École Polytechnique — CSE202 Design and Analysis of Algorithms

Approximation algorithms for solving TSP

In the previous tutorial, we saw *exact* algorithms for solving the Traveling Salesman Problem (TSP). These algorithms are too slow to be used for large inputs (that is, graphs). Unfortunately, this performance issue is seemingly unavoidable (unless $\mathsf{P} = \mathsf{NP}$), since TSP is NP-complete.

In this tutorial, we implement *approximation* algorithms for solving TSP. These algorithms forgo the optimality of the result and aim instead for a faster run time (hence, enabling to tackle larger instances of the problem). They 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, that is, they obey the triangle inequality. Note that this also implies that the respective graph is fully connected.

Download the various Python and text files from Moodle. The file `WG.py` contains a solution of the previous TD, with some additional utility functions. Further, this file contains the skeleton code of the functions you have to implement (at the end of the file). You can test your solutions by running the file `text.py`.

We use the definitions similar to those in the previous TD. In particular, we represent an approximate solution T for TSP (that is, a Hamiltonian cycle) as a list containing the sequence of visited locations nodes in the order they appear in the list. For example, the list `T = ['A', 'B', 'D', 'C', 'E']` represents the cycle starting at `A` and then visiting `B`, `D`, `C`, and `E` consecutively before going back to `A` (which is *not* represented explicitly in the list now!).

In the following, we consider two different algorithms: one for constructing a (good) sub-optimal cycle, and one for improving an existing cycle using *local* operations.

1 Constructing a Hamiltonian cycle greedily

We consider a greedy algorithm that resembles Kruskal's algorithm. It aims to greedily construct a sequence of *edges* C such that the nodes along these edges form a Hamiltonian cycle. Note that this is always possible, since we assume that the graph is fully connected.

The algorithm starts by sorting the list of edges by increasing weight, and it then iterates the edges in that order. In each iteration, the algorithm tries to add the current edge e to C. We add e to C if e connects to vertices that have a degree of at most 1 when considering the edges in C *and* if this does not create a cycle that does not contain all nodes. Otherwise, we skip e and do not add it to C. The algorithm stops once all nodes are part of C, and it returns C.

We 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 increasing weight, and the field `self.adj` is a dictionary storing the adjacency matrix. The list of nodes can be obtained via `self.adj.keys()`.

Question 1. Complete the method `greedily_select_edges(self)`, which should return a `set` of edges that compose a Hamiltonian cycle, following the greedy approach above. Note that such a set has as many edges as the graph has vertices. Further note that in our setting, an edge is a `tuple` of three elements: the first two entries are its incident nodes, and the last entry is its weight!

Hint. Use the Union–Find class `UF` in order to check if an edge closes a cycle that is not Hamiltonian. See the method `weight_min_tree` as an example, or recall the use of Union–Find from TD10.

Question 2. Complete the method `build_cycle_from_edges(self, edges)`, which is given a set of edges describing a cycle. Let `T` be a list of *nodes* that make up the cycle given by `edges`, and let `W` denote its weight. The method should return the tuple `(W, T)`. Note that the first node of `T` does not matter, only the order of nodes is relevant. Recall that `T` does *not* repeat its first node at the end.

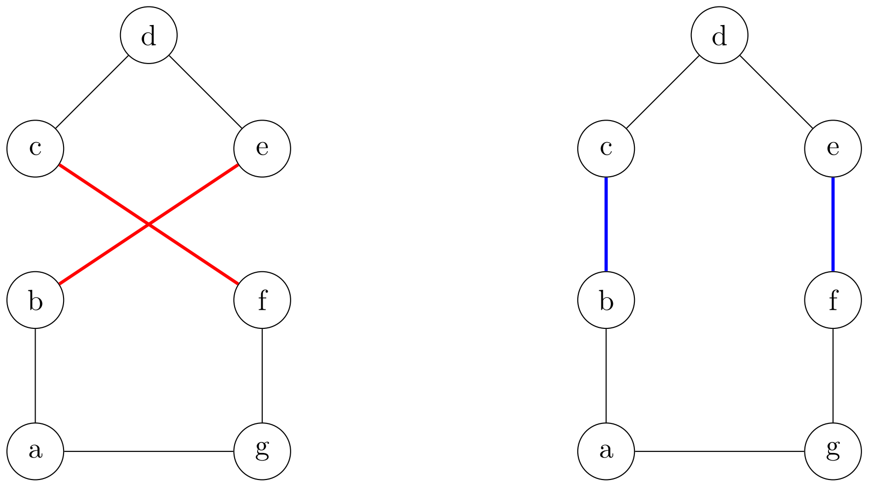

Figure 1: Left: cycle **abedcfg**. A crossing (in red) appears. Right: cycle **abcdefg** obtained after flipping the edges **be** and **cf**.

