UNIVERSITÀ DI SIENA
1240

Family of Local Search for TSP
Implemented classes and functions

Network Optimization

Students:

- Filippo Guerranti
- Alessandro Carfora
- Cai Haihua
- Emre Özbas

Professor:

Marco Pranzo

Assignment

The project focus on implement and test a family of local search algorithms for TSP. Computational tests can be carried out on instances from the TSPLIB.

Contents of this paper

In this paper we will provide a quick description of the classes and functions implemented in order to carry out the project.

Library map

```
localsearch_tsp
|---- README.md
|---- code
|      |---- localsearch
|      |      |---- __init__.py
|      |      |---- solvers.py
|      |      |      |---- Solver
|      |      |      |---- NN
|      |      |      |---- RepNN
|      |      |      |---- TwoOpt
|      |      |      |---- NN2Opt
|      |      |      |---- RepNN2Opt
|      |      |      |---- NN2OptDLB
|      |      |      |---- RepNN2OptDLB
|      |      |      |---- ThreeOpt
|      |      |      |---- NN3Opt
|      |      |      |---- RepNN3Opt
|      |      |      |---- NN3OptDLB
|      |      |      |---- RepNN3OptDLB
|      |      |---- tsp.py
|      |      |      |---- Generator
|      |      |      |---- Loader
|      |      |      |---- TSP
|      |---- test.ipynb
```

Initialization To use the classes and the functions for this project, one must first download the codes from the GitHub page in which they are hosted [`https://github.com/L4plac3/localsearch_tsp.git`]. Once done that, it is easy to import them using the following code:

```
import localsearch.solvers as slv
import localsearch.tsp as tsp
```

Generator() The `Generator` class is used in order to generate a random set of nodes with given cardinality and arranges them into a `numpy` matrix having two columns (correspondent to the `x`-coordinate and to the `y`-coordinate of each node) and a number of row that is equal to the number of nodes generated.

The class is simply composed by its constructor which takes in input the `number` of nodes we want to generate and the dimensions of the rectangle in which they have to be placed (`width` and `height`).

```
generator = tsp.Generator(nodes=30,width=900,height=600)
```

The code shown above will generate 30 random nodes having `x`-coordinate between 0 and 900 and `y`-coordinate between 0 and 600.

Loader() The `Loader` class is used to load a `.tsp` file into a `numpy` matrix having two columns (the components `x` and `y` which are the coordinates of the node) and `N` rows, where `N` is the number of total nodes.

The class has only one method (the constructor) which, given the path of the `.tsp` file as an argument, reads each line of the file and saves the coordinates of the nodes in `self.nodes`, one of the class' properties.

We show the usability of the class: after importing the `localsearch` library we can instantiate an object of class `Loader` as follows, passing it the path of the file we want to load as a parameter.

```
loader = tsp.Loader(r'..\\data\\uy734.tsp')
```

TSP() The `TSP` class is the main class, used to handle all the properties of the TSP instance.

```
t = tsp.TSP(loader.nodes)
```

or

```
t = tsp.TSP(generator.nodes)
```

With its constructor it is possible to initialize:

- an empty dictionary that will contain all the routes computed applying the different solvers to the tsp instance with their cost, their path and the time elapsed to compute the route;
- the numpy matrix containing the nodes loaded via the `Loader` class or generated by the `Generator` class;
- the distance matrix which is computed via the method `distMatFromNodes`.

We provide the class with the following methods:

```
def distMatFromNodes(self, nodes, dist='euclidean')
def solve(self, solver)
def getResults(self)
def printResults(self)
def plotData(self)
def plotSolution(self, route_key)
```

The `distMatFromNodes` method computes the distance matrix of the nodes. It takes the list of nodes and the type of distance to compute as input parameters and saves in `self.dist_mat` the matrix of the distances between each node.

```
self.dist_mat = squareform(pdist(nodes, dist))
np.fill_diagonal(self.dist_mat, np.inf)
```

The functions `squareform` and `pdist` are imported from `scipy.spatial.distance`.

The `solve` method is the one which provides the solution of the TSP by applying the solver that is given as an input parameter. After calling the `solve` method of the `Solver` instance, it saves the computed path, cost and time into `self.routes` which is the dictionary that will contain all the information about the solution provided by the given solver.

