HAACS: Hybrid Adaptive Ant Colony System for the Travelling Salesman Problem

Suro Lee

KAIST
Daejeon, Korea
slee2444@kaist.ac.kr

Abstract. The Ant Colony System (ACS) is one of the most used meta-heuristics to solve the Travelling Salesman Problem (TSP). However, conventional ACS may get easily trapped in local optima due to its static parameters. Furthermore, static parameters also result in slow convergence speeds for large TSP instances. To increase convergence speed and encourage exploration, we introduce Hybrid Adaptive Ant Colony System (HAACS). HAACS uses randomized local search to speed up convergence speeds at the initial stages of HAACS, and dynamically tunes ACS parameters throughout the algorithm to encourage exploration away from local optima. Experiments on 50, 100, and 200-city TSP instances show that HAACS outperforms the randomized two-opt algorithm, conventional ACS, and a previous version of adaptive ACS.

Keywords: Ant Colony Systems · Travelling Salesman Problem · Adaptive Parameter Control · Hybrid Meta-heuristic.

1 Introduction

The Travelling Salesman Problem (TSP) is one of the most intensively studied problems in Computer Science. Due to its NP-hard nature, many meta-heuristic algorithms such as Ant Colony Optimization [2, 3, 5], and Genetic Algorithms [6, 7] have been proposed to solve TSP.

One of the best-performing variants of Ant Colony Optimization used to solve TSP is the Ant Colony System (ACS) proposed by Doringo et al. [2]. However, one of the critical issues faced by ACS is long convergence time and the high potential of trapping in local optima. The long convergence time is caused by the large number of initial iterations required to deposit enough amount of pheromone to accurately guide the ants for exploitation/exploration. That is, the greedy-like behavior of ants in the beginning stages of the algorithm may be very far from optimal, especially in large TSP instances. Moreoever, during conventional ACS, ants may get easily trapped in locally optimal tours because ACS reinforces only the global best tour every iteration. As a result, if the global best tour remains static over multiple iterations, more and more pheromone will be deposited on the global best tour until almost no exploration occurs.

In this paper, we propose a Hybrid Adaptive Ant Colony System (HAACS), which uses 2-opt [1] as a local search algorithm to speed up convergence in the initial stages of the algorithm, and dynamically changes parameters of ACS similar to the work in [3] to encourage exploration when stuck in local optima.

2 Background

2.1 Ant Colony System

The ACS algorithm for TSP [2] consists of three main operations: state transition, local update, and global update. Initially, m ants are placed on random cities. Each ant then chooses the next city for its tour based on the state transition rule. Once the ant traverses to the next city, the pheromone on its path decreases by an amount determined by the local update rule. This local update rule exists to encourage exploration and prevent the ants from only searching in a narrow neighbourhood of the best previous tour. Once every ant has finished its tour, pheromone on the global best tour is increased based on the global update rule. Each rule is explained in detail below.

State Transition Rule An ant positioned on node r chooses city s by applying:

$$s = \begin{cases} \arg\max_{u \in J(r)} \{[\tau(r,u)] \cdot [\eta(r,u)]^{\beta}\} & \text{if } q \leq q_0 \quad \text{(exploitation)} \\ S & \text{otherwise} \quad \text{(biased exploration)} \end{cases} \tag{1}$$

where q is a random variable in range [0,1], q_0 is a parameter in range [0,1], $\eta(r,u)$ is the inverse of the distance $\delta(r,u)$, β is a parameter greater than zero that determines the importance of pheromone verses distance, $J(r)$ is the set of cities that remain to be visited by an ant positioned on city r, and S is a random variable selected according to the probability distribution below.

$$p(r,s) = \begin{cases} \dfrac{[\tau(r,s)] \cdot [\eta(r,s)]^{\beta}}{\sum_{u \in J(r)} [\tau(r,u)] \cdot [\eta(r,u)]^{\beta}} & \text{if } s \in J(r) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Local Update Rule The local update rule is:

$$\tau(r,s) = (1-\rho) \cdot \rho(r,s) + \rho \cdot \tau_0(r,s) \tag{3}$$

where ρ is a parameter in range [0,1] that controls the portion of pheromone loss, τ_0 is the initial amount of pheromone on each edge, and (r,s) is the edge just crossed by an ant.

