Neel Gopal,
Shruthi Sundaranand

PARAMETER-TUNING FOR EFFICIENT PROGRAM EXECUTION THROUGH ANT COLONY OPTIMIZATION

Abstract

As high-performance hardware architectures grow in power and complexity, especially in parallel processing platforms, choosing optimal parameters for efficient program execution can be a challenge. There are many tunable factors that affect the speed of execution of a program and how much power the execution consumes: compiler options, compiler flags, cache block sizes, number of threads, etc. The number of combinations of parameters can grow exponentially so that a brute-force approach to find the best possible set of parameters may be prohibitively expensive. In this project, we propose an Ant Colony Optimization (ACO) solution. ACO is well suited for discrete optimization problems and has been successfully applied in many applications, especially those related to finding shortest paths, optimal schedules, best vehicle routing, etc. The success of ACO hinges on a well-tuned strategy to incrementally promote good solutions and penalize poor ones in the search process. We show how the performance of the ACO varies with different settings of the algorithm and search strategies. We obtain significant improvement when the ACO is extended to update a varying number of neighboring paths, like in simulated annealing. We also compare our results to other approaches, including baseline performances based on other algorithms and also one based on Particle Swarm Optimization.

1. Introduction

Given a computationally intensive program (or a class of similar programs) and a hardware platform to run the program, we are interested in choosing the right parameters so that the program executes as fast as possible. Besides minimizing the time performance, we also like to optimize other related objectives, like minimizing the power consumption of the program or maximize its throughput.

In this project, the program, called *iso3dfd*, is a C++ module that runs on a given platform. There are various parameters that affect the performance of the program: On the compiler side, there are different compiler flags, pragma directives, etc. On the runtime side, factors that determine performance include array sizes, cache blocking sizes, number of threads, etc. The number of combinations can be exponentially large and it is too expensive to test each possible combination in a brute-force manner. So we need an optimization technique that finds the optimum or near-optimum solution without exhaustively running all possible combinations of parameters.

In this project, we focus on four parameters : number of threads and three cache blocking sizes. These are all discrete, numeric, and ordinal parameters. That is, they can be sorted, ranked, and compared in a meaningful way. Even with this simplified version of the problem, there are many possible combinations. In this case, we can choose from 16 possible numbers of threads. The three cache blocking sizes have say 12, 30, and 32 possibilities. So there are a total of 16 x 12 x 30 x 32 = 184,320 possibilities. To solve this discrete optimization problem, we use an ACO algorithm. In this rest of this report, we (1) describe the essence of ACO, (2) provide a very simple example of how ACO solves a two-parameter problem, (3) describe our implementation of ACO to solve the four-parameter problem for an artificial cost function (the sphere equation) as well as the real cost function (calling the *iso3dfd* program), (4) discuss the results we obtain, and (5) conclude with key insights and ideas on how to improve our algorithm.

2. Ant Colony Optimization: A Nature Inspired Solution

ACO is one of the popular swarm-based algorithms that draws from simple reactive behavior of a collection of animals (like ants, bees, birds, termites) to produce, on aggregate, intelligent behavior for tasks like food foraging, navigation around obstacles, temperature regulation, etc. This class of algorithms fall under the broader umbrella of biologically or nature inspired methods (Darwish 2018) that have been successfully applied to a wide variety of problems, especially for optimization.

Here, we describe how ants actually collaborate to find the optimal path for foraging of food. Let us assume a simple scenario where there are 20 discrete paths from the nest of ants to a source of food (Figure 1).

