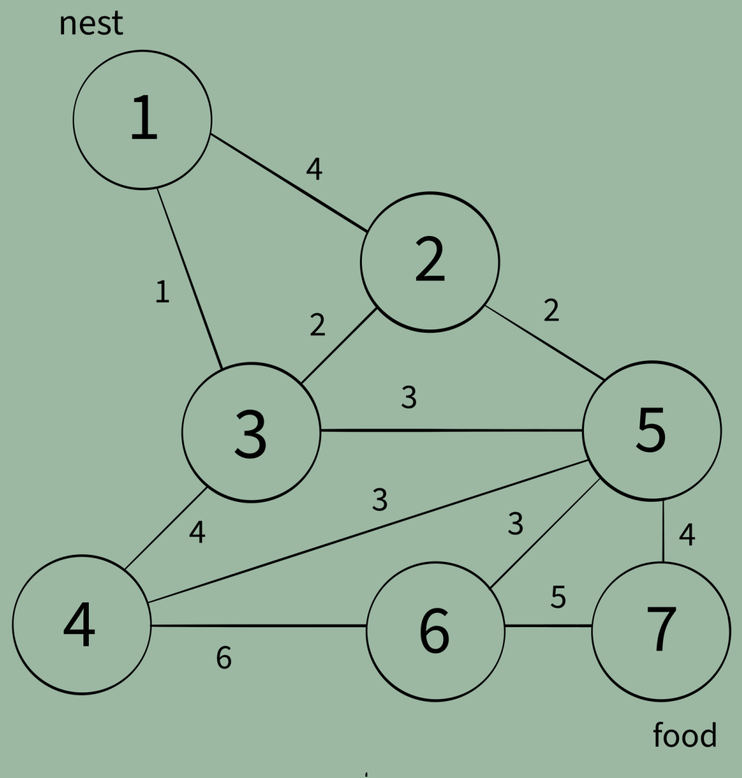
- As shown in the diagrams ant colonies must be and are flexible, in that a colony can respond to external and internal challenges such as when ants are faced with certain obstacles they find a way to still find their destination or "food source"
- robust where tasks are completed even if some ants fail
- decentralized meaning there is no leader in the "colony"
- self-organized wherein paths to solutions are emergent rather than predefined



4 3 components of an Ant Colony

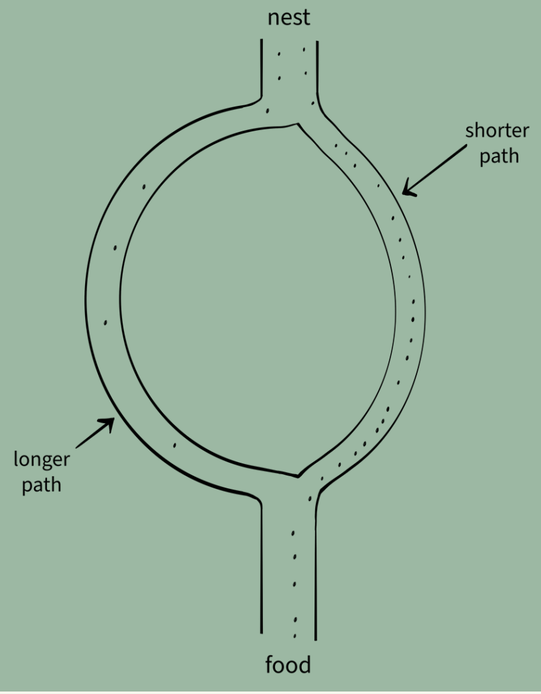
- positive feedback - when a shorter path is found by the agents specifically ants, reinforce it in such a manner that it is able to signal that this path/solution is to be followed
- negative feedback - To reinforce however if solution/path is less optimal or negative pheromone evaporation is introduced. Pheromone evaporation is to prevent premature convergence and/or stagnation. Ants do not know where to go without pheromone of ant ahead
- amplification of fluctuation - lost ant foragers can find new sources of food in the hopes of finding a more optimal path. Revolves around the idea that even if at the moment best path is found, that there may be other more optimal paths other than the current path/solution. Introduces occasionally an ant that moves randomly to find the more optimal path than current path

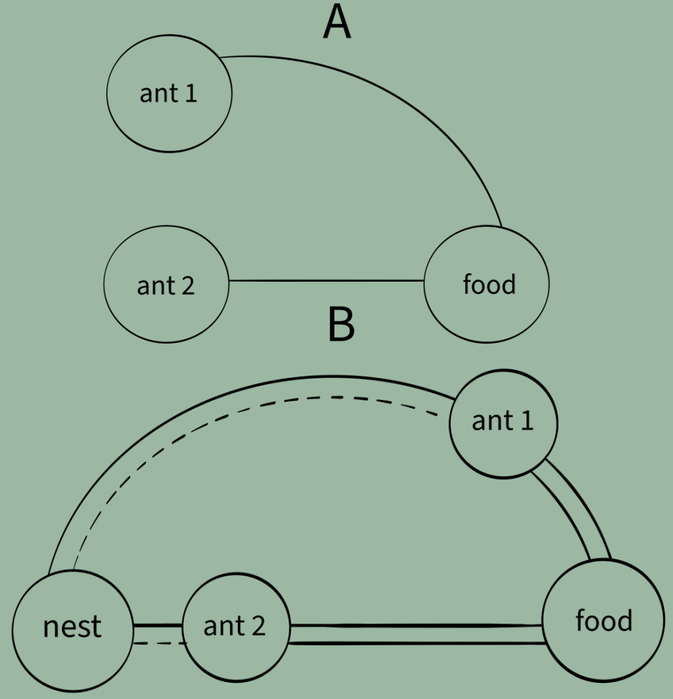


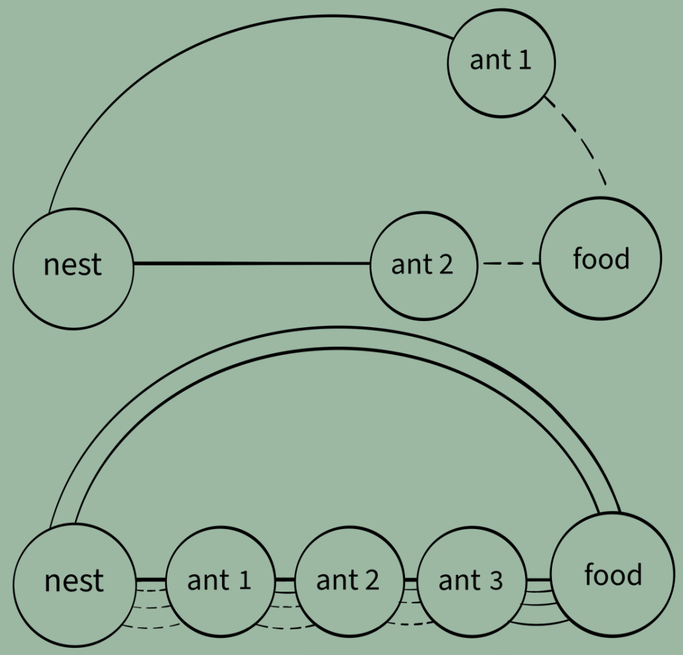


5 finding the shortest path

- In finding the shortest path for instance ants search for an optimal path in a graph like structure
- In this optimal path, the optimal path is the shortest path it takes for the ant from its "nest" to the "food source", without any visible, central, active, coordination mechanisms
- Moreover, to find this shortest path/solution, ants as inspired by biology deposit chemical pheromones while moving around
- In this pheromones, if pheromone concentration/intensity is higher other ants will be more inclined to follow that path, thus making a clear path for other ants to follow to possibly build or establish a path to the food source

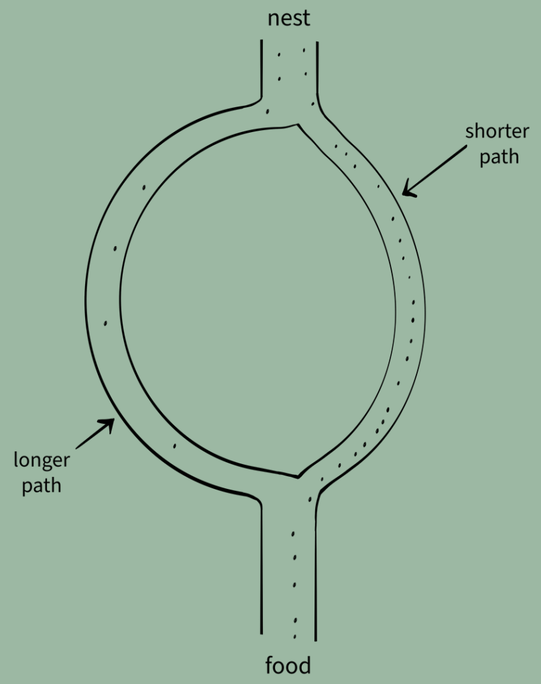


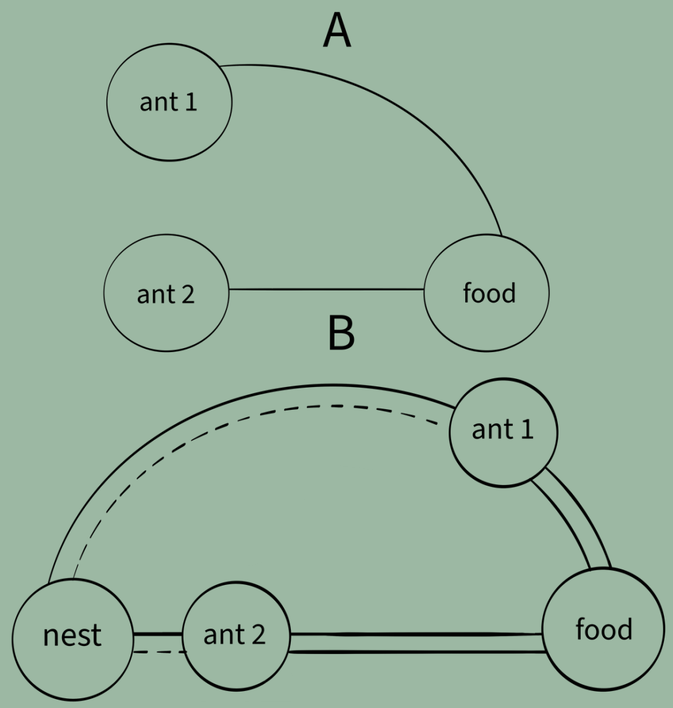


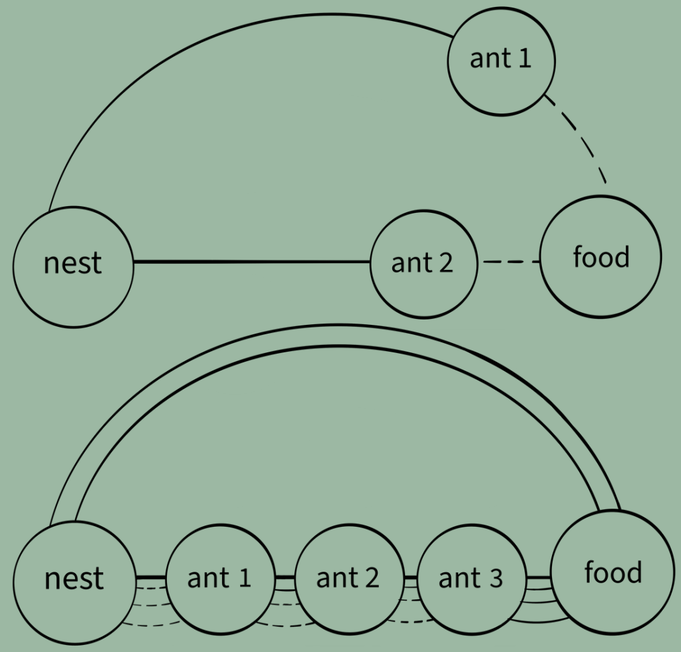


6 finding the shortest path

- since there is always a chance due to amplification of fluctuation of a forager ant to wander and find another more optimal path these ants return to the nest earlier
- When this is the case that indeed a shorter path exists pheromone is then left and then reinforced sooner
- A thing to note also is that in a much longer path pheromone concentration will be lower due to the time it takes for the ant to leave its pheromone
- A shorter path however will be higher and higher in pheromone concentration when using this path since more ants equals more pheromones, and more pheromones equals more ants. In this right-hand side diagram we see that path A has lesser ants treading, this is because path A takes longer to tread thus making the pheromone concentration evaporate faster and be replenished slower, whilst path B because it is shorter will have its pheromone concentrations evaporated slower, and replenished faster due to multiple ants treading it



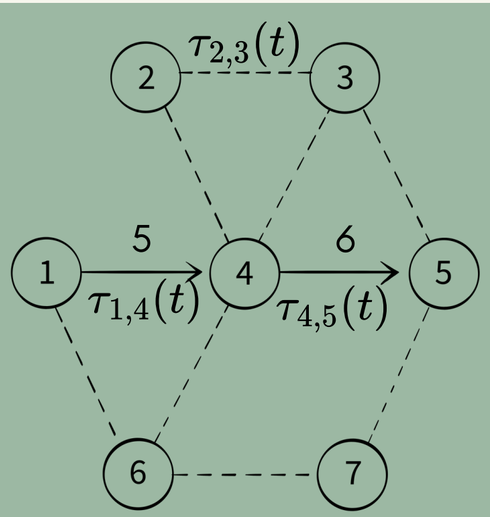


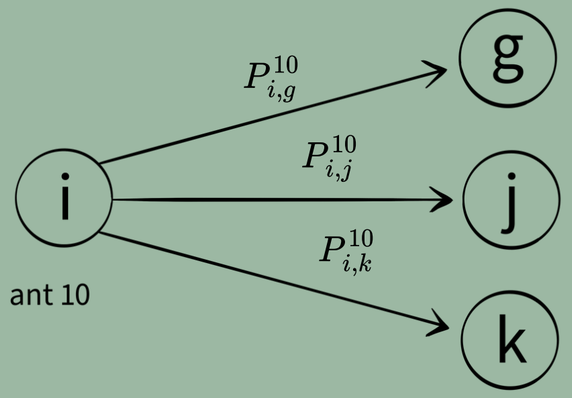


7 stigmergy and artificial pheromone

- stigmergy is a class of mechanisms that mediate animal-animal interactions
- Stigmergy is a form of indirect communication mediated by modifications of the environment (pheromone intensity)
- Some signs observed by agent triggers a response within them that may reinforce/modify signals either positive or negative in order to influence actions of other agents (ants) e.g. leaving more or less pheromones to indicate positive and negative feedback respectively
- Sign based stigmergy moreover is a form of stigmergy where communication between agents is done via signaling
- Such signs of stigmergy are implemented via chemical compounds like pheromones deposited by ants

REPRESENTING ANTS MATHEMATICALLY

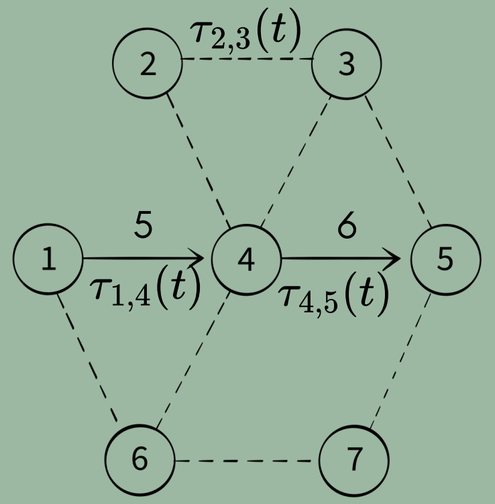


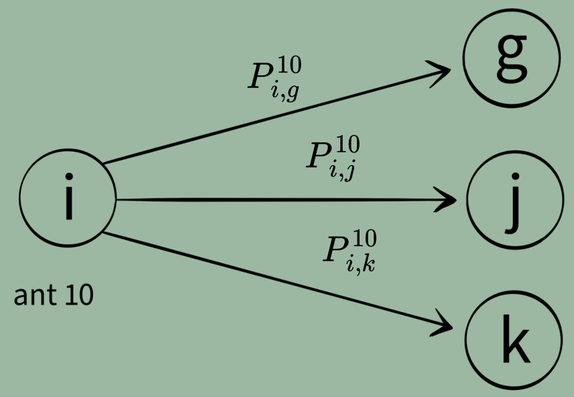


1 constructing the path

To give a brief overview of the variables here, (i, j) for example has variable i where it can be represented by any node in this case it can be the node 1 and j for example node 4 in the first diagram, where as we can see forms an edge or relation between these nodes. Now that we've defined i and j, where it is acutally is used, is as a subscript of the variable $\tau_{i,j}(t)$ where it is the pheromone concentration on edge (i, j) at iteration t for instance $\tau_{i,j}(0)$ here is ph concentration of edge (i, j) at iteration t

