Natural Computing Project Assignment 2 - ACO and Longest Simple Paths in Graphs

Breno Campos Ferreira Guimarães

November 5, 2018

Abstract

In this report, we describe the implementation of an Ant Colony Optimization (ACO) algorithm towards solving the problem of finding the longest simple path between two vertices in a graph. In this problem, the goal is to find the path between two given vertices in a graph with the largest aggregated edge weight, with the restriction that each vertex can appear only once in the solution. We first describe the problem modeling and how it fits into an ACO solution, and go over a few important implementation decisions. Then, we perform a thorough analysis of the execution parameters, which describes how the ideal values were chosen empirically.

1 Introduction

In graph theory, one of the most fundamental structures studied in graphs are *paths*. Given a graph $G = (V, E)$, a path between two vertices $v, w \in V$ is a set of edges $e \subseteq E$ that connect both vertices. There are many categories of paths, each defined by some particularity or restriction. One of the most common and most widely studied type of paths are *simple paths*. A simple path between two vertices in a graph is a path such that each vertex in the graph can only appear in the path at most once.

A popular problem dealing with paths with wide-ranging applications is the longest simple path problem. In this problem, one's goal is to find the simple path in a graph with the maximum value of a specific objective function. Typically, in unweighted graphs, the longest simple path is defined as merely the path with the largest number of edges. In this work, we deal with the weighted version of the longest path problem. In this variation, the goal is to find the path between two given vertices with the largest aggregated edge value.

Figure 1 shows an example of an instance of the longest simple path problem in a weighted directed graph. The goal is to find the longest path from vertex A to vertex F. In the unweighted variation of the problem, the path highlighted in red would be the optimal solution, given that it has more edges. However, in our case, the path in green is the optimal solution, given that its total aggregated weight (23) is larger than the red path's (18).

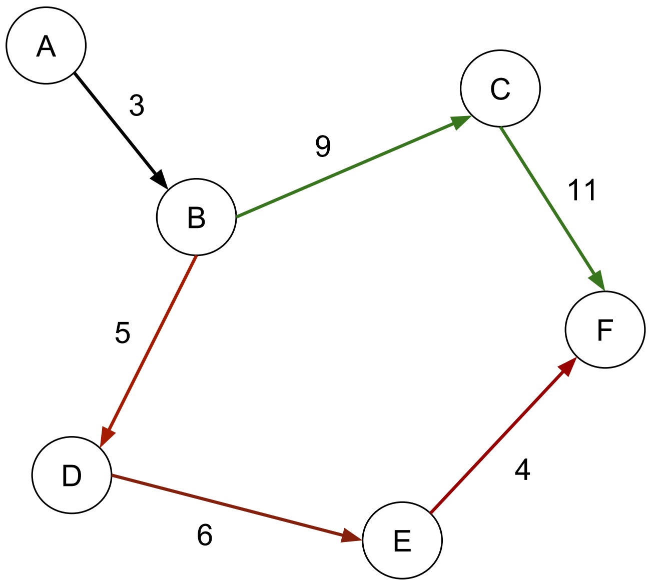

Figure 1: Example of longest simple path instance in a weighted directed graph.

The longest path problem is known to be NP-Hard [1]. Therefore, attempts at solving it in practical time usually apply some sort of heuristics towards finding good solutions, without guaranteeing optimality. In this work, we propose an approach to solve this problem using Ant Colony Optimization (ACO) [2]. In these algorithms, solutions are iteratively built by the simulation of independent agents, ”ants”. The exploration of the search space is guided by a feedback mechanism that simulates pheromones, such that elements of promising solutions are more likely to be chosen in following generations.

In the next few sections, we first go over how the problem was modeled as an instance of an Ant Colony Optimization algorithm. We then provide more details on the implementation of the algorithm itself and explain the most important design decisions. Finally, we evaluate our solution on a dataset of three graphs of varying sizes and densities, tuning the parameters towards achieving the best results.

2 Problem Modeling and Proposed Solution

In an Ant Colony Optimization algorithm, in each iteration every agent independently builds a probabilistically-generated candidate solution to the problem at hand. Then, a feedback step alters the probabilities of different elements of each candidate solution being picked in the future, based on the quality of each solution. Therefore, a few key elements had to be determined during the design of the algorithm. The most important of these are how candidate solutions were modeled, the function that determines the fitness of a solution (and consequently how much its elements should be ”rewarded”) and the probabilistic function that determines which element should be added to a candidate solution in each step.