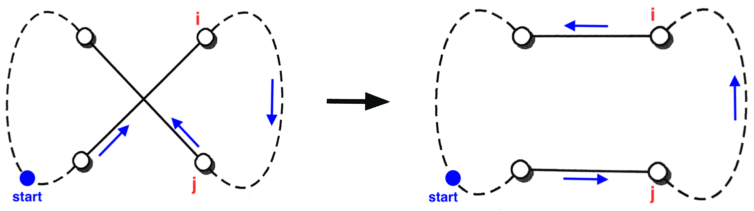

Figure 2: A flip at two given positions i and j.

2 Improving a cycle: a 2-opt neighborhood search

When given a cycle with non-minimal weight, 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 nodes and the set $\mathcal{H}$ of all feasible Hamiltonian cycles of this instance. For a Hamiltonian cycle T, we define the *neighborhood* $N(T)$ of T as the subset of $\mathcal{H}$ of cycles that can be reached from T after executing a certain transformation. In general, different transformations exist and can lead to neighborhoods of various sizes (and different complexity). We consider the 2-*opt* transformation (see Figure 1), which is tailored to resolve crossings in T.

The principle of 2-opt is as follows: Select two distinct edges and swap their beginning and end points. For example, in Figure 1, the edges (b, e) and (c, f) are replaced by (b, c) and (e, f). The cycle on the right is a 2-opt neighbor of the one on the left. In this case, since we consider *Euclidean* TSP instances, the neighbor on the right has a lower weight, as it does not contain the crossing.

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

Question 3. Complete the method `evaluate_flip(self, T, i, j)`, which is given a cycle `T` and two distinct indices `i` and `j`. It should return the gain g of weight when flipping the edges between `i` and `j`. See also Figure 2.

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

The class `WG` contains the function `get(self, T, i)`, which returns `T[i % len(self.adj)]`, that is, it allows to get access to the elements of `T` cyclicly, thus preventing the index going out of bounds.

Question 4. Complete the method `find_best_opt2(self, T)`, which is given a Hamiltonian cycle `T`. It should compute the flip with the largest positive gain `g` in `T` (again, without performing the flip). Let `i` and `j` denote the two nodes with a gain of `g`. The method should return the tuple `((i, j), g)`. If no gain greater than zero can be achieved, the method should return `(None, 0)`.

Question 5. Complete the method `opt_2(self, W, T)`, which performs the best possible flip (if it exists) on `T` and updates its weight `W` accordingly. It should return the pair `(W, T)`. You can use the method `flip(self, T, i, j)`, which reverses the path starting at the node at index `i` and stops at the node at index `j`, thus performing a flip in the list of nodes.

Question 6. Complete the method `neighborhood_search_opt2(self, W, T)`, which repeatedly applies 2-opt to the Hamiltonian cycle `T` with weight `W` until no improving transformation is found. The method should return the pair `(W, T)` of the last improved cycle.

3 Fun extra activities

The file `test.py` contains additional functions for solving larger instances. The function `compare_approx()` compares the performances of the following different approximation algorithms and showcases them in the following colors:

- A randomly generated Hamiltonian cycle (magenta; uses the average of multiple trials).
- The greedy algorithm (blue).
- The 2-opt neighborhood search starting from the greedy cycle (purple).
- The 2-opt neighborhood search starting from a random cycle (green, uses the average of multiple trials).

The function runs two sets of experiments. The first measures the weight of the final solution of each algorithm. The second measures the run time of each approach. The TSP instances are randomly generated. Their sizes range from 5 nodes up to 50.

Remark. Before running `compare_approx()`, try to rank the different methods with respect to both measures (minimum weight and run time).

Now call `compare_approx()` (several times for mitigating the randomness) and see whether your thoughts were correct.

Remark. The initial cycle has a huge impact on the neighborhood search. 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 cycle may lead to a better cycle. In order to achieve better cycles within a reasonable time, you can think about introducing some randomness to the construction procedure of the initial cycle (see, for example, the method *greedy randomized adaptive search procedure* (GRASP)).

Since the neighborhood we use is of fixed size, the search may be stuck in a local optimum. 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 or non-elitist evolutionary algorithms.

Remark. The remaining functions in `tests.py` are `run_eu_instance()`, `run_us_instance()`, and `run_drill_instance()`. Each 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. Each function compares the result of the approximation algorithm with a lower bound on the optimal weight obtained by a minimum spanning tree.

You can try to improve your results on these instances by proposing variants of your approximation algorithms.