Università degli studi di Milano-Bicocca

Decision Models

Final Project

An Hybrid Metaheuristic Approach to the Travelling Salesman Problem

Authors:
Dario Bertazioli - 847761 - d.bertazioli@campus.unimib.it
Fabrizio D'Intinosante - 838866 - f.dintinosante@campus.unimib.it
Massimiliano Perletti - 847548 - m.perletti2@campus.unimib.it

June 21, 2019

Abstract

In order to try to approach a non-linear optimization problem, especially a combinatorial one, and the state-of-the-art in metaheuristics and hybridization, the travelling salesman problem (TSP), in particular the symmetric version, was chosen, together with two specific algorithms: Ant Colony and Genetic Algorithm. Our main goal is to apply this two algorithms on TSP, with the aim of obtain good performances referring to the abundant literature concerning these specific algorithms. After that, our objective is try to improve an hybrid version of each algorithm, in particular implementing Ant Colony with elements of Reinforcement Learning i.e. Q-Learning algorithm, while Genetic Algorithm with techniques of dimensionality reduction, such as clustering.

Contents

1 Introduction

The problem: the travelling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman's goal is to keep the distance travelled as low as possible.

TSP has been studied for decades and several solutions have been theorized. The simplest solution is to try all possibilities, but this is also the most time consuming and expensive method. Many solutions use heuristics, which provides probability outcomes. It must be considered that the results are approximate and not always optimal.

Our approach: in this project we apply two meta-heuristics commonly known as *Ant Colony Optimization* and *Genetic Algorithm*, implementing the "classical" version and a custom one integrating respectively *Reinforcement Learning Algorithm*, namely *Q-learning*, and *Clustering algorithm*, in particular *K-Means*.

1.1 ACO Family Algorithms

In this work we focus on two different approaches to the TSP: implementing some algorithms of the "Ant-type" family, and some Evolutionary Algorithms.
The former approach is based on the exploitation of a set of algorithms called "Ant-Family".

1.1.1 Ant Colony Optimization

The first type of "Ant-like" algorithm we implement is the **Ant Colony Optimization** (ACO) algorithm.

The procedure draws inspiration from a "real" Ant Colony. In nature, such a system is known to accomplish some difficult tasks, being beyond the capabilities of a single ant, exploiting the individuals collaborating with each other.

In particular, ACO algorithm is based on foraging behaviour of some ant species. This behaviour can be summed up as the ability of an ant colony to find the shortest paths between a source of food and the nest. Cooperation among the ants has inspired researchers to apply a similar collaboration based algorithm to those problems whose solutions can be formulated as a least cost path between an origin and a destination. Since most optimisation problems admit such a formulation, this family of algorithms reveals to be pretty interesting.

The first ACO algorithm, Ant System, was proposed by Dorigo [2, 3, 4, 5, 6] (in different versions).

It consists of a **multi-agent** approach that can produce good-quality solutions in a reasonable time for combinatorial optimisation problems [2]. The author demonstrate the performance of this algorithm on Travelling Salesman Problem (TSP) [3].

Regarding the basic mechanism of ACO, here follows a quick biological explanation. Ant species are almost blind, thus they interact with the environment and communicate with each other exploiting some hormones they produce. In particular some ant species are associated with a special kind of hormone called **pheromone**: they lay pheromone trails on the paths they explore, these traces act as stimuli and other ants belonging to the colony are attracted to follow the paths having relatively more tracks. Due to this mechanism, an individual who is following a path because of the pheromone trail also reinforces it by dropping its own pheromone too.

Thus, the more ants follow a specific path, the more likely that path becomes to be followed by the ants in the colony [2, 5, 6].

ACO algorithm makes use of ant-like agents called **artificial ants**, constructing their solutions collaboratively by sharing their experience related to the quality of solutions reached so far. The pheromone trails play a leading role in the exploitation of collective experience. The solutions are built iteratively. Typically (in the most common implementation) artificial ants store the path they

tracked while constructing the solutions. Exploiting such a memory, artificial ants do not deposit the pheromone until they have constructed their solution. Then, They determine the amount of pheromone according to the quality of their solution and upload the pheromone matrix (the data structure in which the pheromone amount for each edge connecting two nodes is stored). Automatically, the paths belonging to better solutions receive more pheromone.

In iteratively building a solution (a total path) for a single ant, a local stochastic transition policy is typically applied, establishing how to decide the next node to visit in a graph according to a probability distribution. Artificial ants make their decisions and transitions to their next state in discrete time steps, deciding whether to follow the main trails, or to random explore a new path (in our implementation, such a decision is made by a threshold check over a random number generation). Exploration is also encouraged by a mechanism of pheromone evaporation, which prevents the colony from getting stuck into a solution corresponding to a (only) local optimum [1].

1.1.2 Ant-Q Metaheuristic

In order to favor a better understanding of the working mechanism of Ant-Q Metaheuristic (presented in details in Sec. (3.2) and to give deeper insight in our implentation (following [7]), let us introduce some theoretical hints about the context.

Hints on reinforcement learning: Reinforcement Learning (RL) is an (almost) unsupervised learning approach.

It consists of an **agent** who tries to learn how to reach a goal by continuously interacting with the environment. There is an evaluation phase where the quality of agent's actions is considered and feedbacks to the agent are given in the form a numerical reward. This type of feedback is known as **evaluative feedback**: in contrary of supervised learning, here the agent is not explicitly told what action is the best to take in a certain situation, instead it should try a set of possible actions and learn the best strategy, i.e. the one yielding the most **reward** itself.

In some cases, the goal state (that is, the agent reaching its objective) cannot be obtained immediately, but only after a set of actions: as a result the reward is delayed[2].

Summing up, the RL problem can be defined according to [8] as the problem of an agent interacting with a complex environment trying to maximise its long-run reward over a sequence of discrete time steps.

The agent follows a **policy** to decide on its action according to the current state and conditions. This policy is typically a stochastic function ($\pi(s,a)$) indicating a probability of choosing an action a given a state s. Notice that agent has the possibility to change its initial policy according to new experiences in order to achieve optimal cumulative reward over time.

The value of a state $V_\pi(s)$ is defined as the expected cumulative reward that will be obtained starting from a state s and acting according to the current policy π. In the same way, the value of a pair state-action($Q_\pi(s,a)$) is the expected return obtained starting from s with action a and then following the policy.