2.1 Modeling solutions

There are several ways to represent a path in graph. One can, for instance, simply list the vertices through which the path traverses, starting from the beginning vertex, to reach the destination vertex. However, given that ACO algorithms focus heavily on manipulating the edges of the graph, we chose to represent candidate solutions as *arrays of edges*. Therefore, a candidate solution in our implementation is represented as simply the list of edges traversed in the path, each with its associated values for weight and pheromone (which are used to guide the probabilistic search while building solutions).

The red and green paths in Figure 1, for example, could be represented in our modeling, respectively, as $\{A \rightarrow B, B \rightarrow D, D \rightarrow E, E \rightarrow F\}$ and $\{A \rightarrow B, B \rightarrow C, C \rightarrow F\}$.

2.2 Pheromones

A key element of ACO algorithms is the concept of pheromones. While in typical weighted graphs edges are associated only with a weight value, in our case each edge in the graph also has a *pheromone* value. This value is used to introduce bias into the process of probabilistically building candidate solutions.

While the weight of each edge remains constant in our graph, pheromones are iteratively modified in a feedback process. After each iteration of the algorithm, all the candidate solutions built by each agent are evaluated. Then, the solution whose fitness value is the largest is selected. Each of the edges in this solution then have their pheromone values increased, proportionally to the fitness of the solution itself.

After the process of updating the pheromones, our algorithm goes through an evaporation step. In this step, the pheromone values for *all* edges in the graph are reduced by a percentage amount, which is controlled by an input parameter.

Therefore, edges in our graph carry two pieces of data:

- Weight - An integer value in the interval $[0, 10]$.
- Pheromone - A real value in the interval $[0, \infty]$

Note that we do not limit the value of pheromones to any specific range. This is one of the big design decisions made in our implementation. This unbounded nature of the pheromone values, combined with the evaporation mechanic, have two major implications. First, it is possible that after multiple evaporation steps an edge's pheromone value approaches zero. This will cause its probability of being included in new solutions to also tend to zero, which effectively eliminates edges from the graph from ever being considered. Second, it is also possible for a particular edge's pheromone value to grow drastically large, making it overwhelmingly more likely to be included in future solutions than any other competing edge. Both of these behaviours were deemed desirable in our implementation, so as to achieve a reasonable balance between convergence and exploration of the search space.

2.3 Choosing Vertices

During the execution of the algorithm, each agent builds its solution incrementally, by choosing to add a vertex to its candidate solution until eventually reaching the destination vertex. Therefore, the process through which edges are chosen to be added to a solution is essential to the quality of the solutions being built.

Our algorithm follows a simple probabilistic formula. Given a current vertex, all of its outgoing edges to vertices that are not yet in the solution are evaluated, being assigned a value equal to its weight multiplied by its current pheromone amount. Then, each edge is assigned a probability proportional to its value over the aggregated values of all valid outgoing edges from the current vertex. This formula can be formalized as follows:

$$p(e) = \frac{w(e) \times ph(e)}{\sum_k w(k) \times ph(k)}$$

Where $p(e)$ is the probability of edge e being chosen, $w(e)$ is the integer weight associated with edge e, $ph(e)$ is the pheromone value associated with edge e and $\sum_k$ is the sum of all k outgoing edges of the current vertex whose endpoint is not currently in the candidate solution. We chose this formula because while simple, it implements the two most important factors in our search: taking an edge's weight into account helps to favor promising edges given their larger contribution to the total cost of a solution, and weighting this value with the pheromones allows bias to be retroactively introduced into the search procedure, which eventually favors edges that may not necessarily be locally optimal but that eventually lead to good global solutions.

2.4 Evaluating Solutions

The structure of the longest path problem itself already gives us a very simple function to evaluate the quality of a solution. The fitness of a candidate solution is defined as simply the aggregated weight of all the edges that are part of it. Note that it is possible that invalid solutions may be built, if a search eventually gets to a point where the destination vertex is no longer reachable. In these cases, we simply assign the solution's fitness to zero.

Evaluating solutions is important not only to determine which candidate will be chosen as the best solution, but also in the feedback mechanism of updating pheromones. In our implementation, in each iteration of the algorithm the solution with the largest fitness value is chosen for pheromone update. Then, each of the edges that make up the solution have their pheromone values increased. The magnitude of this increase is directly proportional to the fitness of the solution, and uses the following formula:

$$\Delta_{ph}(e) = 1 - \frac{1}{\sum_k w(k)}$$

Where Δ_{ph} is the amount of pheromones to be added, and $\sum_k w(k)$ is the sum of the weights of all the edges in the solution (that is, its fitness). Therefore, each edge in the candidate solution receives an equal amount of bias for future iterations.