Global Update Rule The global update rule is:

$$\tau(r,s) = (1-\alpha)\cdot\rho(r,s) + \alpha\cdot\Delta\tau(r,s) \tag{4}$$

where α is a parameter in range [0,1] controlling the amount of pheromone gain and $\Delta\tau$ is defined as follows:

$$\Delta\tau = \begin{cases} (L_{gb})^{-1} & \text{if } (r,s) \in \text{global best tour} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where L_{gb} is the length of the global best tour from the beginning of the trial.

The authors of ACS [2] use the following values for parameters: $\alpha = \rho = 0.1$, $\beta = 2$, $q_0 = 0.9$, and $\tau_0 = (n \cdot L_{nn})^{-1}$ where L_{nn} is the tour length produced my the nearest neighbour heuristic.

2.2 2-opt Algorithm

The 2-opt algorithm [1] is a local search algorithm that repeatedly modifies a tour by the following set of operations.

1. Create the initial tour through a simple heuristic (nearest neighbour) and set it as the local best tour
2. Choose two edges from the local best tour, breaking the tour into two paths
3. Reconnect the two paths in the other possible way, creating a new tour
4. If new tour distance is less than the local best, update the local best tour to the new tour
5. Go back to 2. and repeat until no further modifications are possible

The randomized 2-opt algorithm is a special case of the 2-opt algorithm where the 2-opt algorithm is performed for a predetermined number of iterations. For each iteration, a initial city is chosen randomly for the nearest neighbour heuristc.

3 Hybrid Adaptive Ant Colony System

3.1 Speeding Up Convergence

One of the problems conventional ACS has is that it requires many iterations to build up sufficient pheromone in the edges between cities to begin guiding ants for exploitation/exploration. This problem is exacerbated in large TSP instances as the greedy-like behaviour of ants in the initial stages of ACS may result in tours that are very far from any optimal solution.

Therefore, we modify the conventional ACS algorithm by performing the randomized 2-opt algorithm for the first t iterations and updating pheromone trails at the end of each iteration based on the local best tour found by the 2-opt algorithm. After the t iterations, ACS begins with the pheromone trails accumulated from the randomized 2-opt algorithm. A value of $t = 50$ seemed

adequate for most instances of TSP to effectively speed up initial search and convergence.

In order to further speed up the randomized 2-opt algorithm, we use a special data structure called the *neighbour–list* as suggested by [4]. *Neighbour–list* is created such that each row c contains only the k-nearest neighbours of each city in order of increasing distance from each city c. The list is then used to exploit the fact that each possible 2-opt move can be viewed as a tuple $< t_1, t_2, t_3, t_4 >$ where (t_1, t_2) and (t_4, t_3) are the deleted tour edges and (t_2, t_3) and (t_1, t_4) are the edges that replace them. The edge (x, y) represents an edge where x comes directly before y in tour order. Note that the tuple $< t_1, t_2, t_3, t_4 >$ and $< t_4, t_3, t_2, t_1 >$ are the exact same representations, and for an improving move we must satisfy:

1. $d(t_1, t_2) > d(t_2, t_3)$
2. $d(t_3, t_4) > d(t_4, t_1)$
3. both 1. and 2.

Because of the two facts above, if we fix two cities t_1 and t_2, we can limit our search to finding t_3 that satisfies $d(t_2, t_3) < d(t_1, t_2)$.

Therefore, we iterate through the k nearest neighbours of a city t_2 when searching for cities to perform swaps during 2-opt, and stop search when $d(t_2, x) \geq d(t_1, t_2)$ for some x. Once we find t_3, we are restricted to only one choice for t_4 that creates a tour rather than two cycles.

Although it takes a long time to construct the *neighbour–list*, this method can be justified as the list is used multiple times throughout the 2-opt algorithm, thus amortizing the construction cost. As suggested by [2], we only keep track of $k = 20$ nearest neighbours of each city, as increasing k yields no further benefits.