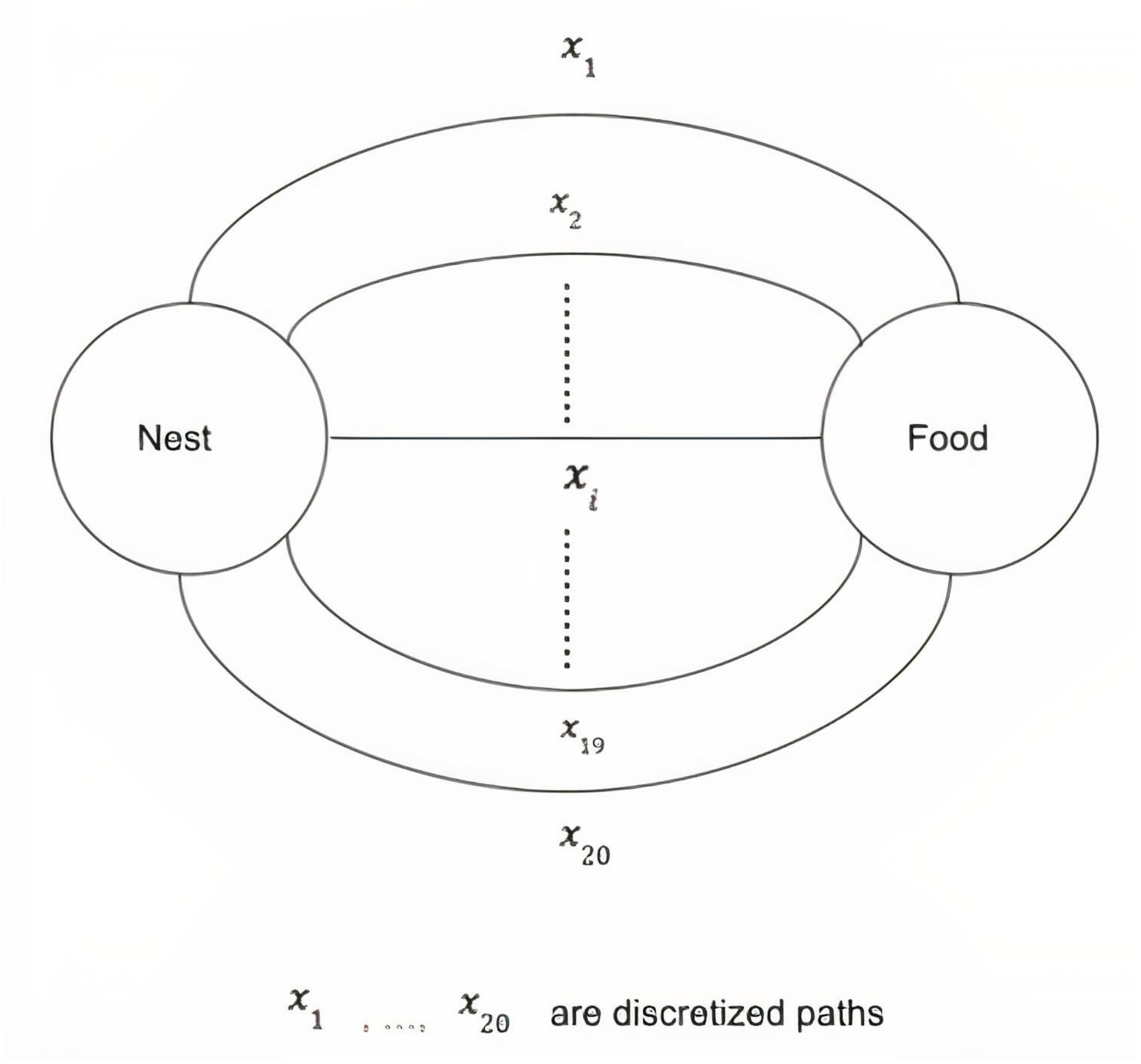

Figure 1

A group of say 5 ants start from their nest and have to find the food source and repeatedly bring food in small pieces back to their nest. The optimal solution is to find the shortest path that all ants can use. At first, the ants randomly select paths. They continuously deposit, at a constant rate, a chemical called pheromone along the explored paths. Ants also reactively choose the paths with the highest concentration of pheromone. Furthermore, the pheromone evaporates at a constant rate. In the initial phase of exploration, the ants will deposit pheromone over all paths randomly while traveling back and forth between the nest and food source. The pheromone level along the shorter paths will increase gradually because more trips will be taken along the shorter paths (less time is taken to complete the shorter paths), even if the paths are chosen randomly. As the pheromone level increases in increments on shorter paths, more ants will choose shorter paths. After many iterations (back and forth trips), all ants will land on the shortest, optimal path. This leads to the remarkable lines of ants that we commonly see on floors and walls.

This amazing feat of nature can be regarded as collectively intelligent because optimization (finding the shortest path) is achieved. The key aspect of this "intelligence" is the delicately tuned balance (achieved over eons of evolution) between the rate of pheromone deposition (that promotes good paths) and pheromone evaporation (that demotes bad paths). Without the right trail of pheromones, the search will be more randomized and the ant colony may not achieve its task and may not survive. Another aspect of intelligence is the communication among ants through pheromone—this type of communication is studied under biosemiotics (Favareau 2010), or communication through signs and codes by life forms. This highly effective multi-agent optimization model inspired Italian researcher Marco Dorigo to develop ACO in his PhD thesis in (Dorigo 1992).

3. A Simple Example of ACO for Discrete Optimization: The Elitist ACO

ACO is naturally suited for path-finding problems like the traveling salesman problem and related practical tasks like vehicle or internet routing, based on networks with nodes and edges. However, ACO can be applied to discrete optimization problems as well, since the task involves discrete path options. Here, we describe a very simple example of a discrete optimization for tuning two parameters for our runtime performance optimization problem. In the next section, we extend the ideas to solve the real problem.

Suppose we have two parameters that can affect the execution time of a given program: the number of threads used and the cache blocking size (we will call these parameters *thread* and *cache*). Assume there are 5 options for threads (say 1, 2, 3, 4, and 5) and four options for cache (say 2, 4, 8, and 16). Each combination of thread and cache constitutes a path. So, there are 5x4

= 20 possible paths. Let's call each path or parameter combination X_{ij} – the i^{th} thread, thread[i] and j^{th} cache, cache[j]. Now we have a colony of say 5 ants to explore one path each in every iteration. Each path X_{ij} has a cost, cost(i,j). Here we choose a simple unimodal objective function as the cost that we want to minimize:

$$\text{cost}(i,j) = \text{threads}[i]^2 + \text{cache}[j]^2$$

For our application, the cost (execution time or power consumed) will have to be computed by running the given program. This is a key difference between the standard ACO, where the cost (typically the length of the path) is known *a priori*. Here, it has to be computed on the fly. Furthermore, the cost function is expected to be non-linear and multi-modal, probably with several local optima.