3 Implementation

The proposed algorithm was implemented fully in C. The main components of the implementation include the graph data structure, the handling of input/output, and the implementation of the ACO algorithm logic. The following sections give an overview of the major factors in the implementation.

3.1 Graph

The graph data structure is fundamental in any ACO implementation, given that it encapsulates how the problem itself is modeled as well as important parameters in the ACO logic, such as the pheromone value for each edge. In our implementation, a graph is composed of the following members:

- **Size** - Stores the number of vertices in the graph.
- **Vertex**[] - Array of vertices representing each of the vertices in the graph.

Vertices store the following data:

- **Neighbours** - Stores the number of adjacent vertices in the graph.
- **Edge**[] - Array of edges representing outgoing edges from the vertex.

Finally, edges consist of the following members:

- **ID** - Stores the ID of the vertex on the endpoint of the edge.
- **Weight** - Integer weight of the edge on the input graph.
- **Pheromone** - Pheromone value used in the ACO algorithm.

3.2 Ant Colony Optimization

The logic of the Ant Colony Optimization algorithm itself was implemented separately. The main aspect of the ACO's code is the structure of an independent agent, or ant. Ants are made up of the following members:

- **Cost** - Stores the fitness of the current candidate solution.
- **Length** - Stores the number of edges in the current solution.
- **Edge**[] - Stores a reference to each edge currently in the solution.

The major functions that implement the algorithm's logic are the following:

- **aco_longest**(graph, decay_ratio, num_ants, num_it) - Performs an execution of the ACO algorithm on the given graph with the parameters provided.
- **build_solution**(ant, graph) - Builds a candidate solution for the given ant for the graph provided.
- **init_pop**(graph, num_ants) - Initializes a population of ants with initially empty solutions.

4 Compiling and Executing

4.1 Folder Structure

All of the code for the project is available in the root folder. Below is the list of files and a brief description of each:

- graph.c/h - Implement the graph data structure and its interface.
- aco.c/h - Implement the Ant Colony Optimization algorithm and its interface.
- main.c - Processes input and kickstarts algorithm execution.
- run.sh - Bash script for running the algorithm with a few default parameters.
- Makefile - Defines compilation targets for the project.

4.2 Compiling

A Makefile is provided within the root folder. Therefore, to compile the program one can simply run "make". A binary called "aco" should be generated in the same folder, which can then be used to run the algorithm.

4.3 Running

To run the program, the command format is:

```
$ ./aco <input_file> <decay_rate> <num_ants> <num_it>
<seed>
```

Where the parameters are:

- input_file - Input file with graph data.
- decay_rate - Rate of pheromone decay per iteration. Range: $[0.0, 1.0]$
- num_ants - Number of ants in the population.
- num_it - Number of iterations the algorithm should run for.
- seed - Random number generator seed.

The Makefile provided includes a target for running the program. To do so, one can simply type "make run" after having compiled the program. The parameters used for these runs are to reproduce the very last experiment described in this report, where the largest graph was tested with 1000 iterations, for a varying number of ants. The program should run once for each random number generator seed used. Also provided is a bash script called "run.sh", which contains the parameters for all parameter tuning experiments performed, in order. Running this script should run all of the experiments described in subsection 5.2.

4.4 Output

The program outputs to the standard output (stdout). It prints the statistics for the evaluation of each iteration. These include average fitness, worst fitness, best fitness, and the value of the global best solution found so far.

5 Experimentation

5.1 Methodology

All the experiments were run using some of the three datasets provided, which are composed of graphs of varying sizes. The hardware used features an Intel i5 4670K CPU at 3.8 GHz, with 16 GB of RAM. The program however was not parallelized, and therefore runs in only one core. The software used was Ubuntu 14.

For each experiment, ten instances of the execution with the same parameters were performed, each using a different seed for the random number generator. Then, the values reported for each execution are the mean amongst all executions combined. While only ten executions may seem like a small number, the standard deviation for all reported metrics was computed and found to be reasonably small, which justifies the fewer number of runs, since extra executions would be unlikely to reduce this much further.

For the largest graph in the dataset, only five executions were performed instead, due to the long running time of each run.

5.2 Parameter Choice

The goal of the first stage of experimentation was to empirically choose the best parameters for execution. To this end, several parameters were tested in order of seeming impact on algorithm performance. We chose to experiment with the datasets provided in order of size. The logic behind this decision is that experimenting with smaller graphs can provide a fast way to propose hypotheses and choose a few initial parameters, whereas experimenting with the bigger graphs takes more time but can help corroborate or refute the hypotheses put forth, and adjust the validity of the parameters chosen.

