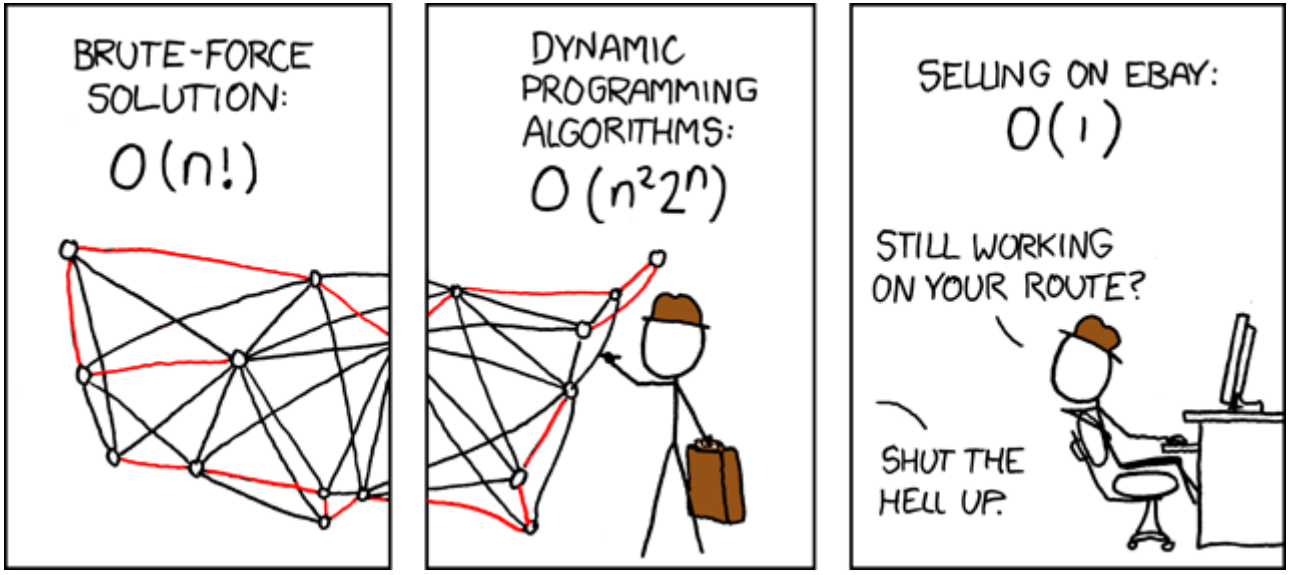
CS 335: Lab Assignment 8

(TA in charge: Manas Bhargava)

The **traveling salesperson problem** (**TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an NP-hard problem in combinatorial optimisation, important in operations research and theoretical computer science. In this assignment you will implement **hill climbing** based search algorithms to solve TSP. We will consider only the *Euclidean* version of TSP, in which the cities all lie on a 2-dimensional plane. Each city will have (x,y) coordinates, and the euclidean distance between two cities will be the weight of the edge connecting them.



(image source: https://xkcd.com/399/.)

Code

Download this code base.

hill climb **Parameters to**

You have to make changes in `hillclimb.py`. This file contains helper functions such as `getRandomTour()` and `getTourLength()` which will be useful while implementing TSP solutions. Running `python3 hillclimb.py -h` will list parameters, as shown here. Go through the file once and make sure that you understand the usage of each of the helper functions.

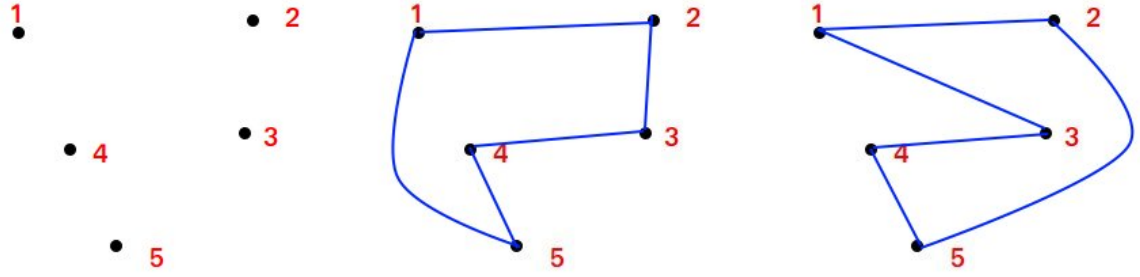
`hillclimb.py`

--file	TSP instance file.
--cities	Number of cities.
--r1seed	Random seed to generate TSP instance file with number of cities provided in '--cities' option (use --cities and --r1seed options only if '--file' option not used).
--r2seed	Random seed to generate random tour which is used for different tasks in this assignment.
--start_city	Starting city for nearest neighbour algorithm (Task 3) and Minimum Spanning Tree-based initialisation (Task 4).
--submit	Option to generate graphs that you need to submit.