Finally the variable $L^k(t)$ represents length of the path (from source to the destination) constructed by ant k This is because when we use the path for example 1, 4, 5, the total cost will be the sum of distances between the nodes of the path e.g. 5+6=11, therefore $L^k(t)$ will be 11





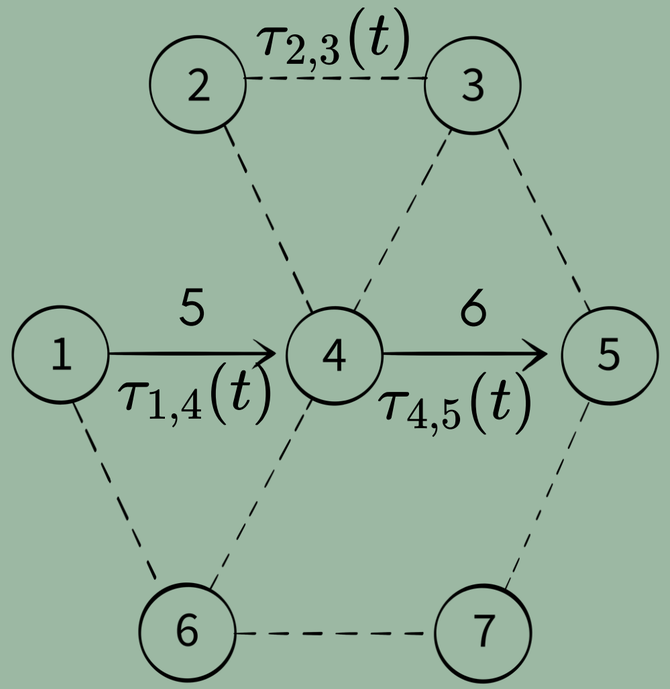
2 transitioning between nodes

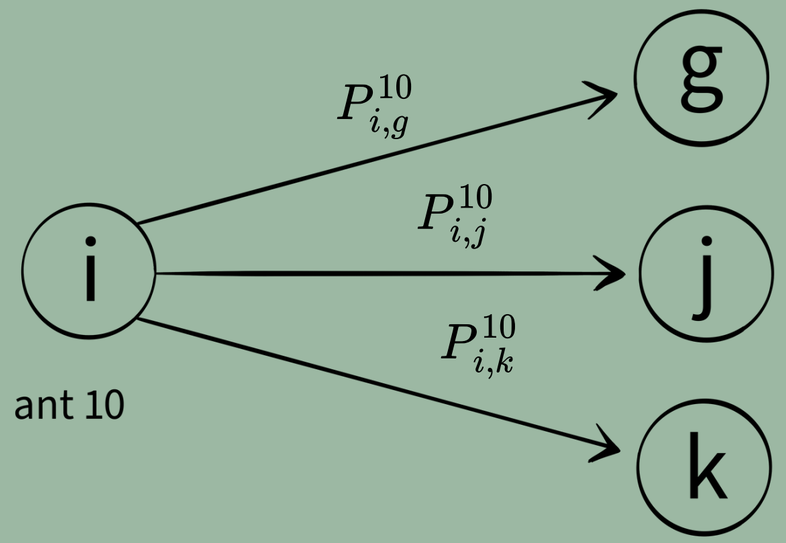
$$P_{i,j}^{k} = \begin{cases} \dfrac{\tau_{i,j}^{\alpha}(t)}{\sum_{u \in \mathcal{N}_i^{k}(t)} \tau_{i,u}^{\alpha}(t)} & \text{if } j \in \mathcal{N}_i^{k}(t) \\ 0 & \text{otherwise} \end{cases}$$

here $P_{i,j}^{k}$ is the transition probability, of ant k selecting the next node j where its constraint $j \in \mathcal{N}_i^{k}(t)$ must be that node j is the set of neighboring feasible nodes connected to node i with respect to ant in iteration t

In our diagram suppose we had ant currently at node 1, then if it has a set of feasible nodes j or nodes not yet visited nor visited already by an ant then we ought to use the corresponding equation which involves the summation notation of all these nodes

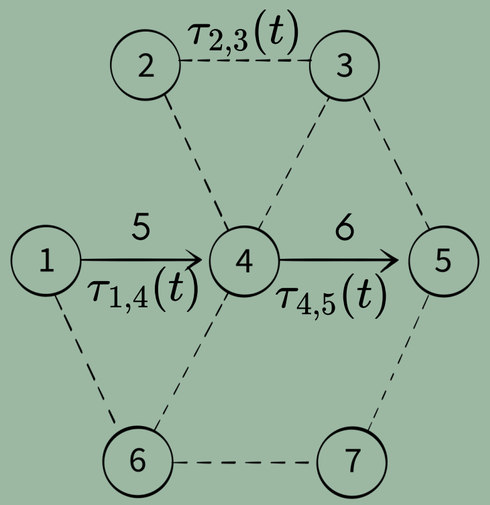
α in $\dfrac{\tau_{i,j}^{\alpha}(t)}{\sum_{u \in \mathcal{N}_i^{k}(t)} \tau_{i,u}^{\alpha}(t)}$ moreover is a constant akin to hyperparameters of a machine learning algorithm which we will have to choose, the only constraint being that it must be greater than 0 or $\alpha > 0$. And as alluded to earlier $\tau_{i,j}$ is the pheromone intensity in between nodes i and j, so for example in our diagram, our nodes i and j were 1 and 4 respectively then $\tau_{i,j}$ would be $\tau_{1,4}$

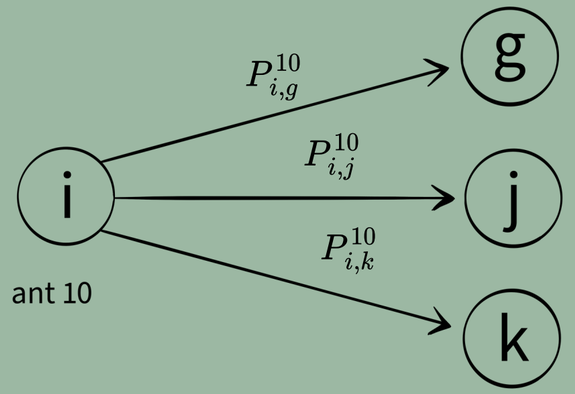




2 transitioning between nodes

Moreover k in the variable $P_{i,j}^{k}$ is also part of a larger set of ants represented as number from 1 to n_k. So if total number of ants n_k is 5 then the set would be $\{1, 2, 3, 4, 5\}$ Like hyperparameters in ML models some useful values for this have been found to be from 10-20 this is because it is again connected to the idea that sometimes ants wander and if a colony has large enough number of ants then it is likely that at least one ant is to stray and find a more optimal solution or path to the "food source"



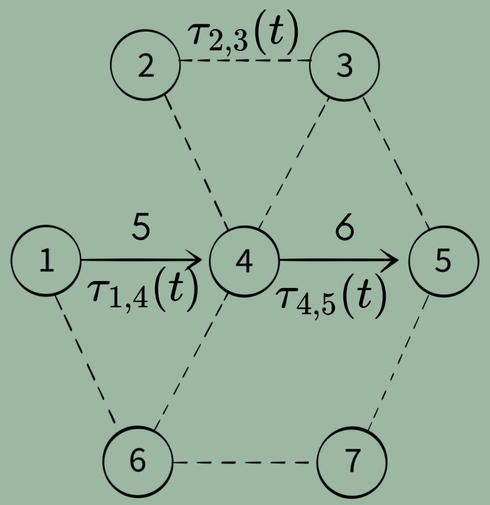


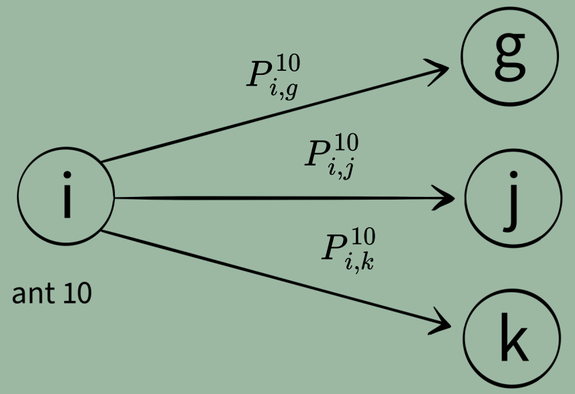
2 transitioning between nodes

defining our transition probability equation earlier its main use is to actually allow the ant to move between nodes, and we will see later on how calculating this transition probability value for all the feasible nodes of ant k at its current node i can serve as a choice to choose the next feasible node in the graph.

In this example here we are assuming $n_k = 10$ as well as assuming ant 10 is currently building a path from node 6. And as we can see , the feasible nodes of node 6 which are the unvisited nodes of an ant, are 1, 4, and 7 denoted as and to calculate the transition probability $P_{i,j}^{10}$ we use our defined equation earlier, which when written is: $\frac{\tau_{6,1}^{\alpha}(1)}{\sum_{u \in \mathcal{N}_6^{10}(1)}^{10} \tau_{6,u}^{\alpha}(1)}$ or $\frac{\tau_{6,1}^{\alpha}(t)}{\tau_{6,1}^{\alpha}(t) + \tau_{6,4}^{\alpha}(t) + \tau_{6,7}^{\alpha}(t)}$

when expanded becuse as we know the feasible nodes of node 6 are nodes 1, 4, and 7 $\mathcal{N}_6^{10} = \{1, 4, 7\}$. In the second example we see ant 10 is at node i, since its feasible nodes are g, j, and k the calculation for the transition probability of ant 10 to these nodes is as follows.





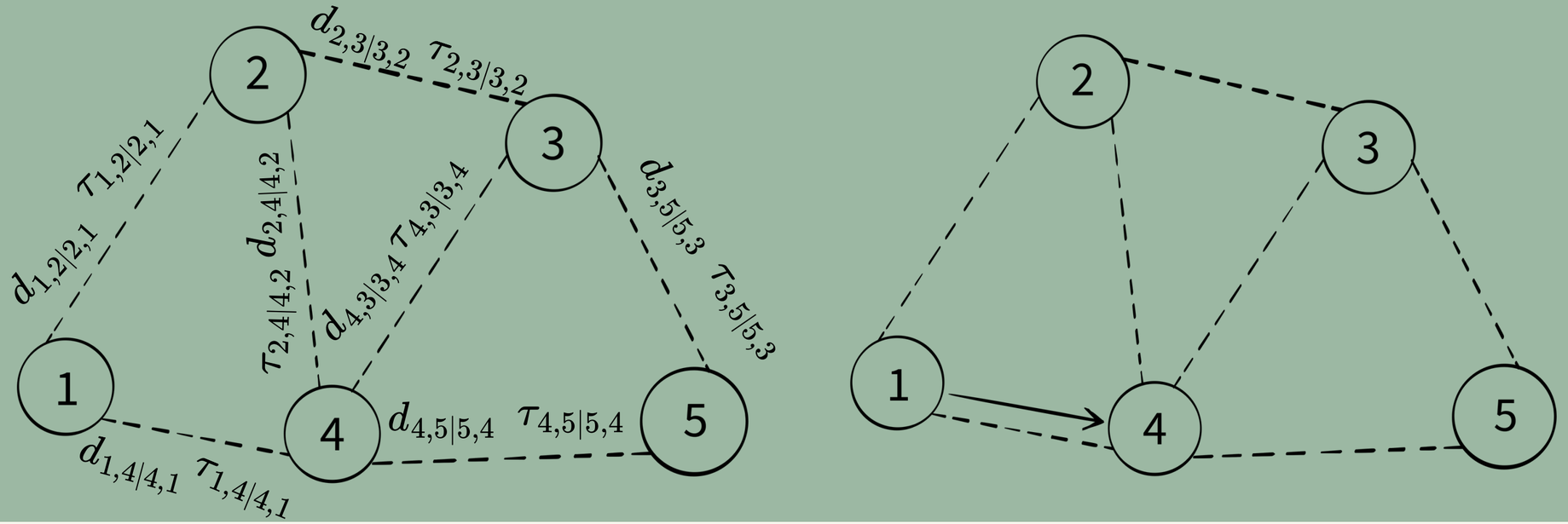
2 transitioning between nodes

as mentioned in the previous slide here are the calculations of each node i's feasible nodes transition probability, which are $P_{i,j}^{10}$ $P_{i,g}^{10}$ and $P_{i,k}^{10}$. And we know that when we now have our feasible nodes we calculate the transition probability of each for each of these feasible nodes of i

$$P_{i,g}^{10} = \frac{\tau_{i,g}^{\alpha}}{\tau_{i,g}^{\alpha} + \tau_{i,j}^{\alpha} + \tau_{i,k}^{\alpha}}$$ is the calculation for transition probability of node i's feasible node g

$$P_{i,j}^{10} = \frac{\tau_{i,j}^{\alpha}}{\tau_{i,g}^{\alpha} + \tau_{i,j}^{\alpha} + \tau_{i,k}^{\alpha}}$$ is the calculation for transition probability of node i's feasible node j

$$P_{i,k}^{10} = \frac{\tau_{i,k}^{\alpha}}{\tau_{i,g}^{\alpha} + \tau_{i,j}^{\alpha} + \tau_{i,k}^{\alpha}}$$ is the calculation for transition probability of node i's feasible node k

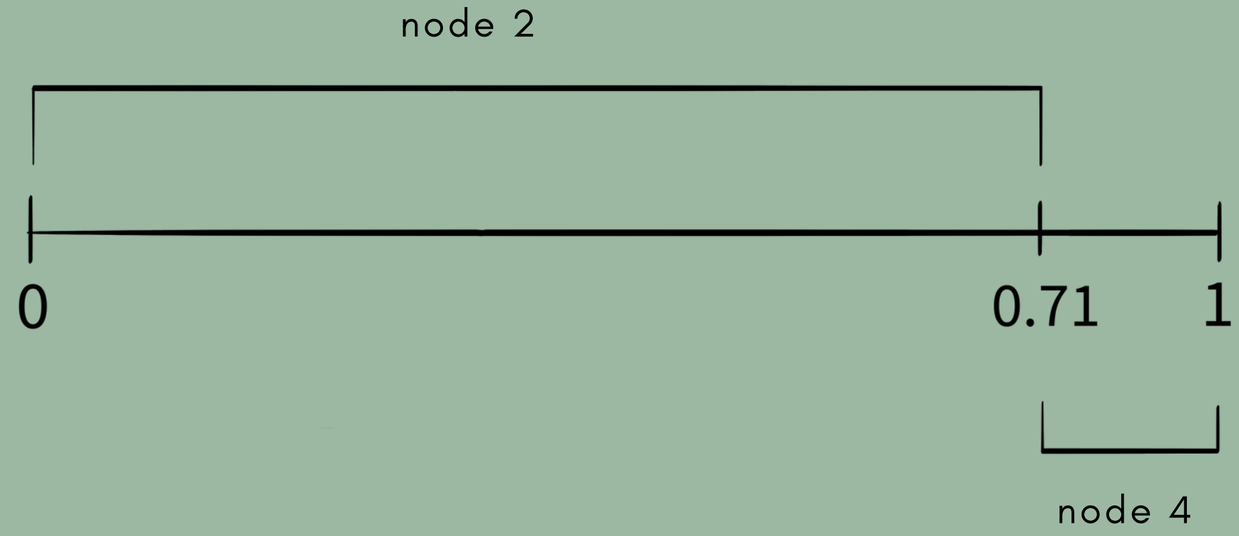


3 choosing the next node

Here ant at node 1 has $\mathcal{N}_1^1(t) = \{2, 4\}$ as its set of feasible neighboring nodes. Starting our path from node 1 we calculate not only the transition probability of feasible neighboring nodes 2 and 4 but also its accumulated transition probability, for this will be important for the ant in choosing its next node

asssuming that we have pheromone values where $\tau_{1,4|4,1}$ is 0.5, $\tau_{1,2|2,1}$ is 0.2, and our hyper parameter $\alpha = 1$ the calculations of each feasible nodes 2 and 4 transition probability is as follows respectively: $P_{1,4}^2(t) = \frac{\tau_{1,4}^\alpha(t)}{\tau_{1,2}^\alpha(t) + \tau_{1,4}^\alpha(t)}$ $P_{1,2}^2(t) = \frac{\tau_{1,2}^\alpha(t)}{\tau_{1,2}^\alpha(t) + \tau_{1,4}^\alpha(t)}$

