Nevergrad Ant colony Algorithm

Comparison and implementation of ant colony algorithm

Jiaqian MA	Vincent Vilmetti	Sauraj Verma	Sauvik Chatterjee	Ying Ding
Master in DSBA	Master in DSBA	Master in DSBA	Master in DSBA	Master in DSBA
ESSEC & Centrale supélec	ESSEC & Centrale supélec	ESSEC & Centrale supélec	ESSEC & Centrale supélec	ESSEC & Centrale supélec

ABSTRACT

Ant colony optimization (ACO), created by Marco Dorigo (Colorni et al. 1991), is a discrete optimization algorithm that is inspired by the food foraging behavior of real ants. The goal, much like how ants operate, by finding the shortest path to the food source. When ants move, ants will leave a chemical pheromone trail on the ground as a path to guide other ants on the route of traversal. Ants tend to choose the paths marked by the strongest pheromone concentration. The indirect communication of the ants is to pheromone trails in order to enable them to find shortest paths between their nest and food. The ACO algorithm is an essential system based on agents that simulates the natural cooperation and adaptation behavior of ants. The ACO simulates the method of real ants to rapidly establish the shortest route from a food source to their nest (Colorni et al. 1991).

The idea of the ACO is to model the problem that is being solved, just as the search for a minimum cost path in a graph that uses artificial ants to search for good paths. The ACO consists of a number of cycles (iterations) of solution construction. A number of ants construct complete solutions by using heuristic information and the collected experiences of previous groups of ants in each iteration. These collected experiences are represented by using the pheromone trail which is deposited on the constituent elements of a solution (Colorni et al. 1991). Small quantities are deposited during the construction phase, while larger amounts are deposited at the end of each iteration in proportion to solution quality.

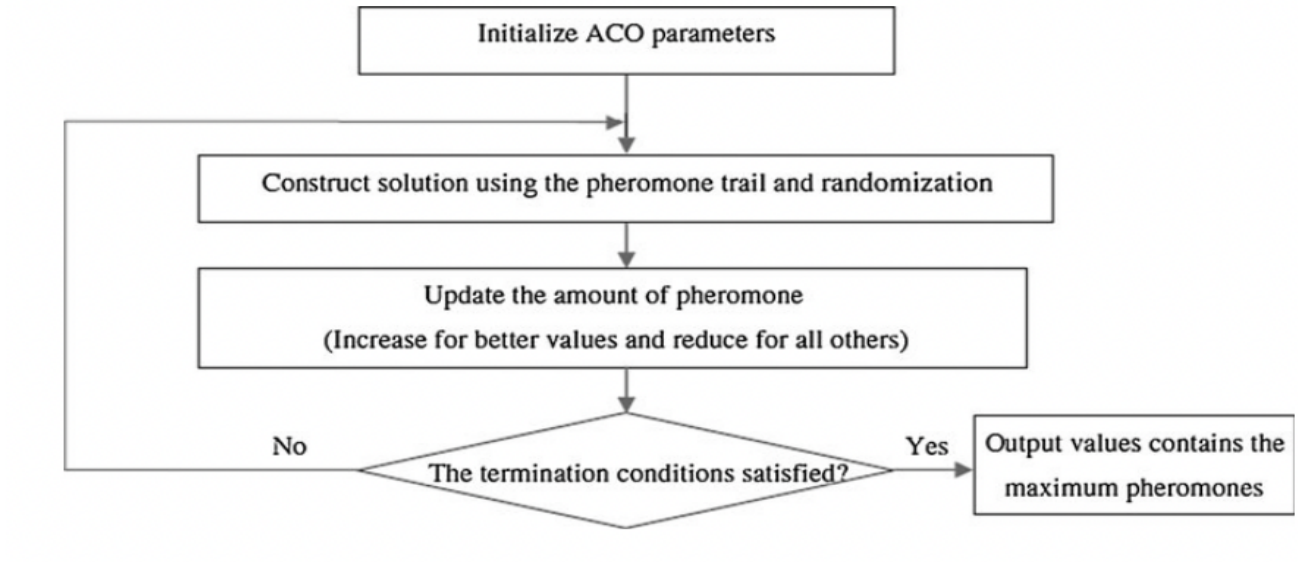

Figure 1: High level overview of the Ant Colony Optimization algorithm.

INTRODUCTION

Nevergrad is an open-source python library which helps to improve the optimization process. It offers a range of algorithms that do not use gradients to optimize the objective function and organize them into a python framework, including differential evolution and FastGA. It has been widely used to solve machine learning problems like Multimodal problems and ill-conditioned problems. The facebook AI Research group has adopted the algorithm in a variety of projects in reinforcement learning and others.