5.2.1 Experimenting with Graph 2

Following the logic explained in the previous section, we began our experiments by using the smallest graph in the dataset, which contains only 20 vertices and 190 edges. The fact that we know the value for the longest path in this graph (168) as well as its relatively small size made it a strong candidate for testing the general workings of the algorithm and how the main parameters affect the execution and the quality of the solutions generated.

The main aspects to be tested in this stage were how varying the number of ants in the population affects the quality of the solutions generated and the convergence of the algorithm itself. Therefore, we fixed a number of iterations and decay rate that seemed sufficient for the algorithm to converge to reasonable values. These were chosen as **100** iterations and **0.05** as the decay rate. Our hypothesis before the experimentation follows the intuition that increasing the number of ants would theoretically provide more diversity in initial solutions, allowing the algorithm to perform a wider exploration step, thus making it able to converge to better solutions overall. Similarly we speculated that a larger number of ants would increase the probability of better solutions being built in each generation, thus increasing the probability of repeating edges in solutions, which would favor earlier convergence.

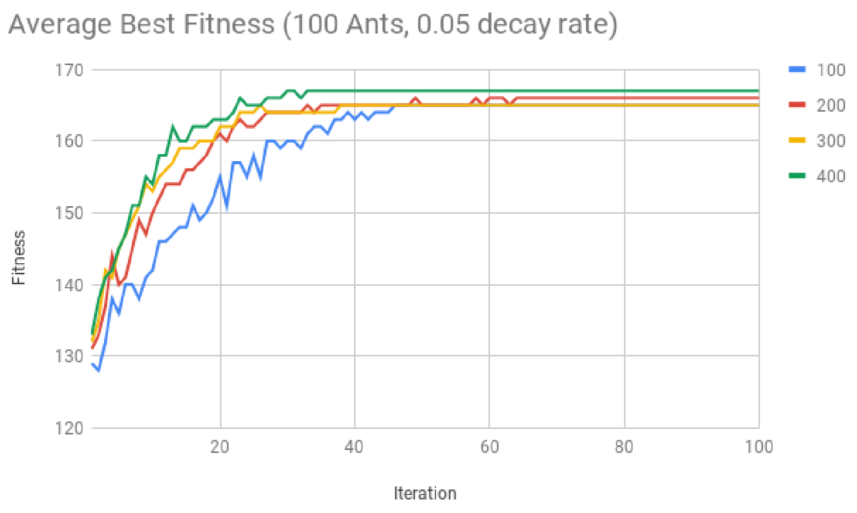

Figure 2: Average best fitness for 100 iterations and 0.05 decay rate, with varying number of ants.

Figure 2 shows the result of this experiment. Observing the data, one can conclude that, indeed, the average best solution gets progressively better as the number of ant increases. While for a small number of ants early solutions are consistently worse and reach values of around 165, for larger numbers of ants the solutions improve much faster, and for the case of the largest number of ants (400), they eventually converge to the optimal solution. The number of ants also clearly affects the convergence rate of the algorithm, with larger number of ants converging in earlier iterations. Both of these observations corroborate our initial hypotheses.

However, given the very small size of the graph in question, it is possible that the results of any thorough experimentation performed on it would not necessarily generalize to larger samples. Therefore, we chose to perform more thorough analyses for other datasets.

5.2.2 Graph 1: The Chosen One

Our second experimentation stage was performed on the first graph in the dataset, which contains 100 vertices and 8020 edges. This graph's medium size and relative density made it the ideal candidate for the most thorough of our experimentation stages: it is large enough for results to be likely to generalize better, yet small enough for execution time to not be a major hindrance. Therefore, we spent the bulk of our experimentation effort generating and analysing data for this graph.

There are essentially two major parameters we needed to experiment with. Given that the number of iterations must simply be large enough to allow the algorithm to converge, the main parameters that actually control the behaviour of the algorithm itself are the pheromone decay rate and the number of ants. However, these parameters are so intertwined that it is unlikely that we could simply fixate one while tuning the other. Our results would likely be biased by the fixation of the parameter. Therefore, we chose to perform a wide analysis of how both these parameters relate to each other. To this end, we chose a number of iterations large enough for the experiments to converge for the vast majority of the executions, and varied the other parameters simultaneously.