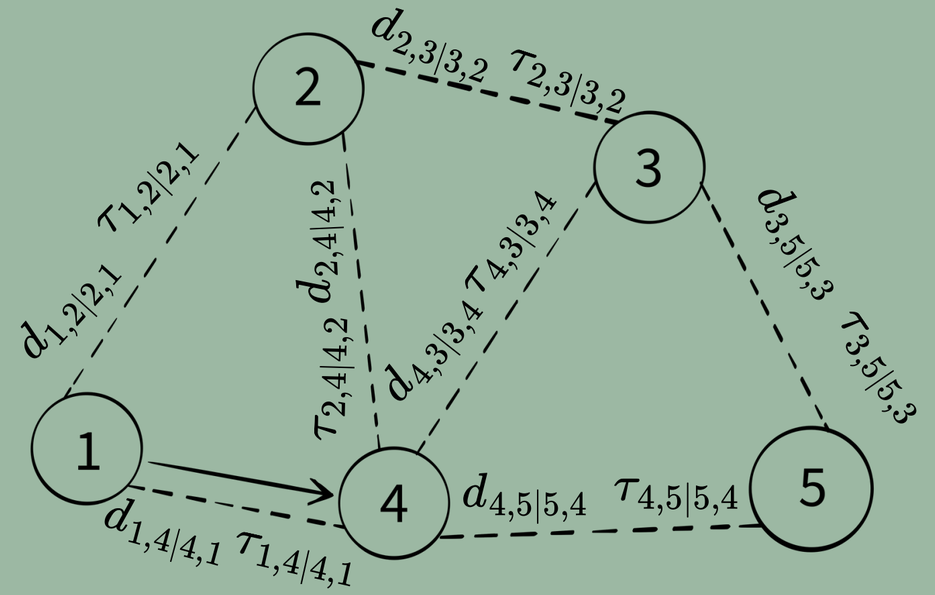
assuming we have substituted these values our transition probabilities for each node 2 and 4 would rseult in the values 0.71 and 0.29 initializing our initial sum, of course, to 0, because the formulae used in calculating the accumulated transition probability for these nodes 2 and 4 are: $acc = 0$, $acc_{P_{1,4}^1(t)} = P_{1,4}^1(t) + acc_{P_{1,2}^1(t)}$, $acc_{P_{1,2}^1(t)} = P_{1,2}^1(t) + acc$. The final resulting accumulated transition probabilities would be 0.71 and 1, which we will use in determining which node the ant should go to next

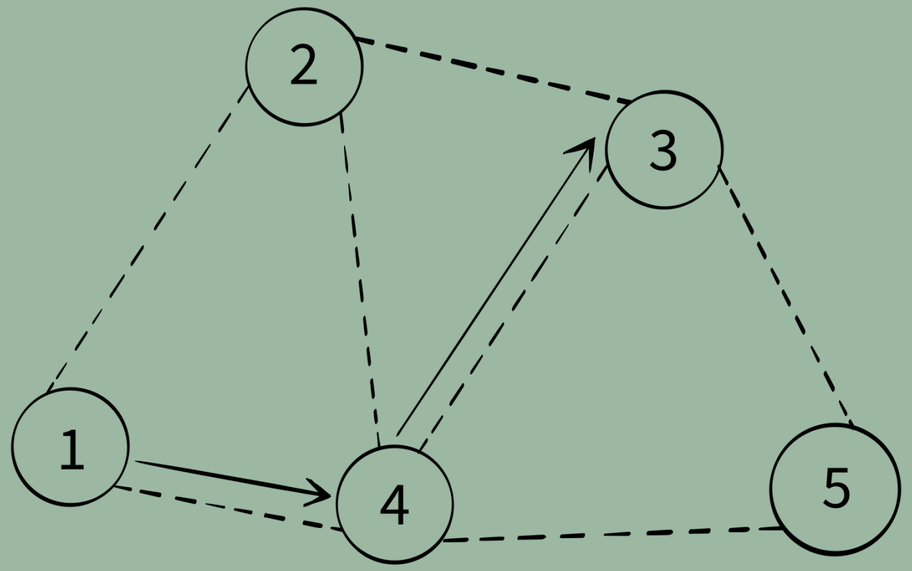


3 choosing the next node

recall that our resulting accumulated transition probabilities for nodes 2 and 4 are 0.71 and 1 respectively, and in the diagram above we label the point until 0.71 as node 2 and the points from 0.71 to 1 as node 4. From here since the highest accumulated transition probability is 1 we will have to generate a random number r between 0 and this highest accumulated transition probability number inclusively $r \leq P_A$ which here is P_A

say the generated number r is 0.8, because in our diagram of accumulated transition probabilities it is in the bounds of 0.71 and 1 the current ant chooses node 4 as its next node. This is why in our previous diagram the ant goes to node 4





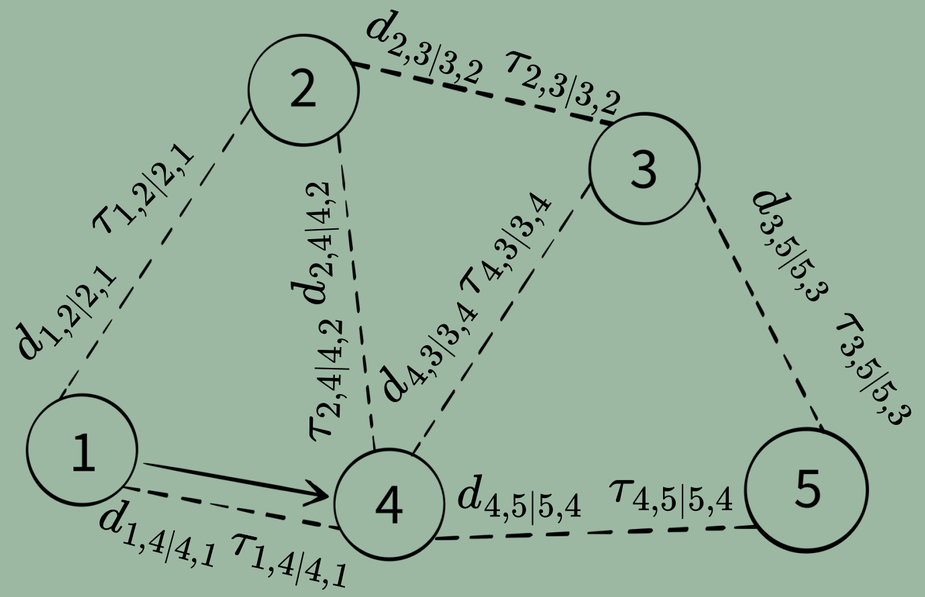
4 constructing the path

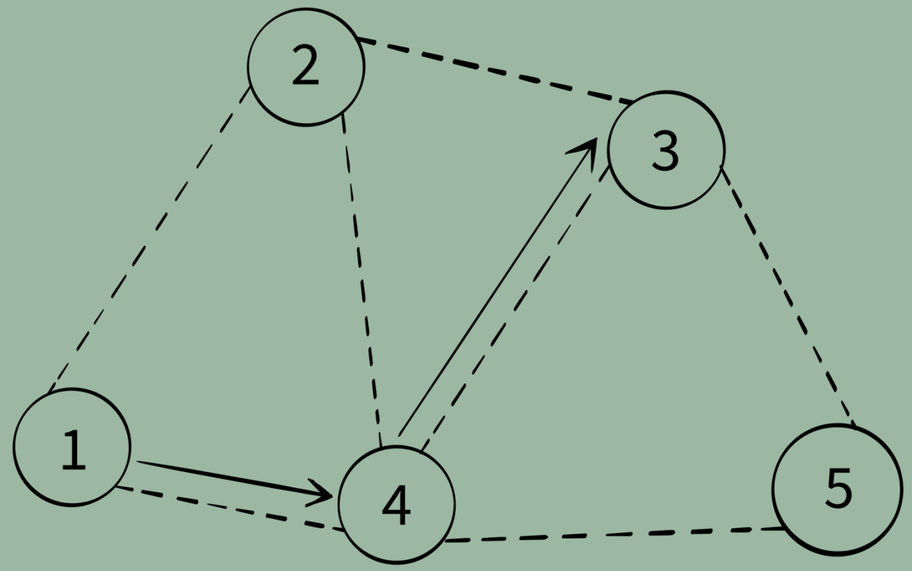
since ant 1 for instance is at node 4, assume we have calculated the transition probabilities for each feasible node denoted as $\mathcal{N}_4^1(t) = \{2, 3, 5\}$ which come out to the following because we have 3 feasible nodes 2, 3, and 5:

$$P_{4,2}^1(t) = \frac{\tau_{4,2}^\alpha(t)}{\tau_{4,2}^\alpha(t) + \tau_{4,3}^\alpha(t) + \tau_{4,5}^\alpha(t)} \qquad P_{4,3}^1(t) = \frac{\tau_{4,3}^\alpha(t)}{\tau_{4,2}^\alpha(t) + \tau_{4,3}^\alpha(t) + \tau_{4,5}^\alpha(t)} \qquad P_{4,5}^1(t) = \frac{\tau_{4,5}^\alpha(t)}{\tau_{4,2}^\alpha(t) + \tau_{4,3}^\alpha(t) + \tau_{4,5}^\alpha(t)}$$

Once we have calculated these transition probabilitie obviously the next step now is to calculate the accumulated transition probability by of course starting our accumulator variable to 0 e.g. $acc = 0$ and then adding the subsequent transition probabilities for each node we calculated earlier which are $P_{4,2}^1$ $P_{4,3}^1$ and $P_{4,5}^1$ which we will also use to subsitute in the following equations in order to calculate the accumulated transition probabilities for each of our nodes

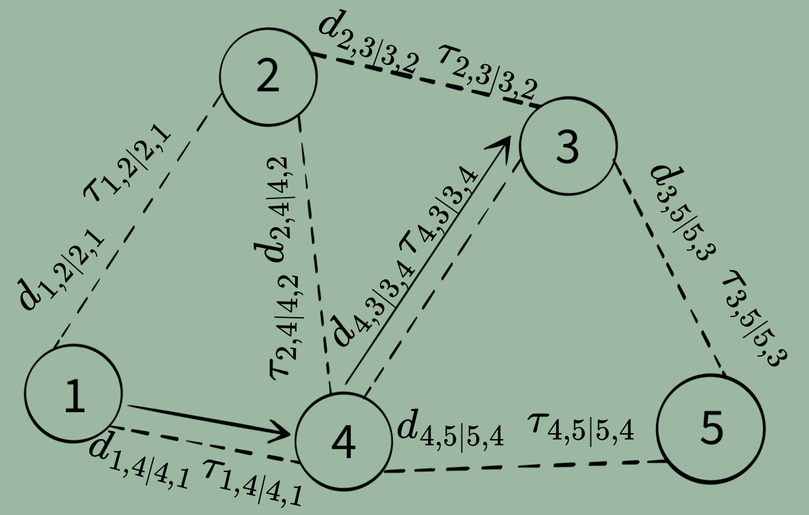
$$acc_{P_{4,2}^1(t)} = P_{4,2}^1(t) + acc \qquad acc_{P_{4,3}^1(t)} = P_{4,3}^1(t) + acc_{P_{4,2}^1(t)} \qquad acc_{P_{4,5}^1(t)} = P_{4,5}^1(t) + acc_{P_{4,3}^1(t)}$$





4 constructing the path

Once done in calculating the accumulated transition probabilities for each of those feasible node we saw earlier we can assume also that we have generated a random number r and this number lied between the first accumulated transition probability which is at node 2 and the second accumulated transition probability which is at node 3. From this we can infer that ant 1 has chosen node 3 as its next node. Moreover we can divide the major steps of an ant constructing a path into two: calculating the accumulated transition probability and generating a random number to determine the node the ant goes to next.

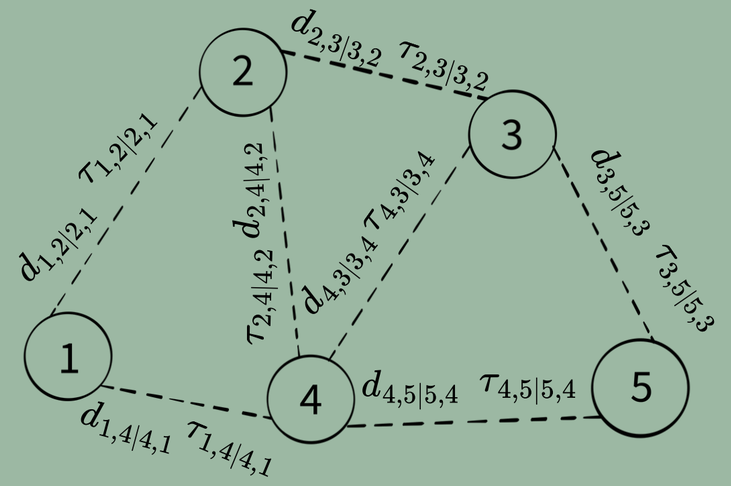


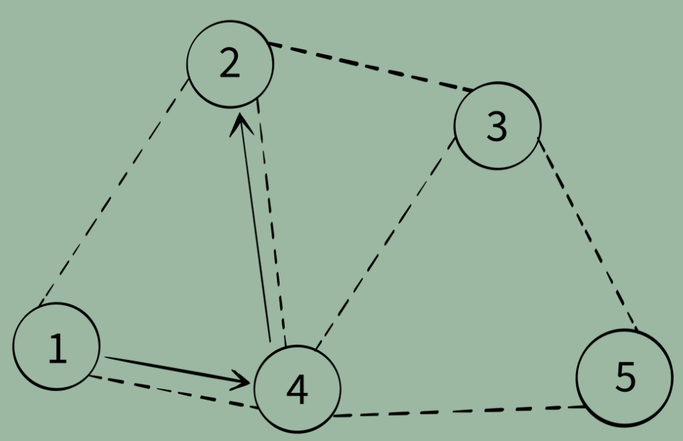
4 constructing the path

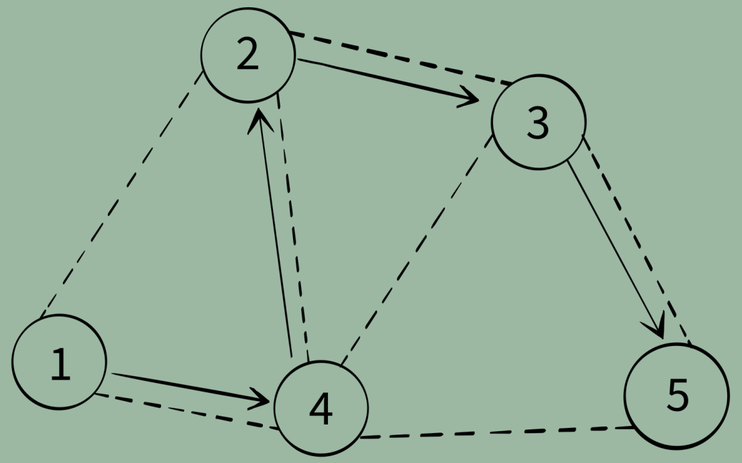
At node 3 we see that our feasible nodes our 2 and 5 denoted as $\mathcal{N}_3^1(t) = \{2, 5\}$ therefore making our calculations for each transition probability the following for feasible nodes 2 and 5 respectively of node 3:

$P_{3,2}^1(t) = \dfrac{\tau_{3,2}^\alpha(t)}{\tau_{3,2}^\alpha(t) + \tau_{3,5}^\alpha(t)}$ for node 2 and $P_{3,5}^1(t) = \dfrac{\tau_{3,5}^\alpha(t)}{\tau_{3,2}^\alpha(t) + \tau_{3,5}^\alpha(t)}$ for node 5

Here we assume we have already done the steps involved in choosing a node such as calculating the transition probability the accumulated transition probability and then generating a random number r. Again to understand take for instance that our transition probability values for feasible nodes 2 and 5 have come out to be 0.4 and 0.6 because we want to start our accumulator to 0 programmatically ($acc = 0$), our first accumulated transition probability for node 2 will be 0.4 + 0 denoted as $acc_{P_{3,2}^1(t)} = P_{3,2}^1(t) + acc$ then for the accumulated transition probability of node 5 it will be the accumulated transition probability value of node 2 + 0.6 denoted as $acc_{P_{3,5}^1(t)} = P_{3,5}^1(t) + acc_{P_{3,2}^1(t)}$ Assuming our ant has choosen node 5 which in our example is our destination node, our constructed path would now be $x^1(t) = \{1, 4, 3, 5\}$ which allows us to calculate the total distance of our path denoted as $f(x^1(t)) = d_{1,4} + d_{4,3} + d_{3,5}$







4 constructing the path

Because we are iterating over potentially multiple ants also we do the operations we have done previously for ant 2 as well. For this example say the source node was still 1 and the destination still 5. Again the previous processes are applied on the ff. here.