In formulas, V is defined as:

$$V^\pi(s) = E_\pi\{\sum_i \gamma^i r_{t+1,i+1\ |s_t=s}\}\ , \tag{1}$$

and accordingly:

$$Q^\pi(s,a) = E_\pi\{\sum_i \gamma^i r_{t+1,i+1\ |\ s_t=s,\ a_t=a}\}\ , \tag{2}$$

where r_{t+1} is the reward associated with the $t-th$ transition between system states s_t and s_{t+1}.

[1] However, notice that in real ant colonies pheromone evaporation is too slow to be a significant part of their search mechanism.

[2] This particular feature will be a central part of the implementation that will be presented later in eq.14.

Randomly initialise Q(s, a)
Repeat for each episode
  Initialise the current state s
  While s is not terminal state
    Choose action a at the current state s according to the policy (e.g. $\varepsilon - greedy$)
    Take action a, observe reward r and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;

Figure 1: The pseudo code of a typical Q-learning algorithm implementation.

The RL problem consists of the agent trying to find the optimal policy π^* that maximizes the value functions, obtaining:

$$V^{\pi^*}(s) = max_{a \in A(s)}(Q^{\pi^*}(s,a)) \ . \tag{3}$$

However, the policy estimation can be in general a complex problem, and an optimal policy, where it exists, can be obtained with various algorithms, such as Policy Iteration and Value Iteration. There are also kind of learning mechanism defined as "off-policy", because they do not exploit a proper a-priori policy procedure.

Q-Learning: is an off-policy method, meaning that it updates the Q-values iteratively basing this process on the action that gives the maximum value (that is, such an algorithm tries to directly learn Q^* instead of learning Q_π first). Fig. (1) shows the pseudocode of the algorithm: the agent uses a so called ϵ-greedy policy, but updates the current Q-value estimate considering the action that provides the maximum value at the successor state instead of considering the (current-)policy-suggested action.

Ant-Q Algorithm: one of the core points of this project consists of the implementation of the Ant-Q algorithm, introduced by Gambardella in collaboration with Dorigo[7], while attempting to ameliorate the "classic" ACO performances.

In this approach, the pheromone update rule is borrowed from the Q-learning praxis:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma[max_{a'} Q(s',a') - Q(s,a)] \ . \tag{4}$$

Thus, the Ant-Q algorithm is a crossover between a "simple" Ant Colony algorithm and a Q-learning algorithm. The basic structure belongs to the Ant family, whereas the update rule as well as the decision schedule for exploring the nodes (that is, the modality an artificial ant selects the next node to visit, starting from a given node) is imported from the cited reinforcement learning algorithm.

1.2 Evolutionary Algorithms

1.2.1 Genetic Algorithm

The latter approach we faced along this work is the **Genetic Algorithm** (GA), which we initially implement in its classical version. This algorithm is based on a biological metaphor: the resolution of a problem is seen as a competition among a population whose evolving individuals become better and better candidates solutions over time. A "fitness" function is used to evaluate each individual to decide whether it will contribute to the next generation. Then, in analogy with the biological metaphor (the gene transfer in sexual reproduction), a crossover operator is applied in order to generate the next generation of the population. This process, in analogy to the evolutionary theory (Darwinism), should lead after a certain number of iterations to a much more fit ensemble of individuals representing "good" candidate solutions to the considered problem.

1.2.2 K-Means GA Metaheuristic

With the aim of improve classic GA performance on big dataset, different hybridization techniques have been created. One of these consists of splitting the main problem in multiple sub-problems, reduced in dimensions, in order to reach a simplified form for the general problem.

K-Means GA meta-heuristic (KGA) briefly consists of three stages: aggregate data using a simple clustering method, the K-Means method, apply the GA algorithm on every data cluster, with a path dimension smaller than the original, and finally rebuild the final solution starting from the smaller multiple solutions. The K-Means method is shortly described below.

Clustering with K-Means: At first, specifically for the TSP problem, we need to cluster our cities into close groups with a clustering method.

The *K-Means* method is designed to partition a set of data into K classes with K chosen as desired. This method constructs some partitions of the data-matrix so that the squared Euclidean distance between the row vector for any object and the centroid vector of its respective cluster is at least as small as the distances to the centroids of the remaining clusters. The centroid of cluster C_k is a point in a P-dimensional space obtained by averaging the values on each variable over the objects within the cluster. For instance, the centroid value for j-th variable in cluster C_k is

$$\bar{x}_j^{(k)} = \frac{1}{n_k} \sum_{i \in C_k} x_{ij} \ , \tag{5}$$

and the complete centroid vector for cluster C_k is given by [9]

$$\bar{x}^{(k)} = (\bar{x}_1^{(k)}, \bar{x}_2^{(k)}, \ldots, \bar{x}_p^{(k)}) \ . \tag{6}$$

2 Datasets

The datasets used in this work are taken from TSP-lib, a large source of TSP datasets largely cited in literature. There are datasets of variable dimension and for everyone is also available the optimal solution so that is possible for us to compare our results with the optimal one. Every solution is available at the following link. Every dataset is composed of a list of "*cities*" associated with $2D$ coordinates; the only preprocessing we apply is to compute a matrix containing the distance between every point and the other ones.

In particular we select 5 datasets of different dimension:

- dj38
- berlin52
- ch130
- d198
- pr1002

Such a choice because they are often used and largely cited in literature. Those datasets will be used to test the performance of our implementations.

3 The Methodological Approach

3.1 ACO

As introduced in Sec. (1.1.1), Ant Colony Optimisation is a meta-heuristic proposed to solve hard optimisation problems. The ACO meta-heuristic, from a high-level point of view, is composed of two main stages:

- **Construct a single ant solution**: the artificial ants construct their solutions. The transition policy controls the ants' next step to one of the adjacent nodes. Once the ants have completed their path, the quality of the current solution is evaluated, and used in the next step. The decision policy is based on following a probability distribution of the type [2]:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \quad \text{if j} \in \text{N}_\text{i}^\text{k} \ , \tag{7}$$

  where

  - η_{ij} indicates an heuristic value specified according to the problem; in the TSP case, we considered it to be $1/d_{ij}$, coherently with the choice of [2].
  - τ_{ij} is the pheromone quantity on the path between the i-th and j-th nodes,
  - N_i^k the neighbourhood region of the k-th ant.
  - α and β are the parameters used to set the relative importance of the pheromone trail and the heuristic value. In our implementation, we attempted to change the values of those parameters in order to explore a set of possible configurations. We experienced that:
    - as $\alpha \to 0$, the pheromone track become less important and the ants tend to choose the closest cities, resulting in a much more "greedy" search;
    - vice-versa, when $\beta \to 0$, heuristic values are almost ignored and only the tracks are considered in the decision making;

    From the little experience we made, it seems quite worth to notice that the optimal setting of those parameters depends on the particular dataset. More in details it might depend on the considered dimension and on the variance associated to the average distance between all the nodes: indeed, with small datasets (let us say having n 30 nodes or less) the ACO seems to better perform when more importance is given on the pheromone effect, since the artificial ants quickly converges to a unique path (thus $\alpha \geq \beta$), whereas whenever a large dataset is considered the pheromone action of guiding artificial ants in their exploration seems to be as relevant as the heuristic value weight, resulting with a optimal configuration being a compromise between those actions (thus it might be better to have $\alpha \sim \beta$).