The values chosen were decay rates $\{0.01, 0.05, 0.1, 0.3, 0.5\}$ and numbers of ants $\{100, 400, 700, 1000\}$, which gives us a total of 20 parameter combinations. This experiment concerned two major contention points. First, whether the results obtained for the first graph remained consistent when scaling up to larger graphs. That is, does increasing the number of ants still consistently increase the quality of the solution and speed up convergence? Second, we now needed to analyse how the pheromone decay rate affects the behaviour of the algorithm, seeing as we kept its value constant in our first experiment.

Note that the data we show is for best fitness for each iteration, not best global candidate solution, therefore, it is possible for these values to fluctuate, particularly while the algorithm is still in its exploration stage.

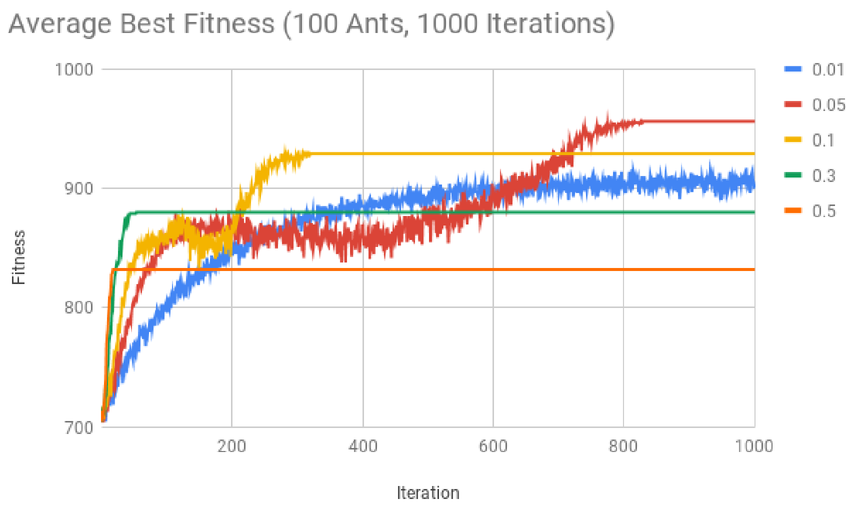

Figure 3: Average best fitness for 100 ants and 1000 iterations, for varying decay rates.

Figure 3 shows the result for a value of 100 ants, with all variations of the decay rate. Interestingly, the larger amount of data allows us to very clearly observe the stages of the algorithm, with the exploration stage showing a pattern of fluctuating results, while the values flatline once convergence is reached.

The easiest conclusion we can reach from this data is that the decay rate clearly seems to favor earlier convergence. While for a value of 0.01 the algorithm does not converge after 1000 iterations, it begins to do so with a value of 0.05, after roughly 800 iterations. For larger values, the algorithm converges progressively earlier, reaching convergence in a staggering 40 iterations for a value of 0.5.

Another conclusion regards the quality of the solutions in relation to the decay rate chosen. Clearly, larger values of decay rate cause the algorithm to converge to much worse solutions, as can be seen for the curves for rates of 0.3 and 0.5. This is most likely due to the algorithm exploring for very little time before entering an exploitation stage, where it can only improve the current best solutions marginally before plateauing.

However, decreasing this rate indefinitely does not always improve results. As one can see for the rate of 0.01, while the algorithm does not converge, it also seems to not be improving solutions any longer, instead fluctuating around a fitness value of 900. The 0.05 decay rate, on the other hand, seems to provide an ideal balance between exploration and exploitation, allowing the algorithm to search a wide region of the search space, before improving solutions gradually up to a maximum fitness value of 956, which is reasonably close to the optimal value of 990.

Lastly, this experiment also helps us observe the importance of convergence. If one were to analyse the behaviour of the algorithm for only 400 iterations, for instance, one could wrongly conclude that a ratio of 0.1 provides better performance, as it clearly has converged to a value much higher than all other rates. However, while this remains true for several hundred iterations, the superior exploration/exploitation balance of the 0.05 rate allows it to gradually improve solutions until it overtakes the previous one. Thus, it is important when comparing parameters that we attempt as much as possible to have all executions converge, so as to perform a fairer comparison.