$\mathcal{N}_1^2(t) = \{2, 4\}$

$$P_{1,4}^2(t) = \frac{\tau_{1,4}^\alpha(t)}{\tau_{1,2}^\alpha(t) + \tau_{1,4}^\alpha(t)}$$

$$P_{1,2}^2(t) = \frac{\tau_{1,2}^\alpha(t)}{\tau_{1,2}^\alpha(t) + \tau_{1,4}^\alpha(t)}$$

$acc = 0$

$acc_{P_{1,2}^2(t)} = P_{1,2}^2(t) + acc$

$acc_{P_{1,4}^2(t)} = P_{1,4}^2(t) + acc_{P_{1,2}^2(t)}$

assume ant 2 chose node 4

$\mathcal{N}_4^2(t) = \{2, 3, 5\}$

$$P_{4,2}^2(t) = \frac{\tau_{4,2}^\alpha(t)}{\tau_{4,2}^\alpha(t) + \tau_{4,3}^\alpha(t) + \tau_{4,5}^\alpha(t)}$$

$$P_{4,3}^2(t) = \frac{\tau_{4,3}^\alpha(t)}{\tau_{4,2}^\alpha(t) + \tau_{4,3}^\alpha(t) + \tau_{4,5}^\alpha(t)}$$

$$P_{4,5}^2(t) = \frac{\tau_{4,5}^\alpha(t)}{\tau_{4,2}^\alpha(t) + \tau_{4,3}^\alpha(t) + \tau_{4,5}^\alpha(t)}$$

$acc = 0$

$acc_{P_{4,2}^2(t)} = P_{4,2}^2(t) + acc$

$acc_{P_{4,3}^2(t)} = P_{4,3}^2(t) + acc_{P_{4,2}^2(t)}$

$acc_{P_{4,5}^2(t)} = P_{4,5}^2(t) + acc_{P_{4,3}^2(t)}$

assume ant 2 chose node 2

$\mathcal{N}_2^2(t) = \{3\}$

$$P_{2,3}^2(t) = \frac{\tau_{2,3}^\alpha(t)}{\tau_{2,3}^\alpha(t)} = 1$$

$acc = 0$

$acc_{P_{2,3}^2(t)} = P_{2,3}^2(t) + acc$

assume ant 2 chose node 3

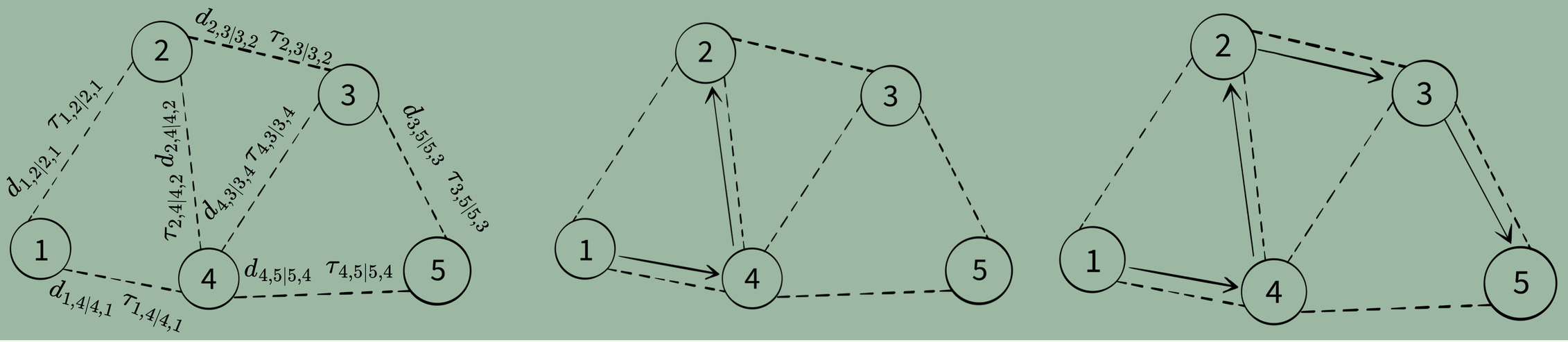
$\mathcal{N}_3^2(t) = \{5\}$

$$P_{2,5}^2(t) = \frac{\tau_{2,5}^\alpha(t)}{\tau_{2,5}^\alpha(t)} = 1$$

$acc = 0$

$acc_{P_{2,5}^2(t)} = P_{2,5}^2(t) + acc$

assuming ant 2 chose node 5, this ant has now arrived at its destination



4 constructing the path

since ant 2 has now arrived at node 5 which is the destination node this ant therefore has now constructed a path or its solution set denoted as $x^2(t) = \{1, 4, 2, 3, 5\}$ meaning it has passed through the nodes 1, 4, 2, 3, and 5 as show in the diagram above with the arrows akin to ant 1 which recall had the solution set $x^1(t) = \{1, 4, 3, 5\}$. This is now the solution of ant 2 and when calculating the cost the calculation amounts to the equation $f(x^2(t)) = d_{1,4} + d_{4,2} + d_{2,3} + d_{3,5}$

where $d_{1,4}$ $d_{4,2}$ $d_{2,3}$ and $d_{3,5}$ are actually the distance values between each node, so in our example if ant 2 has crossed from node 1 to node 4 then its distance value is denoted as $d_{1,4}$ which we see in our diagram above, the only difference is that since the graph is undirected it's why edges can be interchangeable which when denoted are the following

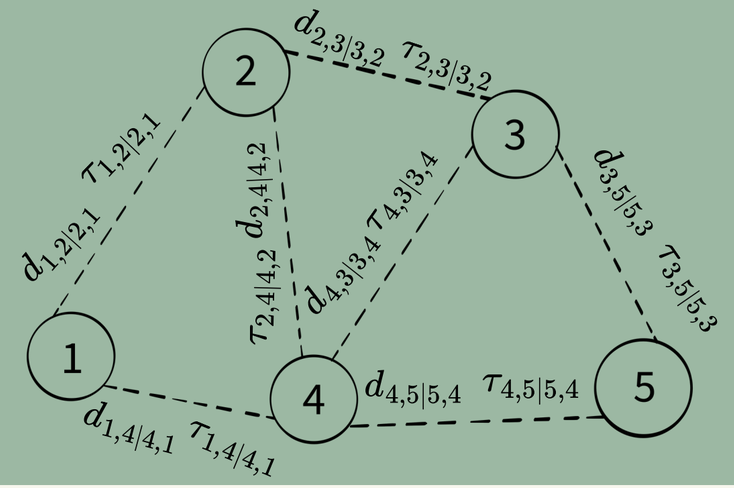
$d_{1,4|4,1}$ which is the edge from node 1 to 4 and vice versa

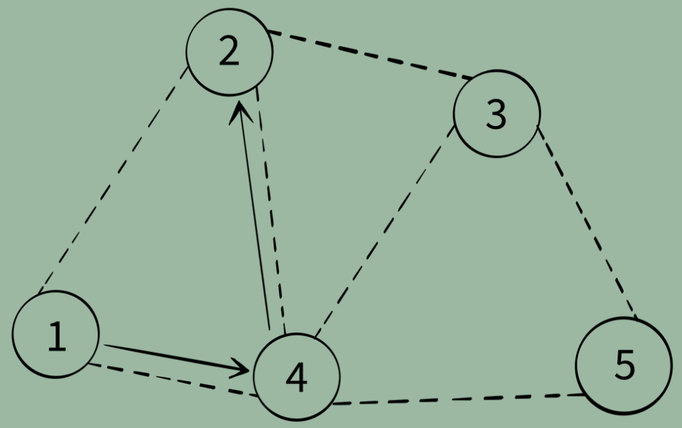
$d_{2,4|4,2}$ which is the edge from node 4 to 2 and vice versa

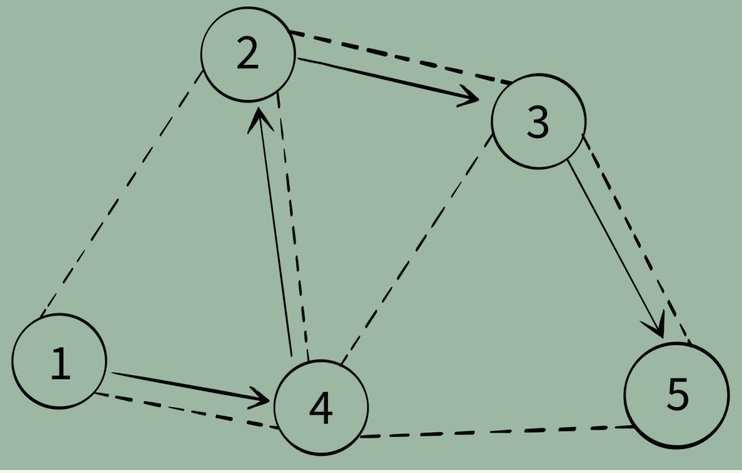
$d_{3,5|5,3}$ which is the edge from node 3 to 5 and vice versa

$d_{2,3|3,2}$ which is the edge from node 2 to 3 and vice versa

and which we will use onwards, on subsequent steps. But to conclude this part suppose our distance values for these edges were 6, 14, 2, 7, since we have 4 edges all in all using path 1, 4, 2, 3, and 5, our final cost would be 29





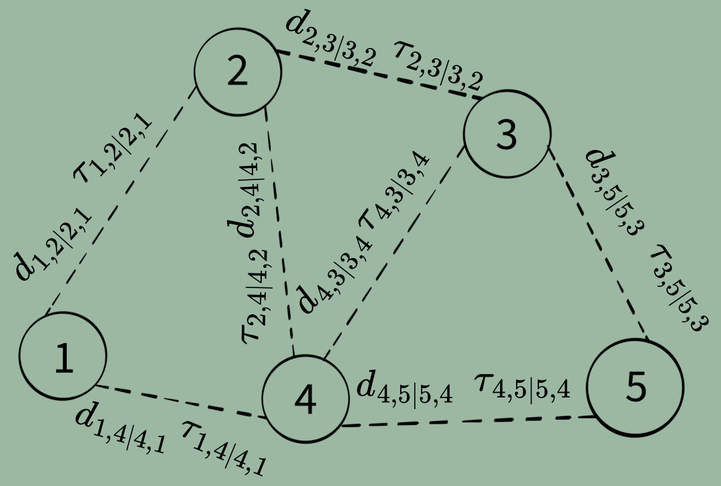


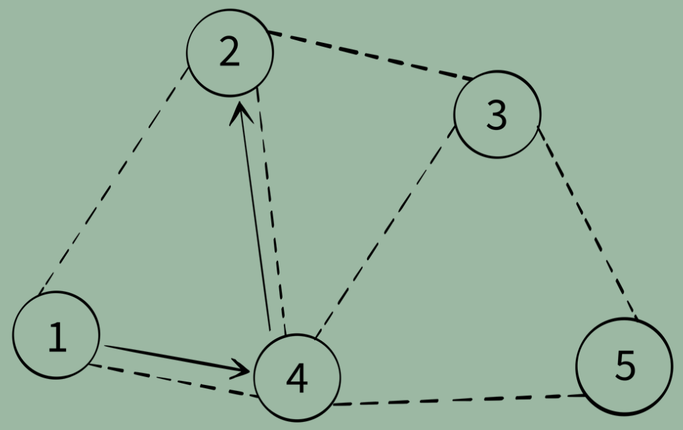
5 evaporation of pheromone intensity/negative feedback

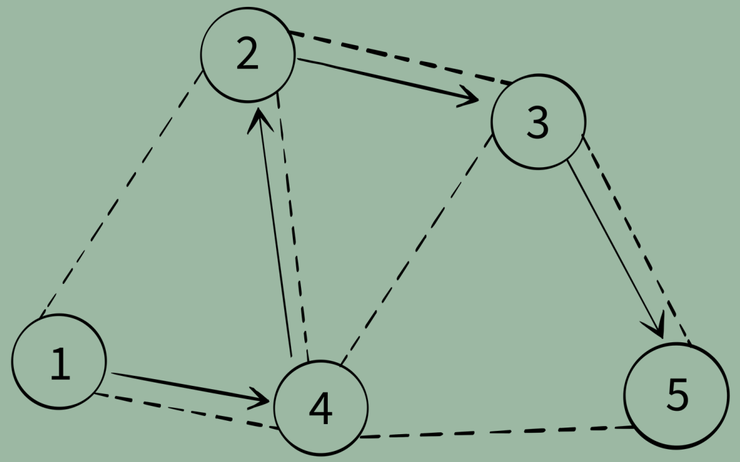
akin to updating the pheromone intensity to represent the process of ants having to follow a much more optimal path which is shown later, the path in which to avoid by ants must also be reinforced and this is done through the evaporation of the pheromone intensity of paths that are not optimal

that is for each edge (i, j) pheromone intensity is reduced through the formula $\tau_{i,j}(t) = (1 - \rho) * \tau_{i,j}(t)$ where greek letter rho ρ is a set of values between 0 and 1 exclusively, which represents also as our evaporation rate $\rho \in (0, 1)$

that is when given our pheromone adjacency matrix $\mathrm{T} = \begin{bmatrix} 0 & \tau_{1,2} & 0 & \tau_{1,4} & 0 \\ \tau_{2,1} & 0 & \tau_{2,3} & \tau_{2,4} & 0 \\ 0 & \tau_{3,2} & 0 & \tau_{3,4} & \tau_{3,5} \\ \tau_{4,1} & \tau_{4,2} & \tau_{4,3} & 0 & \tau_{4,5} \\ 0 & 0 & \tau_{5,3} & \tau_{5,4} & 0 \end{bmatrix}$ we will have to reduce the pheromones of our paths to a degree in order to further reinforce other ants not to follow a path with lesser pheromone values, and to follow a path with more pheromone values which we will see later



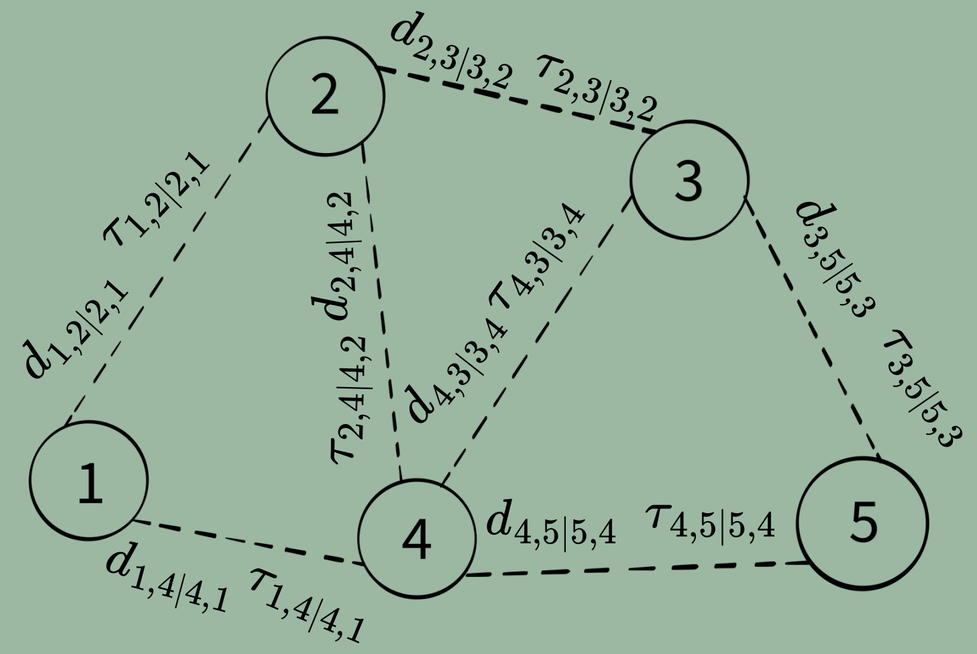




6 updating the pheromone concentration/positive feedback

In our example since we merely had 2 ants in our sample problem but once all ants $1, \ldots, n_k$ have constructed their path by calculating their respective transition probabilities $P_{i,j}^k$ and subsequently their accumulated transition probabilities from source to destination and all pheromone intensities on edge (i, j) is updated at the next generation/time t the process (that is when pheromone evaporation has also finished) then repeats again for each ant in the next iteration $t + 1$