3.2 Motivating the Adaptive Ant Colony System

As explored in previous works [3, 8], conventional ACS is susceptible to getting stuck in local optima even with the local update rule that is meant to increase exploration. The main reason conventional ACS cannot easily escape local optima is the set of static parameters $\rho, \alpha,$ and q_0 that control exploration/exploitation. In this section, we investigate the effects of each parameter on ACS and motivate the need for dynamic parameters in ACS.

Once ACS begins converging on a local optimal solution and exploration decreases due to the increasing amount of pheromone on the local optimal tour, conventional ACS has no mechanism that can save the local best solution and allow the ants to begin exploring again. In other words, ACS crosses the point-of-no-return (or point-of-no-exploration). In fact, conventional ACS falls into a negative-feedback loop where increased pheromone on the local optimal tour causes ants to remain in the local optimal tour, which continues to reinforce the local optimal tour by increasing pheromone on the local optimal tour. Once this continues, biased exploration (when $q \geq q_0$ in eq. (1)) is rendered useless because the pheromone levels on the local optimal tour are significantly higher

than the pheromone levels on any other edges, resulting in a very high probability of taking the local optimal tour and a very low probability of exploring. Thus at this point, lowering q_0 does not effectively encourage exploration. In fact, lowering q_0 only slows down convergence as shown in Section 4.1

Another candidate in alleviating ACS's tendency of getting trapped in local optima is lowering α. Since α controls the amount of pheromone gain of the local best solution after each iteration, lowering it would result in a better global searching ability. However, this would also make the search very undirected and lead to slower convergence, potentially undoing the effects of Section 3.1. Furthermore, once ACS begins to converge at a local optimum, decreasing α will only decrease the rate at which pheromone accumulates on the local optimal tour, instead of decreasing the amount of pheromone on the local optimal tour. Thus, decreasing α will not pull ACS out of its negative feedback loop.

As a result, we rely on ρ to pull ACS out of local optima and begin exploration again. Because ρ controls the decay of pheromone on the edges just-crossed by an ant, increasing ρ will cause the pheromone to decay more rapidly on edges that ants frequently cross. In fact, as the number of ants traversing an edge increase, the amount of pheromone on that edge will decrease proportionally as fast. Although as mentioned by [3], increasing ρ will make the search less directed, that is exactly what we need to escape local optima. Furthermore, unlike q_0 and α, ρ will actually fight against the effects of accumulating pheromone on the local optimal tour by directly decreasing the amount of pheromone on the tour.

Another added advantage of adaptively adjusting ρ is that we can potentially find the optimal value of ρ for each instance of TSP. Because optimal parameters are instance-dependent, static parameters will cause ACS to be highly effective on some instances of TSP and relatively ineffective on other instances of TSP. Consequently, dynamic parameters will provide ACS with the flexibility to better adapt to each instance of TSP.

3.3 Adaptive Ant Colony System

Motivated by Section 3.2, we dynamically adjust ρ throughout HAACS. To do so, we need an indicator of the convergence state or the optimization state of HAACS. Inspired by [3], we use normalized Average Tour Similarity (nATS) as a metric of optimization state.

Average Tour Similarity (ATS) is defined as:

$$ATS = \frac{1}{m}\sum_{k=1}^{m} s(T_k, T_{best}) \tag{6}$$

where m is the number of ants, T_k is the tour of ant k, T_{best} is the global best tour, and $s(T_k, T_{best})$ is defined as the number of edges contained in both T_k and T_{best}. Our definition of ATS differs from that of [3] in that T_{best} is defined as the global best tour, not the iteration best tour.

Since $ATS \in \{0, 1, ..., n\}$ where n is the number of cities, we defined nATS as:

$$nATS = \frac{ATS}{n} \tag{7}$$

where $nATS \in [0, 1]$. In words, nATS measures the proportion of traversed edges in an iteration that overlaps with the global best tour. From Figure 1, we indeed see that nATS is a good measure of optimization state. Once nATS saturates, no drastic improvements in global best distance are observed.