The `getResults` and `printResults` methods are the ones designed for returning the `self.routes` as a dictionary (the first one) and as a screen-printed table (the second one).

The `plotData` method is used to plot the original nodes imported via the `Loader` class or generated by the `Generator` class.

Lastly, the `plotSolution` method prints the solution provided by the `route_key`, which represents the name of the solver whose solution we are interested in.

Solver() The `Solver` class is the basic class used as a parent for all the solvers we implemented in the project. It has a constructor which takes the `initial_node`, the `initial_path` and the `initial_cost` as input parameters. Then an abstract method `solve` is defined in order to be inherited by all the child classes and a `cost` method is implemented to compute the cost of the heuristic path for the specific solver.

NN(Solver) The `NN` class implements the Nearest Neighbour solver. The `solve` method (inherited from `Solver`) works as follows:

1. start from an initial node (passed as input parameter to the constructor), assign it to the variable containing the current node and add the current node to the path;

2. set the total cost equal to zero;
3. find the nearest node above all the not visited yet nodes in the TSP instance to the current node;
4. add the nearest node to the path;
5. add the distance between the current node and the nearest node to the total cost;
6. set the current node equal to nearest node;
7. go to step 3 if any of the node has still not been added to the path;
8. return the path and the total cost.

RepNN(NN) The `RepNN` class implements the so called Repeated Nearest Neighbour. It has the usual `solve` method (inherited from `NN`) that apply the Nearest Neighbour algorithm starting from each of the nodes and keeps the cheapest path among all.

TwoOpt(Solver) The `TwoOpt` class is the parent class responsible for implementing the `2-opt` algorithm. This algorithm takes an already complete but not so good path and finds a better neighbour solution. It works in the following way: given an initial path repeat the next steps until no other improvement can be done:

1. let i iterates over all the nodes till $N-2$ where N is the number of nodes of the TSP instance;
2. let j iterates over all the nodes starting from $i+2$ till $N-1$ (if $i == 0$) or N (otherwise);
3. compute the gain that you will get by applying the 2-opt move to the i-th and j-th nodes;
4. if the gain is greater than 0, then you will apply the 2-opt move, else you continue.

The previous steps gets their meaning once we describe the two following functions:

```
def gain(self, i, j, dist_mat)
def swap(self, i, j)
```

The first one (`gain`) is the one which computes the gain of the 2-opt move. So, given the indexes i, $i+1$, j, $j+1$ and the distance matrix of the TSP instance, we find the four nodes they are correspondent to: `A`, `B`, `C`, `D`. Then we compute the distances as shown below:

```
dAB = dist_mat[A,B]
dCD = dist_mat[C,D]
dAC = dist_mat[A,C]
dBD = dist_mat[B,D]
```

After that, it is possible to define the initial distance `d1 = dAB + dCD` and the final distance (after the 2-opt move) `d2 = dAC + dBD`. The function returns the difference `d1 - d2`.
The second function (`swap`) makes the 2-opt move. So, given the indexes of the nodes to swap, it reverses the path between those nodes (`[---,A,B,---,C,D,---] -> [---,A,C,---,B,D,---]`).

The `TwoOpt` class is also able to apply the Don't Look Bits speed-up providing it as boolean value to the class instance. In that case the function is slightly modified in order to carry out the aforementioned speed-up.

- **DLB**: we start by setting the DLB flags for each node to `False`. Then we apply the usual 2-opt algorithm: if the i-th node's flag is set to `True` we skip it, otherwise we proceed with the algorithm. Once the inner cycle (`for` loop over j) is completed without improvements, the flag of the i-th node is set to `True`.

NN2Opt(TwoOpt) The `NN2Opt` class is the one that applies the 2-opt algorithm to an initial path computed via the Nearest Neighbour algorithm. It simply has the `solve` method which is inherited from `TwoOpt` class and which set the initial path and the initial cost to the ones computed via Nearest Neighbour and, in particular, by applying the `solve` method of the `NN` class.

```
solver = NN()
solver.solve(tsp)
self.initial_path = solver.heuristic_path
self.initial_cost = solver.heuristic_cost
super().solve(tsp)
```

RepNN2Opt(TwoOpt) As for the `NN2Opt` class, the `RepNN2Opt` applies the 2-opt algorithm to an initial path, this time computed via the Repeated Nearest Neighbour approach.