In this example because our ants our finished finding paths, the process of reinforcing the solution or path found by a certain path can be represented and now done using the following calculations, which will do as such, making ants in the next iteration indeed follow this more optimal path. And because previously our negative feedback reduced all our pheromone values, this process of positively reinforcing the optimal paths that certain ants made therefore only reinforces certain pheromone values in our pheromone adjacency matrix T

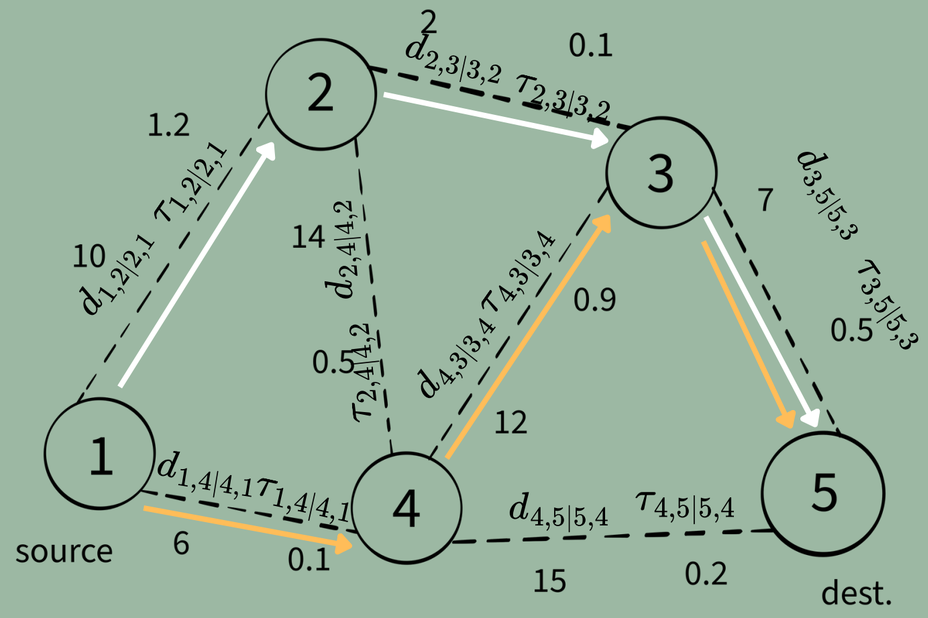


$$\mathrm{T} = \begin{bmatrix} 0 & \tau_{1,2} & 0 & \tau_{1,4} & 0 \\ \tau_{2,1} & 0 & \tau_{2,3} & \tau_{2,4} & 0 \\ 0 & \tau_{3,2} & 0 & \tau_{3,4} & \tau_{3,5} \\ \tau_{4,1} & \tau_{4,2} & \tau_{4,3} & 0 & \tau_{4,5} \\ 0 & 0 & \tau_{5,3} & \tau_{5,4} & 0 \end{bmatrix}$$

6 updating the pheromone concentration/positive feedback

$\tau_{i,j}(t+1) = \tau_{i,j}(t) + \sum_{k=1}^{n_k} \Delta\tau_{i,j}^{k}(t)$ is what we use to update our pheromone concentrations for each edge (represented as an adjacency matrix) where $x^k(t)$ is the chosen solution of ant k, $\Delta\tau_{i,j}^{k}(t) = \begin{cases} \frac{Q}{f(x^k(t))} & \text{if } edge(i,j) occurs in path for x^k(t) \\ 0 & \text{otherwise} \end{cases}$ is a summation of a constant where its value is $Q > 0$ and the function where $x^k(t)$ is passed calculates the quality of the solution or the sum/length of the path used by ant k

k

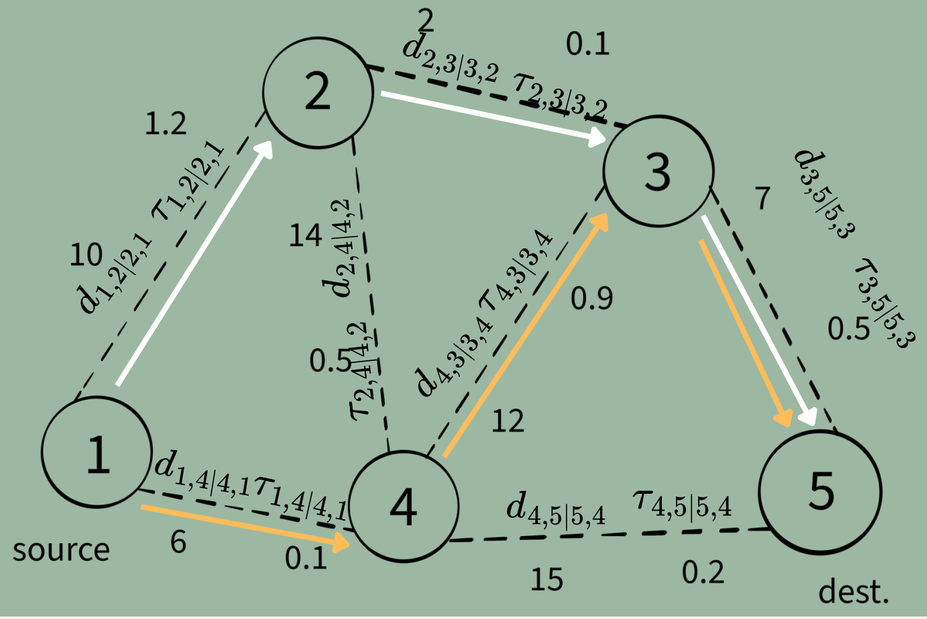


$$D = \begin{bmatrix} - & 10 & - & 6 & - \\ 10 & - & 2 & 14 & - \\ - & 2 & - & 12 & 7 \\ 6 & 14 & 12 & - & 15 \\ - & - & 7 & 15 & - \end{bmatrix} \qquad \mathrm{T} = \begin{bmatrix} - & 0.3 & - & 0.8 & - \\ 0.3 & - & 1.5 & 0.1 & - \\ - & 1.5 & - & 0.9 & 0.5 \\ 0.8 & 0.1 & 0.9 & - & 0.2 \\ - & - & 0.5 & 0.2 & - \end{bmatrix}$$

7 putting it all together

Recall that evaporation pheromone concentration is first and foremost before updating pheromone concentration values for the next iteration. Assuming the evaporation rate ρ is 0.2, then $(1-\rho)$ would be 0.8, therefore making each pheromone values in T reduce by about 80%. Once done we now move on to the final part of a single iteration of the ACO algorithm, that is updating our pheromone adjacency matrix, by taking into account the solutions/paths found by our ants, which in our example have been $x^1(t) = \{1,4,3,5\}|\{(1,4),(4,3),(3,5)\}$ for ant 1, and $x^2(t) = \{1,2,3,5\}|\{(1,2),(2,3),(3,5)\}$for ant 2.

because we already have our solutions and because the pheromone concentration update requires the cost of our two ants solutions we need to calculate first these cost values which actually comes out to $f(x^1(t)) = 6+12+7$ or 25 for ant 1 and $f(x^2(t)) = 10+2+7$ or 19 for ant 2



$$D = \begin{bmatrix} - & 10 & - & 6 & - \\ 10 & - & 2 & 14 & - \\ - & 2 & - & 12 & 7 \\ 6 & 14 & 12 & - & 15 \\ - & - & 7 & 15 & - \end{bmatrix} \qquad \mathrm{T} = \begin{bmatrix} - & 0.3 & - & 0.8 & - \\ 0.3 & - & 1.5 & 0.1 & - \\ - & 1.5 & - & 0.9 & 0.5 \\ 0.8 & 0.1 & 0.9 & - & 0.2 \\ - & - & 0.5 & 0.2 & - \end{bmatrix}$$

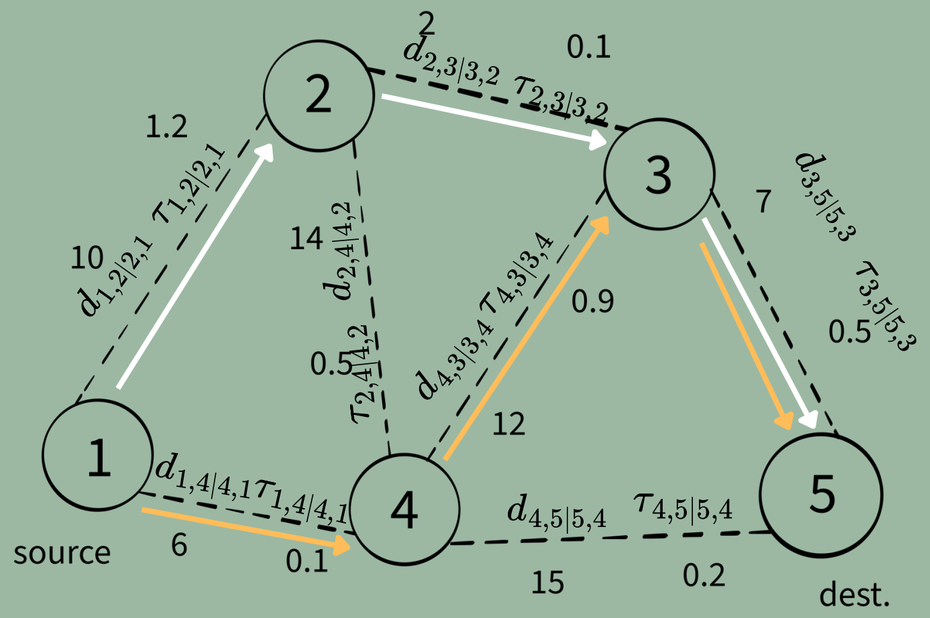
7 putting it all together

From here it's all a matter of substitution where the values of each tau and distance value is shown. The variable or constant Q here however is the only thing we need to define, which for most cases will be 1. Now assuming we are currently at iteration 1 of our optimization process we should want to update our Tau or pheromone intensity matrix T values for the second iteration which makes the variable t in our equations below 2

because ant 1 passes on edge $(1,4|4,1)$ but otherwise ant 2 in its solution our calculations would be

$\tau_{1,4|4,1}(t+1) = \tau_{1,4|4,1}(t) + \dfrac{Q}{f(x^1(t))} + 0$ and our substitutions $\tau_{1,4|4,1}(2) = 0.1 + \dfrac{1}{25} + 0$ resulting in the new tau value for edge

between nodes 1 and 4 to be 0.19. This is only for this edge that ant 1 just so happens to go through in its solution, we must do this for every edge corresponding to our pheromone intensity adjacency matrix

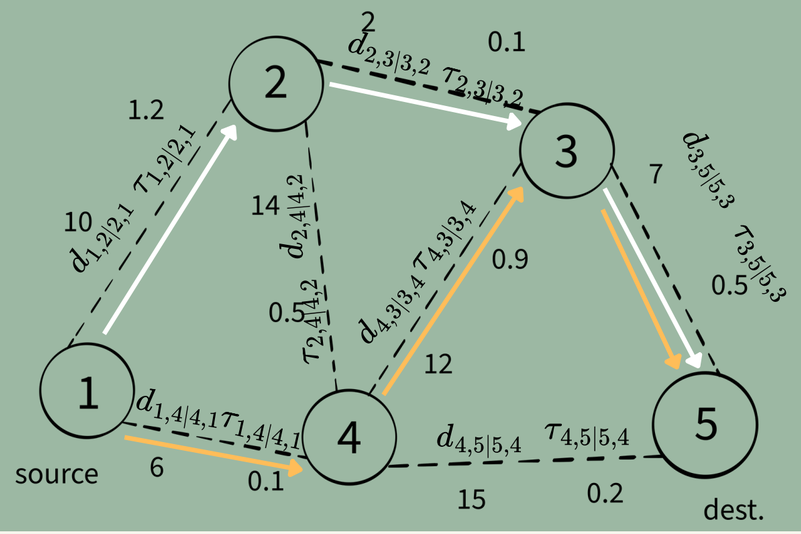


$$D = \begin{bmatrix} - & 10 & - & 6 & - \\ 10 & - & 2 & 14 & - \\ - & 2 & - & 12 & 7 \\ 6 & 14 & 12 & - & 15 \\ - & - & 7 & 15 & - \end{bmatrix} \qquad \mathrm{T} = \begin{bmatrix} - & 0.3 & - & 0.8 & - \\ 0.3 & - & 1.5 & 0.1 & - \\ - & 1.5 & - & 0.9 & 0.5 \\ 0.8 & 0.1 & 0.9 & - & 0.2 \\ - & - & 0.5 & 0.2 & - \end{bmatrix}$$

7 putting it all together

recall that $x^1(t) = \{1,4,3,5\}|\{(1,4),(4,3),(3,5)\}$ was the solution/path for ant 1, and $x^2(t) = \{1,2,3,5\}|\{(1,2),(2,3),(3,5)\}$ for ant 2 , we will base our equations entirely on these solutions. now like in the previous slide on calculating the updated pheromone value of a specific edge particularly edge (1, 4) because ant 2 passes on edge $(1,2|2,1)$ but otherwise ant 1, our calculations would be:

$\tau_{1,2|2,1}(t+1) = \tau_{1,2|2,1}(t) + 0 + \dfrac{Q}{f(x^2(t))}$ and our substitutions $\tau_{1,2|2,1}(2) = 1.2 + 0 + \dfrac{1}{19}$ which when calculated results in

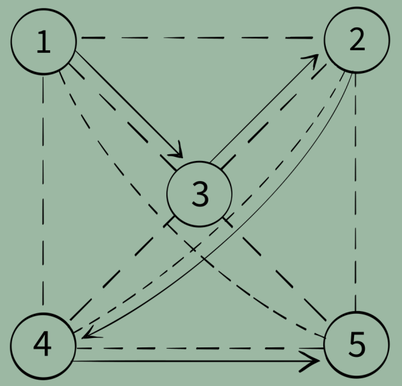


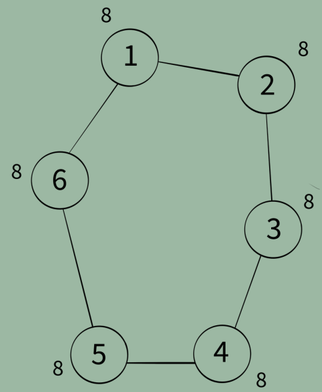
$$D = \begin{bmatrix} - & 10 & - & 6 & - \\ 10 & - & 2 & 14 & - \\ - & 2 & - & 12 & 7 \\ 6 & 14 & 12 & - & 15 \\ - & - & 7 & 15 & - \end{bmatrix} \qquad \mathrm{T} = \begin{bmatrix} - & 0.3 & - & 0.8 & - \\ 0.3 & - & 1.5 & 0.1 & - \\ - & 1.5 & - & 0.9 & 0.5 \\ 0.8 & 0.1 & 0.9 & - & 0.2 \\ - & - & 0.5 & 0.2 & - \end{bmatrix}$$

7 putting it all together

Now that we have done it for the first two edges of the solutions of both ant 1 and 2 we just do it for all the pheromone values of our pheromone intensity adjacency matrix

because ant 1 passes on edge $(4,3|3,4)$ but otherwise ant 2, our calculations would be $\tau_{4,3|3,4}(t+1) = \tau_{4,3|3,4}(t) + \frac{Q}{f(x^1(t))} + 0$ and our substitutions $\tau_{3,4|4,3}(2) = 0.9 + \frac{1}{25} + 0$. For $(3,5|5,3)$ because both ant 1 and 2 pass on this edge our calculations would be $\tau_{3,5|5,3}(t+1) = \tau_{3,5|5,3}(t) + \frac{Q}{f(x^1(t))} + \frac{Q}{f(x^2(t))}$ and our substitutions $\tau_{3,5|5,3}(2) = 0.5 + \frac{1}{25} + \frac{1}{19}$. For $(2,3|3,2)$ because ant 2 passes on this edge but otherwise ant 1, our calculations would be $\tau_{2,3|3,2}(t+1) = \tau_{2,3|3,2}(t) + 0 + \frac{Q}{f(x^2(t))}$ and our substitutions $\tau_{2,3|3,2}(2) = 0.1 + 0 + \frac{1}{19}$. Now for the last & final edge $(4,5|5,4)$ that connects the two nodes 4 and 5 in our graph because both ant 1 and 2 never pass on this edge our calculations would be $\tau_{4,5|5,4}(t+1) = \tau_{4,5|5,4}(t) + 0 + 0$ and our substitutions $\tau_{4,5|5,4}(2) = 0.2 + 0 + 0$