We update ρ according to the following set of heuristics:

- If global best tour has not improved in the past z iterations and $nATS > 0.95$, increase ρ by 0.05
- If global best tour has not improved in the past z iterations and $nATS < 0.90$, decrease ρ by 0.05

Only updating ρ when the global best tour has stopped improving in the past n iterations is critical. Without using this condition and simply constraining nATS to be in the range [0.90, 0.95] each iteration, the search simply becomes less directed, resulting in HAACS settling for solution even worse than the local optimum. By adding the z iterations condition, we have the ability to (1) make the search more directed when the search is too undirected to cause convergence, and (2) encourage exploration when HAACS seems to have converged on a local optimum. A value of $z = 50$ seems adequate enough to improve performance on most TSP instances. Furthermore, in order to not lose the local optimum solution, we only replace the global best solution when a shorter tour is found.

The full algorithm is described in detail in Appendix 6.2., Algorithm 1.

4 Experiments

All experiments were performed on the *kroC100.tsp* TSP instance for 1000 iterations. All figures are in Appendix 6.2. **For instructions on how to replicate results or how to run the code, please see Appendix 6.1**.

4.1 Effect of each parameter on nATS and convergence speed

The effect of various q_0, α, and ρ values in conventional ACS are displayed in Figure 1,2,3, and 4. For plotting, the tour distances were max-min normalized with $min = 20749$ (known minimum of kroC100.tsp) and $max = 26000$.

From Figure 1 and 2, it is evident that although lowering q_0 from 0.9 to 0.6 significantly decreases nATS, it comes at a cost of slower convergence to a very suboptimal solution. For testing the effects of lowering α, we increased α to 0.4 from the default value of 0.1 and analyzed the change in nATS and the convergence rate from $\alpha = 0.4$ to $\alpha = 0.1$. From Figures 3 and 1, it is evident that lowering α slows down convergence, but does not lower nATS. nATS simply saturates slightly slower for a lower α. One can also see that at high values of α, ACS gets stuck at a local optimum.

On the other hand, from Figure 1 and 4, it is evident that increasing ρ from 0.1 to 0.4 produces the desired effects. First, nATS does not fully saturate, remaining slightly above 0.9 due to increased number of explorations. In addition, noticeable deterioration in convergence speed is not observed, implying that it is safe to increase ρ to a value around 0.4-0.5 without risking slowing down convergence speed.

4.2 Performance Comparisons between conventional ACS and HAACS

From Figure 1 and 5, we see the clear performance advantages of HAACS over ACS. Due to the randomized 2opt algorithm, HAACS starts at a significantly lower best distance than ACS. Around iteration 50 where HAACS switches to adaptive ACS, one can see a spike in best distance caused by exploration of the ants. However, this spike rapidly decreases much below the best distance found by 50 iterations of 2opt. Furthermore, one can see that ACS suffers from a local optimum from iteration 200 to 600, where HAACS only suffers from such local optimum from iteration 130 to 330.

By comparing the performance of ACS and HAACS, it is clear that HAACS outperforms conventional ACS in terms of both convergence speed and global search.

5 Conclusion and Future Work

We introduce HAACS, a novel Hybrid Adaptive Ant Colony System. By using a combination of randomized 2opt and ACS, we obtain faster convergence rates than conventional ACS. In addition, by adaptively tuning parameters throughout ACS iterations by monitoring nATS and global improvements, we enable a much more global search of the search space than conventional ACS.

Potential improvements of the algorithm are: trying a variety of other heuristics for adapting ρ, partitioning the TSP graph for large TSP instances to speed up convergence, and finding a way to collectively tune all parameters $\rho, \alpha,$ and q_0 to further improve performance.