    In the presented results α, β are fixed and equals to their most stable values taken from the literature (mostly from [2, 3], but): $\alpha = 1$ and $\beta = 3$.

- **Update the pheromone matrix**: the pheromone trails are adjusted considering the latest iteration of the colony search process. Two different updates happen:

  - the pheromone evaporates according to the equation:

$$\tau_{ij} = (1 - \rho)\tau_{ij} \ , \tag{8}$$

    where ρ is the evaporation coefficient. In our implementation, and following the literature, ρ is taken equal to 0.5. Such a value should represent a compromise between exploring and exploiting the panorama of possible solutions;

– new pheromone is deposited on the followed path. The amount of pheromone to deposit is typically decided according to the quality of the particular solutions that each path belongs to:

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \ , \tag{9}$$

where $\Delta\tau_{ij}^{k}$ is the pheromone increase amount deposited by the k-th ant, which can be (e.g. in Dorigo initial work) taken either as a constant, or $\Delta\tau_{ij} = 1/L_k$, where L_k is the $k-th$ ant path length.

However the entity of pheromone update and it's weight on how the search will be biased towards the best solution found so far is an implementation decision.

Note that the summation is extended up to a parameter m representing the number of artificial ants. The value of this parameter is taken to be $m = 4\ n$ with n being the number of nodes in the graph. This particular value is a completely experimental value, differing from the classical literature value being $m_{literature} = n$. The explanation of such a choice can be found in paragraph 3.2, being related to the parallel implementation.

– the previous equations can be combined together obtaining:

$$\tau_{ij} = (1-\rho)\tau_{ij} + \Delta\tau_{ij} \ . \tag{10}$$

The **pseudocode** of the ACO solution is presented in Tab. 1.

3.2 Ant-Q

In this subsection some more details about what anticipated in Sec. 1.1.2 will be given, and the implementation of the Ant-Q algorithm will be discussed.

As it was previously stated, Ant-Q borrows the update rule from the Q-learning algorithm. Thus, Eq. (10) changes as the following:

$$\tau_{ij} = (1-\alpha)\tau_{ij} + \alpha(\Delta\tau_{ij} + \gamma max_{l\in N_j^k}\tau_{jl}) \ , \tag{11}$$

where α has a role similar to the previous evaporation coefficient ρ, and γ is the well-known parameter called in RL literature discount rate. In literature, in particular in [7] the typical choice of such parameters is $\alpha = 0.1$ and $\gamma = 0.3$. We adopted this value for α, whereas we found to be a better choice for our implementation to have $\gamma \to 1$. The better performance of the $\gamma = 1$ value in our case might be related to the parallel implementation, as further explained in Sec.3.2.

The next action (what next node connecting to) is chosen according to:

$$s = \begin{cases} arg\ max_{a\in J_k(s)}[Q(s,a)]^{\delta}[\eta(s,a)]^{\beta} & \text{if } q \le q_0 \\ S & \text{otherwise} \end{cases} \ , \tag{12}$$

where, correspondingly to Eq. (7), η is an heuristic value associated to the inverse of distance between a pair of nodes, and δ , β are parameters for the relative importance of pheromone effects (given by $Q \approx \tau$) and the distance measure (indeed $\eta \sim 1/d$). Accordingly to [7], it is taken $\delta = 1$ and $\beta = 2$. The q_0 parameter is analogous to ϵ_0 in the well known ϵ-greedy reinforment learning policy. Following this analogy, the q variable is simply taken as a random number. When the condition $q < q_0$ is not met, a decision process correspondent to Eq. (7) applies. In this work, following the main reference [7], this parameter is taken as $q_0 = 0.9$ (minor tuning attempts were made, not leading to an apparently better setting).

Observing Eq. (11) in relation to the Q-learning update rule Eq. (4), we underline how much similar the Ant-Q pheromone matrix is compared to a "standard" Q-table. Indeed, it is worth to notice that:

Algorithm Ant Colony Optimization

Main Algorithm

```
0: initialize best_dist and best_path to None
1: for generation in generations:
2:    create n_ants artificial ants
3:    for one_ant in ants:
4:       make a single ant path (see Make path)
5:       compute the path length
6:       update best_dist and best_path
7:    update the pheromone matrix
      (local update only for child processes, according to Eq. (10))
8:    every a certain n of iterations:
9:       update the global pheromone matrix
         (shared in MPI environment among master&child.)
10: return best_dist, best_sol
```

Make path

```
1: start from a vertex
2: add start vertex to visited nodes
3: for each remaining vertex:
4:    list the neighbours
5:    list the not yet visited neighb
6:    calculate the probability of choosing a vertex (according to Eq. (7))
7:    choose the vertex according to probability
8:    add the chosen vertex to the visited list
9:    return the chosen vertex id
```

Local update pheromone matrix

```
1: for ant in ant_colony :
2:    for each vertex of one_ant_path :
3:       increase pheromone_matrix between current and next vertex of Δτ
         (according to Eq. (10))
```

Global update pheromone matrix (parallelism)

```
1: gather from MPI env all the pheromone matrices
2: if process is the parent process (rank==0):
3:   for each element average over the n_cores matrices.
4: broadcast obtained pheromone matrix to the other
   processes
```