8 alternative transition probability formula

moving on from simple Ant colony we introduce Ant System which improves on the former simple ant colony optimization (SACO) method, which includes heuristic info to the transition probability, and includes a tabular list to the set of feasible nodes $\mathscr{N}_i^k(t)$. Below is the new equation defined as:

$$P_{i,j}^k = \begin{cases} \frac{\tau_{i,j}^\alpha(t)\eta_{i,j}^\beta(t)}{\sum_{u\in\mathscr{N}_i^k(t)}\tau_{i,u}^\alpha(t)\eta_{i,u}^\beta(t)} & \text{if } j \in \mathscr{N}_i^k(t) \\ 0 & \text{otherwise} \end{cases}$$

where: $\tau_{i,j}$ is still the pheromone value at edge (i,j), $\eta i,j$ is a priori effectiveness of the move from node i to node j, meaning attractiveness of moving to such a node. For this equation as well, the constant alpha and beta has constraints $\alpha > 0$ and $\beta > 0$

moreover $\eta i,j$ is the formula in which this new transition probability equation is able to improve the degree to which an edge or node is attractive. It is defined by $\eta_{i,j}(t) = \frac{1}{d_{i,j}(t)}$ where 1 when divided by a large distance value will result in a smaller value, maybe say a decimal value less than 1 and approximating almost 0. And when this value approximating zero is multiplied to $\tau_{i,j}^\alpha$ it results in a significantly smaller value e.g. 0.01 of $\tau_{i,j}^\alpha$ it results in a significantly smaller value e.g. 0.01 of $\tau_{i,j}^\alpha$ is essentially 1% of this pheromone value. In the priori effectiveness rational expression, $d_{i,j}$ is the cost/distance/length between node i and j

part 3

REPRESENTING AN ANT COLONY PROGRAMMATICALLY

```python
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_cs
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

from utilities.data_preprocessor import preprocess
from utilities.data_visualizer import view_train_cross

from aco_algorithm.ant_colony import Colony

# ## load and preprocess data
df = pd.read_csv('./data.csv')
X, Y = preprocess(df)

colony = Colony(X.T, Y.T, epochs=80, num_ants=20, visualize=False)
best_ants, best_ant = colony.run()

# save each each best ant at each oteration to pkl file

print(*best_ants, sep='\n\n')

# save the overall best ant to pkl file
print('best ant: \n')
print(best_ant)
```

Ant Colony Optimization (ACO) is not only limited to optimization problems but can also be applied to feature selection in machine learning. Feature selection aims to identify the most relevant features from a given dataset to improve model performance and reduce computational complexity. In the context of ACO, features are treated as the "nodes" in the graph, and ants represent the search process.

In this approach, ants construct solutions by selecting a subset of features iteratively. They deposit pheromone trails on the edges connecting the features based on their quality and relevance to the problem at hand. The pheromone trail intensity represents the attractiveness of a feature subset. Ants prefer paths with higher pheromone levels, biasing the search towards promising feature combinations.

As the algorithm progresses, ants collaborate by reinforcing pheromone trails of successful feature subsets, leading to the discovery of more informative combinations. By exploiting the collective intelligence of the ant colony, ACO effectively explores the feature space, focusing on subsets that contribute the most to the learning task.

The final result is a set of features with high pheromone levels, indicating their importance in the model. These selected features can then be used to train machine learning models, improving their learning accuracy, reduce learning time, and simplify learning results [1], [2], [3] as well as improving their generalization ability while mitigating as much as possible overfitting which is a significant problem posed always to machine learning experts and researchers

```python
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

from utilities.data_preprocessor import preprocess
from utilities.data_visualizer import view_train_cross

from aco_algorithm.ant_colony import Colony

# ## load and preprocess data
df = pd.read_csv('./data.csv')
X, Y = preprocess(df)

colony = Colony(X.T, Y.T, epochs=80, num_ants=20, visualize=False)
best_ants, best_ant = colony.run()

# save each each best ant at each oteration to pkl file

print(*best_ants, sep='\n\n')

# save the overall best ant to pkl file
print('best ant: \n')
print(best_ant)
```

1 applying ACO algorithm to a classification problem in machine learning

how ACO works in this problem is that like nodes of the ants colony the features of a dataset represent these nodes, and what it basically does is select iteratively the features that when fed to a classification algorithm in this case yields the lowest cost value, or what we have established in previous slides the solution/path of an ant that yields the shortest path, when all of its visited nodes distances are added up

1. first and foremost we will have to read our dataset which in this application uses the breast cancer dataset of 569 training examples and 30 features overall

```
df  = pd.read_csv('./data.csv')
X, Y = preprocess(df)
```

Here we set pass our dataset to the preprocessor function in which its return value will be assigned to variables X and Y. Moreover it is necessary to import in this case the pandas library in order to read the data.

```python
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

from utilities.data_preprocessor import preprocess
from utilities.data_visualizer import view_train_cross

from aco_algorithm.ant_colony import Colony

# ## load and preprocess data
df = pd.read_csv('./data.csv')
X, Y = preprocess(df)

colony = Colony(X.T, Y.T, epochs=80, num_ants=20, visualize=False)
best_ants, best_ant = colony.run()

# save each each best ant at each oteration to pkl file

print(*best_ants, sep='\n\n')

# save the overall best ant to pkl file
print('best ant: \n')
print(best_ant)
```

2 initializing the algorithm

2. as we can see we have set the epochs or the number of iterations to 80 in order to ensure that the ants converge/find the best path/solution over a relatively sufficient period of time. We have also set the number of ants in our algorithm to 20.

In this we will also call the method of the instantiated Colony class which is the whole of the ACO algorithm, called .run().

```
colony = Colony(X.T, Y.T, epochs=80, num_ants=20, visualize=False)
```

In later steps we will see, after the initialization and instantiation of this class how the .run() method works and how it implements the aforementioned equations involved in the artificial ants processes in finding the optimal path/solution in each iteration and in all iterations.

Note: the extra utility libraries like utilities.data_preprocessor, utilities.data_visualizer, and

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05,
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

3. here in the file where the Colony class is implemented, because we use the main equations to determine where each ant should go, evaporate the pheromone intensity, and update the pheromone intensity positively we place default values for the hyperparameters that these equations depend upon, in the definition of our class.

These are...

Q our constant value which we set to 1

α our alpha term

β our beta term

ρ our rho term

which are denoted in the arrows on the side and used in the following equations we have thus far established below:

$$P_{i,j}^{k} = \begin{cases} \dfrac{\tau_{i,j}^{\alpha}(t)\eta_{i,j}^{\beta}(t)}{\sum_{u \in \mathcal{N}_i^k(t)} \tau_{i,u}^{\alpha}(t)\eta_{i,u}^{\beta}(t)} & \text{if } j \in \mathcal{N}_i^k(t) \\ 0 & \text{otherwise} \end{cases}$$

where we know that again j is an element of the set of feasible nodes of the node i at iteration t denoted by $\mathcal{N}_i^k(t)$ where for example if an ant is sitting at node 14 and it has not visited the neighboring nodes for instance 2, 3, and 5 then $\mathcal{N}_i^k(t)$ would be...

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05,
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

$\mathcal{N}_4^2(t) = \{2, 3, 5\}$ where the super script 2 is the second ant in our colony finding trying to find a path, and the subscript 4 is the node where this ant is currently at and, and where we can also see that indeed its set of nodes are 2, 3, and 5.

Moreover the variable rho (ρ) and Q as we've previously established is used not in the transition probability equation we've seen thus far but used in our negative feedback or pheromone concentration evaporation formula:

$$\tau_{i,j}(t) = (1 - \rho) * \tau_{i,j}(t)$$

where rho is a value between 0 and 1 inclusively. And where Q is used, is in our positive feedback's or pheromone update formula's $\Delta\tau_{i,j}^k(t)$ term below which we will see later

$$\tau_{i,j}(t+1) = \tau_{i,j}(t) + \sum_{k=1}^{n_k} \Delta\tau_{i,j}^k(t)$$

Here $\tau_{i,j}(t)$ we know as the pheromone value in between node i and j or at edge i and j , and the summation term $\sum_{k=1}^{n_k} \Delta\tau_{i,j}^k(t)$ where the lower limit k starts at 1, since the index of the first ant is of course 1

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05,
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

and stops at the upper limit denoted as n_k in that term which we know as the number of our total ants to use which we can also just so happen to use as our stopping value. But more than that, this summation actually sums up the values $\Delta\tau_{i,j}^{k}(t)$ of each ant, which when expanded are these constraints

$$\Delta\tau_{i,j}^{k}(t) = \begin{cases} \frac{Q}{f(x^k(t))} & \text{if } edge(i,j) occurs in path for x^k(t) \\ 0 & \text{otherwise} \end{cases}$$

Now this may seem intimidating for a layman but as we recall each ant will eventually build a solution, for ant 1 its paths are let's say $x^1(t) = \{1,4,3,5\}|\{(1,4),(4,3),(3,5)\}$ and for ant 2 $x^2(t) = \{1,2,3,5\}|\{(1,2),(2,3),(3,5)\}$. And how this works is that lets say we are now summing up all the $\Delta\tau_{i,j}^{k}(t)$ terms of each ant from 1 to n_k , the current pheromone value we are trying to update at edge (i,j) or in between node i and node j must first and foremost have such an edge occur in the solution or path made by an ant, so for example if we are now trying to update pheromone concentration in between node 3 and 4 (or 4 and 3) or at edge $(4,3)$ denoted as $\tau_{4,3|3,4}$ we would check based on the given constraint if indeed this edge occurs in the path made by ants 1 and 2, and in this example because the edge does

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05,
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

occur for ant 1's constructed path then we would follow the constraint $\frac{Q}{f(x^k(t))}$ as our delta tau term $\Delta\tau_{i,j}^k(t)$ to be used in our summation where here we finally use Q our constant value which we set to 1, and as we've previously discussed the function $f()$ here in the denominator takes in the solution of ant k denoted by $x^k(t)$ and calculates the total cost or distances covered by the solution of said ant. In our previous discussions we've also set the cost of the solution of ant 1 $f(x^1(t))$ for instance to 25, now because we are using this constraint since the current edge does occur in ant 1's path, $\frac{Q}{f(x^k(t))}$ when substituted this results in $\frac{1}{25}$ This is but for ant 1's delta tau term $\Delta\tau_{i,j}^k(t)$, for ant 2's delta tau term however because the edge $(4,3)$ does not occur in its path set or solution, the constraint we will have to follow is to set the delta tau term for ant 2 to 0. Which then ultimately results in the ff. final calculations of the pheromone update equation when substituted these values:

$$\tau_{3,4|4,3}(2) = 0.9 + \frac{1}{25} + 0$$

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05,
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q
        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

hese values we will set by default to 1, 1, 0.05, and 1 espectively in the parameters of the .__init__() method of the Colony class. This will be our Colony class' default values so that we don't have to explicitly et it when we invoke the class.

2 initializing the algorithm

Where $\tau_{3,4|4,3}(2)$ again would be the new pheromone concentration value at edge $(4,3)$ or in between node i and node j for the next or in this case 2nd iteration assuming we are at the 1st iteration; 0.9 being the current pheromone intensity value at this edge denoted as $\tau_{3,4|4,3}(1)$, and finally the summation of the delta tau values being $\sum_{k=1}^{2} \Delta\tau^{k}_{3,4|4,3}(1)$ which when expanded is $\frac{1}{25} + 0$ only having two terms since we have 2 ants in this example all in all. All of this ultimately going back to our pheromone update equation earlier which was:

$$\tau_{i,j}(t+1) = \tau_{i,j}(t) + \sum_{k=1}^{n_k} \Delta\tau^{k}_{i,j}(t)$$

and when its variables substituted with the respective values results in:

$$\tau_{3,4|4,3}(2) = 0.9 + \frac{1}{25} + 0$$

This is to reiterate again all in all as alluded in previous discussions how the pheromone concentration update equation works and will work in code, and how the aforementioned constants like α β ρ and Q fit in their respective equations

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05,
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

And as mentioned these constant values or hyper parameters we will set by default to 1, 1, 0.05, and 1 respectively in the parameters of the .__init__() method of the Colony class. This will be our Colony class' default values so that we don't have to explicitly set it when we invoke the class.

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

4. we will also have to, in this implementation import the linear algebra library called numpy as np, for the instance attributes self.eta and self.tau will have to be initialized to a matrix of ones with dimensionality of the number of features of our input dataset X, which actually implements the matrix we saw earlier with the adjacency matrix of pheromone intensity values:

$$\mathrm{T} = \begin{bmatrix} 0 & \tau_{1,2} & 0 & \tau_{1,4} & 0 \\ \tau_{2,1} & 0 & \tau_{2,3} & \tau_{2,4} & 0 \\ 0 & \tau_{3,2} & 0 & \tau_{3,4} & \tau_{3,5} \\ \tau_{4,1} & \tau_{4,2} & \tau_{4,3} & 0 & \tau_{4,5} \\ 0 & 0 & \tau_{5,3} & \tau_{5,4} & 0 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} * \tau_0$$

Note that this matrix is to be initialized however we want and does not have a fixed value, this is why when we initialize the self.tau attribute the we initialize our pheromone intensity adjacency matrix to a matrix of ones and then multiplied by tau_0 which is a variable that controls for the kind of values we want our matrix to have because we know when 1 is multiplied by any value the result becomes that "any value". So if for instance our matrix of ones is multiplied by 0.2 as our tau_0 then the our matrix comes out to a matrix of 0.2's

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

Because we use now a much typical approach for calculating our transition probability $P_{i,j}^k$ namely the ant system equation:

$$P_{i,j}^k = \begin{cases} \frac{\tau_{i,j}^{\alpha}(t)\eta_{i,j}^{\beta}(t)}{\sum_{u \in \mathcal{N}_i^k(t)} \tau_{i,u}^{\alpha}(t)\eta_{i,u}^{\beta}(t)} & \text{if } j \in \mathcal{N}_i^k(t) \\ 0 & \text{otherwise} \end{cases}$$

instead of the simply ant colony equation we have thus far used in previous discussions, we would have now to define now a matrix called eta since we know this equation uses a variable $\eta i,j$ which measures the attractiveness of a feasible node as mentioned earlier we will have to define a matrix of ones again since this variable like the tau matrix is taken into account when calculating the probability value of the ant transitioning to a certain feasible node it has. So all in all, in all a graphs nodes and not just a single node i or j, η (eta) is a matrix of ones

```
self.eta = np.ones((X.shape[0], X.shape[0]))

self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))
```

```python
class Colony:
    def __init__(self, X, Y, epochs=15, num_sampled_features=15, num_ants=3, Q=1, tau_0=1, alpha=1, beta=1, rho=0.05, visualize=Tru
        # X must be a 1024 x 100 matrix and Y must be 1 x 100 matrix
        self.X = X
        self.Y = Y

        # 1024 features
        self.num_features = X.shape[0]

        # 100 instances
        self.num_instances = X.shape[1]

        # desired number of selected feaures
        self.num_sampled_features = num_sampled_features

        # ACO algorithm hyper parameters
        self.epochs = epochs
        self.num_ants = num_ants
        self.Q = Q

        # initial intensity of pheromone values in pheromone matrix 'tau'
        self.tau_0 = tau_0
        self.alpha = alpha
        self.beta = beta
        self.rho = rho

        # initialize heuristic info matrix to be 1024 x 1024
        self.eta = np.ones((X.shape[0], X.shape[0]))

        # init pheromone matrix to be 1024 x 1024
        # multiplied by initialized tau_0 value
        self.tau = tau_0 * np.ones((X.shape[0], X.shape[0]))

        # list to hold best cost values out of all ants in each iteration
        # e.g. ant 1 out of all ants holds best path/cost of iteration/epoch 1
        self.best_ants = []

        self.ants = np.empty(shape=(num_ants, 1), dtype=np.dtype(Ant))

        # initially best ants cost is an infinite value
        self.best_ant = Ant()
        self.visualize = visualize
```

2 initializing the algorithm

we also initialize self.best_ants attribute to an empty list, self.ants to an empty numpy array with the dimension of the nubmer of ants by 1 and of type Ant which we will see later, self.best_ant to an instance of the Ant class which we will again see in later steps to represent a single artificial ant

```python
class Ant:
    def __init__(self):
        self._tour = []
        self._cost = np.inf
        self._output = np.inf

    def __str__(self):
        return f"""
        tours: {self.tour}\n
        length: {len(self.tour)}\n

        costs: {self.cost}\n
        length: {len(self.tour)}\n

        outputs: {self.output}\n
        length: {len(self.tour)}\n
        """

    @property
    def tour(self):
        return self._tour

    def append_tour(self, val):
        self._tour.append(val)

    @property
    def cost(self):
        return self._cost

    @cost.setter
    def cost(self, val):
        self._cost = val

    @property
    def output(self):
        return self._output

    @output.setter
    def output(self, val):
        self._output = val
```

3 representing an Ant programmatically

5. Our representation of an Ant consists of the instance attributes those being, self._tour, self._cost, and self._output which are in this case privatized so to speak and use getter and setter functions as implemented to maintain good practice, in the figure on the right.

