ACO vs PSO - An analysis on the Traveling Salesman Problem

Francisco Santos, Leonardo Vieira, Pedro Carvalho
2015238068, 2015236155, 2015232484
Departament Of Informatics Engineering
Faculty of Science and Technology
University Of Coimbra
Evolutionary Computation 2018/2019
fjrsantos@student.dei.uc.pt, lmavieira@student.dei.uc.pt, pfcarvalho@student.dei.uc.pt

1 Brief Introduction

The traveling salesman problem has been extensively studied and is one of the most popular NP-Hard problems. The difficulty of such problems increases drastically as the problem becomes more complex, this means that there is no algorithm that can find the best solution for very difficult cases of TSP in useful time. The answer to this problem lies in approximations. While it is not (yet) possible to find the best solution in a reasonable amount of time we can use algorithms that can reach solutions that are close to the best one. These algorithms cannot guarantee that their solution is the best but they run much faster than exact ones. In this assignment we are proposed to perform a study in the performance of two swarm intelligence algorithms discussed in class. Later on this report, metrics about the multiple runs made are going to be studied and analyzed in order to reach a final conclusion. While our focus will be comparing the two proposed algorithms we will also try to compare our results with the state of the art whenever possible.

2 Problem Description

Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) are the two swarm intelligence algorithms we will be tackling for this project. The *No Free Lunch Theorem* tells us that there's not an algorithm that's optimal for every problem, however, for specific problems, certain algorithms may perform better than others. As such, our goal is to compare the ACO and the PSO algorithm for the TSP problem. Three instances of the problem with different difficulties were selected for the experiment and several different sets of parameters combination were tested. It is important to note that the TSP problem is a permutation problem, and as such, some algorithms like the PSO might have to suffer some adaptation.

2.1 Traveling Salesman Problem

The Traveling Salesman Problem is a classical problem. A salesman must visit one and only one set of cities, starting in one city and returning to that same city minimizing a certain criterion (distance in our case). There is a dynamic variant of this problem where the distance between cites may changes during each run, however we will not study this variant.

The difficulty of a TSP problem increases with the number of cities there are in the said problem. We were proposed to study 3 different TSP problems, one in a small map, one in a medium sized map, and one in a large map.

We looked over some TSP problems, mostly on the *TSPLIB95* but also on other sources and selected those who best fitted our needs. The problems are:

- *Berlin52* - 52 nodes [1]
- *kroA100* - 100 nodes [1]
- *CH150* - 150 nodes [1]

We chose a very simple representation to model our solutions. Our algorithm will be working with lists that identify the order the node should be visited in. This solution is based on *permutations*. We will talk about how each algorithm will work around our chosen representation in their specific sections.

2.2 Ant Colony Optimization

Ant Colony Optimization is a multi-agent algorithm to solve path finding problems. Each agent is called an "ant", and just like real ants, pheromone-based communication is used in the most paradigms of this algorithm. The simulated ants, replicate the real ones by leaving a trail of pheromones, this trail attracts other ants that are wondering in the search space. The pheromone intensity is inversely proportional to the distance that the ant traveled. On top of that, the pheromone evaporate as iterations go by to simulate the concept of erosion. In a practical sense this will help avoid a premature conversion, if there were no pheromone evaporation, the trail left by the first ants would excessively attract the others.

Algorithm Implementation

[2] The ACO implementation uses a multi-threaded version in python 3.6. It uses the default mathematical formulas for the core operations of the algorithm. The general structure of the algorithm goes as following:

Initialization of the map, ants and pheromones. For each iteration of the algorithm, each ant makes a move in its tour, the shortest path so far is updated and the pheromones in each edge are updated. This process repeats a predefined number of times (in our case the stop condition is **time** in order to compare both algorithms).

The pheromones are updated with a delta value in each iteration, as said, inversely proportional to the distance already traveled, so delta is equal to pheromone constant divided by

the current path distance. At each iteration, when the ant has to decide a new destination, it picks a based with cumulative probability using the attractiveness rate of each new location. This rate is calculated using a the level of pheromone and the size of each possible way. This values are then weighted using an alpha and a beta algorithm parameter. In this assignment, the euclidean method was used to calculate the distance metric.

