Ant Colony and Firefly Optimization

Daniel Eberharter, Christoph Haas, Maximilian Suitner

June 2021

1 Ant Colony Optimization

1.1 Overview

The ant colony optimization (ACO) is an algorithm based on the behaviour of real world ants. It is usually used within graph based problems. When ants travel through the graph, they deposit pheromones. The more pheromones deposited on a given edge the more ants used this edge which results in a higher probability for a future ant to take the edge. Each ant selects its next edge x or y based on the following probability.

$$p_{xy}^k = \frac{\left(\tau_{xy}^{\alpha}\right)\left(\eta_{xy}^{\beta}\right)}{\sum_{\in \text{allowed}_z}\left(\tau_{xz}^{\alpha}\right)\left(\eta_{xz}^{\beta}\right)}$$

Where τ_{xy} indicates the amount of pheromones on the given edge and η_{xy} the desirability of the edge (usually $\frac{1}{d_{xy}}$ where d_{xy} denotes the distance). The parameters α and β control the influence of either the deposited pheromones or the desirability. After all ants have completed one step, the pheromones on all edges get updated.

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

with

$$\Delta\tau_{xy}^k = \begin{cases} \frac{Q}{L_k} & \text{if ant } k \text{ uses edge } xy \text{ on its tour} \\ 0 & \text{otherwise} \end{cases}$$

where m is the number of ants and $\Delta\tau_{xy}^k$ the amount of pheromones deposited by kth ant. Q denotes a constant value and L_k the cost of the tour taken by the ant (typically sum of all distances).

1.2 Implementation

Our implementation simply follows the rules describe by the ACO and the Traveling Salesman Problem (TSP). In order to integrate the algorithm we implemented the optimizer using Opt4J's `IterativeOptimizer`. However, each iteration does not compute one step for each ant, rather each iteration computes a new solution. So for each iteration all ants travel through the whole graph. In order to abstract the ACO from the TSP we created two additional classes which describe the graph. `AntNode<T>` which holds the original node (in the TSP case a `City`) and all neighboring nodes. `AntEdge<T>` which describes an edge in a the weighted graph. Therefore, in addition to the two nodes, it includes a distance and the amount of pheromones deposited. The distance will be used for calculating the probabilities and the amount of pheromones an ant deposits. The `AntColony` interface provides a high level API for the algorithm. Initialization is done by passing the start node (since a node includes all its neighbors we know all nodes implicitly). The `step` method returns the next best solution for the current state of the graph. It returns all paths taken by each ant. The implementation holds a given number of `Ant`s. The `Ant` class describes the movement of an ant and can calculate the next best path for the current state of the graph. Therefore, the ant chooses an edges until it reaches the end edge. To further abstract the problem we introduced the `AntStepper` interface which provides functionality for one step for a given ant and corresponds to the ACO update method. The exposed function `getNextEdge` simply returns the edge selected by the algorithm. The ant then collects all edges (and the nodes) and creates an `AntPath` which holds the path taken by the ant and the cost of the path. The last step is simply updating the deposited pheromones by using each `AntPath`.

1.3 Challenges

1.3.1 Performance

Since this was an optimization task it is very important, in order to keep optimization time as low as possible, that the data structures where chosen carefully. The first step was to abstract the graph from the TSP. Introducing a map in each node that holds all its neighbors was crucial in order to get short lookup time for a given edge. Additionally, when implementing the ACO it is very tempting to iterate over all edges in a nested loop. To avoid this we introduced a `Selector` interface which selects the best edge from a list of given edges and a list of respective probabilities. Each edge contains a weight of $\left(\tau_{xy}^{\alpha}\right)\left(\eta_{xy}^{\beta}\right)$ which is calculated *once* for each edge before moving. Selecting possible edges does scale badly with the problem size (number of cities in the TSP). In order to mitigate this issue we introduced an additional hyper-parameter `consideredEdges` which controls the number of edges to consider for moving (based on their fitness). This *window size* allows the algorithm to perform efficiently even with a large problem-size.

1.3.2 Abstraction

Since `Opt4J` already makes use of `Guice` it was easy to abstract certain parts from the algorithm. Additionally, testing was made easy since only a few lines of code had to be tested at once.

