Tryna Solve the Travelling Salesman Problem

David Nardi
MSc in Artificial Intelligence
david.nardi@edu.unifi.it
https://github.com/david-inf/Python-TSP

June 2024

Abstract

With this work we tackle the Travelling Salesman Problem (TSP) using few known heuristic algorithms. We start with two local search approaches that swap the edges of the cycle, once we saw that the local searches get trapped in local minima we moved forward to a multi-start meta-heuristic for each local search. Simulated annealing is then tested using a routine to initialize the temperature parameter.

Results show that the local search gets trapped in local minima due to the starting solution, multi-start solves this problem; the simulated annealing takes much more computation time but ends always in a better solution. Further studies may focus on reheating procedures for the simulated annealing like the simulated tempering.

Python implementation is available at this GitHub repo.

Contents

Modelling the TSP

1. Ops perspective
2. Smoothing the hard constraint, the pragmatic perspective

Tryna solve the TSP (for each solver the circular and random layouts)

1. Exact
    - Brute-force: check all possible permutations
2. Local search
    - `swap`
    - `swap-rev` or `reverse`
3. Simulated annealing (SA)
4. Multi-start (GRASP)

Results

- Convergence analysis and other diagnostics
- Energy view using multi-start
- What's next? Simulated tempering (SA with reheating), genetics algorithms

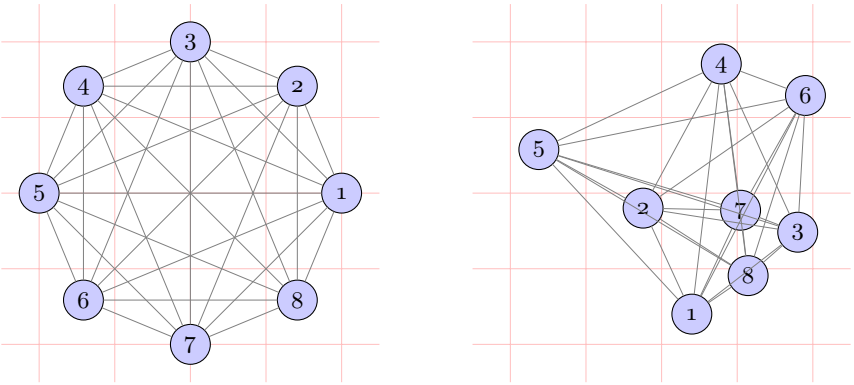
References

- Collega di Pisa? Quelli che trovavo su Medium? Paper trovati per caso?

1 Modelling the travelling salesman problem

Given an undirected (weighted) and complete graph $G = (V, E)$, where the set of all nodes $|V| = N$ and the set of all edges $\{i, j\}$ is $|E| = N(N-1)/2$ and their relative weight $c_{ij} > 0$. an Hamiltonian cycle is a cycle which touches every node in V once and only once, an Hamiltonian cycle might not exist.

We consider the two following layouts with all possible edges



on the left, a circular graph, for which we know that the optimal solution is a circular path; on the right each node is in a random position, so we don't know exactly the optimal solution, that is 1, 2, 3, 4, 5, 6, 7 and 8 but the same reversed too $f(\bar{x}_1) = f(\bar{x}_2)$.

We constraint the Hamiltonian cycle to start with the first node 1, so the total number of possible cycles reduces to $(N-1)!$.

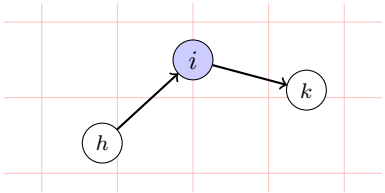
1.1 The optimization perspective

The Travelling Salesman Problem (TSP) consists in finding the *shortest Hamiltonian cycle*. The feasible set F is the set of all Hamiltonian cycles in G and the objective function $c(P)$ is the length of the Hamiltonian cycle P.

We make use of a logical variable for each edge

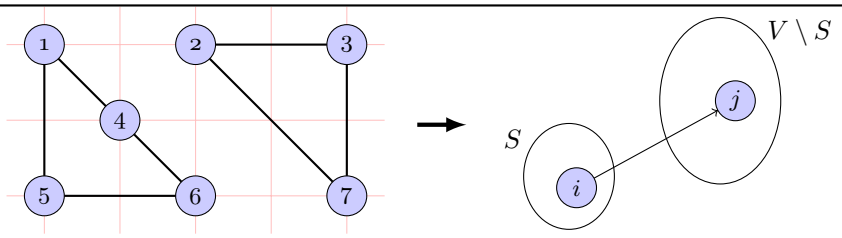
$$x_{ij} = \begin{cases} 1 & \text{if } i \to j \\ 0 & \text{otherwise} \end{cases}$$

hence the objective function si the sum of all costs associated with the edges in the cycle. This is the binary constraint on variables. Since a feasible solution is a cycle, we must impose the constraint that each node has one incoming and one outgoing edge



that is the *cycle covering* constraint.