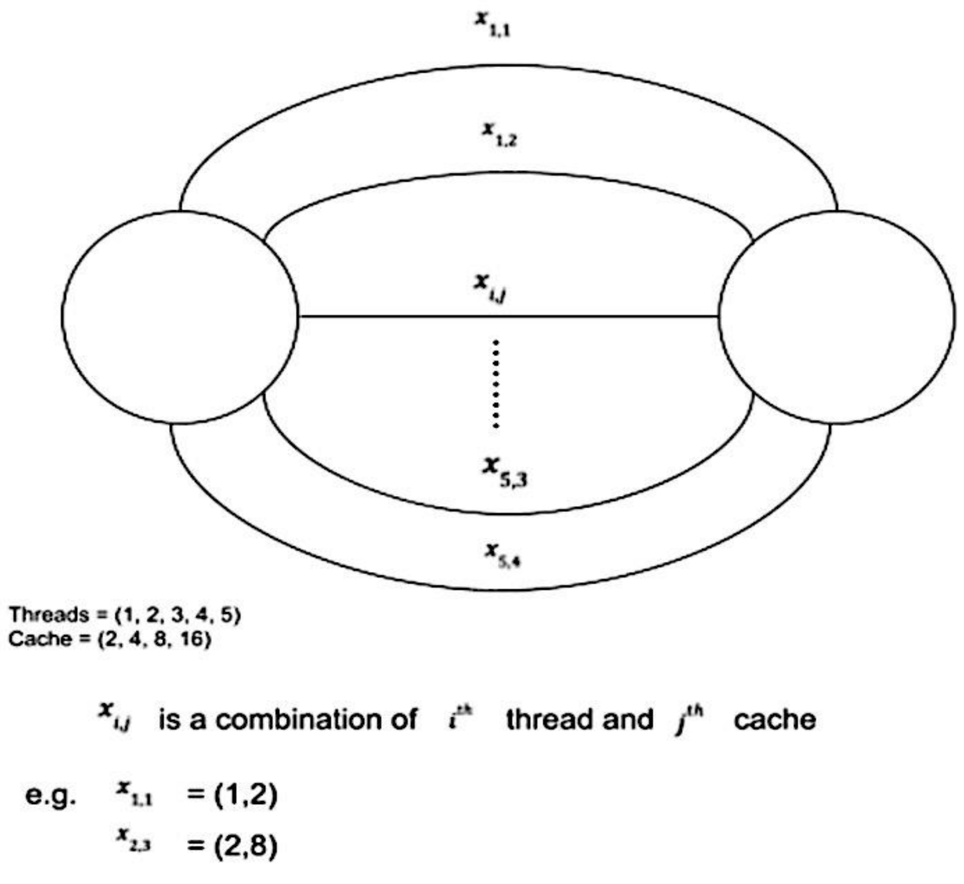

Figure 2

We keep two key data: (1) the pheromone level τ_{ij} associated with every path X_{ij}, which reflects the quality of the path, and (2) the probability with which to pick each of the options for thread and cache. Let probability vectors P_i and Q_j be probabilities with which thread i and cache j should be chosen. These probabilities are calculated based on the pheromone levels (the more pheromones, the larger the probability). In each iteration of the algorithm, every ant randomly picks a path X_{ij}; after all ants explore their chosen paths, the pheromone levels τ_{ij} and probabilities P_i and Q_j are updated iteratively. This iterative process continues until all ants choose the shortest path or the maximum number of allowed iterations is reached.

Initially, all τ_{ij} are set to 1, all $P_i = 1/5$ for i = 1 to 5, and $Q_j = ¼$ for j = 1 to 4. Then, each ant will choose a path X_{ij} stochastically by the roulette wheel selection method to pick the two parameters. The roulette method works as follows for each parameter: first a random number r (between 0 and 1) is chosen. It is compared with the cumulative probabilities (of P or Q). We choose the parameter option that has the largest cumulative probability less than the random number r. For example, in the first iteration, the probabilities and cumulative probabilities for the four cache sizes are as follows (Table 1):

Table 1. Roulette method

Cache Size	**Probability**	**Cumulative Probability**
cache[1] = 2	$Q_1 = 0.25$	0.00
cache[2] = 4	$Q_2 = 0.25$	0.25
cache[3] = 8	$Q_3 = 0.25$	0.50
cache[4] = 16	$Q_4 = 0.25$	0.75

If r = 0.55, then we choose cache[3]. If r = 0.92, we choose cache[4].