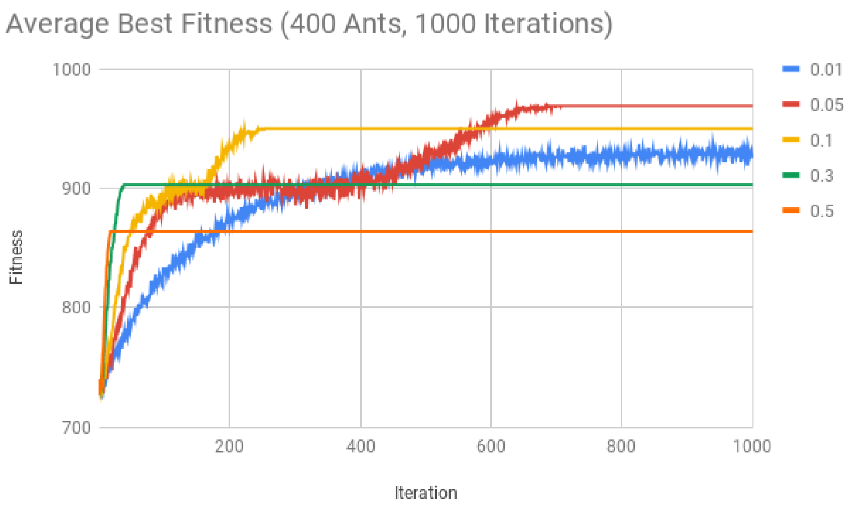

Figure 4: Average best fitness for 400 ants and 1000 iterations, for varying decay rates.

Figure 4 shows the same results but for a value of 400 ants. While the overall behaviour of the varying decay rates seems to remain consistent for a larger number of ants, there are several hypotheses we can raise by comparing the two graphs.

First of all, the number of ants clearly seems to have an effect on the rate of convergence. Whereas for a rate of 0.05 the algorithm initially took over 800 iterations to converge with 100 ants, for 400 ants it converges around the 700th iteration. This is consistent with the conclusions we had reached in section 5.2.1. While harder to visualise, the data is also consistent with our second conclusion from that experiment: the solutions are on average of better quality for larger number of ants. While the best solution for 100 ants had a fitness value of 956, the best solution in this experiment had a fitness of 969, even closer to the optimal value.

Lastly, one can observe that the number of ants also affects the *variance* of the solutions in each iteration. From analysing the exploration stages of all curves, it is clear to see that the values fluctuate much more for 100 ants than they do for 400. This intuitively makes sense. Since we are generating a larger number of solutions, the probability of good solutions appearing in every iteration increases, which when selecting only the best solution from each iteration will cause this value to vary to a lesser degree. This is a case of the classic observation in statistics that larger sample sizes lead to smaller variance.

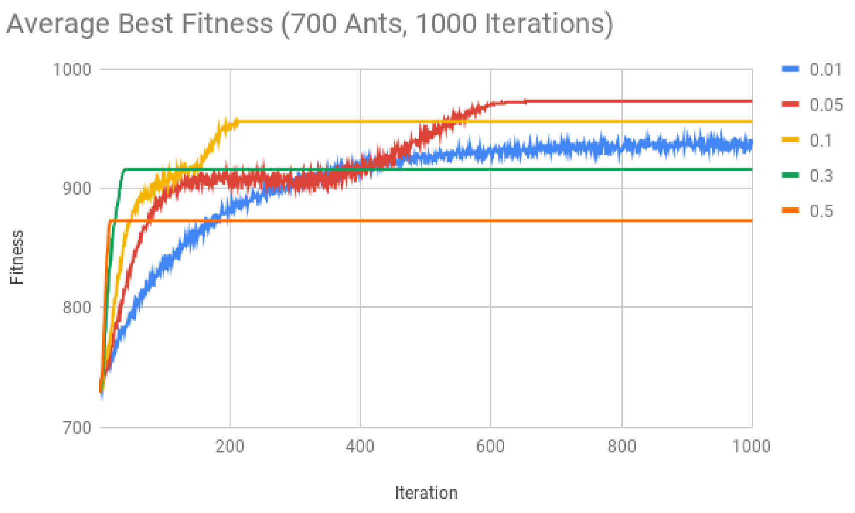

Figure 5: Average best fitness for 700 ants and 1000 iterations, for varying decay rates.