Table 1: Ant Colony pseudocode

- Eq (11) updates the pheromone value of the transition (i, j) according to the pheromone value of the next transition (j, l),
- Eq. (11) uses the second part of Eq. (4) (known as TD Error) to weight the pheromone quantity associated to the current edge with a learning rate α and a discount rate γ,
- the equation

$$\tau_{ij} = (1 - \alpha)\tau_{ij} + \alpha(\gamma max_{l \in N_j^k} \tau_{jl}) \ , \tag{13}$$

  is used for the pheromone matrix update (namely a local update) during each path construction (of each ant), and it does not include the delayed reward $\Delta\tau_{ij}$,
- $\Delta\tau_{ij}$ is calculated according to the solution quality, as anticipated circa Eq. (9), and assigned in a "delayed" mode: thus the value of $\Delta\tau_{ij}$ for all i and j will be 0 while the ants apply the update rule Eq. (13) during their construction of the current solution. Therefore, the update rule Eq. (11) is reapplied at the completion of the current solution, but with the value of the next transition considered to be equal to zero. (thus updating only by $\Delta\tau_{ij}$):

$$\tau_{ij} = (1 - \alpha)\tau_{ij} + \alpha(\Delta\tau_{ij}) \ , \tag{14}$$

Algorithm Ant-Q algorithm

Main Algorithm

```
0: initialize best_dist and best_path to None
1: for generation in generations:
2:    create n_ants artificial ants
3:    for each ant:
4:       make a single ant path (see Make path)
5:       compute the path length
6:       update best_dist and best_path
7:       update pheromone matrix with delayed rewards
         (according to Eq. (14))
8:    update the global pheromone matrix
      (shared in MPI environment among master&child.)
9: return best_dist, best_sol
```

Make path

```
1: start from a vertex
2: add start vertex to visited nodes
3: for each remaining vertex:
4:    list the neighbours
5:    list the not yet visited neighb
6:    generate a random number q ∈ {0, 1}
7:    if q < q_0: (threshold)
8:       select next vertex according to Eq. (12)
9:    else:
10:      calculate the probability of choosing a vertex
11:      (according to Eq. (7))
12:       choose the vertex according to probability
13:   add the chosen vertex to the visited list
14:   give local rewards (local update pheromone matrix)
15:   return the chosen vertex id
```

Local update pheromone matrix

```
1: for ant in ant_colony :
2:    for each vertex of one_ant_path :
3:       increase pheromone_matrix between current
         and next vertex of a Δτ (according to Eq. (11))
```

Global update pheromone matrix (parallelism)

```
1: gather from MPI env all the pheromone matrices
2: if process is the parent process (rank==0):
3:   for each element average over the n_cores matrices.
4: broadcast obtained pheromone matrix to the other
   processes
```

Table 2: Ant-Q pseudocode

- still according to [7],[8] $\Delta\tau_{ij}$ can be updated with an iteration best rule (update every single colony iteration) or global best (update based on the global best value.

Notice that many other attempts were made in the direction of hybridizing ACO and reinforcement learning based algorithms, some particularly interesting and successful, such as [10],[11],[12]. However, due to the few time available, we could head our forces in implementing only the in-detail-described Ant-Q.

The pseudocode regarding our implementation of such an hybrid meta-heuristic is reported in Tab. 2

Parallel Implementation: since both the ACO and Ant-Q are memory and computational-expensive, we suggest a parallel implementation of each algorithm.

The type of proposed algorithm is naturally easy to parallelize, since it is enough to split the total number of ants into subsets (called *n_ants_per_core* in the code) of ants that will be distributed on each single core.

Thus, a parent process initializes the algorithm, creating n_core children each operating on a "per-

sonal" memory area. Every child has its own data structures, in particular its own pheromone matrix. The MPI/OpenMPI[3] [4] software (in particular mpi4py[5], a pythonic Api for OpenMPI) is used to host a shared pheromone matrix which will be updated according to the pseudocode in Tab. 1,2 (as a mean-matrix), and which every "local" matrix will be update to after every iteration, so that every artificial ant in every child process will "feel" a as similar as possible pheromone effect.

As anticipated previously, we made slightly different choice in setting some parameters values:

- in both ACO and Ant-Q implementation, we take m, the number of ant agents, to be $m = n\_core \times n$, where n is the number of nodes of the graph and n_core is the number of cores available for running parallel processes. This choice is explained due to the fact that in our implementation on each core one child (or the parent) process is running alone, and the initial amount of artificial ants is equally distributed among the cores. Every iteration the pheromone matrix created or locally updated by a process running a single core is globally updated and a pheromone mean matrix is calculated and shared among the other processes. Thus for each node the number of ants for each iteration is the classical $m_{single\_core} = n$, because locally (while constructing the local pheromone matrix) the ant-ant interaction is the classical (and known in literature to be optimal) one;

- in Ant-Q, the parameter γ is taken to be equal or close to 1, differently from the main Ref. [7]. This choice is quite empirical, since we obtained slightly better performances with $\gamma \in [0.9, 1]$. For simplicity in producing the final results a value of $\gamma = 1$ is taken. A possible explanation of such a choice might be found once again in the parallel implementation: since the global pheromone matrix is the result of an average of the results of n_cores systems, the future choice taken considering such a matrix (having the role of a Q-table in this circumstances) might be more trustable with respect to the classical situation of a single system sequential implementation. Thus, if this is confirmed, a value of $\gamma \to 1$ would lead to a better performance.

3.3 GA

With reference to Sec. 1.2.1, the pseudo code of the standard genetic algorithm is summarized in the Tab. 3, where T_m is the mutation rate that determines the rate at which the mutation operator is applied, T_p is the population size (number of chromosomes), $elite\_n$ is the number of individuals selected with the Selection mechanism, based on the fitness function value, that have to be saved for the new generation without mutation in order to preserve good individuals and $MaxG$ the number of generations used in the experiment[13].

Finally, aiming to explore a new variation of the standard algorithm, we try to integrate the *K-Means* clustering algorithm in order to reduce the problem dimensionality and hoping to improve the performances of the genetic procedure.

As mentioned, evolutionary algorithms like *Genetic algorithm* use operators inspired by natural selection such as reproduction, mutation, recombination and selection. *Genetic algorithm* is quite customizable in every component; in this work we focus our attention on the selection part. In particular we experiment the implementation of the following selection strategies [14]:

- roulette-wheel selection;

- tournament selection.