Now, after every iteration, the pheromone levels and probabilities are updated by the following rules for the ith thread and jth cache:

$$\tau_{ij}^{next} = (1 - \rho)\, \tau_{ij}^{prev} + \alpha * \max_k(cost(i,j))/cost(i,j) \qquad (1)$$

where τ_{ij}^{next} is the next pheromone level for path X_{ij}, τ_{ij}^{prev} is the previous pheromone level, ρ is the pheromone evaporation rate (typically 0.5), $\max_k(cost(i,j))$ is the highest cost over all ants k. α is a constant factor. Typically, $\alpha = 2$. The higher the cost of a path, the less the pheromone increases. The division of two cost values in (1) normalizes the values of τ.

The probability P_i of picking the ith thread and the probability Q_j of picking jth cache in the every iteration are calculated based on new levels of pheromone by the following equations:

$$P_i = \Sigma_j \tau_{ij} / \Sigma_i \Sigma_j \tau_{ij} \quad \text{where } 1 \le i \le 5$$
$$Q_j = \Sigma_i \tau_{ij} / \Sigma_j \Sigma_i \tau_{ij} \quad \text{where } 1 \le j \le 4 \qquad (2)$$

Essentially, the probabilities are updated in favor of paths with higher levels of pheromone.

The algorithm is as follows:

```
Initialize thread, cache, number of ants, $\tau$, P, Q, $\rho$, $\alpha$
While (convergence or maximum iterations not reached)
    All ants pick paths randomly by the roulette wheel method
    Find costs of path for all ants using the cost function
    Find the best and worst paths
    Update pheromone matrix $\tau$ as per equation (1)
    Update probability vectors P and Q, as per equation (2)
```

In this version of ACO, called the *elitist* method, pheromone is updated for the path the best ant finds. Our implementation of this simple algorithm makes all ants converge to the best path (thread = 1 and cache = 2) much quicker than by a purely random search method. However, as we shall see in the next section, the elitist approach works well for small problems or problems where many paths are visited. But for this parameter-tuning problem, a better pheromone strategy is needed.

4. Implemented Methods

In this section, we present an extension of the simple algorithm described above. The approach is the same in essence, except that proposed algorithm is used to solve our real parameter-tuning problem, where we have 4 parameters: (1) thread with 16 possible values, (2) cache1 with 32 possible values, (3) cache2 with 12 possible values, (4) cache3 with 30 possible values. That is, the total number of possible combinations or paths is 16 x 32 x 12 x 30 = 184,320.

In the simple example above, there were only 20 paths, and all ants quickly converged to the optimum path. Also, the computation of the cost function was inexpensive. In the real problem, the computation of the cost is very expensive; calls to the given program have to be made by every ant in every iteration. So, the search strategy has to be as efficient as possible. The real problem has far more paths and many of them may not be visited, so their pheromone level never gets updated.

The first version of the algorithm, called the elitist method, is like a typical ACO that updates only paths visited. We observe that visited paths are quickly favored and many ants settle on a suboptimal path.

This motivates the following heuristic that improves performance through a better pheromone update strategy. When exploring a path, we do a local search around that path to find the cost of immediate neighbors by changing one parameter at a time. Using this information, the pheromone levels of many surrounding neighbors (not just immediate ones) are updated, without calculating the cost. This is based on the reasonable assumption that the cost function is smooth in local neighborhoods. By updating pheromones for neighbors as well, we broaden the search to many more options, and we have a better chance of escaping local optima, and a better chance of finding the global optimum.

The pheromone levels of many paths are updated based on a function that combines the quantum of gain in performance and the Euclidean distance to the neighbor. We normalize the gain or loss in performance and the Euclidean distance so that they can be meaningfully combined.

The core idea is the following: if by just changing the number of threads, say from 4 to 5, we get a performance increase, then we promote a few paths with threads 5 or greater (other parameters being the same), and demote a few paths less than 4. The farther away the path, the lesser the pheromone changes. Here, we can use a formula to decrease the pheromone increment with increasing Euclidean distance. It can be linear or we can use a normally distributed function over the Euclidean distance to update the pheromone level. However, this approach can be applied to ordinal, numerical functions i.e. they are naturally and numerically ordered.

Here is an example. Consider a 3-parameter problem, with the following values:

thread = [1 2 3 4 5 6 7 8 9 10]
cache1 = [2 4 8 16 128 256 512 1024 2048]
cache2 = [2 4 8 16 128 256 512 1024 2048]

Suppose one ant explores path $X_{4,5,6}$ where thread[4] = 4, cache1[5] = 128, cache2[6] = 256. Let the associated cost be cost(4,5,6). The ant explores the immediate neighboring paths by changing only one parameter at a time. We can explore both directions or choose between the two directions of change randomly (to avoid the cost of computing on both sides). Suppose the ant finds these costs of neighboring paths: cost(4+1,5,8), cost(4,5-1,8), cost(4,5,5,6+1). The cost of these local neighbors are computed and the pheromone level of the following paths are updated by increasing the pheromone on the "good" side (side where cost of the neighbor decreases) and decreasing on the other.

$X_{i,4,5}$ i = 1 to 10
$X_{4,j,6}$ j = 1 to 9
$X_{4,5,k}$ k = 1 to 9

By changing just one parameter at a time, we argue that the search is more informed and effective. We can choose to compute the cost of both sides as well, but that will increase the cost of the ACO search.