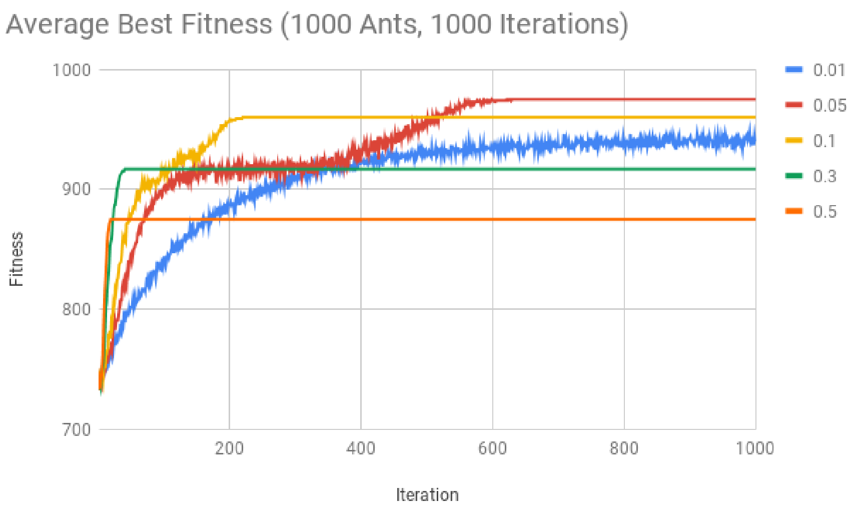

Figure 6: Average best fitness for 1000 ants and 1000 iterations, for varying decay rates.

Figures 5 and 6 show the results for 700 and 1000 ants, respectively. There is not much to learn from these experiments that has not yet been observed. Clearly the conclusions from the previous experiments remain consistent, with solutions becoming progressively better and less varied with each increase in the number of ants. However, the effects of these seem to be reaching diminishing returns. The value of the best solution, for example, improves from 969 to 973 from 400 to 700 ants, and then to 975 for 1000 ants. While better and very close to the optimal solution, the rate of improvement is clearly slowing down. This is a strong indicator that increasing the number of ants any further would not bring much benefit to the algorithm's performance.

To better facilitate the observation of the difference in solution quality amongst these differing values for number of ants, we organized the data for the decay rate with the best results (0.05), and for each value of number of ants calculated how the gap between the best solution in each iteration and the optimal solution varies across all iterations. Figure 7 shows this data. From this graph it is clear to see that larger numbers of ants not only significantly improve the solution towards which the algorithm eventually converges, but also improve the overall quality of the solutions reached for most stages of the execution.

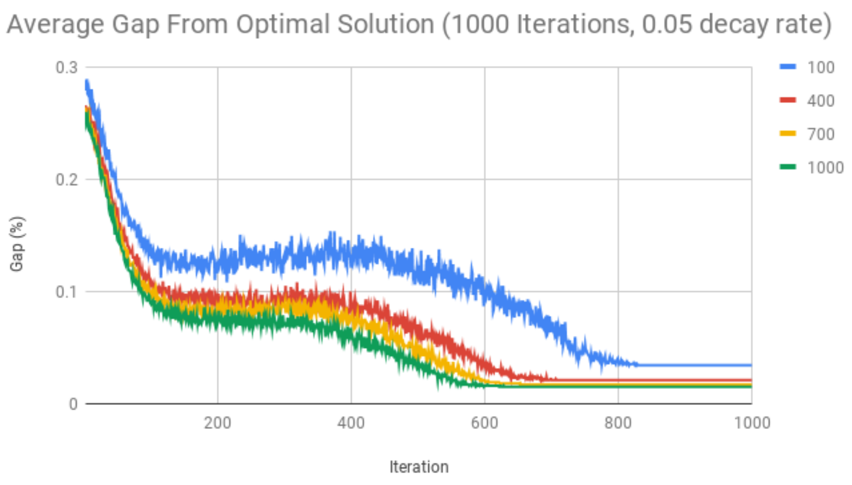

Figure 7: Percentage gap between average best solution in iteration and the optimal solution.

From these experiments, we concluded that the ideal values that are more likely to produce good results for really large graphs are a decay rate of 0.05 and 1000 ants. As to the number of iterations, one could assume that 700 iterations could be a better value, given that for our parameters the algorithm already converges by then. However, this is not likely to remain the case for larger graphs which consequently have larger solution spaces to explore. Therefore, we chose 1000 iterations as the ideal number as well.

5.2.3 Graph 3: The Behemoth

The third graph in the dataset was massively larger than the other two, and was particularly dense, with a ratio of edges to vertices much larger than the other two. Therefore, the running time of the algorithm for this graph was also understandably much longer, which made experimenting with it a little more challenging. Our goal with our experimentation stage for this graph was simply to check whether our hypotheses established in the earlier stages would hold when scaling up to graphs of significant size. To this end, we kept the values of

1000 iterations and decay rate of 0.05. However, we were still interested in how a varying number of ants affects execution, so we varied this parameter for the values determined beforehand.