NN2OptDLB(TwoOpt) The `NN2OptDLB` class derives from `TwoOpt` and applies that algorithm to an initial path computed via the Nearest Neighbour approach. Differently from the class `NN2Opt`, in this case the Don't Look Bits speed-up is applied (by setting `dlb=True` in the constructor of the base class).

```
def __init__(self):
    super().__init__(dlb=True)
```

RepNN2OptDLB(TwoOpt) Same as `NN2OptDLB` class but in this case the initial path is computed via the Repeated Nearest Neighbour algorithm.

ThreeOpt(Solver) The `ThreeOpt` class is the parent class responsible for implementing the `3-opt` algorithm. This algorithm takes an already complete but not so good path and finds a better neighbour solution. It works in the following way: given an initial path repeat the next steps until no other improvement can be done:

1. let i iterates over all the nodes till $N-4$ where N is the number of nodes of the TSP instance;
2. let j iterates over all the nodes starting from $i+2$ till $N-2$;
3. let k iterates over all the nodes starting from $j+2$ till $N-1$ (if $i == 0$) or N (otherwise);
4. compute the gain that you will get by applying the 3-opt move to the i-th, j-th and k-th nodes and also return the case which gives the best gain;
5. if the gain is greater than 0, then you will apply the 3-opt move related to the specific case, else you continue.

The previous steps gets their meaning once we describe the three following functions, as we did for the `TwoOpt` class:

```
def gain(self, i, j, k, dist_mat)
def move(self, i, j, k, opt_case)
def swap(self, i, j)
```

The first one (`gain`) is the one which computes the gain of the 3-opt move and returns the case which provides the best move. So, given the indexes i, $i+1$, j, $j+1$, k, $k+1$ and the distance matrix of the TSP instance, we find the six nodes they are correspondent to: `A`, `B`, `C`, `D`, `E`, `F`. Then we compute the seven possible distances (each one related to the correspondent 3-opt move). Then the case corresponding to the best gain and the gain itself are returned.
The second function (`move`) takes in input the indexes i, j, k and the case corresponding to the best gain and computes the 3-opt move using the `swap` function (defined as in the 2-opt framework).

The `ThreeOpt` class, like the `TwoOpt`, is also able to apply the Don't Look Bits speed-up providing it as boolean value to the class instance. In that case the function is slightly modified in order to carry out the aforementioned speed-up.

- **DLB**: we start by setting the DLB flags for each node to `False`. Then we apply the usual 3-opt algorithm: if the i-th node's flag is set to `True` we skip it, otherwise we proceed with the algorithm. Once the inner cycles (`for` loop over j and over k) are completed without improvements, the flag of the i-th node is set to `True`.

NN3Opt(ThreeOpt) The `NN3Opt` class is the one that applies the 3-opt algorithm to an initial path computed via the Nearest Neighbour algorithm. It simply has the `solve` method which is inherited from `ThreeOpt` class and which set the initial path and the initial cost to the ones computed via Nearest Neighbour and, in particular, by applying the `solve` method of the `NN` class.

```
solver = NN()
solver.solve(tsp)
self.initial_path = solver.heuristic_path
self.initial_cost = solver.heuristic_cost
super().solve(tsp)
```

RepNN3Opt(ThreeOpt) As for the `NN3Opt` class, the `RepNN3Opt` applies the 3-opt algorithm to an initial path, this time computed via the Repeated Nearest Neighbour approach.

NN3OptDLB(ThreeOpt) The `NN3OptDLB` class derives from `ThreeOpt` and applies that algorithm to an initial path computed via the Nearest Neighbour approach. Differently from the class `NN3Opt`, in this case the Don't Look Bits speed-up is applied (by setting `dlb=True` in the constructor of the base class).

```
def __init__(self):
    super().__init__(dlb=True)
```

RepNN3OptDLB(ThreeOpt) Same as `NN3OptDLB` class but in this case the initial path is computed via the Repeated Nearest Neighbour algorithm.