In this algorithm, multiple structures are used. *Nodes* is a dictionary structure with the initial map, this structure will then be used by the distance function to create and update a *distance_matrix* which is populated with the distance between cities. On top of that there's a *pheromone_map* matrix and a *ant_updated_pheromone_map* that have the pheromone level for each edge and values to add to each edge in the next iteration, respectively.

The mathematical formulas used in the algorithm are the following:

Pheromone evaporation update

$$(1 - \rho) \times \tau_{ij}(t) \tag{1}$$

Pheromone deposit update

$$\tau_{ij}(t) = \sum \Delta\tau_{ij}^{k}(t) \tag{2}$$

- τ_{ij} Pheromone value of the edge from node i to j

Ant transition rule

As it was said above, some operators are parameters that can vary in order to create new configurations. Those are:

- Population Size
- Runtime of the algorithm
- Alpha value - used to control the influence of pheromone in the new city choice
- Beta value - used to control the influence of the distance in the new city choice
- Pheromone Evaporation Constant

The representation we proposed for the problem ends up being an auxiliary structure of the algorithm (each ant must keep a list of the visited nodes in order to choose its next step) so the algorithm did not suffer any alterations to accommodate our decisions.

2.3 Particle Swarm Optimization

Particle Swarm Optimization is a method in the swarm intelligence branch that tries to optimize a solution iterating and improving a candidate solution. The algorithm holds multiple candidate solutions at the same moment, here called particles. These particles can move in the search space, as such, each particle has a unique position, velocity and moves according the best known position and global best. Improved versions that take in consideration neighbourhoods are also possible, but will not take part in our study. As our previous algorithm, this one is also susceptible to finding a good solution that is not the best one.

Algorithm Implementation

The PSO algorithm implementation was adapted to our permutation problem [4] from an open source PSO implementation in python for float problems [3]. Although multiple improvements based on particle swarm are possible, we only used the vanilla implementation, as we do in the ACO algorithm. The basic structure goes as following:

Initialize candidate solutions (particles) and compute the best one. Then evaluate and update each particle velocity and position (solution values in our case), and update the best value globally (social component) and the previous best solution of each particle (cognitive component). This process repeats until a stop condition occurs (like in ACO, time was our stop condition). The particle movement is determined by the velocity which, in turn, is determined by its social and cognitive components. Each component has different weights defined as parameters.

The mathematical formulas used in the algorithm are the following:

Velocity update formula

$$\begin{aligned}\vec{v}(t+1) = \frac{\omega}{e_p} \times \vec{x}_i(t) + r_1 \times \frac{\phi_1}{e_c} \times (\vec{p}_i - \vec{x}_i(t)) \\ + r_2 \times \frac{\phi_2}{e_s} \times (\vec{p}_g(t) - \vec{x}_i(t))\end{aligned} \tag{3}$$

- $\vec{v}$ - particle velocity
- ω - personal weight
- e_p - personal error
- e_c - cognitive error
- e_s - social error
- $r_1 r_2$ - random values between 0 and 1
- ϕ_1 - cognitive weight
- ϕ_2 - social weight
- $\vec{p}_i$ - personal best position
- $\vec{p}_g$ - global best position
- $\vec{x}_i$ - current personal position

As it was said above, some operators are parameter that can vary in order to create new configurations. Those are:

- Weight value - used to control the influence of the previous velocity when updating a particle
- Cognitive value - used to control the influence of the particle best solution, when updating a particle
- Social Value - used to control the influence of the global best solution, when updating a particle

Adapting PSO to permutations problem

The PSO algorithm holds the information about the particles in a list with particle structure instances, each having information regarding a candidate solution. In this structure there are also 3 *boolean arrays* structures, a *personal mask*, a *cognitive mask* and a *social mask*. These masks are used by each particle to indicate which slices of solution are selected from each component (personal, cognitive and social) and then added to the spliced together to form a new solution.

There are some additional masks that are used to facilitate the creation of valid solutions. An availability mask is created each time a particle is updated, this mask contains information about the nodes that are not yet in the new solution. Performing an *and* operation between this mask and the boolean array of each component (personal, cognitive and social) results in a new auxiliary mask. This auxiliary mask tells us which of the available nodes the component mask intends to use. Performing a *xor* operation between this new mask and the availability mask updates the availability mask to show only the cities that are available after choosing the cities from each component.