The **roulette-wheel** selection consists of giving a weight to each chromosomes (the individuals in our population), with each weight corresponding to a portion of a roulette wheel. In this way chromosomes

[3] `https://www.open-mpi.org/`

[4] a nice introduction to MPI basic usage: `https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf`

[5] https://mpi4py.readthedocs.io/en/stable/

Algorithm Genetic Algorithm

1: **procedure** Genetic(Tm, Tp, elite_n, Selection, MaxGen)
2: $Pop \leftarrow GeneratePopulation(Tp)$
3: $Pop \leftarrow Evaluation(Pop)$
4: **for** $i = 1 \dots MaxGen$ **do**
5: $Pop \leftarrow Selection(elite\_n\ from\ Pop)$
6: $Pop \leftarrow Crossover(Pop)$
7: With probability Tm do:
8: $Pop \leftarrow Mutation(Pop)$
9: **end for**
10: **return** the best solution in Pop
11: **end procedure**

Table 3: Genetic Algorithm pseudocode

with higher fitness will have higher weight and a larger portion of the wheel

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \ , \tag{15}$$

with f_i as the fitness value for the ith individual.

Spinning the wheel for different times allows selecting the individuals for the next generation. Eventually, the roulette wheel is nothing more than a weighted sort mechanism.

The **tournament** selection, instead, consists of randomly selecting a group of individuals from the larger population and taking only the ones with the highest fitness values. The number of individuals competing in each tournament is commonly set to 2 but in this work we decide to implement a variable tournament dimension for each iteration.

The other elements, however, are standard. Among these we remember:

- **crossover:** the simplest crossover strategy called *single-point* is applied;
- **mutation:** the basic mutation version called *point mutation*, with a mutation probability described by a **mutation rate**, is applied.

The *single-point* crossover is one of the simple crossover technique used for random GA applications. This crossover uses the single point fragmentation of the parents and then combines the parents at the crossover point to create the offspring or child [15].

The *point mutation* consists of change each individual gene value according to a certain probability. The method is easy in its implementation, but it cannot effective control the mutation results [16].

Having seen the results provided by the different selection techniques, we decide to use *tournament* in classical GA and *roulette-wheel* in KGA. In fact the *roulette-wheel* selection implements more the exploitation aspects of GA while the *tournament* selection allow to improve a more exploratory approach, which is exactly what we need when working on big permutations [17].

3.4 KGA

The **K-Means Genetic Algorithm** (KGA) as mentioned in Sec. 1.2.2, is composed of different phases described below.

K-Means Clustering: The first step consists of create an arbitrary number of clusters using the K-Means clustering algorithm, which pseudocode is presented in Tab: 4.

Algorithm K-Means Algorithm

1: Set the K cluster centers randomly;
2: **repeat**
3: **for** *each vertex* **do**
4: Calculate distance measure to each cluster;
5: Assign it to the closest cluster;
6: **end**
7: recompute the cluster centers positions;
8: **until** stop criteria are met;

Table 4: K-Means pseudocode

Intra-group evolution operation: The aim of the intra-group evolution operation is to find the shortest path for the given vertices in each cluster. GA is performed in each cluster aiming to obtain an approximate solution by a couple of genetic operations like selection, crossover, and mutation. Running the GA algorithm on smaller portion of original data allows to improve time performances and reach better solutions. Eventually all those clusters could be handled in parallel. The result of this step is the creation of tours $T_1, T_2, \ldots, T_k$ for each cluster $C_1, C_2, \ldots, C_k$.

Inter-group connection: In the previous step, the shortest path between the given vertices in each cluster is obtained. With the aim of reconstructing the whole shortest path we need to properly connect each cluster to the others. By connect two clusters, one should determine which edges will be deleted from the adjacent shortest path among each cluster, and which edges will be linked for combining two adjacent clusters into one.

Assume that $i \in G_i$ and $j \in G_j$ are two closest vertices between two clusters G_i and G_j, consider $i-1,\ i+1 \in G_i$ to be two adjacent vertices of the vertex i, and $j-1, j+1 \in G_j$ to be two adjacent vertices of the vertex j. Given G_i and G_j, in order to combine the two clusters into a single one, we need to select two vertices i^* and j^* for deleting and linking edges, according to Eq. (16)

$$\{i^*, j^*\} = \text{argmin}_{i',j'} \begin{cases} d_{ij} + d_{i'j'} - d_{ii'} - d_{jj'} \\ d_{ij'} + d_{i'j} - d_{ii'} - d_{jj'} \end{cases} \tag{16}$$

where $i' \in \{i-1; i+1\}, j' \in \{j-1; j+1\}$ as in [18].

Following this strategy, the first two clusters are combined together forming a new supercluster, which combines with a third cluster, and so on, step by step. At last, all those clusters are joined into one tour, and the shortest whole travelling tour is derived as shown in Fig. 2.

The whole process of the KGA is listed in Tab. 5 inspired by[18],[19]. It is worth to underline that, in order to achieve better performance from the inter-group connection operation, we use the GA also to find the shortest path between the clusters *centroids*. This allows to improve the final result and restricts the number of intersections through the path.

4 Results and Evaluation

Each algorithm is tested on every dataset listed in Sec. 2. For every algorithm each run consists of five iterated executions, in order to compute the mean and standard deviation of the resulting optimal values.

The performance tests are executed on a machine with quad-core Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz. Every parallel execution exploits all the four cores (thus, $n\_cores = 4$).

Each table reported in this section also provide, for each run,the number of iterations for each algorithm, the best value, the averaged displacement (%) of each solution from the globally optimal

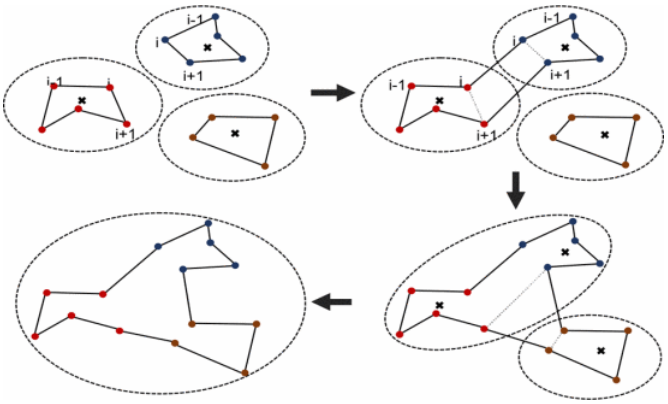

Figure 2: The inter-group connection procedure illustrated.

Algorithm KGA

1: **input** an TSP;
2: K-Means is adopted to cluster the TSP into k *sub-problems*
3: **For** each sub-prob $i = 1$ to k, do:
4: **repeat**
5: GA procedure
6: **until** *stop criteria are met*
7: **Output** shortest path for sub-problem i;
8: **End**
9: Seek for the best combining seq S with GA
10: Combine all those shortest path into one tour
11: **Output** the shortest whole travelling tour.