In the project, the issue #616 is related to the implementation of the ant colony algorithm. Ant colony algorithm (ACO) is one of the gradient-free optimization methods based on the foraging behaviour of ants. It adopts probability models that follow a Markovian process and can be used to solve computational problems to find the optimal path within the graph problem. The algorithm will start with a random wander of the ants. Once an ant finds a source of food, it will travel back to the colony and leave the pheromones which signal the food source. Thus, other ants will follow the designated path with a probability factor. Next, the ants will populate the path and leave their own pheromones as they bring back the food. The shortest paths are more likely to come stronger as ants leave pheromones every time they find and bring the food. This algorithm can work in a very dynamic environment and especially work well with the graphs. The typical optimization problems tested are the travelling salesman problem (TSP) and the Knapsack problem.

This issue mentioned above has been picked to make contributions, through the implementation and evaluation of the several ant colony algorithms. As contributors, we have sent the message to ask for the permission on the assignment and have performed thorough research about the current ant colony algorithm implementation. It is noticeable that there are various implementations of the algorithm. Therefore, we have implemented a wide range of ant colony algorithms identified to find the most efficient algorithm and make a further improvement.

Below are the findings and summary related to the ant colony issue in the Nevergrad open-source project. The tests are run based on the TSP and Knapsack datasets.

DATASETS DESCRIPTION

Library used: tsplib95, numpy, pandas, time

To test on the functionality of different algorithms, two types of datasets have been chosen, which are Travelling salesman problem and the knapsack problem. it is noticeable that the ant colony algorithm will vary between TSP and Knapsack datasets. Based on the difference, we have selected datasets as below.

1) TSP datasets:

Library: TSPLIB - A Traveling Salesman Problem Library,

- ATT48 is a set of 48 cities (US state capitals) from TSPLIB. The minimal tour has a length of 33523.
- Rbg443.vrp.gz (ATSP) is a set of 443 coordinates. It is an asymmetric traveling salesman problem, which means the distance from node i to node j might be different from node j to node i. The optimal path has a length of 2720.

2) Knapsack datasets:

- P07 is a set of 15 weights and profits for a knapsack with capacity 750, from Kreher and Stinson, with an optimal profit of 1458.
- P08 is a set of 24 weights and profits for a knapsack with capacity 6404180, from Kreher and Stinson, with an optimal profit of 13549094.

IMPLEMENTATION

Among all the ant colony algorithms, we have selected several algorithms to be evaluated on the two types of datasets, TSP and knapsack. Also, to create the smme environment, the tests have been run on Google Colab in order to utilize a larger resource pool of CPU and RAM available to test the datasets and their performance.

Algorithms with Travelling path problems

Akaval/Ant colony algorithm [1]

The Akaval ant colony algorithm will have a random starting point and visit all cities to the final destination. It is assumed that they travel back using the same path and leave the pheromone on the way back. If it is shorter, they will deposit more pheromone on the shortest paths they traveled. Then, an individual ant makes decisions based on level of pheromone and the distance to the nearest city.

In the algorithm, we have imported the tsp file using the tsplib95 packages. Then we have calculated the distance between nodes and made a symmetric matrix as the input of the function:

```
AntColony(self, distances, n_ants, n_best, n_iterations, decay, alpha, beta)
```

With the initiation of number of ants, number of best ants who deposit pheromone, number of iterations, pheromone decay rate). Alpha means exponent on pheromone, higher alpha gives pheromone more weight. Beta means exponent on distance, higher beta gives distance more weight. The parameters can control the speed of pheromone decay and distance weight. The ant made decision based on the formula below:

```
city_to_city_score = pheromone ** alpha * (1.0 / distance) ** beta.
```

The dataset we used is the att48.tsp

Results:

By tuning the parameters of alpha, decay and beta, we adjusted the speed of the pheromone decay and compared the results of the optimal solutions with theoretical optimal tour 33523. It is noticed that the function will be close to the optimal solution 33523 when the parameters are set as (`n_ants=100, n_best=10, n_iterations=100, decay=0.2, alpha=1, beta=5`). Decay, which means evaporation, allows the learning of new policies by allowing ants to forget bad choices made in the past. Evaporation is carried out by decreasing pheromone trails at exponential speed. In this case, we use 0.2 as the decay parameter.

Figure 2 below shows the performance of the function under different alphas. The optimal solution with lower alpha is less fluctuation, this is because lower alpha means that the initial node is less likely to be chosen randomly.