2.4 Parameters Assumptions

Since we had little to no experience working with these algorithms before we did some undocumented experiences before-hand to gain some intuition on what adequate values for each parameter might look like. In order to find these values we did some test runs where we fixed all parameters except one and experimented with different orders of magnitude.

2.5 Experimental Setup

To perform this experiment we measured a few metrics in both algorithms. Important to note both of the algorithms return the same metrics for the sake of comparison between them.

The metrics gathered were the following ones:

- Time to initialize population.
- Best individual (Solution)
- Best individual (Fitness)
- Average Fitness
- Number of iterations

In the ACO case, where populations are initialized in each iteration, the time to initialize population metric is the sum of all the initialization in all iterations.

We expect to see a premature conversion towards a local optima in the PSO. This is a known issue with the base version of the algorithm since there is no way a particle would tend explore an completely separate and unexplored part of the search space (global search). This could be solved with some improvements like introducing random immigrants in the implementations or implement and adaptive particle swarm optimization (*APSO*) that can perform both global and local search.

A test script was developed where multiple sets of parameters were tested and reported into an *.csv* files.

2.6 Experimented Parameters

We fixed the population size at 100 individuals for both algorithms and for each map tested, each algorithm will have 15 seconds for the smallest map, 30 seconds for the medium sized map and 45 seconds for the largest map. The values tested for each algorithms are described in the following sections. The reason we fixed these values was so we could make comparisons between the algorithms on an even playing field.

PSO

Regarding PSO's parameters, the weight values tested were 0.6 and 0.9 (we chose leaning toward the upper bound of the range to try and fight early convergence). The cognitive and social values were tested in pairs, these pairs being (1,3), (2,2) and (3,1) for both parameters respectively as we found some recommendations that they should sum up to 4. [8]

ACO

The ACO parameters evaluated were the following:

- Alpha - 1 and 5
- Beta - 2.5 and 5
- Pheromone Evaporation Coefficient - 0.3 and 0.6

Since we did not have as many expectations of the results for ACO as we had for PSO these parameters were the ones that seemed the most promising during our undocumented tests.

3 Experiment Results

3.1 *ACO*

There is a relevant detail about our results using the ACO algorithm. The method we used to calculate the distance yielded different results from the ones present in the solutions to the chosen problems. In order to make meaningful comparisons we reevaluated our best individuals using the method utilized to get the values for the optimal solutions. The recalculated values are marked with an asterisk.

Map	Alpha	Beta	Evaporation Coefficient	Average Best Fitness	Optimal Solution
Small	1	2.5	0.3	8154.03*	7544
Medium	1	5	0.6	24387.07*	21282
Large	1	5	0.3	7959.24*	6528

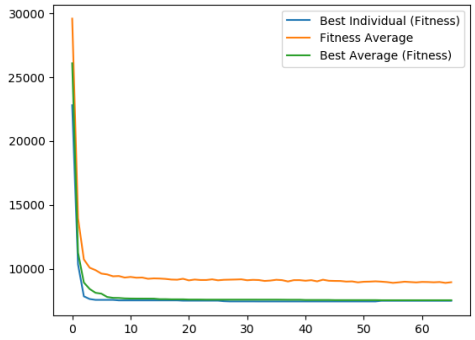

Figure 1: Best ACO on small map

3.2 *PSO*

Map	Weight	Cognitive	Social	Average Best Fitness	Optimal Solution
Small	0.6	1	3	18007.78	7544
Medium	0.6	1	3	127559.77	21282
Large	0.6	1	3	42416.26	6528