Table 5: KGA pseudocode

value (OptimalTour)[6], the percentage displacement (%) of the best solution found[7], and the averaged cpu-time.

The average and best displacements are meant to be a sort of quality measure for the performances, whether the average time is taken into account as well.

Tab. 6 presents the results of such a testing procedure for the ACO algorithm, whereas Tab. 7 provides the same information for the Ant-Q approach.

Dataset	ACO						
	OptimalTour	Mean	SD	Best	% Avg Dist	% Best Dist	Avg time (s)
dj38 (1000 it)	6656	6763.03	75.16	6708.04	1.61	0.78	75
berlin52 (1000 it)	7542	7925.39	162.00	7677.66	5.08	1.80	320
ch130 (500 it)	6110	6487.19	74.20	6385.46	6.17	4.51	9240
d198 (100 it)	15780	17376.23	105.81	17235.44	10.12	9.22	6540
pr1002 (10 it)	259054	337976.61	833.96	337309.96	30.47	30.21	6180

Table 6: The resulting ACO performances on the test datasets.

[6]values taken from `https://wwwproxy.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/STSP.html`
[7]i.e. "% Best Dist" is calculated as % Best Dist = (OurBestSolution − OptimalTour)/OptimalTour) ∗ 100

Dataset	Ant-Q						
	OptimalTour	Mean	SD	Best	% Avg Dist	% Best Dist	Avg time (s)
dj38 (1000 it)	6656	6663.99	4.16	6659.43	0.12	0.05	37
berlin52 (1000 it)	7542	7666.17	113.87	7548.99	1.65	0.09	170
ch130 (500 it)	6110	6491.31	70.01	6383.42	6.24	4.47	8400
d198 (100 it)	15780	17075.81	39.69	17032.75	8.21	7.94	2880
pr1002 (10 it)	259054	319646.49	1009.39	318960.00	23.39	23.12	6120

Table 7: The resulting Ant-Q performances on the test datasets.

As regards the secondary part of this work, including GA and KGA, the following implementation details are provided:

- Keeping in mind Tab. 3, mutation rate (Tm), is typically set to $Tm = 0.01$. This choice is explained with the aim of limiting an excessive randomization while still maintaining a good trade-off between the exploitation and exploration behaviour.
- Population size (Tp) is set to $Tp = 300$ for every dataset with classic GA with the exception of *pr1002*, where the dataset dimension would have not allowed such a setting considering the computational limits. In this particular case we take $Tp = 125$. For each dataset used with KGA $Tp = 125$ considering the reduced dimensionality of the sub-problems which such an algorithm is applied to.
- Selection mechanism as mentioned in Sec. 3.3 is set to *tournament* in classic GA, and $roulette - wheel$ in KGA.
- The action of elitism mechanism is controlled by a parameter, $elite\_n$, corresponding to the number of individuals selected due to elitism, taken equal to 10% of Tp in most cases. In KGA $elite\_n$ is taken equal to 25.
- Number of generations ($MaxGen$) is set to $MaxGen = 5000$ in classic GA attempting to achieve a decent performance. Instead, in KGA a smaller value of $MaxGen = 1000$ is used for each dataset still considering the dimensional reduction associated with the clustering.
- Specifically for KGA algorithm, k is the number of clusters created with the K-Means algorithm in order to split the main problem into smaller sub-problems. In this implementation k is set to different values according to the dimension of the original problem.

  In particular

  - $k = 2$ for dj38;
  - $k = 3$ for berlin52;
  - $k = 8$ for ch130;
  - $k = 30$ for d198;
  - $k = 60$ for pr1002.

  The k dimensions are chose referring to the shape of the points into the datasets.
- it is worth to notice that KGA in addition to the number of generations into each cluster, has a number of iterations also to determine the optimal path between the centroids of the clusters as mentioned in Sec. 1.2.2. This parameter is set to 1000 for each dataset regardless the number of clusters created. This is due to the will to equal compare the performance of the algorithm for each dataset.

Finally, Tab. 8 presents the results of such a testing procedure for the GA algorithm, whereas Tab. 9 provides the same information for the KGA approach.

Dataset	GA						
	OptimalTour	Mean	SD	Best	% Avg Dist	% Best Dist	Avg time (s)
dj38	6656	7032.91	574.34	6659.43	5.66	0.05	263
berlin52	7542	8095.35	167.27	7868.72	7.34	4.33	286
ch130	6110	7933.02	1041.27	6929.81	29.84	13.42	435
d198	15780	27579.73	1869.54	25537.85	74.78	61.84	604
pr1002	259045	4666413.09	12608.57	4652556.08	1701.39	1696.04	2641

Table 8: The resulting GA performances on the test datasets.

Dataset	KGA						
	OptimalTour	Mean	SD	Best	% Avg Dist	% Best Dist	Avg time (s)
dj38	6656	9608.77	1817.02	7720.03	44.36	15.99	140
berlin52	7542	8879.59	446.69	8218.95	17.74	8.98	188
ch130	6110	7790.85	312.17	7314.08	27.51	19.71	409
d198	15780	24023.86	3066.94	20830.94	52.24	32.01	1413
pr1002	259045	379495.87	6343.57	373865.26	46.50	44.32	4456

Table 9: The resulting KGA performances on the test datasets.

5 Discussion

Regarding the ACO and Ant-Q results, the following conclusions can be drawn:

- Both algorithms seem to perform well, with quite good (or at least "decent" for the largest dataset) average percentage displacements from the globally optimal solution.

- in general, both algorithms are pretty fast and well performing on small or medium datasets ($n \lesssim 100$), but the execution time becomes pretty important when approaching datasets with $n \gtrsim 100$ nodes. This behaviour is explained considering that Ant family algorithms are multiagent algorithms, and the best practice for an optimal tuning is known to require setting the number of ant agents $m \sim n$, thus the larger the dataset is the more internal iterations the algorithm will be doing.

- a comparison between the two algorithms can be done in terms of the percentage variation. Observing the plot in Fig. 3 it can be clearly stated that the Ant-Q has slightly but systematically better performances, especially in providing the best solution and typically having a better average execution time (even though the magnitude order remains similar between the two algorithms).