But more important than these good practices of OOP, is what these attributes entail. Most important of all in the least is the self._tour and the self._cost attributes which as shown in previous parts will represent the list of all nodes a certain ant has traversed in order to create a solution/path that may either be optimal or not $x^1(t) = \{1, 4, 3, 5\}$

And the self._cost attribute being the representation of the quality of the solution/path made by said ant $f(x^k(t))$. Which in the context of machine learning will be the total cost incurred by certain ants solution/path or in this case selected features when fed to a machine learning algorithm which we will see later in the use of a standard Artificial Neural Network

setter method .append_tour is a method that will build the list of nodes or in this case the features which each ant will select

the setter method .cost is another method that will be invoked multiple times in the algorithm to assign the cost incurred by an ants made solution/path

```python
def run(self):
    # loop from 0 to 14
    for epoch in range(self.epochs + 1):
        print(f'epoch {epoch} starting\n')

        # loop from 0 to 2
        for k in range(self.num_ants):

            # instantiate an Ant object
            temp_ant = Ant()

            # since we have 1024 features for ex, generate a random
            # number from 0 to 1023 inclusively, 1024 is excluded
            temp_tour = np.random.randint(0, self.num_features)
            temp_ant.append_tour(temp_tour)

            self.ants[k, 0] = temp_ant

            # loop from [1] to [1023], instead of [0] to [1023], but stop at 1024
            for l in range(1, self.num_features):

                # since we are accessing last element of tour
                # attribute of ant make sure, .tour is never
                # empty or statemetn will raise error
                i = self.ants[k, 0].tour[-1]

                # P when calculated is a 1 x 1024 row vector
                # or will always be a 1 x num_features row vector
                P = np.power(self.tau[i, :], self.alpha) * np.power(self.eta[i, :], self.beta)


                # sets the visited spots of the ants in the P matrix to 0
                # e.g. [1000] accesses P[[1000]], or element at 1000th index
                # [1000, 241] accesses elements at 1000th and 241st index and
                # sets them to 0
                P[self.ants[k, 0].tour] = 0

                # sum all elements in P row vector and use as denominator
                P = P / np.sum(P)

                j = self.roulette(P)
                self.ants[k, 0].append_tour(j)
```

4 implementing and running the transition probability

7. And for the primary component of the algorithm here we implement the equation used in calculating the transition probability of ant going through each node or feature in the dataset. And subsequently the accumulated transition probability. Denoted by the equation below as well as its equivalent code statements

$$P_{i,j}^{k} = \begin{cases} \dfrac{\tau_{i,j}^{\alpha}(t)\eta_{i,j}^{\beta}(t)}{\sum_{u \in \mathcal{N}_i^k(t)} \tau_{i,u}^{\alpha}(t)\eta_{i,u}^{\beta}(t)} & \text{if } j \in \mathcal{N}_i^k(t) \\ 0 & \text{otherwise} \end{cases}$$

The statements below serve actually as the numerator of the transition probability for the first condition

```
i = self.ants[k, 0].tour[-1]
P = np.power(self.tau[i, :], self.alpha) * np.power(self.eta[i, :]), self.beta)
```

The statements below however serve as the denominator part of the transition probability's equation for the first condition

```
P[self.ants[k, 0].tour] = 0
P = P / np.sum(P)
```

```python
def roulette(self, P):
    """P - is the transition probability vector with dimensionality 1 x num_features
    or in this case 1 x 1024 if number of features is 1024
    """
    # generate random float between (0, 1) exclusively
    r_num = np.random.uniform()

    # since P is a 1 x num_features matrix
    # np.cumsum(P) will be same shape as P
    p_cum_sum = np.cumsum(P)

    bools = (r_num <= p_cum_sum).astype(int)

    # return the index of the first occurence of
    # a true/1 value in the bools array
    return np.where(bools == 1)[0][0]
```

4 implementing and running the transition probability

8. And finally because calculating the transition probability does not end there for the algorithm, we need to calculate also the accumulated transition probability, which actually is invoked through the use of a helper function self.roulette which calculates the accumulated transition probability given the calculated transition probability P

```
j = self.roulette(P)
self.ants[k, 0].append_tour(j)
```

This then finally produces a value j which we append through the use of our setter function in our instantiated ant object self.ants[k, 0]. This value j is actually the ants chosen node which lets us know the our implementation works.

In the diagram on the right-hand side we see that this method (part also of our Colony class) calculates the accumulated transition probability given the transition probability P we calculated earlier.

```
        # calculate cost given the paths made by the and
        cost, output = self.J(epoch, k, self.ants[k, 0].tour, self.num_sampled_features, {
            'X': self.X,
            'Y': self.Y,
            'num_features': self.num_features,
            'num_instances': self.num_instances
        })

        self.ants[k, 0].cost = cost
        self.ants[k, 0].output = output

        # use current cost of ant k at iteration i and compare
        # to current best ant cost, then continually update the best ant
        if self.ants[k, 0].cost < self.best_ant.cost:
            self.best_ant = self.ants[k, 0]

    # updating pheromones for positive feedback
    for k in range(self.num_ants):
        # append the first node to the whole path made by ant
        tour = np.append(self.ants[k, 0].tour, self.ants[k, 0].tour[0])

        # go through now all features from index [0] to [1023]
        for l in range(self.num_features):
            i = tour[l]
            j = tour[l + 1]
            self.tau[i, j] = self.tau[i, j] + self.Q / self.ants[k, 0].cost

    # updating evaporation rate for negative feedback
    self.tau = (1 - self.rho) * self.tau

    # store all ants at each iteration with the best cost
    self.best_ants.append(self.best_ant)

    if epoch % 10 == 0:
        print(f'epoch {epoch} finished\n')

# return the best ant and best ants which
# contain the paths and their sampled paths
return [self.best_ants, self.best_ant]
```

4 implementing and running the transition probability

9. At the end of constructing our paths by a certain ant k, we will have to calculate the quality of the solution/path made by not just a certain ant, but all ants in each iteration and to keep the best ants in each iteration, and from all these best ants pick the ant with the greatest quality of solutions.

To calculate $f(x^k(t))$ we will need a function/method to measure the quality of the solution/path made by an ant. In our case because it is applied to an ML problem what we can do instead of measuring the length of the path made by an ant, we calculate the cost value to be incurred by the features selected/constructed by the ant, which is done through the helper method self.J()

Note: The method to calculate the cost will not be covered in this article since it is out of teh scope of the ACO algorithm, however to have an idea on what cost function this method uses, this uses the binary cross entropy loss/cost function which is commonly used in binary classification problems like the dataset we have thus far used. Some useful articles about this can be visited here: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

```
        # calculate cost given the paths made by the and
        cost, output = self.J(epoch, k, self.ants[k, 0].tour, self.num_sampled_features, {
            'X': self.X,
            'Y': self.Y,
            'num_features': self.num_features,
            'num_instances': self.num_instances
        })

        self.ants[k, 0].cost = cost
        self.ants[k, 0].output = output

        # use current cost of ant k at iteration i and compare
        # to current best ant cost, then continually update the best ant
        if self.ants[k, 0].cost < self.best_ant.cost:
            self.best_ant = self.ants[k, 0]

    # updating pheromones for positive feedback
    for k in range(self.num_ants):
        # append the first node to the whole path made by ant
        tour = np.append(self.ants[k, 0].tour, self.ants[k, 0].tour[0])

        # go through now all features from index [0] to [1023]
        for l in range(self.num_features):
            i = tour[l]
            j = tour[l + 1]
            self.tau[i, j] = self.tau[i, j] + self.Q / self.ants[k, 0].cost

    # updating evaporation rate for negative feedback
    self.tau = (1 - self.rho) * self.tau

    # store all ants at each iteration with the best cost
    self.best_ants.append(self.best_ant)

    if epoch % 10 == 0:
        print(f'epoch {epoch} finished\n')

# return the best ant and best ants which
# contain the paths and their sampled paths
return [self.best_ants, self.best_ant]
```

5 positive feedback phase

10. As we've previously established the positive feedback mechanism is defined by the equations below and expressed in code on the side denoted by the arrow:

$$\tau_{i,j}(t+1) = \tau_{i,j}(t) + \sum_{k=1}^{n_k} \Delta\tau_{i,j}^{k}(t)$$

where:

$$\Delta\tau_{i,j}^{k}(t) = \begin{cases} \frac{Q}{f(x^k(t))} & \text{if } edge(i,j) occurs in path for x^k(t) \\ 0 & \text{otherwise} \end{cases}$$

in its implementation programmatically we see that we again have to loop over all the ants that have constructed their respective solutions/paths and use the quality of the solution we have calculated in the previous phase to update our pheromone intensity adjacency matrix.

```
        # calculate cost given the paths made by the and
        cost, output = self.J(epoch, k, self.ants[k, 0].tour, self.num_sampled_features, {
            'X': self.X,
            'Y': self.Y,
            'num_features': self.num_features,
            'num_instances': self.num_instances
        })

        self.ants[k, 0].cost = cost
        self.ants[k, 0].output = output

        # use current cost of ant k at iteration i and compare
        # to current best ant cost, then continually update the best ant
        if self.ants[k, 0].cost < self.best_ant.cost:
            self.best_ant = self.ants[k, 0]

    # updating pheromones for positive feedback
    for k in range(self.num_ants):
        # append the first node to the whole path made by ant
        tour = np.append(self.ants[k, 0].tour, self.ants[k, 0].tour[0])

        # go through now all features from index [0] to [1023]
        for l in range(self.num_features):
            i = tour[l]
            j = tour[l + 1]
            self.tau[i, j] = self.tau[i, j] + self.Q / self.ants[k, 0].cost

    # updating evaporation rate for negative feedback
    self.tau = (1 - self.rho) * self.tau

    # store all ants at each iteration with the best cost
    self.best_ants.append(self.best_ant)

    if epoch % 10 == 0:
        print(f'epoch {epoch} finished\n')

# return the best ant and best ants which
# contain the paths and their sampled paths
return [self.best_ants, self.best_ant]
```

5 positive feedback phase

In particular the code block we have to implement is the following which follows the equation we have established earlier that updates our pheromone adjacency matrix

```
# updating pheromones for positive feedback
      for k in range(self.num_ants):
          # append the first node to the whole
path made by ant
         tour = np.append(self.ants[k, 0].tour,
self.ants[k, 0].tour[0])

           # go through now all features from
index [0] to [1023]
        for l in range(self.num_features):
          i = tour[l]
          j = tour[l + 1]
             self.tau[i, j] = self.tau[i, j] +
self.Q / self.ants[k, 0].cost
```

This in turn will serve as positive reinforcement for ants with the best solutions since ants with lower costs in this case would (equivalent to a shorter path) will have their constructed solution/path be followed by other ants

```
        # calculate cost given the paths made by the and
        cost, output = self.J(epoch, k, self.ants[k, 0].tour, self.num_sampled_features, {
            'X': self.X,
            'Y': self.Y,
            'num_features': self.num_features,
            'num_instances': self.num_instances
        })

        self.ants[k, 0].cost = cost
        self.ants[k, 0].output = output

        # use current cost of ant k at iteration i and compare
        # to current best ant cost, then continually update the best ant
        if self.ants[k, 0].cost < self.best_ant.cost:
            self.best_ant = self.ants[k, 0]

    # updating pheromones for positive feedback
    for k in range(self.num_ants):
        # append the first node to the whole path made by ant
        tour = np.append(self.ants[k, 0].tour, self.ants[k, 0].tour[0])

        # go through now all features from index [0] to [1023]
        for l in range(self.num_features):
            i = tour[l]
            j = tour[l + 1]
            self.tau[i, j] = self.tau[i, j] + self.Q / self.ants[k, 0].cost

    # updating evaporation rate for negative feedback
    self.tau = (1 - self.rho) * self.tau

    # store all ants at each iteration with the best cost
    self.best_ants.append(self.best_ant)

    if epoch % 10 == 0:
        print(f'epoch {epoch} finished\n')

# return the best ant and best ants which
# contain the paths and their sampled paths
return [self.best_ants, self.best_ant]
```

5 negative feedback phase

11. For the final phase of a single iteration of ants constructing a path, negative feedback or the reinforcement of ants in subsequent iterations to be pushed to avoid some certain paths that are not optimal by further reducing the pheromone intensity of these less optimal paths, is implemented here.

```
self.tau = (1 - self.rho) * self.tau
```

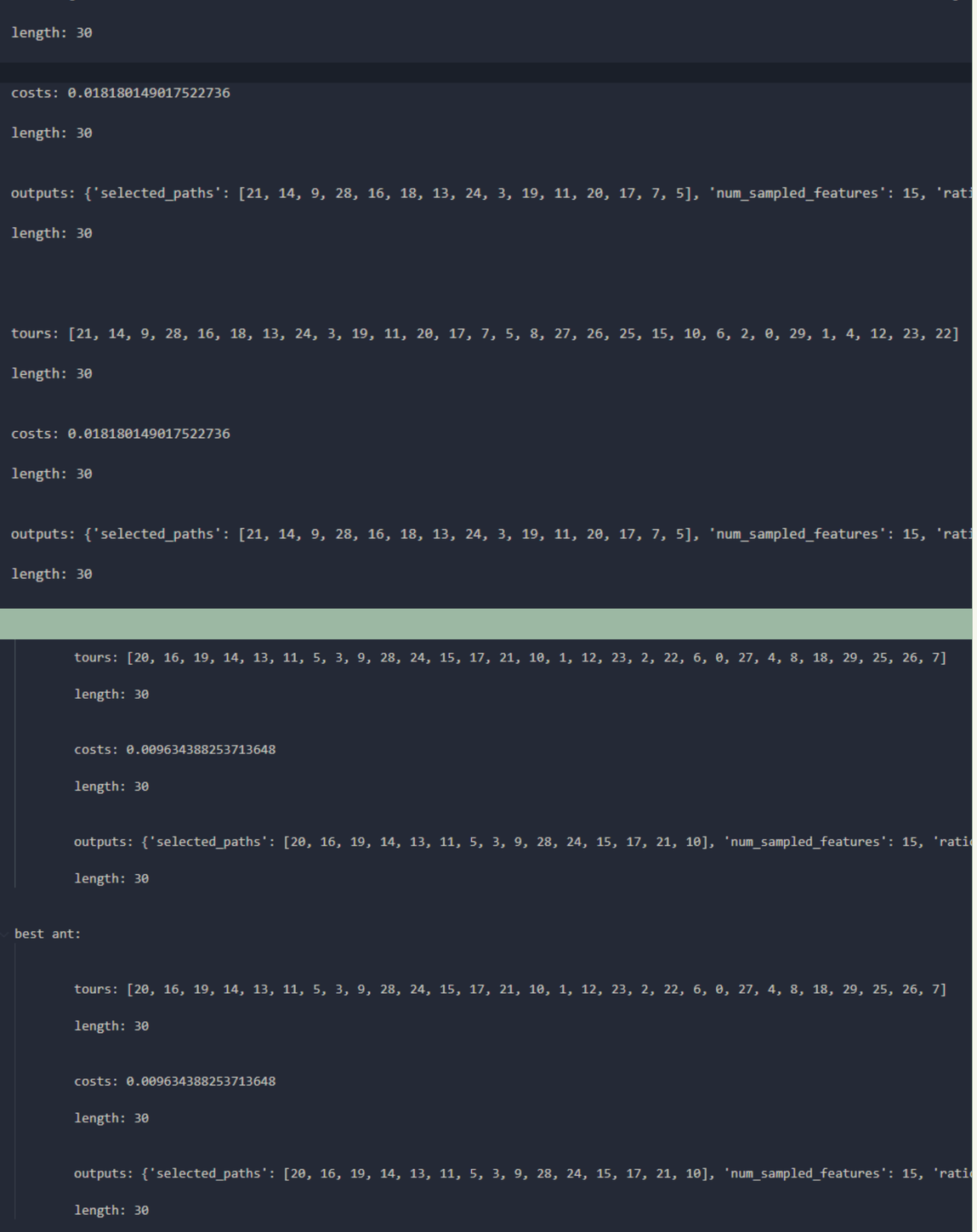
The statement above is a translation to the the equation we have discussed in earlier steps which is $\tau_{i,j}(t) = (1 - \rho) * \tau_{i,j}(t)$. However instead of individually reducing these pheromone values as indicated in this equation, we use the power of vectorization which in this case we use our pheromone intensity adjacency matrix and multiply it by a scalar value which is the difference of 1 and the evaporation rate (rho value). This in turn makes our computations more efficient and faster.

```
        # calculate cost given the paths made by the and
        cost, output = self.J(epoch, k, self.ants[k, 0].tour, self.num_sampled_features, {
            'X': self.X,
            'Y': self.Y,
            'num_features': self.num_features,
            'num_instances': self.num_instances
        })

        self.ants[k, 0].cost = cost
        self.ants[k, 0].output = output

        # use current cost of ant k at iteration i and compare
        # to current best ant cost, then continually update the best ant
        if self.ants[k, 0].cost < self.best_ant.cost:
            self.best_ant = self.ants[k, 0]

    # updating pheromones for positive feedback
    for k in range(self.num_ants):
        # append the first node to the whole path made by ant
        tour = np.append(self.ants[k, 0].tour, self.ants[k, 0].tour[0])

        # go through now all features from index [0] to [1023]
        for l in range(self.num_features):
            i = tour[l]
            j = tour[l + 1]
            self.tau[i, j] = self.tau[i, j] + self.Q / self.ants[k, 0].cost

    # updating evaporation rate for negative feedback
    self.tau = (1 - self.rho) * self.tau

    # store all ants at each iteration with the best cost
    self.best_ants.append(self.best_ant)

    if epoch % 10 == 0:
        print(f'epoch {epoch} finished\n')

# return the best ant and best ants which
# contain the paths and their sampled paths
return [self.best_ants, self.best_ant]
```

5 negative feedback phase

This is the whole of the algorithm which again involves the construction of a path through the calculation of the accumulated transition probabilities, then updating the pheromone intensity/concentration adjacency matrix by both decreasing all the pheromone intensity values but also increasing the pheromone intensity values of some edges, which when part of a solution yields the best quality or a lower cost.

