École Polytechnique — CSE202 Design and Analysis of Algorithms

Approximate algorithms for solving TSP

In the previous tutorial we saw algorithms for solving the Traveling Salesman Problem (TSP) in a precise manner. These algorithms are too slow to be used for large inputs (graphs). This performance issue is seemingly unavoidable since TSP is NP-complete (unless P=NP).

In this tutorial we will implement approximate algorithms for solving TSP : algorithms which trade the optimality of the result for a better complexity (hence enabling to tackle larger instances of the problem). These algorithms apply to *Euclidean* TSP instances, where the graph nodes represent points in a plane and the weights of the edges represent Euclidean distances between these points. (Every two nodes are connected by an edge.)

Download the different python files on moodle and the text files containing the data of some Euclidean TSP instances. The file `WG.py` contains the correction of the previous tutorial with some additional utility functions and the signature of the functions you will have to implement (at the end of the file). The previous tutorial definitions are used. In particular, a given (approximate) solution circuit T for TSP is represented in Python as a list containing the sequence of visited locations (nodes) (in the order they appear in the list). For example, a list `T = ['A','B','D','C','E']` represents the circuit starting at A visiting B, D, C and E consecutively and going back to A.

You will see in the following two different algorithms : one for constructing a (good) sub-optimal circuit, and one for improving an existing circuit using "local" operations.

Debugging : The file `test.py` contains test functions for each question. Call `testi()` where i is the question number to test the corresponding code. The tests are not exhaustive.

1 Constructing a circuit : a greedy algorithm

We consider a greedy algorithm to construct a circuit, which resembles the Kruskal algorithm for computing a minimum spanning tree. The algorithm starts with the list of edges sorted by increasing weight (distance). It picks the first edge in this list (with minimal weight) and considers it to be part of the circuit that is being constructed. The next iterations traverse the sorted list of edges (in increasing order), take edges one by one, and try to add them to the constructed path if it is still possible to end up with an Hamiltonian circuit (no vertex of the circuit has a degree larger than 2 and every vertex is connected to every other vertex). Therefore, when extending a path with a new edge, one must check that the new edge does not create a *sub-circuit* (a circuit traversing a strict subset of the nodes), and it does not increase the degree of a node beyond 2 when looking only at the current selection of edges (otherwise, a node will be traversed more than once). The algorithm stops once a Hamiltonian circuit (going through all locations exactly once) is obtained.

We will implement this algorithm in the context of the class `WG` which represents a weighted undirected graph. The field `edges` stores the list of edges sorted by weight, and the field `self.adj` is a dictionary storing the adjacency matrix. The list of nodes can be obtained as `self.adj.keys()`.

Question 1. Implement the method `greedy_select_edges(self)` that, given the list of edges ordered by weight (`self.edges`), returns a set of n edges composing a circuit selected according to the above description (n is the number of graph vertices).

Hint. Use the Union-Find class `UF` to check if two nodes are connected in the constructed circuit. See the method `weight_min_tree` as an example, or recall the use of Union-Find in TD10.

Question 2. Implement the method `build_circuit_from_edges(self,edges)` that takes as input a set of edges describing a circuit and builds the list T of visited locations in the order defined by those edges. Returns a tuple (w, T) where w is the weight of the circuit T. This method should have linear complexity in terms of the length of `edges`.

Then use the previous methods to implement the method `greedy_min(self)` which builds a circuit using the greedy algorithm.

2 Improving a circuit : a 2-Opt neighborhood search

When given a non-optimal circuit, one can try to improve it by exploring some potential modifications to it. This principle is called *Neighborhood search* or *Local search*.

Consider a TSP instance with n locations and the set H of all feasible circuits on this instance. A neighborhood $N(T)$ of the circuit T can be defined as the subset of H of circuits that can be reached from T after executing a given transformation pattern. Different patterns exist and can lead to neighborhoods of various sizes (and different complexity). In the case of TSP we are interested in the 2-Opt pattern, which is typically tailored to resolve crossings in the circuit T. Figure 1 illustrates the 2-Opt operation.

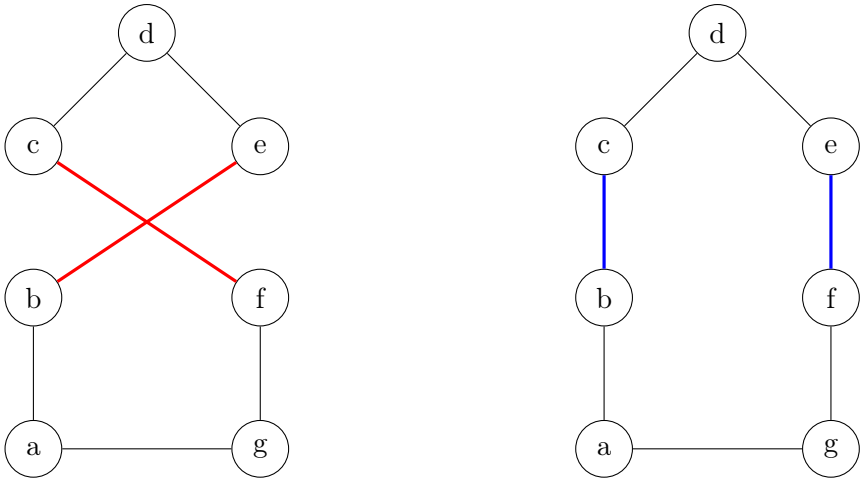

FIGURE 1 – Left : Circuit **abedcfg**. A crossing (in red) appears. Right : Circuit **abcdefg** obtained after flipping the sub-path **edc**.