Using only this constraint can result in a solution with sub-tours, so we add a constraint for connecting all possible sub-tours



from a subset S there must be at least an edge to the complementary set $V \setminus S$, the *connection* constraint, that is the *hard* constraint of the optimization problem.

The resulting optimization problem will be

$$\begin{aligned} \min \quad & c(P) = \sum_{\{i,j\} \in E} c_{ij} x_{ij} \\ & \sum_{\{i,j\} \in E} x_{ij} = 2 \\ & \sum_{i \in S, j \notin S} x_{ij} \geq 1 \quad \emptyset \subset S \subset V \\ & x_{ij} \in \{0, 1\} \end{aligned} \tag{1}$$

In practice, for simplicity, the cost for each edge is the distance between the two cities, so we use the distance matrix $D \in \mathbb{R}^{N \times N}$

$$D = (d_{ij}) = \begin{pmatrix} 0 & d_{12} & d_{13} & \cdots & d_{1N} \\ d_{21} & 0 & d_{23} & \cdots & d_{2N} \\ d_{31} & d_{32} & 0 & \cdots & d_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{N1} & d_{N2} & d_{N3} & \cdots & 0 \end{pmatrix}$$

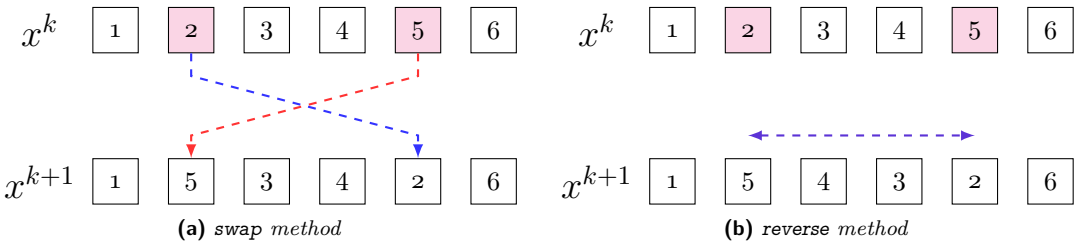
that is a symmetric matrix, whether an edge is in the cycle or not, the variables x_{ij} become the elements of the adjacency matrix A of the same shape of D. The objective function will be $c(P) = A \odot D$ that we call f for simplicity; the cycle covering constraint becomes the sum over rows and columns of A that must be equal to $2N$.

1.2 Smoothing the hard constraint

Each one of the algorithms considered uses of a starting solution x^0 and then performs a perturbation on that solution. We can smooth the hard constraint by keeping this solution each time and performing permutations of the nodes so that the new sequence is still a Hamiltonian cycle, hence a new point.

Here we have used two different methods, figure 1 on the following page, that randomly draw two different nodes given a solution x^k:

- `swap`: given the sequence, swap node i position with node j, figure 1a;
- `reverse`: reverse the nodes between i and j inclusive, figure 1b



(a) *swap method* **(b)** *reverse method*

Figure 1: Perturbation methods for generating new solutions

2 Heuristic algorithms

Heuristic algorithms do not guarantee the optimal solution but may give good solutions, we made use of local search and meta-heuristics.

2.1 Local search

The idea behind this class of algorithms is simple: given a feasible solution, there might be some other similar feasible solutions with lower objective function value. So the idea is to optimize the objective function by exploring the neighbourhood of the current point x^k in the solution space.

The local search starts with a feasible solution randomly drawn from the feasible set $x^0 \in F$. A generation mechanism is then successively applied in order to find a better solution, in terms of the objective function value $f(x^k)$, by exploring the neighbourhood of the current solution. Exploring the neighbourhood means perturbing the current solution, for this purpose we use the methods in figure 1.

The algorithm ends when no improvement can be found (or a maximum number or iterations k^* has been exceeded), and the current solution is considered as the approximate solution of the optimization problem.

Algorithm 1: Local search framework

Input: f, D, x^0
$x^* \leftarrow x^0$;
$k \leftarrow 0$;
while *stopping criterion not satisfied* **do**
  Generate a feasible solution x^k, see figure 1;
  if $f(x^k) < f(x^*)$ **then**
    $x^* \leftarrow x^k$;
  end
  $k \leftarrow k + 1$;
end
Output: x^*

2.2 Multi-start

Local search algorithms always ends in a local minima that is not the global optimum, this is due certainly to the perturbation method and the starting solution x^0 as well.