1.4 Evaluation

All evaluations were performed on the same system with eight CPU cores and 32GB RAM. In graph 1 one can see the optimization progress of the ACO algorithm. In comparison, graph 2 shows the optimization process for the same problem utilizing the given evolutionary algorithm. The ACO algorithm reached its optimum very fast in comparison the EA for the same amount of iterations. We can call a value an optimum if the relative change to its successors is neglectable. If we compare the computation time for one iteration, it was around 20ms for ACO, whereas EA only needed 3ms on average. But due to the fact that the ACO algorithm reached a optimum after only 6 iterations, it still outperformes the EA.

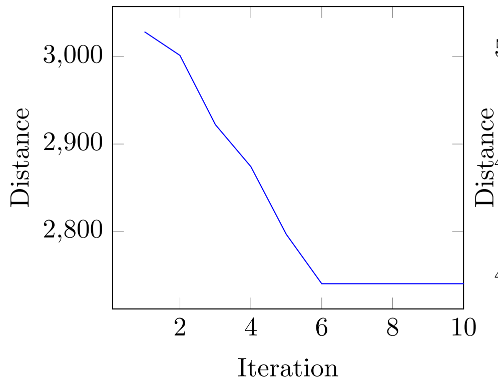

Figure 1: ACO Algorithm

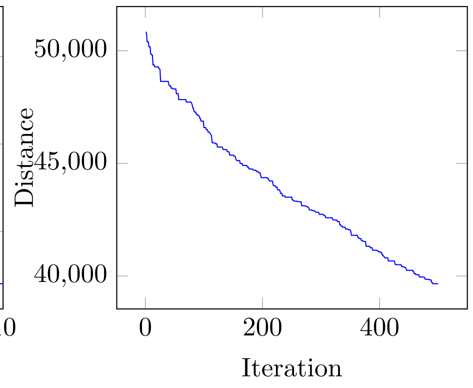

Figure 2: EA algorithm

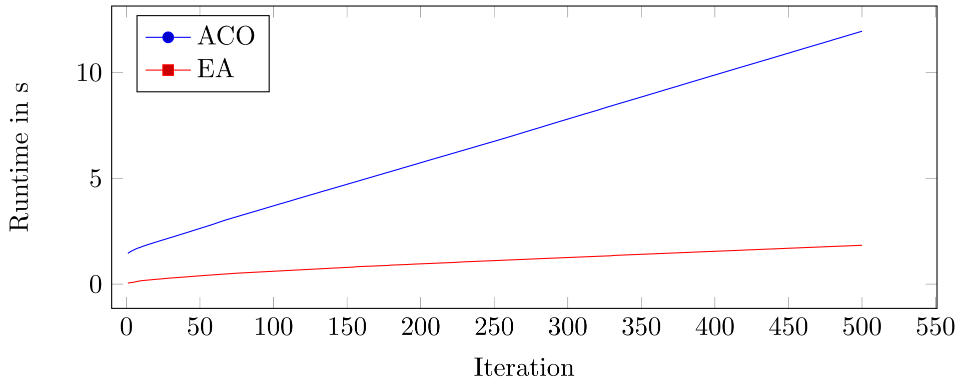

Figure 3: Runtime comparison for ACO and EA algorithm

2 Firefly Optimization

2.1 Overview

The firefly optimization algorithm is a nature-inspired iterative meta-heuristic particle-swarm-optimization (PSO) approach. The main idea is to based on the behavior of a swarm of fireflies, where each firefly moves towards the most brightest individual in its proximity. In terms of the algorithm a firefly's light intensity is defined by its fitness. Its fitness value in return is defined by the position of the firefly, where the position corresponds to a specific solution in the search space. After enough iterations the fireflies positions should converge to the optimal solution. In order to not starve on local sub-optima the algorithm introduces a random walk component, where each firefly moves a certain distance in a random direction per iteration.

The update function for the position of a single firefly x_i in relation to another firefly x_j boils down to

$$x_i^{t+1} = x_i + \beta e^{-\gamma r_{ij}^2} + \alpha \epsilon$$

where α, β and γ are hyper-parameter for controlling random-walk, attractiveness and light absorption.

2.2 Implementation

The core of the implementation is the `FireflyAlgorithm` class, which implements the Opt4J `IterativeOptimizer` interface, which requires an `initialize` method as well as a `next` method. In the `initialize` method the Opt4J selector is initialized with the configured number of fireflies and we create an initial population of individuals. The `next` method represents a single optimization step, which can be described by the following steps:

- Calculate the fitness of each individual. Since we our goal is to minimize the objective values the fitness of firefly i is indirect proportional to the sum of the objectives.

  $$fitness_i = 1/\sum_{o \in obj_i} o$$

  This calculation is done using the `FitnessCalculator` class.