When the algorithm finishes ultimately, this method will return all the best ants and the out of all these best ants the best ant out of all, which along with them comes their respective solutions/paths they have constructed which in this case are the feature indeces of our dataset, accessible through the Ant class' self.tour getter method.



6 obtaining the best ants and the best ant out of all iterations

Upon returning from the method self.run() of the our colony object which was the statements:

```
colony = Colony(X.T, Y.T, epochs=80, num_ants=20, visualize=False)
best_ants, best_ant = colony.run()
```

we obtain the best_ants of each iteration as well as the best_ant out of all these best ants in each iteration. As you'd guess this method returns all the best ants in each iteration and the overall best ant in all iterations and will be assigned to variables best_ants and best_ant.

In the top-left most diagram we see the best ant at iteration 1 and after it the best ant at iteration 2 and so forth.

And in the bottom-left most diagram we see the output of the print statement of the ant_colony.py file upon reaching the end of runtime of this script, which shows us the best_ant out of all best_ants in each iteration which has selected the feature indeces 20, 16, 19, 14, 13, 11, 5, 3, 9, 28, 24, 15, 17, 21, and 10.

part 4

TRAINING A BINARY CLASSIFIER WITH THE CONSTRUCTED SOLUTION

```
# use path below if in local machine
df = pd.read_csv('./data.csv')

# use path below if in google collab
# df = pd.read_csv('./sample_data/breast_cancer_data.csv')


X, Y = preprocess(df)
X_trains_orig, X_, Y_trains_orig, Y_ = train_test_split(X, Y, test_size=0.3, random_state=0)
X_cross_orig, X_tests_orig, Y_cross_orig, Y_tests_orig = train_test_split(X_, Y_, test_size=0.5, random_state=0)
# view_train_cross(X_trains_orig, X_cross_orig, Y_trains_orig, Y_cross_orig)
```

- [20, 16, 19, 14, 13, 11, 5, 3, 9, 28, 24, 15, 17, 21, 10] is the path of the best ant so use these feature indeces in loading the dat
- this dataset is the one with carefully selected features

```
features = df.columns[[20, 16, 19, 14, 13, 11, 5, 3, 9, 28, 24, 15, 17, 21, 10]]
features

```

```
Index(['symmetry_se', 'smoothness_se', 'concave points_se', 'perimeter_se',
       'texture_se', 'fractal_dimension_mean', 'area_mean', 'texture_mean',
       'concave points_mean', 'concavity_worst', 'perimeter_worst', 'area_se',
       'compactness_se', 'fractal_dimension_se', 'symmetry_mean'],
      dtype='object')
```

```
X_reduced, Y_reduced = preprocess(df, feat_idxs=features)
X_trains_reduced, X_, Y_trains_reduced, Y_ = train_test_split(X_reduced, Y_reduced, test_size=0.3, random_state=0)
X_cross_reduced, X_tests_reduced, Y_cross_reduced, Y_tests_reduced = train_test_split(X_, Y_, test_size=0.5, random_s
# view_train_cross(X_trains_reduced, X_cross_reduced, Y_trains_reduced, Y_cross_reduced)
```

Using the constructed features of the best_ant which was feature indeces 20, 16, 19, 14, 13, 11, 5, 3, 9, 28, 24, 15, 17, 21, and 10, in the breast cancer dataset these correspond to the features symmetry_se, smoothness_se, concave points_se, perimeter_se, texture_se, fractal_dimension_mean, area_mean, texture_mean, concave points_mean, concavity_worst, perimeter_worst, area_se, compactness_se, fractal_dimension_se, and symmetry_mean.

And from here we use a simple artificial neural network to train a binary classifier, since the dataset only has 2 classes of outputs. Moreover we will compare the results of two trained binary classifiers, one that uses the original dataset and the other only using the selected features by the best_ant

baseline model training and validation

- train the baseline model on both original dataset and reduced dataset

```
# import then load baseline model architecture
baseline_model_orig = load_baseline()
baseline_model_red = load_baseline()

# begin model training
baseline_history_orig = baseline_model_orig.fit(
    X_trains_orig, Y_trains_orig,
    epochs=100,
    validation_data=(X_cross_orig, Y_cross_orig),
    callbacks=[EarlyStopping(monitor='val_binary_crossentropy', patience=10)]
)

baseline_history_red = baseline_model_red.fit(
    X_trains_reduced, Y_trains_reduced,
    epochs=100,
    validation_data=(X_cross_reduced, Y_cross_reduced),
    callbacks=[EarlyStopping(monitor='val_binary_crossentropy', patience=10)]
)

# build the dictionary of results based on metric history of both models
baseline_results_orig = {}
baseline_results_red = {}
for metric in ['loss', 'binary_crossentropy', 'binary_accuracy', 'val_loss', 'val_binary_crossentropy', 'val_binary_accuracy']:
    if metric not in baseline_results_orig:
        baseline_results_orig[metric] = baseline_history_orig.history[metric]
    if metric not in baseline_results_red:
        baseline_results_red[metric] = baseline_history_red.history[metric]
```

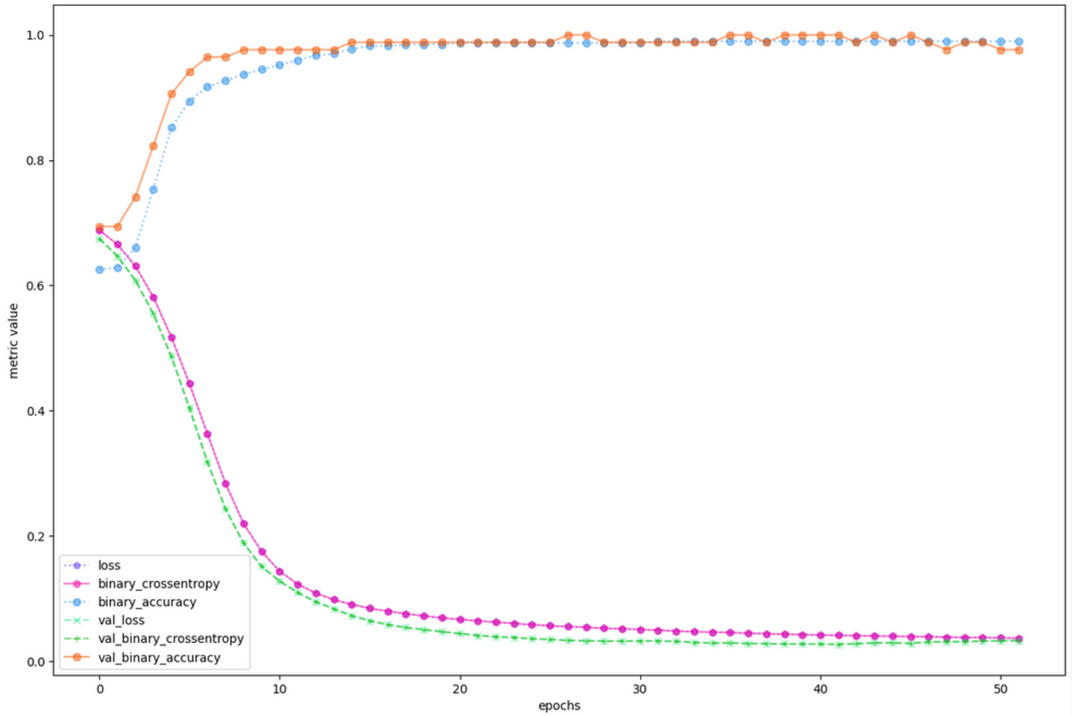
1 training both classifiers

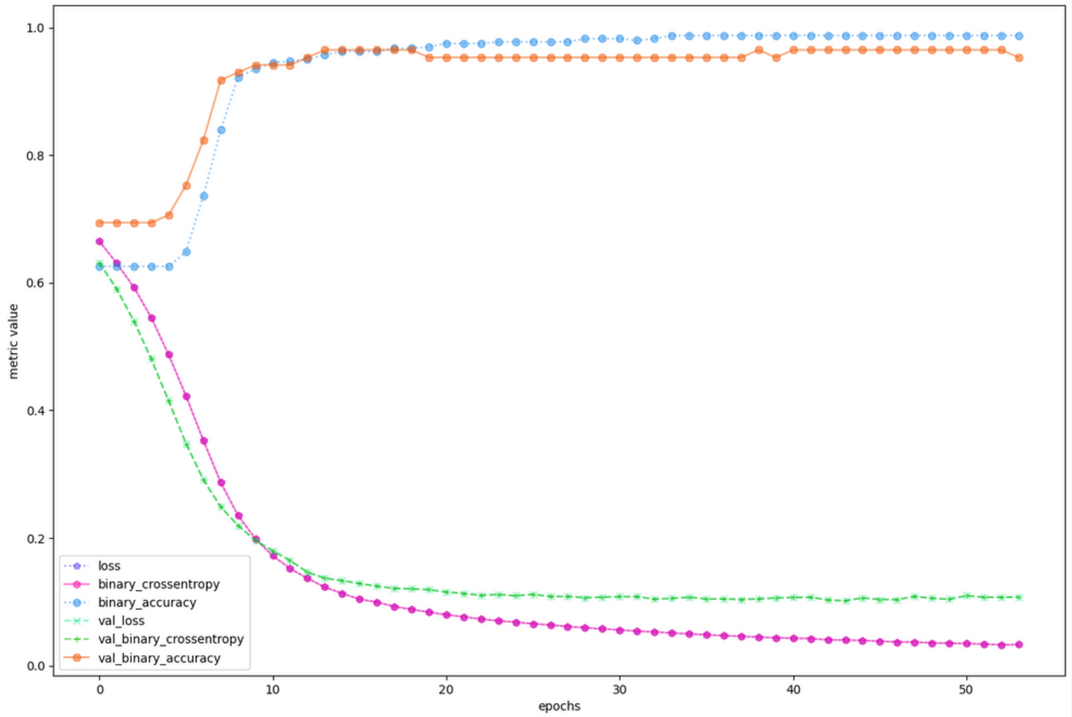
previously we saved the feature names in the variable features, and then used this variable as query to the pandas DataFrame object to select only the features the best_ant has selected.

we then pass it to an instantiated simple neural network model object which are the ff. statements:

```
baseline_model_orig = load_baseline()
baseline_model_red = load_baseline()
```

the former instantiated object being the model to use the original dataset and the latter to use the reduced dataset. After this we simply call the .fit() method (both being passed the validation data as well as the training data) of the neural network models to train both models and subsequently validate them as well to gauge the performance of both models through its loss/cost and accuracy values





2 analyzing results

Having trained both neural networks along with validating it using the validation dataset, we see that we have similar results.

In the top-left most diagram which is the neural network that trained on the original dataset we see that its loss/cost values as expected go down approximating a value of zero, which in both the training and validation set the model generalizes easily. How can we prove this? By easily looking at our accuracy in both training and validation which seems to peek at around 98% to 97% respectively, which is a good sign that the model has generalized well to unseen data

in the bottom-left most diagram which is the neural network that trained on the reduced dataset we see that its loss/cost values as expected still go down approximating a value of zero however on the validation the loss/cost value seems to flatten at around 0.1. Still we can see that this model generalizes well because when we take a look at both the training and validation accuracy we see that the values are around 98% and 95% respectively which is still a good number albeit not entirely at the same level as using all the features of the dataset as the previous model did.

```
BASELINE USING ORIGINAL DATASET RESULTS:
loss: 0.0364415012300014 5
binary_crossentropy: 0.0364415012300014 5
binary_accuracy: 0.9899497628211975
val_loss: 0.032482240349054 34
val_binary_crossentropy: 0.032482240349054 34
val_binary_accuracy: 0.976470589637756 3

BASELINE USING REDUCED DATASET RESULTS:
loss: 0.032802432775497437
binary_crossentropy: 0.032802432775497437
binary_accuracy: 0.9874371886253357
val_loss: 0.107366278767585 75
val_binary_crossentropy: 0.107366278767585 75
val_binary_accuracy: 0.952941179275512 7
```

3 conclusions

Albeit the baseline model that used the original dataset performed better by just a small margin to that of the baseline model that used the reduced dataset, both achieved relatively good results still and can be inferred that even with fewer features the model can perform reasonably well and reduce computational complexity because feature selection provides an effective way to solve the problem of higher dimensional data by removing irrelevant and redundant data, which can reduce computation time, improve learning accuracy, and facilitate a better understanding for the learning model or data [4].

Thus the first and foremost question we had of whether using ACO can positively or negatively affect the performance of an ML model like an artificial neural network is answered, this being that indeed ACO can help in terms of reducing computational complexity by reducing redundant features while still achieving relatively good accuracy metrics as compared to a dataset with all its features intact.

```
BASELINE USING ORIGINAL DATASET RESULTS:
loss: 0.03644150123000145
binary_crossentropy: 0.03644150123000145
binary_accuracy: 0.9899497628211975
val_loss: 0.032482240349054 34
val_binary_crossentropy: 0.032482240349054 34
val_binary_accuracy: 0.976470589637756 3

BASELINE USING REDUCED DATASET RESULTS:
loss: 0.032802432775497437
binary_crossentropy: 0.032802432775497437
binary_accuracy: 0.9874371886253357
val_loss: 0.107366278767585 75
val_binary_crossentropy: 0.107366278767585 75
val_binary_accuracy: 0.952941179275512 7
```

3 conclusions

Although 30 features may not be much of a contributing factor to the efficiency of the model now, as our number of features grow however it will be important to scale them to a certain degree where important features are only selected. In the case of image processing a 100 x 100 pixel image when preprocessed will have 10000 features and obviously this may indeed affect the performance of the model and cause overfitting therefore it is reasonable to conclude that using feature selecting techniques whether manually or by the use of algorithmic paradigms such as that of the ACO algorithm may indeed greatly affect the performance, efficiency, and computational time of our machine learning model.

References

1. A. Goltsev et al. Investigation of efficient features for image recognition by neural networks. Neural Netw. (2012)
2. E. Rashedi et al. A simultaneous feature adaptation and feature selection method for content-based image retrieval systems. Knowl.-Based Syst. (2013)
3. F. Amiri et al. Mutual information-based feature selection for intrusion detection systems. J. Netw. Comput. Appl. (2011)
4. C. Jie et al. Feature selection in machine learning: A new perspective. Neurocomputing, Volume 300. (2018)

Full code and presentation is in the Github repository at https://github.com/08Aristodemus24/breast-cancer-classifier