In order to make the local search independent from x^0, we can use multi-start meta-heuristics so that we can perform local searches starting from different starting solutions. Basically, we choose a number B of simulations, for each iteration a starting solution is randomly generated, then a local search is performed; finally the best solution is chosen.

This procedure is called Greedy Randomized Adaptive Search Procedure (GRASP), for large B we can be sure to obtain the global optimum since it is part of the feasible set from which the starting solutions are drawn.

Algorithm 2: Multi-start framework

Input: f, D
x^* s.t. $f(x^*) = \infty$;
for $b = 1, 2, \ldots, B$ **do**
  Generate a starting feasible solution x^0;
  Perform a local search (see algorithm 1) starting from x^0 to obtain $\hat{x}^b$;
  if $f(\hat{x}^b) < f(x^*)$ **then**
    $x^* \leftarrow \hat{x}^b$;
  end
end
Output: x^*

2.3 Simulated annealing

Local search falls in a subdomain over which the objective function is convex, in order to avoid being trapped in a local minima, it is necessary to define e process likely to accept current feasible solutions that momentarily reduce the objective function value.

Simulated Annealing (SA) can implement this idea, the acceptance of new solution is controlled by a *temperature* parameter, in such way the algorithm can consider past informations about the optimization process: once the algorithms ends in a good solution, similar solutions can be quite near the current one.

The simulated annealing uses the following parameters:

- T_k: temperature, starting from a initial value T_0 (that might be iteratively tuned), this parameter drives the search of the global optimum;
- α: cooling rate for the temperature parameter according to geometric cooling $T_{k+1}\alpha T_k$;
- L_k: length of the Markov Chain for which T_k = cost, it is the number of inner iterations for each k, on which new solutions are generated, each on of these iterations is called *transition*.

Inside each transition a new feasible solution x^t is generated as in the local search through methods from figure 1 on the preceding page; from statistical mechanics perspective, these perturbations allows to work in the canonical ensemble, the energy E_t (the objective function) is free to fluctuate but the number of particles (nodes in the graph) remains constant.

Once the solution is generated the algorithm checks if there is an improvement over the current best solution x^* (the sequence of $f(x^*)$ is constrained to be decreasing); then the Metropolis

acceptance criterion is applied using the energy gap $\Delta E_t = f(x^t) - f(x^k)$, this rule checks if there is an improvement over the current best inner solution x^k the criterion is as follows

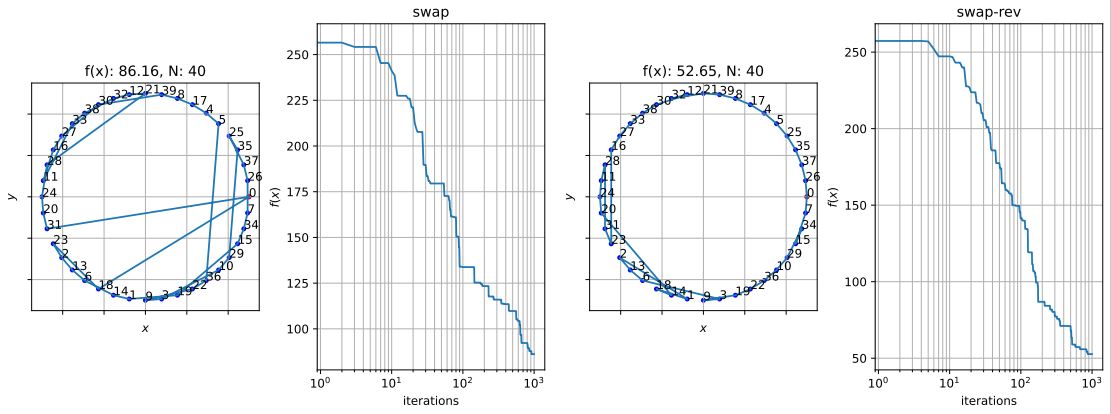
$$\mathbb{P}(\text{accept } x^t) = \begin{cases} 1 & \text{if } f(x^t) < f(x^k), \text{ i.e. } \Delta E_t < 0 \\ \exp(-\Delta E_t / T_k) & \text{otherwise, i.e. } \Delta E_t \geq 0 \end{cases}$$

the rule accepts the new solution based on the Boltzmann distribution, this criterion allows to accept up-hill moves that increase the objective function value $f(x^k)$, so the sequence $\{f(x^k)\}_k$ it is not necessarily decreasing.