Furthermore, we can also adopt the following strategy: initially, we search a larger local neighborhood and update many paths. With more iterations, we decrease the size of the neighborhood, like the temperature in simulated annealing. The rate of decrease of this neighborhood size can be linear or based on a suitable distribution, like a normal distribution over an Euclidean distance.

In our implementation, we tried a very simplified version of the strategy of looking at neighbors: we find the cost and update the pheromone for only *one* neighbor in every direction (by changing just one parameter at a time). We found that even this simple strategy helps tremendously in escaping local optima.

We also experimented with a Particle Swarm Optimization (PSO) solution (Kennedy & Eberhart 1995). PSO solves a problem by having a population of candidate solutions (or particles) which move around in the search space based on simple formulae over the particle's position and velocity. The movement of each particle is determined by its local best known position, and is also guided toward the best known positions in the search space, which are updated as better positions are found by other particles. This is expected to move the entire swarm toward the best solutions. This closely resembles the idea behind ACO.

5. Results

We ran the algorithm (called neighbor) based on the simple neighbor search and compared it to baseline methods:

- Elitist: the elitist ACO algorithm with pheromone update for the best ant only
- All ants: ACO algorithm with update pheromone for all ants
- Random: very random method for updating pheromone, which is expected to perform badly

We use the following as measures of quality of the algorithm on (1) the cost of the best solution found, (2) the number of times we have to call the iso3dfd program, and (3) the number of iterations needed to converge to the best solution. Calling the given program is the biggest determinant of the efficiency of the algorithm. Also, we keep track of the cost for each path that has been computed so we avoid unnecessarily re-computing the cost for a given path. We first tried our implementation where the cost is the sphere equation:

$$\text{Cost} = \Sigma\, x_i^2 \text{ where } x_i \text{ is the value of the i}^{\text{th}} \text{ parameter.}$$

Table 2 and Figure 3 compares the different algorithms based on two performance criteria (every other parameter being the same: maximum iterations = 30, number of ants = 20, $\alpha = 2$, $\rho = 0.5$)

Table 2. Comparison of the four ACO algorithms

Algorithm	**Lowest Cost**	**Calls**
Neighbor	769	227
All Ants	6208	566
Elitist	5648	356
Random	13177	469

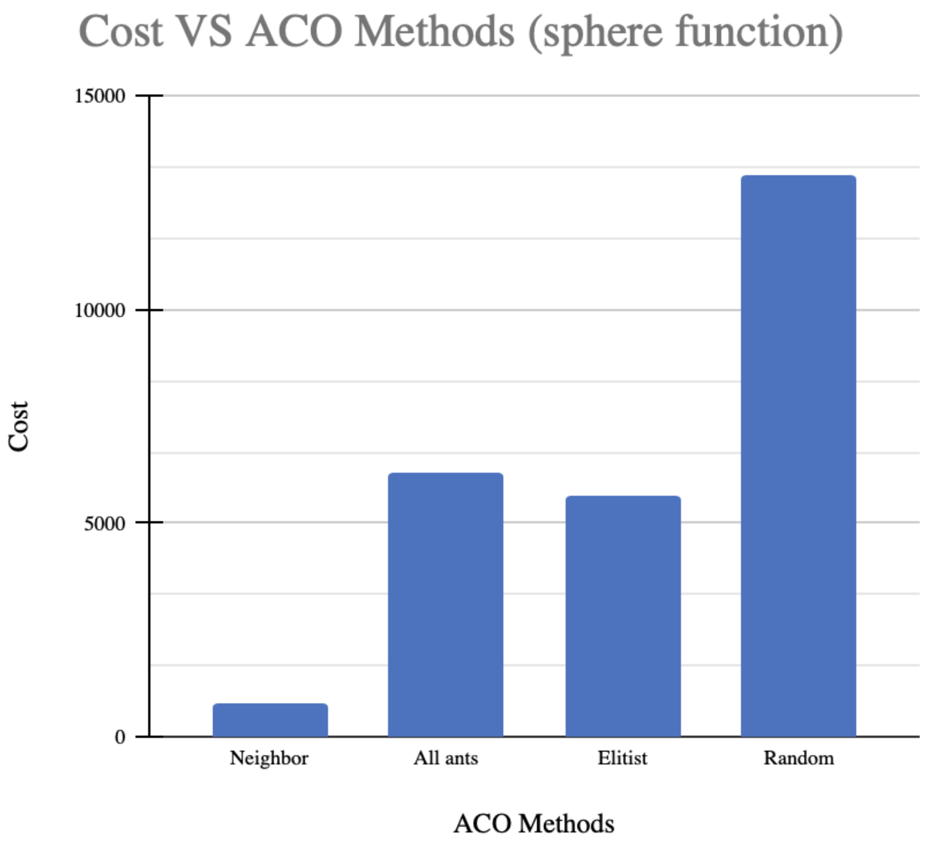

Figure 3. Comparing the four ACO algorithms

We observe that the neighbor performs the best: it reaches the optimal solution (769) after 9 iterations with the least number of calls to the program.