References

1. Croes, G.: A method for solving traveling salesman problems. Operations Research **6**(6), 792–812 (1958)
2. Doringo, M., Gambardella, L.-M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation.**1**(1), 53–66 (1997)
3. Yu, W., Hu, X., Zhang, J., Huang, R.: Self-adaptive ant colony system for the traveling salesman problem. In: 2009 IEEE International Conference on Systems, Man and Cybernetics, pp. 1399–1404, San Antonio, TX (2009) https://doi.org/10.1109/ICSMC.2009.5346279
4. Johnson, D., McGeoch, L.-A. : Local Search in Combinatorial Optimization. 1st edn. John Wiley and Sons, London (1977), pp. 215–310
5. Li, B., Wang, L., and Song, W.: Ant Colony Optimization for the Traveling Salesman Problem Based on Ants with Memory. In: 2008 Fourth International Conference on Natural Computation, pp. 496–501, Jinan (2008) https://doi.org/10.1109/ICNC.2008.354
6. Potvin, J.: Genetic algorithms for the traveling salesman problem. Annals of Operations Research **63**, pp. 337—370 (1996). https://doi.org/10.1007/BF02125403
7. Hussain, A., Muhammad, Y., Nauman, M., Hussain, I., Shoukry, A., Gani, S.: Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. Computational Intelligence and Neuroscience, (2017), https://doi.org/10.1155/2017/7430125
8. Satukitchai, T., and Jearanaitanakij, K.: An early exploratory method to avoid local minima in Ant Colony System. In: 2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), pp. 1–5, Hua Hin (2015) https://doi.org/10.1109/ECTICon.2015.7206969

6 Appendix

6.1 How to run the code

Dependencies To run straight from the python file, the required dependencies are the latest versions of:

- argparse
- numpy
- matplotlib
- networkx

In a Windows environment, simply run the self-contained executable *tsp_solver.exe* under the directory tsp → *tsp_solver*

Running the code For an explanation of command line arguments, please see *util.py* or enter in bash:

```
$ tsp_solver.py -h
```

To run the code on a specific TSP instance with 10 ants and 10000 fitness evaluations. Note that the default values of ants are 10 as suggested by [2] and the default number of fitness evaluations are infinity. If you do not set the number of fitness evaluations, the program will default to 1000 iterations:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000
```

Because the randomized 2opt algorithm uses an incomparable amount of fitness evaluations compared to ACS, the number of fitness evaluations used for the randomized 2opt algorithm during the HAACS algorithm are not counted. If you wish to count the fitness evaluations of randomized 2opt as well, please specify the -c flag:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -c
```

To control (initial) parameters q_0, β, ρ, α, use the -q, –beta, –rho, and –alpha flags, respectively:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -v -q 0.9
  --beta 2 --rho 0.1 --alpha 0.1
```

To control the number of randomized 2opt iterations in HAACS, use the -t flag:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -t
```

You may choose between four algorithms [2opt — ACS — AACS — HAACS] using the -a flag. AACS is HAACS without the 2opt phase.

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -a HAACS
```

To print out useful debugging/testing messages, please specify the -v flag:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -v
```

To graph nATS and normalized best distance for [ACS — AACS — HAACS], please specify the -s flag:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -s
```

To graph the best tour obtained at the end of the algorithm, please specify the -g flag:

```
$ tsp_solver.py tsp_instance.tsp -p 10 -f 10000 -g
```

6.2 Algorithm and Figures

Algorithm 1: HAACS Algorithm

```
Result: best_tour_dist, best_tour
initialize matrices, parameters, and ants
while stop_condition is not met do
    if iterations ≤ t then
        get randomized nearest neighbour tour
        perform 2_opt algorithm
        global_update_pheromone
    else
        clear all ant data
        for i ← 1 to num_cities do
            choose next city
            local_update_pheromone
        end
        all ants complete tour by returning to initial city
        local_update_pheromone