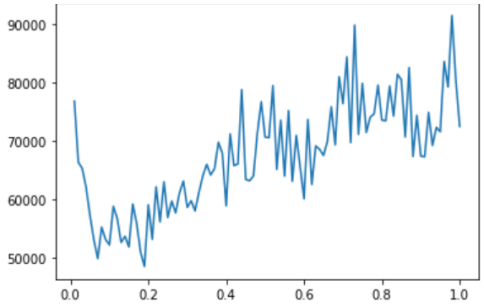

Figure 2 : impact of pheromone decay parameter alpha on results

Johnberroa - Ant-Colony-Optimization[2]

The AntColonyOptimization module is different from the previous implementation discussed above as it takes into account the probabilistic factor of the ants staying on the optimal path if they discover one or continue to trail off to other paths in hunt of a better food source. This implementation allows a wider range of explorative search in the algorithm as it can avoid local traps or cycles where there is no edge which allows us to exit the inner cycles.

After further research and tuning of the parameters, we found that the optimal parameters should be tuned accordingly for the given graph problem we are trying to solve, with larger distance matrices taking in a larger population of ants that networks itself into a colony. The implementation of AntColonyOptimization is defined by the class module as:

```
AntColonyOptimization(self, ants, evaporation_rate, intensification, alpha, beta, beta_evaporation_rate, choose_best)
```

The parameters specified for this implementation are the same as the abovementioned implementation, with the significant difference coming from the use of the parameters “intensification” and “choose_best”.

Results:

The implementation was performed on att48.tsp, with a total of 100 trials run under the parameter specifications of alpha = 1, beta = 5, ants=100, intensification=1 and choose_best = 0.10. We ran each trial for 200 iterations, with a total of 100 ants randomly initialized across the graph. We also looked into timing the algorithm’s runtime and averaging across the 100 trials in order to find the average time taken for ACO to produce a solution.

In order to test the implementation for robustness, we also tested the dataset on **DANTZIG42**, a smaller implementation of the TSP. The central goal of using a smaller dataset was to see if the algorithm converges to the global optimum. The parameters were kept the same as above for even comparison across the datasets, with the findings reporting that the dataset performs better on the smaller dataset, with the optimal solution converging to 699, which is also the true solution.

Alisonzille – Antsys [3]

The implementation of Alisonzille’s algorithm consists of three classes that have many sub-functions which interact to create the Ant Colony Optimization algorithm. The first, “AntWorld”, is a collection of nodes and edges from the problem. Taking in a list of nodes and edges, it constructs a “world” which will later be “traveled” through. The second, “Ant” is a single solution finder which traverses the “AntWorld” and deposits pheromones, while checking if the path had already been traversed. The last, “AntSystem”, takes in a world and the number of ants, and after randomly initializing the starting point, lets the ants travel through the possible nodes. After one iteration, the “elite_p_ants” are kept and pheromone deposits decay w.r.t. time.

To implement this algorithm, we import the tsp file with the tsplib95 package. Since we know the EDGE_WEIGHT_TYPE: EUC_2D, we can calculate the euclidean distance between nodes as the Edge/paths between colonies for our ant to traverse. This, along with the dataset is fed into the AntWorld class to create the web of possible paths. Finally, we create 100 Ants to input into the AntSystem, which iterates 1000 times with beta=2 and an evaporation coefficient of 0.05.

Results:

We see that after running the algorithm, the model does not find the optimal solution but has a stable solution revolving around 34,460 with 2.491% error rate.

Algorithms with Knapsack problems

Astrastrastra[4]

The probability of a specific object's selection is given by its value, and the remaining knapsack capacity and a variable called pheromone which is deposited on objects forms a single ant's solution. By introducing an evaporation system which in every iteration lessens the amount of pheromone, the final optimal solution is formed. This is done using a random.choices function on the solutions and probabilities to pick one randomly and testing with it. Here the heuristic is simply capacity / weight.

Results:

Converges towards optimum for P07 and P08

Alisonzille – Antsys [5]

The implementation of Alisonzille's algorithm consists of three classes that have many sub-functions which interact to create the Ant Colony Optimization algorithm. The first, "AntWorld", is a collection of nodes and edges from the problem. Taking in a list of nodes and edges, it constructs a "world" which will later be "traveled" through. The second, "Ant" is a single solution finder which traverses the "AntWorld" and deposits pheromones, while checking if the path had already been traversed. The last, "AntSystem", takes in a world and the number of ants, and after randomly initializing the starting point, lets the ants travel through the possible nodes. After one iteration, the "elite_p_ants" are kept and pheromone deposits decay w.r.t. time.

To implement this algorithm, we import the tsp file with the tsplib95 package. Since we know the EDGE_WEIGHT_TYPE: EUC_2D, we can calculate the euclidean distance between nodes as the Edge/paths between colonies for our ant to traverse. This, along with the dataset is fed into the AntWorld class to create the web of possible paths.

Results:

Converges towards optimum for P07, does not converge for P08

EVALUATION AND COMPARISON

With the same parameters, each run of the algorithms may generate a different optimal path. After running 100 loops, we have obtained the runtime statistics and optimal paths as summarized in the below table.