We also found how α affects the time to converge convergence. Table 3 compares performance for different values of α (neighbor algorithm, maximum iterations = 30, number of ants = 20, ρ = 0.5). In this case, all runs converge to the optimal cost (769).

Table 3. Varying α in the pheromone increment formula

Alpha	Cost	Calls	Iterations
1.0	769	487	29
2.0	769	469	30
3.0	769	351	19
4.0	769	227	15
5.0	769	192	9
6.0	769	194	10
7.0	769	199	10
8.0	769	209	15
9.0	769	253	18

Figure 4 shows that the best value of α in this situation is around 5.0, based on the number of iterations needed to converge to the optimal solution.

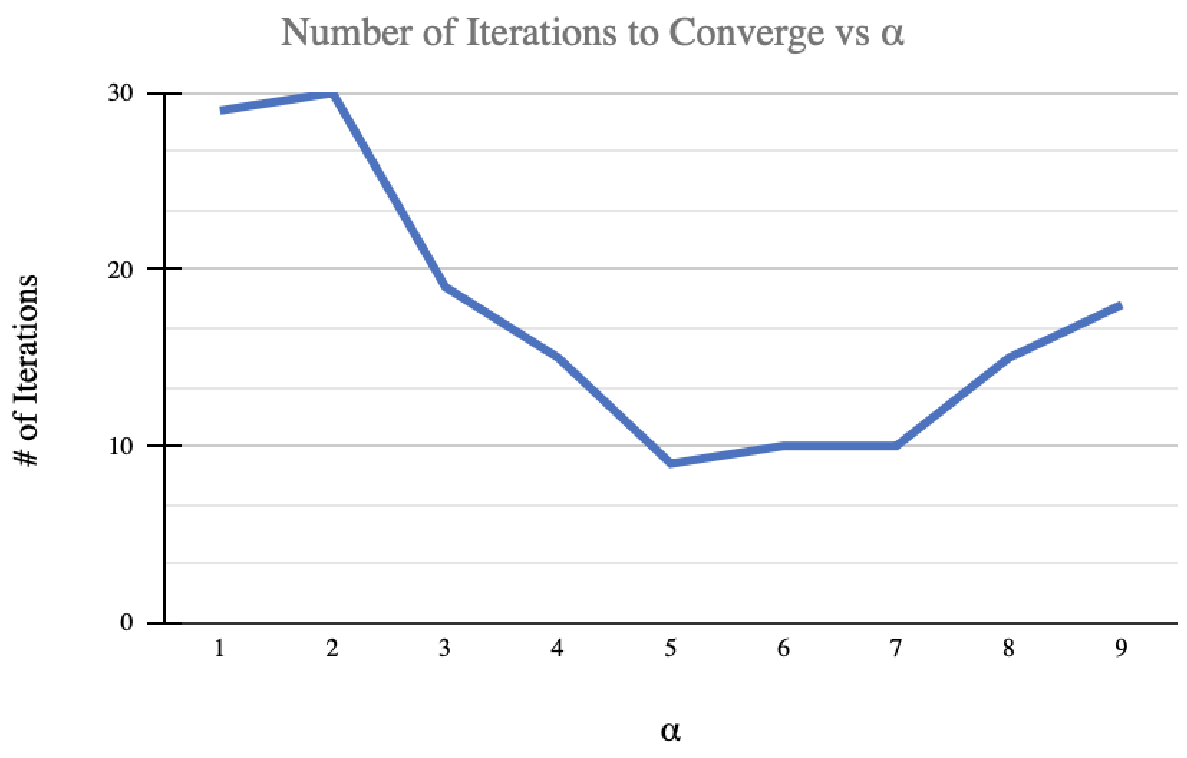

Figure 4

Table 4 shows the impact of evaporation rate ρ, everything else being the same (neighbor algorithm, 20 ants, maximum number of iterations = 30, α = 5). A suboptimal solution is obtained with ρ = 0. Figure 5 shows that the best ρ in this scenario is about 0.5, which is what is commonly used in ACO algorithms.

Table 4. Impact of Evaporation Rate ρ

ρ	Lowest Cost	Calls	# of Iterations
0	1552	790	30
0.1	769	654	26
0.2	769	487	20
0.3	769	414	30
0.4	769	401	29

ρ	Lowest Cost	Calls	# of Iterations
0	1552	790	30
0.5	769	192	9
0.6	769	199	13
0.7	769	179	22
0.8	769	163	23
0.9	769	151	21
1	769	187	25