  Thus, the introduction of the Q-learning-like updating rule (Eq. 11) seems to improve the performance over the "simple" ACO. Therefore the hybrid approach performs better than the classic one. This results is in agreement with those of Gambardella and Dorigo in Ref. [7].

- it is worth to notice that one could increase the performances of such algorithms even in case of larger dataset, just providing more computational resources and time. The good news of this approach consists of being able to scale out pretty easily, just by setting up an MPI cluster[8] and configuring an host-file for the parent-children scheduling. However, for our limited resources, the last runs on large datasets were pretty limited in the number of iterations, thus producing a wider gap with the globally optimal solution.

[8]For more info see `https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/`

In regard to the evolutionary approach, GA and KGA in particular for this work, the following conclusions are drawn:

- Both algorithms perform well but, due to their nature, they require a large amount of iteration i.e. time, to reach good performance in terms of average displacements from the globally optimal solution.
- both algorithms are rather slow, especially if compared in terms of ratio between time and displacements from the optimal solution in average.
- comparing the two algorithms in terms of percentage variation is possible observing Fig. 4. GA seems to be more efficient on small dataset while KGA seems to suffer not in finding the optimal solutions for the sub-problems but in reconstruct the total optimal solution linking the sub-problems ones. This is due to the linking mechanism that could cause a small distortion for each linking operation causing its accumulation and amplification. Indeed bigger the dataset is, more efficient seems to become KGA compared with GA; where GA seems to substantially diverge from the optimal solution, KGA improve his performance as the problem dimension grows.
- it is worth to notice that both algorithms performance could be improved with a fine parameters tuning and providing more computational resources on a larger amount of time, maybe implementing a parallel computation approach.

Going further in comparison, it is worth to make a parallel between the performance of the Ant-family algorithms and the evolutionary ones:

- As shown in Fig. (3,4) the ACO and Ant-Q algorithms seems achieve better performance than GA and KGA on small and medium-size datasets. This is probably due to the nature of this algorithms. While Ant family algorithms use a "brute-force" approach, evolutionary algorithms works on entire permutations and this prevents them from achieving good results;
- Always referring to the figures already mentioned, but especially to Tab. (6,7,8,9) where is possible to see also the information about times required for each algorithm to reach their respective results, evolutionary family algorithms appears good for exploratory works on large datasets especially in absence of high performance hardware. In fact GA for small and KGA for larger datasets, represents a good solution for initial approach to a combinatorial problem, in order to save computational time to achieve pretty good approximate solutions, maybe with the aim to find a good starting point for another algorithm. Instead, if the aim is to try to find the global optimal solution, and especially if in condition to have powerful hardware and enough time, then Ant family algorithms are more suited.

6 Conclusions

The main core objective of this work was to implement and test two interesting hybridized metaheuristic approaches to the Travelling Salesman Problem.

The fist approach presented in Sec. (1.1.1) and (1.1.2) consisted of developing the Ant Colony Optimization method, and integrating a reinforcement learning procedure into the basic structure of such algorithm, producing the so called Ant-Q algorithm.

The second part reported in Sec. (1.2.1) and (1.2.2) was based on the genetic algorithm, whose classic implementation was improved with the integration of a K-means algorithm and a custom method of obtaining the shortest path by sub-sequentially joining the clusters.

After executing some performance tests of several runs over the datasets presented in Sec. (2), the provided results demonstrate an effective improvement of the hybridized approach with respect to the classic version of each algorithm.

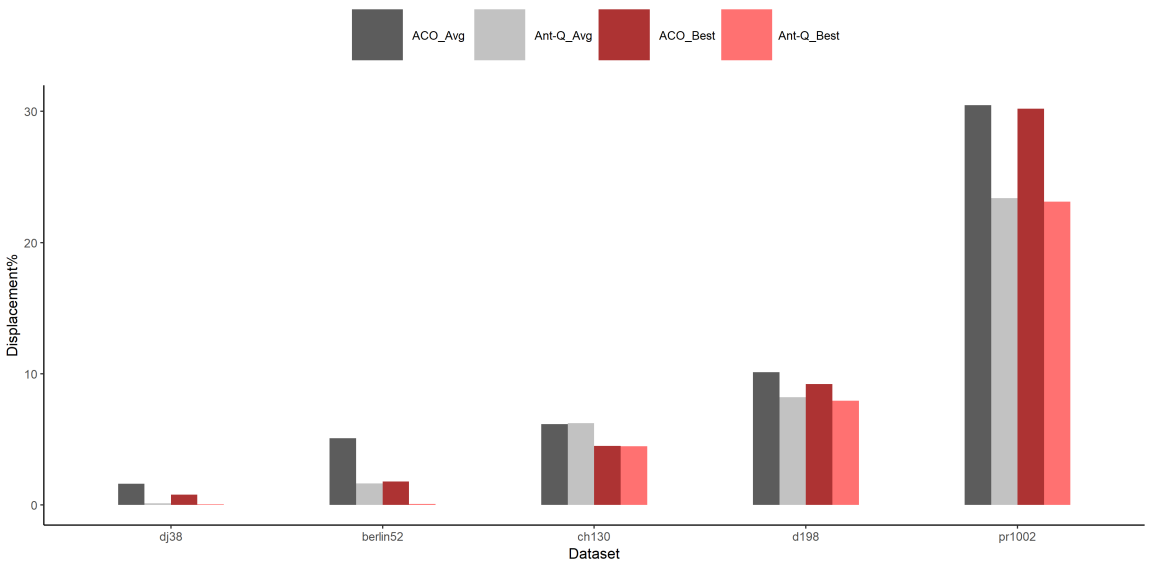

Figure 3: Percentage variation between AVG and Best ACO and Ant-Q results.

Furthermore, a comparison among the two types of methods can be done. The Ant family seems more likely to be applied on small or medium-sized problems, where it shows the best performances, whereas the evolutionary approach integrated with a clustering procedure is much more performant on large datasets, where it outperforms the classic Genetic implementation, with a reasonable cpu-time.

On large datasets both the families of algorithms do not obtain greatest results. The KGA utilization seems to be appropriate in order to have a "decent" sequence, that might be used for example as a starting point of another exploitation-based algorithm. The ACO and the Ant-Q becomes slow on higher dimensions running on four cpu-cores. However, further improvements in this direction could be obtained thanks to the parallel implementation, if a MPI cluster is provided. This would probably allow Ant-Q (in particular) to provide a good result even on large-scale problems.

Even further hybridization would be possible to implement, as a future scenario. For instance, it could be interesting to combine the ACO (or even better, the Ant-Q) and the KGA itself, leading to a method similar to that in Ref. [20].