- Find the fittest individual in the population (the firefly with the highest light intensity) using the `FireflySelector` class. While this is not strictly necessary it greatly helps increasing performance

- Calculate the new position for each for each firefly using the `ParticleMover` class.

- Update the position in the individuals genotype and set the individuals state to *genotyped*. This ensures that Opt4J executes phenotyping and evaluation.

2.3 Remarks

The following remarks regarding the implementation should be considered:

- Basic firefly optimization has a bad worst-case runtime complexity since the position of each firefly is updated with respect to each other firefly. We mitigated this issue by finding the brightest individual first and moving each other individual only in relation to the fittest one. This leads to worse results for a small number of iterations but in return greatly speeds up execution time.
- Initially we re-created each individual in the population after calculating the new phenotype (i.e. the position of the firefly). The final version omits this and always works on the same individual instance for each firefly which is far more efficient.
- Usage of Javas *parallelStreams* when possible in order to leverage systems with multiple cores.
- Excessive use of dependency injection (the optimizer class barely contains any logic by itself - implementations have been distributed over multiple highly coherent services).
- High code coverage (100% classes, 91% lines).

2.4 Evaluation

The optimization results of the firefly algorithm can be seen in figure 4. In comparison to the results of the evolutionary algorithm seen in figure 5, one can see that the firefly algorithm reaches the optimum faster. The firefly algorithm reaches an optimium, which we defined as 20 % difference to the global optimum that was found, after 2247 iterations. The evolutionary algorithm needs for the same goal 7095 iterations. There is also a significant time improvement which can be seen in figure 6. The firefly algorithm is nearly twice as fast as EA, it takes only 15.8 seconds for 10000 iterations, where the EA needed 29.1 seconds. Trying out different parameter settings for the algorithms changes the results drastically. The settings that are used in the config file were optimized by randomly trying and comparing the results.

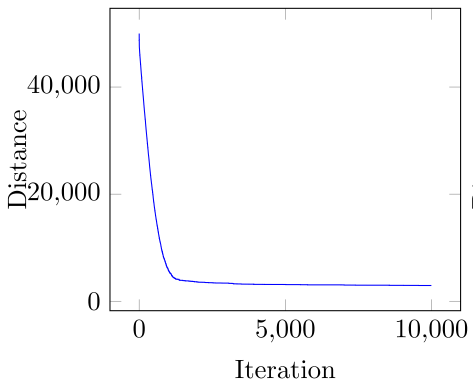

Figure 4: Firefly Algorithm

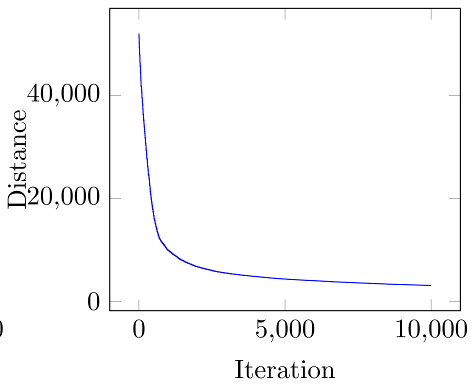

Figure 5: EA algorithm

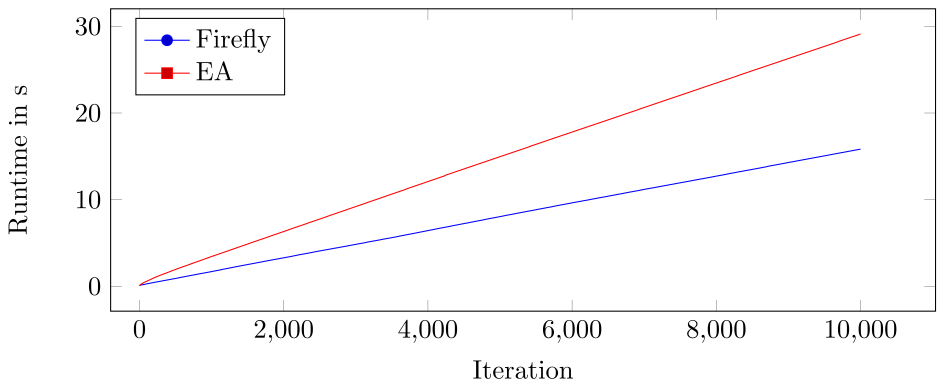

Figure 6: Runtime comparison for Firefly and EA algorithm