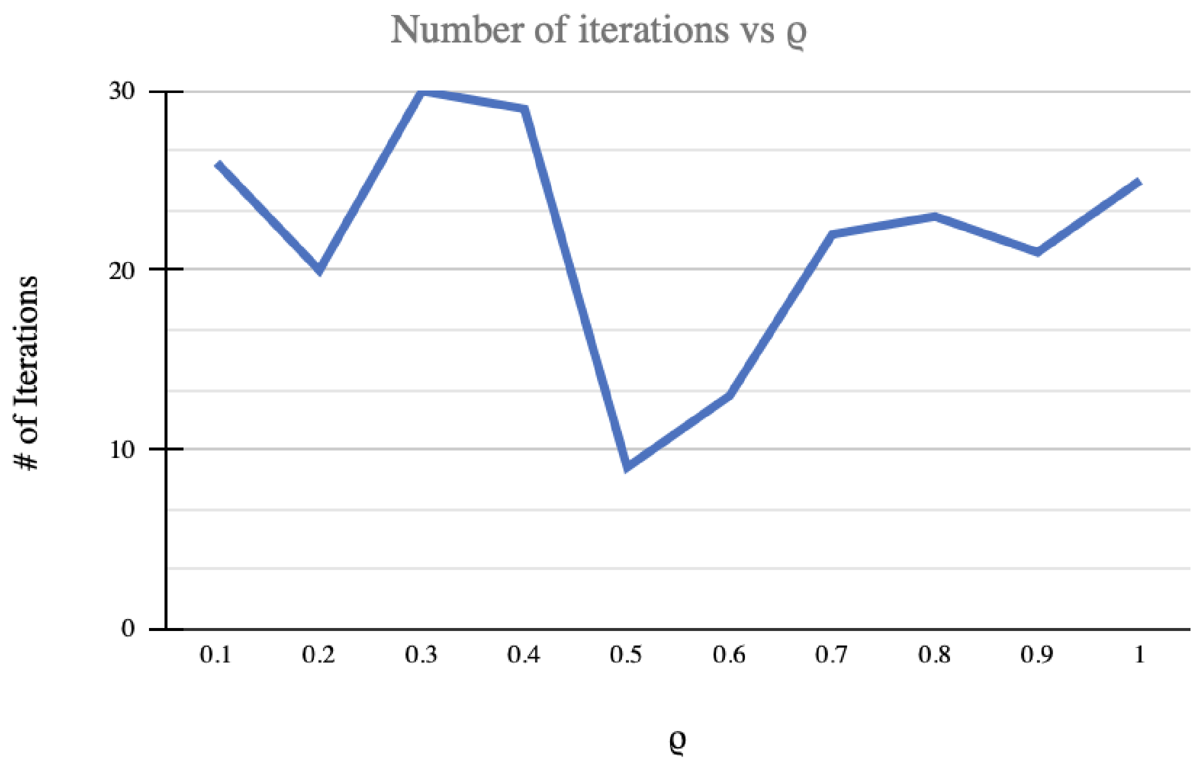

Figure 5

Table 5 shows how the number of ants impacts performance, everything else remaining the same. More ants lead to the optimum cost, but then more calls are made to the given program. The number of ants is related to the number of iterations; it is unclear what is the sweet spot for the number of ants and the number of iterations that keeps the number of calls at a minimum and still find the best solution possible.

Table 5. Impact of changing the number of ants

Number of ants	**Lowest cost**	**Number of calls**
1	52089	78
5	13849	113
10	993	141
15	793	169
20	769	192

We run the four algorithms for ACO on the real problem that makes calls to *iso3dfd,* the C++ program. Table 6 shows the performance for the four algorithms for 20 ants, 30 iterations, $\alpha = 2$, $\rho = 0.5$). Interestingly, we see the same pattern of performance as when the function is the sphere function: neighbor performs the best; elitist and "all ants" perform the same, and as expected, random performs the worst (Table 6 and Figure 6).

Table 6. Comparing the four ACO algorithms on the real problem

ACO Algorithm	**Lowest cost**	**Number of calls**	**Execution time (seconds)**
Neighbor	0.5526	549	478.98
All ants	0.6143	449	399.96
Elitist	0.6046	280	252.47
Random	0.6556	406	368.18