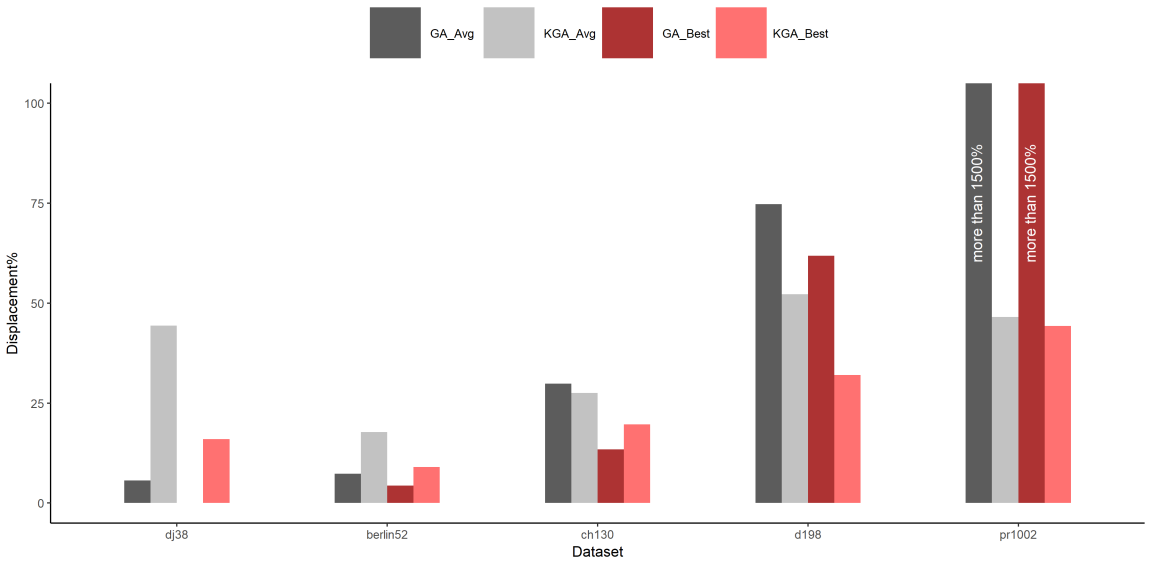

Figure 4: Percentage variation between AVG and Best GA and KGA results.

References

[1] Colorni A., M. Dorigo and V. Maniezzo, 1991.Distributed Optimization by Ant Colonies.Proceedings of ECAL91 - European Conference on Artificial Life, Paris, France, F.Varela and P.Bourgine(Eds.), Elsevier Publishing, 134–142.

[2] Dorigo, M., Maniezzo, V., and Colorni, A. Ant System: Optimization by a Colony of Cooperating Agents. In: IEEE Transactions on Systems, Man, and Cybernetics. Vol. 26, No. 1, 1996.

[3] Dorigo, M., and Gambardella, L. M. Ant Colony System: A Cooperative Learning Approach to the Travelling Salesman Problem. In: IEEE Transactions on Evolutionary Computation, Vol.1 No.1, pp.53-66, April 1997.

[4] Dorigo, M., Di Caro, G., and Gambardella, L. M. Ant algorithms for discrete optimisation. In: Artificial Life 5(2),pp.137-172, April 1999.

[5] Dorigo, M., and Stützle, T. Ant Colony Optimization, MIT Press, Cambridge, MA, 2004.

[6] Dorigo, M., Birattari, M., and Stützle, T. Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique. In: IEEE Computational Intelligence Magazine, November 2006.

[7] Gambardella, L. M. and Dorigo, M. Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem. In: Proc. ML-95, 12th Int. Conf. Machine Learning. Palo Alto, CA: Morgan Kaufmann, pp. 252–260, 1995.

[8] Sutton, R. S., and Barto, A. G. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 1998.

[9] Steinley, D. (2006), K-means clustering: A half-century synthesis. British Journal of Mathematical and Statistical Psychology, 59: 1-34.

[10] Sun, R., Tatsimu, S., Zhao, G., Multiagent Reinforcement Learning with an Improved Ant Colony System. In:IEEE Transactions on Systems, Man, and Cybernetics, Vol.3, pp.1612-1617, 2001.

[11] Miagkikh, V. and Punch, W. F. An Approach to Solving Combinatorial Optimization Problems Using a Population of Reinforcement Learning Agents. In: Genetic and Evolutionary Computation Conference, pp. 1358-1365, 1999.

[12] Monekosso, N., and Remagnino, P., The Analysis and Performance Evaluation of the Pheromone-Q-learning Algorithm. In: Expert Systems, Vol.21, No.2, pp.80-91, May 2004.

[13] Chagas De Lima Junior, Francisco & Neto, Adriao Duarte & Melo, J.D.. (2010). Hybrid Metaheuristics Using Reinforcement Learning Applied to Salesman Traveling Problem. 10.5772/13343.

[14] Razali, Noraini Mohd and John Geraghty. "Genetic Algorithm Performance with Different Selection Strategies in Solving TSP." (2011).

[15] A.J. Umbarkar and P.D. Sheth, Crossover Operators in genetic algorithms: a review. In: Ictact journal of soft computing, October 2015, Vol: 06.

[16] Suvarna Patil, Manisha Bhende, Comparison and Analysis of Different Mutation Strategies to improve the Performance of Genetic Algorithm. In: International Journal of Computer Science and Information Technologies, Vol. 5 (3) , 2014, 4669-4673.

[17] Kumar, Rakesh and Jyotishree. "Blending Roulette Wheel Selection & Rank Selection in Genetic Algorithms." (2012).

[18] L. Tan, Y. Tan, G. Yun and Y. Wu, "Genetic algorithms based on clustering for traveling salesman problems," 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), Changsha, 2016, pp. 103-108.

[19] Krishna, K and Murty, Narasimha M (1999) Genetic K-Means Algorithm. In: IEEE Transactions on Systems Man And Cybernetics-Part B: Cybernetics, 29 (3). pp. 433-439.

[20] An Analysis of Ant Colony Clustering Methods: Models, Algorithms and Applications Gong Zhe, Li Dan, An Baoyu, Ou Yangxi, Cui Wei, Niu Xinxin, Xin Yang, `https://pdfs.semanticscholar.org/6b8e/36adaa1cbe7781ebac1ff7903dfea492c59a.pdf`