Additionally, given that we were not sure whether the algorithm would converge for this much larger dataset, we show the data not only for best solution in each iteration but also for the worst and average solution in each iteration, as well as the fitness of the best global solution found in each step of the algorithm. We stress that these values all eventually converged to one single value in earlier experiments, which is why we chose to omit them.

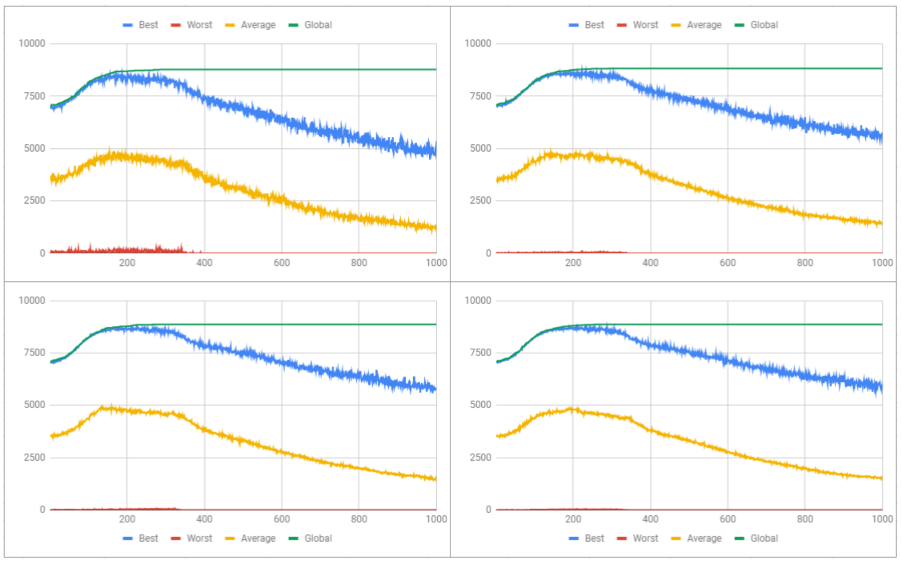

Figure 8: Best, worst, average and global best average fitness for graph 3 with varying number of ants and 1000 iterations.

Figure 8 shows the result of this experiment for 100 ants (top left), 400 ants (top right), 700 ants (bottom left) and 1000 ants (top right). Our earlier hypotheses for how the number of ants affects solutions remains consistent for this experiment. Clearly the variance for both best and worst (and consequently also average) solution diminishes as the number of ants increases. Also, while not quite as easy to visualise in the graphs, the value of the best global solution also increases, though this improvement, from 8776 for 100 ants to 8866 for 1000 ants is not quite as impressive at this scale. Therefore, interestingly enough, a smaller number of ants could theoretically still provide reasonably satisfying results for larger graphs.

However, the most striking result of this experiment is the fact that the algorithm does not converge at all. Not only this, but after an initial improvement stage where solutions gradually get better, it eventually reaches a point where solutions are not being improved but rather worsened. This behaviour also remains consistent up to the final iteration of the algorithm. We attempted to investigate why this is the case and what could cause the algorithm's behaviour to change in such a way given only a larger input, but could not reach a conclusive explanation. Unfortunately, the algorithm seems to simply not generalize well for larger inputs, and we do not know why.

6 Conclusions

In this work, we proposed an implementation of an Ant Colony Optimization algorithm towards solving the Longest Simple Weighted Path problem in graphs. After describing our modeling and implementation, we performed a thorough process of experimentation, and provided an analysis of the results reached. The main hypotheses we were able to corroborate were the fact that larger decay rates in our algorithm favor earlier convergence, but also tend to worsen the quality of solutions built. Rates that are too low, on the other hand, may cause the algorithm to never converge and instead simply explore suboptimal regions of the solution space indefinitely. Therefore, the decay rate must be chosen with care with a value that balances these two behaviours out. We also found evidence that increasing the number of agents progressively improves the quality of solutions, as well as causes the algorithm to converge earlier. However, these effects have diminishing returns, so increasing this parameter further is unlikely to cause much difference. Finally, we observed that for large datasets the algorithm's behaviour changes dramatically, with convergence not being reached and solution quality actually decreasing over time. Unfortunately, we did not have time to investigate this phenomenon deeply enough to reach any conclusive theory as to why this is the case, so it remains a mystery for Scooby-Doo and his gang to solve.

References

[1] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, Introduction to Algorithms, Third Edition, The MIT Press, 2009.

[2] Dorigo, Marco and Birattari, Mauro and Sammut, Claude, Ant Colony Optimization, Encyclopedia of Machine Learning, Springer US, 2010.