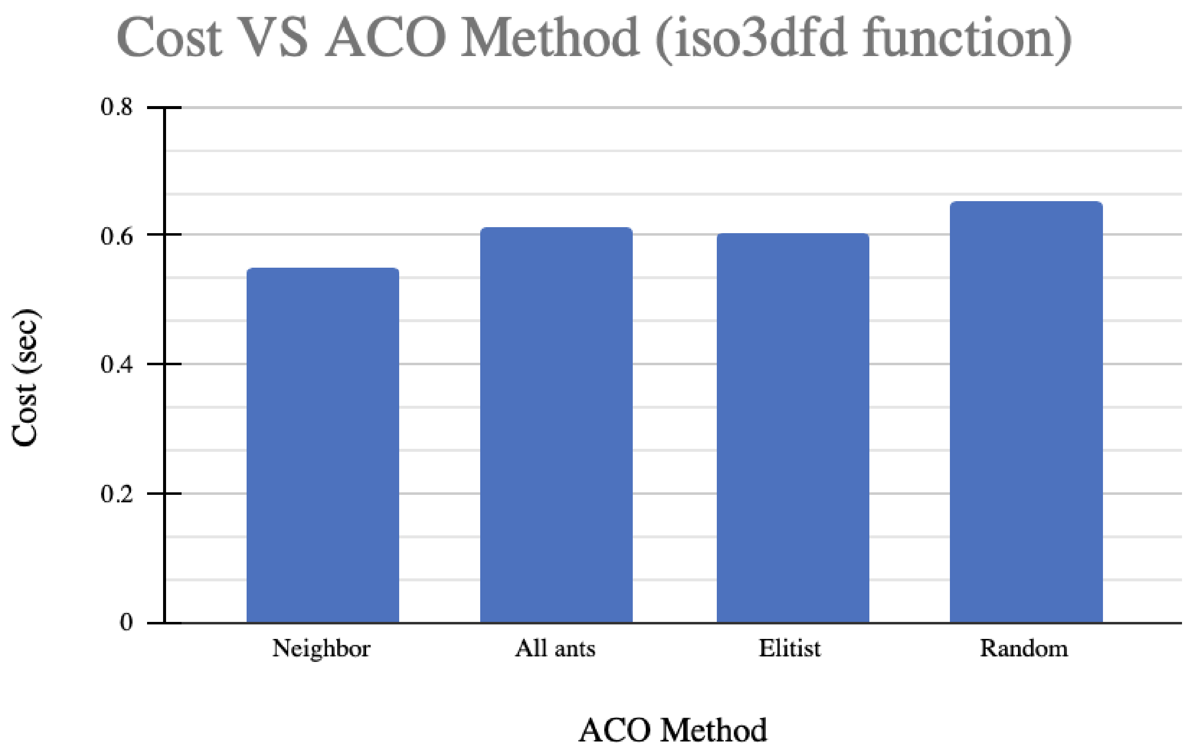

Figure 6. Comparing four ACO algorithms on the real problem

6. Conclusion & Future Work

The following are the key insights:

1. Although ACO is a good method for discrete optimization, the elitist ACO method is not adequate because not enough paths are visited and there is a tendency to settle on sub-optimal solutions.
2. The local neighborhood search in the *neighbor* ACO algorithm helps to escape local minima.
3. Our ACO approach, the neighbor algorithm in particular, gives a better solution faster than our PSO implementation.

4. The ACO is a good approach for our parameter-tuning problem: if the pheromone increment is properly tuned, it can lead to good solutions very quickly. We determined an appropriate evaporation rate ρ and found that $\rho = 0.5$ works well.
5. The constant α used for incrementing pheromone should not be excessively high as well. We determined an appropriate value for and found that $\alpha = 2$ works well when the cost is the sphere function and $\alpha = 5$ works well for the real problem involving calls to the given program.
6. The results are very similar for the two versions of the problem: using the sphere function as well as calling the given C++ program. However, while the sphere function is deterministic, the cost of calling the given function can fluctuate. We ran the optimization on exactly the same or combination of parameters a number of times: the cost in terms of time performance (in seconds) varies significantly. The mean was 0.604525 seconds while the standard deviation was 0.049887365.

As future work, the following can be explored:

1. Increase the size of the neighborhoods for which the pheromone will be updated in the *neighbor* algorithm. However, this approach is applicable to numeric, ordinal parameters.
2. Starting from a large size of a neighborhood, decrease the size with every iteration, like in simulated annealing. This is a promising approach since initially we would like to explore many paths, but as we get a large set of relatively good paths, the search can be narrowed down gradually.
3. Introduce other parameters that impact the performance.
4. Optimize on other cost functions, or a combination of functions, like power consumed and throughput.
5. Use a pheromone update formula that ranks the importance of each parameter (determined automatically or based on domain knowledge).
6. More informed way to set initial conditions based on domain knowledge.

7. References

A.Darwish, *Bio-inspired computing: Algorithms review, deep analysis, and the scope of applications*, Future Computing and Informatics Journal, Volume 3, Issue 2, pp. 231-246, 2018.

F. Donald (ed.), *Essential Readings in Biosemiotics: Anthology and Commentary*, Biosemiotics 3, Berlin: Springer, 2010.

Ghoseiri, Keivan, and Behnam Nadjari. "An Ant Colony Optimization Algorithm for the Bi-Objective Shortest Path Problem." *Applied Soft Computing*, Elsevier, 24 Oct. 2009, https://www.sciencedirect.com/science/article/pii/S1568494609001963.

Glabowski, Mariusz, et al. "Shortest Path Problem Solving Based on Ant Colony Optimization Metaheuristic." *ResearchGate*, Pozna´n University of Technology, Nov. 2012, https://www.researchgate.net/publication/234065233_Shortest_Path_Problem_Solving_Based_on_Ant_Colony_Optimization_Metaheuristic.

Ines Alaya, Christine Solnon, Khaled Ghedira. Ant Colony Optimization for Multi-objective Optimization Problems. 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Oct 2007, Patras, Greece. pp.450-457. Hal-01502167, https://hal.archives-ouvertes.fr/hal-01502167/document.

M.Dorigo, *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italy, 1992.

J. Kennedy, R. Eberhart, *Particle swarm optimization*, Proceedings of the 1995 IEEE International Conference on neural networks, Volume 4, pp. 1942-1948, 1995.

Research Paper on Teams (documents.pub_ant-colony-optimization-55eb1ca6e0ebb.pdf)

https://github.com/johnberroa/Ant-Colony-Optimization
https://github.com/pjmattingly/ant-colony-optimization