Algorithm	Avka	Johnberroa	Antsys	Astrastrastra	Antsys (knapsack)	Astrastrastra	Antsys (knapsack)
Dataset	att48	att48	att48	P07	P07	P08	P08
Dataset Type	TSP	TSP	TSP	KnapSack	KnapSack	KnapSack	KnapSack
Runtime [(Mean)]	39.76	14.5	15.5	13.5	12.07	26.15	25.08
Runtime [(STD)]	0.15	4.7	2.3	0.48	3.74	0.95	7.29

Best Solution	34,432	33,524	34,357	1,458	1,458	13,549,094	13,501,943
Average Solution	35,692	34,018	34,460	1,457.3	1,453.48	13,539,180.53	13,323,853.53
Optimal Solution	33,523	33,523	33,523	1,458	1,458	13,549,094	13,549,094
Converged Trials	0	0	0	87	16	67	0
Error Rate (Average)	6.47%	1.48%	2.80%	0.048%	0.31%	0.073%	1.66%
Error Rate (Best)	2.71%	1.47%	2.49%	0	0	0	0.35%
Global Optimal?(Y/N)	N	N	N	Y	Y	Y	N

For the tsp problems, all of the three algorithms will generate the results close to the optimal path after tuning the parameters, but fail to reach the global optimal. Since the functions cannot converge well, every run of the function will yield different results.Among the three algorithms, it is concluded that the Johnberroa will have the shortest runtime and the performance of runtime is also relatively stable. In terms of optimal solution, Johnberroa has the lowest best solution and average solution with relatively small runtime. And, Avka performs the worst in terms of runtime and error rates.

For the knapsack problem, after tuning the parameters and running a 100 trials with a small and a large dataset, we find that the algorithm Astrastrastra converges to optimum in both the datasets. It's runtime performance is slightly poorer than Antsys for both datasets, however it has a much lower standard deviation in terms of runtime, thereby proving to be more stable. The Antsys algorithm also converges for the smaller dataset but only 16/100 trials, compared to 87/100 for the former, and has a much higher average error rate. It does not converge at all for the larger dataset and has a best error rate of 0.35% and average error rate of 1.66%. Astrastrastra converges 67/100 times with comparable runtime. Hence, we go ahead with Astrastrastra.

Due to the limitation of the computation power and capacity of our laptop computers, we were unable to apply the larger datasets for testing the ant colony algorithm. Thus, the algorithms have been implemented and evaluated on the small to medium size datasets.

Conclusion

To conclude, the implementations of ant colony algorithms can help to generate the results close to optimal solution with chosen parameters, while some of them cannot converge and fail to have the constant optimal solution for each run. Some results may even deviate far away from the optimal solutions. We will put forward the best implementation to Nevergrad and further try to solve the issue of local convergence. We will promote Johnberroa's implementation for TSP problems and Astrastrastra for the knapsack problems. We can potentially add a flag/extra argument in our nevergrad.ACO function for switching between the two. Of course, we will default to TSP, since it is the more popular problem, especially with ACO solutions.

Also, the ant colony algorithms lack generalization to be applicable on a wide range of datasets. The algorithms cannot fit both the TSP and Knapsack datasets. It will call for further improvements on the code structure. It is also very slow compared to more robust alternatives within the Genetic Algorithm space, and also compared to optimizations outside the GA umbrella. As such, it is not recommended to use ASO on large datasets, as we pointed out earlier.

REFERENCES AND LINKS

[1] https://github.com/Akavall/AntColonyOptimization
[2] https://github.com/johnberroa/Ant-Colony-Optimization
[3] https://github.com/alisonzille/antsys/blob/main/antsys/antsys.py
[4] https://github.com/astrastrastra/ACO-AS
[5] https://github.com/alisonzille/antsys/blob/main/examples/knapsack_antsys.ipynb
[6] https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html (tsp dataset)
[7] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/atsp/(tsp dataset)
[8] Colorni A, Dorigo M, Maniezzo V. (1991) Distributed optimization by ant colonies. In: Proceedings of the first European conference on artificial life, Paris, France, pp 134–142
[9] X. Yu and T. Zhang, 2009. Convergence and Runtime of an Ant Colony Optimization Model. Information Technology Journal, 8: 354-359.**DOI**: 10.3923/itj.2009.354.359；**URL**: https://scialert.net/abstract/?doi=itj.2009.354.359

CONTRIBUTION NOTES :

Issue: https://github.com/facebookresearch/nevergrad/issues/616

Github code: https://github.com/janding123/Nevergrad_Ant-colony-_issue-616

Each of the team members has performed the algorithm search, dataset search, algorithm implementation and evaluation. The findings of the projects have been summarized and analysed within the team members as mentioned above.