        find best performing ant
        global_update_pheromone

        if no improvements have occurred in the past z iterations then
            adapt_rho
        end
    end
end
```

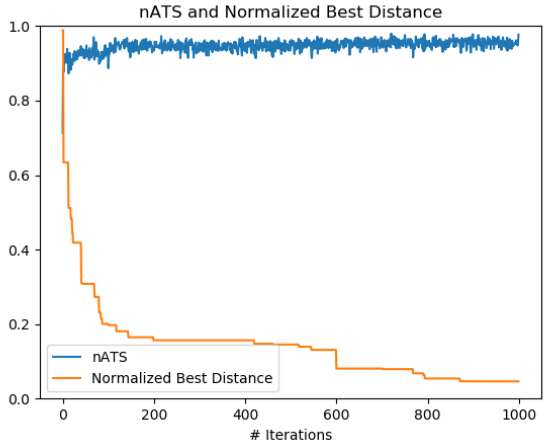

Fig. 1. ACS with $q_0 = 0.9, \alpha = \rho = 0.1$

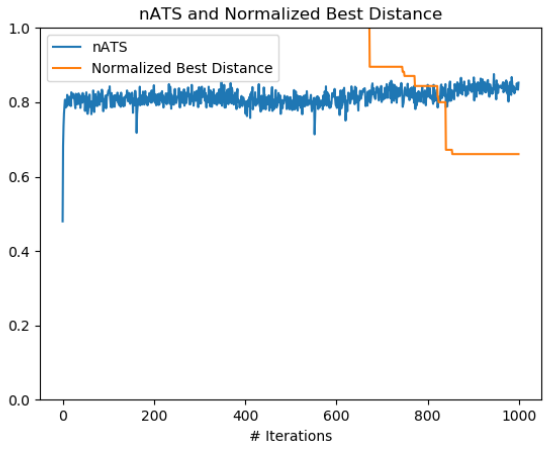

Fig. 2. ACS with $q_0 = 0.7, \alpha = \rho = 0.1$

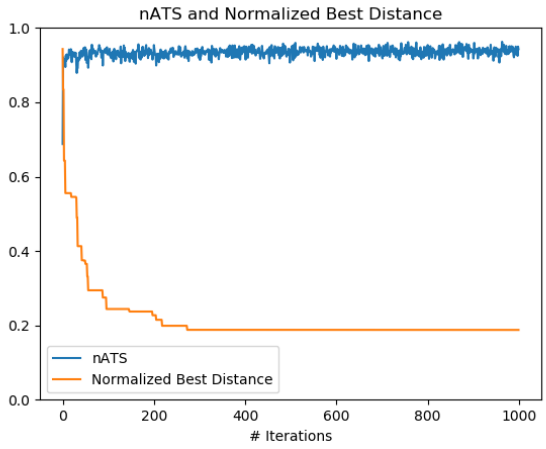

Fig. 3. ACS with $q_0 = 0.9, \alpha = 0.4, \rho = 0.1$

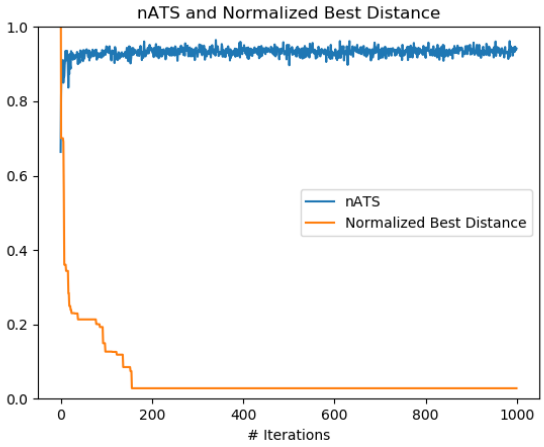

Fig. 4. ACS with $q_0 = 0.9, \alpha = 0.1, \rho = 0.4$

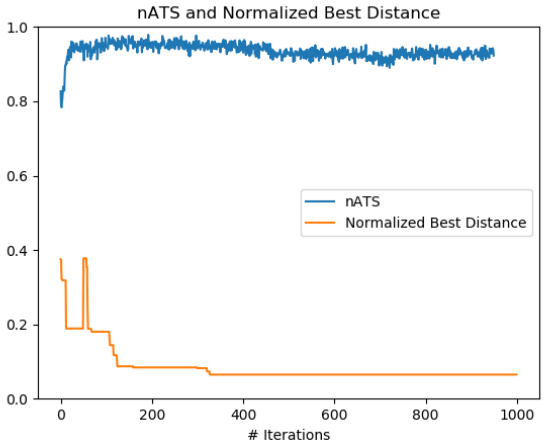

Fig. 5. HAACS