The principle of a 2-Opt transformation of a circuit is as follows : select two distinct edges and swap their beginning and endpoints so as to build another circuit. In Figure 1, the edges (b, e) and (c, f) are replaced by (b, c) and (e, f). The circuit on the right is a neighbor of the one on the left according to 2-Opt. In this particular case, the neighbor on the right has a lower weight as it does not contain the crossing (recall that weights represent Euclidean distances).

You will have to implement a Neighborhood search based on 2-Opt : given a circuit T, explore its neighborhood $N(T)$ and find a better candidate circuit T'. Repeat the process on T' until no better circuits exist in the neighborhood.

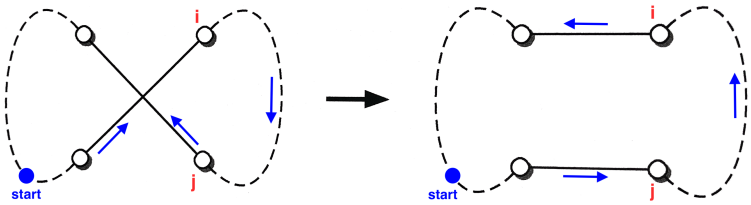

FIGURE 2 – A flip at two given positions i and j.

Question 3. Implement the method `evaluate_flip(self,T,i,j)` that, given a circuit T and two distinct indices i and j, returns the gain g of flipping the order of the locations between i and j (included). See Figure 2.

The gain g is defined as the weight of the removed edges minus the weight of the added edges involved by the flip. (Do not perform the flip in this function.).

Question 4. Implement the method `find_best_opt2(self,T)` that, given a circuit T, computes the flip leading to the largest positive gain (again, without performing the flip). Returns a tuple $((i, j), g)$

consisting of the flip as a pair of indices (i, j) and the corresponding gain g. If no gain greater than zero can be achieved, returns `(None,0)`.

Question 5. Implement the method `do_flip(self,T,i,j)` that flips (reverses) the order of the elements of T between the indices i and j (included).

Implement the method `opt_2(self,w, T)` that performs the best possible flip (if it exists) on T and updates its weight w accordingly. It returns a pair (w, T).

Question 6. Implement the method `neighborhood_search_opt2(self,w, T)` that repeats 2-Opt transformations on the circuit T until no improving transformation can be found. It returns the weight-circuit pair (w, T) of the improved circuit.

3 Experiments

The file `test.py` contains a function `compare_approx()` which compares the performances (best solution achieved and computational time) of the different approximation algorithms :

- Randomly generated circuit (red, minimal weight obtained and average computational time over several trials)
- The greedy algorithm (blue)
- The 2-Opt neighborhood search starting from the greedy circuit (purple)
- The 2-Opt neighborhood search starting from a random circuit (green, minimal weight obtained and average computational time over several trials).

The TSP instances are randomly generated, from 5 locations up to 50.

Question 7. Before trying `compare_approx()`, try to rank the different methods according to which one you think will bring the best results (minimal weight and minimal computational time).

Now call `compare_approx()` (several times for mitigating the randomness). What do you observe ?

Remark. The influence of the initial circuit on the Neighborhood search is important. A bad one may require a very long search to converge. A good one can make the search converge quickly, but this does not necessarily imply the best quality : a random initial circuit may lead to a better circuit. In order to achieve better circuits within a reasonable time, you can think about introducing some randomness within the construction procedure of the initial circuit (for example see the method GRASP, Greedy Randomized Adaptive Search Procedure).

As the neighborhood we use is of fixed size, it may happen to be stuck at a non-optimal and bad quality circuit, failing to find an improving circuit among its neighbors. You can try to solve this situation by increasing the size of the neighborhood when necessary (for example, switching to a k-opt). Another possibility is to accept some non-improving moves inside the neighborhood. This is the principle of methods like Simulated Annealing.

Remark. The functions `run_eu_instance()`, `run_us_instance()` and `run_drill_instance()` contains a call to the greedy algorithm and its improvement with 2-Opt neighborhood search on respectively an instance with 24 European cities, 48 US cities and 280 drill locations on an electronic chip. It compares the result of the approximate algorithm with a lower bound on the optimal weight obtained by the minimal spanning tree.

Try to improve your results on these instances by proposing variants of your approximate algorithms.