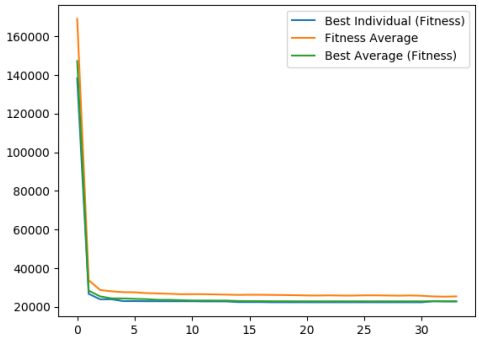

Figure 2: Best ACO on medium map

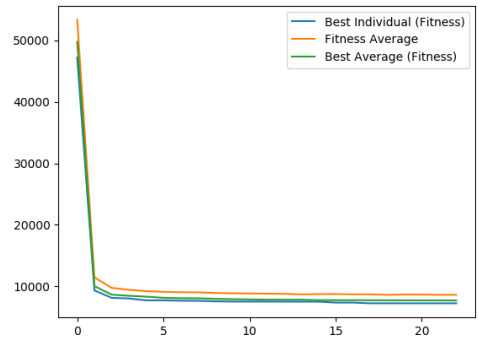

Figure 3: Best ACO on large map

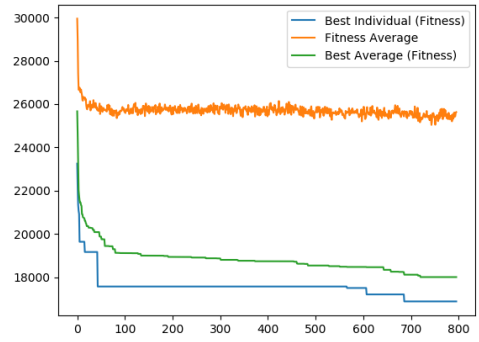

Figure 4: Best PSO on small map

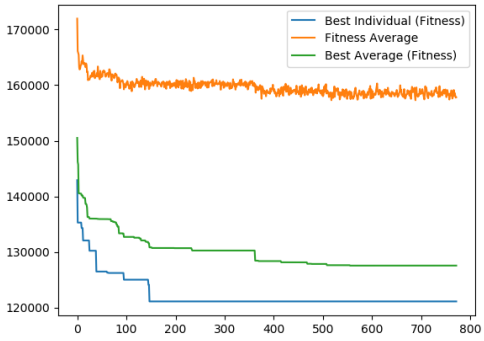

Figure 5: Best PSO on medium map

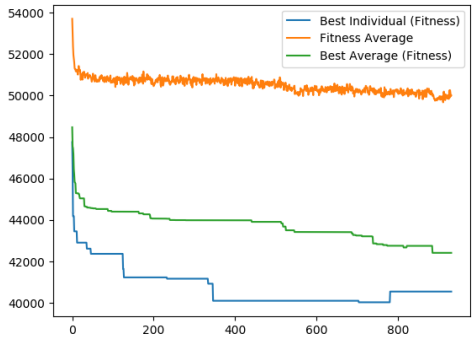

Figure 6: Best PSO on large map

3.3 Results and Discussion

We posed the following hypothesis based on our observation of the data: "Is the average of the best individuals in the last generation for ACO better than the average of the best individuals in the last generation for PSO, given our experimental setup?". In order to answer this questions we posed 6 null hypotheses that need to be rejected:

- h_0 - "Average Best PSO = Average Best ACO, on small map"
- h_1 - "Average Best PSO = Average Best ACO, on medium map"
- h_2 - "Average Best PSO = Average Best ACO, on large map"
- h_3 - "Average Best PSO <Average Best ACO, on small map"
- h_4 - "Average Best PSO <Average Best ACO, on medium map"
- h_5 - "Average Best PSO <Average Best ACO, on large map"

We were able to reject all of these hypotheses with $p < 0.01$. This analysis can be found in detail in *statistical_analysis.py*. It is interesting to note that PSO seems to not only has a worse performance overall (for the same computational resources) but also deals with the increasing complexity very poorly. This can be observed as the average best fitness is worse for the Large map than the Small one, even though the optimal fitness is lower for the Large map. This is specially meaningful as it disproves hypothesis such as "PSO should be used over ACO if the problem has an overwhelming complexity and the computational resources are limited", looking at the scenario explained above we find that there is no evidence to support this type of claim, as PSO seems to be very ineffective in navigating complex problems.
Another interesting insight is that the ideal parameters we found for PSO were the same for every map. Not only that, these parameters seem to geared towards exploitation over exploration (lowest possible Weight, highest possible Social component). This might be because the exploration mechanisms of PSO are naturally weak, which favors effective