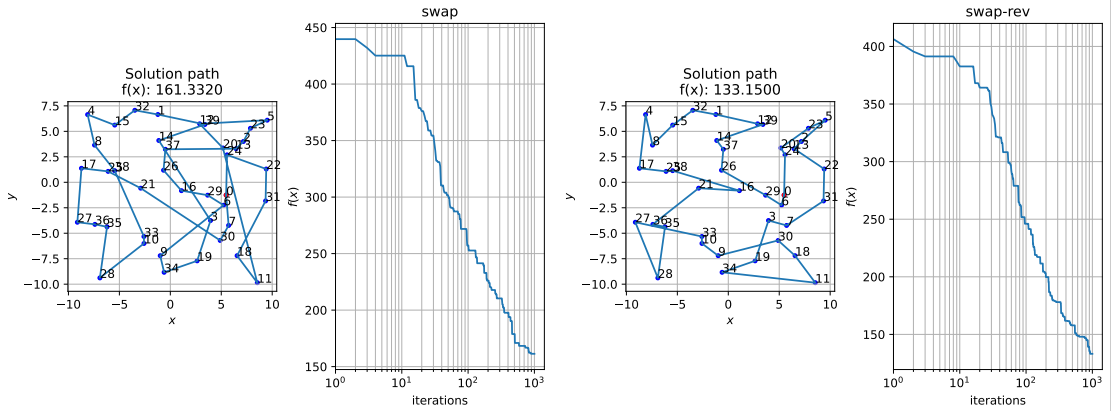
Algorithm 3: Simulated Annealing (SA)

Input: f, D, x^0, T_0, α, L_k
$x^* \leftarrow x^0$;
$k \leftarrow 0$;
while *stopping criterion not satisfied* **do**
  $x^k \leftarrow x^*$;
  for $i = 1, 2, \ldots, L_k$ **do**
    Generate a feasible solution x^t, see figure 1;
    if $f(x^t) < f(x^*)$ **then**
      $x^* \leftarrow x^t$; // down-hill, new best solution
    end
    if $f(x^t) < f(x^k)$ **then**
      $x^k \leftarrow x^t$; // down-hill note that $f(x^k) \geq f(x^*)$
    else
      Generate a random number $r \sim U(0, 1)$;
      $\Delta E \leftarrow f(x^t) - f(x^k)$; // energy gap
      if $r < \exp(-\Delta E / T_k)$ **then**
        $x^k \leftarrow x^t$; // up-hill, lower quality solution accepted
      end
    end
  end
  $T_{k+1} \leftarrow \alpha T_k$;
  $k \leftarrow k + 1$;
end
Output: x^*

3 Results

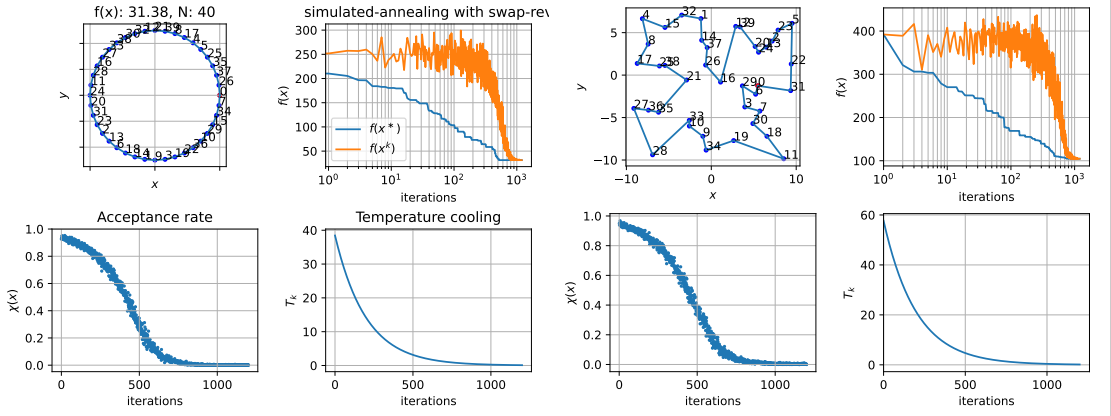


(a) *swap method on circular layout* **(b)** *reverse method on circular layout*



(c) *swap method on random layout* **(d)** *reverse method on random layout*

Figure 2: Local search algorithms performance



(a) *Circular layout* **(b)** *Circular layout*

Figure 3: Simulated annealing performance