exploitation over mediocre exploration. It is also possible that we were misguided in our initial assessment of the parameters and that ended up hurting the algorithm's results. We also did some experiments with additional mechanisms to improve performance for PSO (Random Immigrants and Predator-Prey model) but we were not able to run enough tests for a proper analysis (the results, however, were not very promising). ACO performed much better than we anticipated, our first impressions were that the algorithm was too slow to compete with PSO with the same computational resources, note that since we used run time as our stopping condition ACO sometimes runs as little as less than a 10th of the iterations of PSO (for example, ACO runs between 20-30 iterations for the large map with PSO runs 800+). The results show that, contrary to our initial expectations, ACO is able to convincingly outperform PSO from the very first iteration. The reason behind this is that ACO is able to find improvements (and significant ones) in almost every iteration while PSO lingers in local optima for large amounts of time.
The best parameters found for ACO are also noteworthy. As the complexity of the problem grows, Beta started becoming more important, a higher Beta leads to a slower conversion so we found it interesting that this matched up with our expectation. It is interesting, however that the Evaporation Coefficient (that can be increased to slow down conversion) decreased from map Medium to Large, but since this only happened after the recalculations mentioned in 3.1 we believe this odd behaviour is due to the effect of the different methods of calculating distance.

3.4 Conclusion

In this work we learned a lot about the flexibility and adaptability of swarm intelligence algorithms for problems outside of the scope we first encountered them in. We also found that in our specific experimental setup ACO performed better universally, proving to be better equipped for the task at hand. If we had more time (or computational resources) we would expand this work in several ways:

- Test a wider range of parameters, our assumptions are serviceable (as shown by some of the good results we found) but are far from the ideal when trying to test the limits of these algorithms. Ideally we would use another Evolutionary Algorithm to evolve the parameters for each algorithm for each problem.
- Implement additional mechanisms, both of these algorithms have many variations and *add-ons* to compensate for their shortcomings. Although we explored these ideas for PSO we did not have the computational power to explore them properly on time.
- Explore additional representations for PSO. After we had already finished our initial implementation we learned of an alternative representation for PSO that we would've liked to explore and compare with the current one.

4 User Manual

In regards to the utilization of this program, there are two main modes that correspond to the two different algorithms implemented.

To run ACO the following should be written in a console containing Python version 3:

```
$ python3 main.py \
--aco \
--map <map index> \
-p <population size> \
-i <number of iterations> \
-a <alpha value> \
-b <beta value> \
-pec <evaporation coefficient> \
-f <filename>
```

To run PSO the following:

```
$ python3 main.py \
--pso \
--map <map index> \
-p <population size> \
-i <number of iterations> \
-w <weight value> \
-c <cognitive value> \
-s <social value> \
-f <filename>
```

To fully replicate the experiments done one must only run the script named *algorithm_tests.sh* with the following command:

```
$ ./algorithm_tests.sh
```

The results will be written to the following directory: *./test_results*

References

[1] Ruprecht-Karls-Universität Heidelberg. *TSPLIB - TSP*. 2018. URL: https://wwwproxy.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/.

[2] pjmattingly. *Implementation of the Ant Colony Optimization algorithm (python)*. 2016. URL: https://github.com/pjmattingly/ant-colony-optimization.

[3] Nathan Rooy. *Particle Swarm Optimization from Scratch with Python*. 2016. URL: https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python/.

[4] George Swan. *Solving the Travelling Salesman Problem With a Particle Swarm Optimizer*. 2017. URL: https://www.codeproject.com/Articles/1182794/Solving-the-Travelling-Salesman-Problem-With-a-Par.

[5] Wikipedia contributors. "Ant colony optimization algorithms". In: Wikimedia, 2019. URL: %5Curl%7Bhttps://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms%7D.

[6] Wikipedia contributors. "Particle swarm optimization". In: Wikimedia, 2019. URL: %5Curl%7Bhttps://en.wikipedia.org/wiki/Particle_swarm_optimization%7D.

[7] Ramadan Zeineldin. *Solving the Travelling Salesman Problem With a Particle Swarm Optimizer*. 2017. URL: https://www.codeproject.com/Articles/1182794/Solving-the-Travelling-Salesman-Problem-With-a-Par.

[8] Ramadan Zeineldin. 2015. URL: https://www.researchgate.net/post/What_are_the_best_PSO_parameter_values.

[9] Ernesto Costa. *Swarm Intelegence*. University Lecture. 2019.