An input instance of a TSP problem (for example, see data/berlin52.tsp in the `data` folder) has one line corresponding to each city. This line specifies the index of the city, the x coordinate, and the y coordinate. In `hillclimb.py` , you will find the `Node` data structure. A node contains the index of a city, along with its x and y coordinates.

A tour is defined as a permutation of indices of nodes a_1, a_2, ..., a_n (where a_i is the index of node provided in the input file). Suppose b_1, b_2, ..., b_n is a tour: it means the salesperson starts the tour at b_1, then goes to b_2, and so on until finally returning to b_1 from b_n. So a tour is just a list of node indices.

For example, some tours in the instance shown below (left) are [1, 2, 3, 4, 5] (middle) and [2, 1, 3, 4, 5] (right). Any rotated version or mirror image of a tour is the same tour (e.g. 1,2,3,4,5 is the same as 2,3,4,5,1 which is the same as 3,4,5,1,2 and so on. Similarly 1,2,3,4,5 is same as 5,4,3,2,1).



In the code, the following variables, functions and data structures are implemented for your convenience.

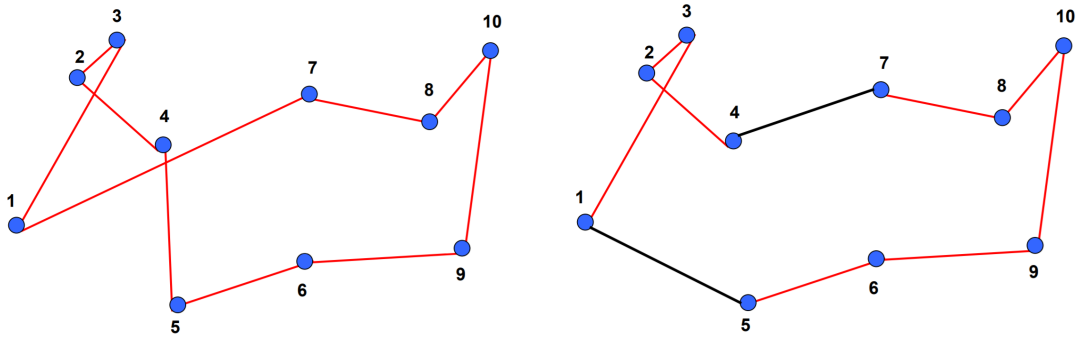
- **Node**: Class which has three member variables: index of city, x-coordinate and y-coordinate.
- **cities**: denotes the number of cities in the graph.
- **nodeDict**: This is a dictionary with its key as index of a node and the Node instance as its value.
- **getTourLength**: This function takes a tour as an argument and returns the length of the tour.
- **getDistance**: This function takes as input two nodes and returns the euclidean distance between them.
- **Tour**: a tour is just a list of indices (not Nodes) of cities.

As presented in class, you will first implement the **2-neighbourhood** of nodes to perform hill climbing. You will also implement two non-trivial initialisation routines, which should help hill climbing find better solutions. We will then experiment with a variation of 2-neighbourhood of nodes called **3-neighbourhood** and perform hill climbing with different intialisation routines.

Note that although we apply "hill climbing" in this assignment, we only maintain tour lengths, which we aim to **minimise**. It would be unnatural to consider the negative of tour lengths and find a maximum. So we'll stick with minimising tour lengths. Consequently, although we call our method hill climbing, we'll be performing valley-descending.

Task 1: Generate 2-opt neighbours (2 marks)

Two tours are called 2-neighbours if one can be obtained from the other by deleting two non adjacent edges and adding two edges. For example, the two tours shown below are 2-neighbours of each other.



For a given tour, generate a list of all its 2-neighbours (also sometimes called 2-opt neighbours). You will be implementing the function `generate2optNeighbours(tour)`. Here tour is a list of indices of cities/nodes. When you fill out the function, it should return a list of tours. For example, `generate2optNeigbours` called on [1, 2, 3, 4] should return unique tours given by [[1, 3, 4, 2], [1, 3, 2, 4]]. The order of the tours does not matter. In principle it is a good idea not to have rotated/mirrored copies of tours in the list returned, as they add unnecessary complexity to the algorithm. However, your code will still work if

the list returned by `generate2optNeighbours(tour)` has such copies. For example, the above call might as well have returned ([[4,1,3,2], [3,2,1,4], [2,1,3,4],[1,3,2,4]], where [4,1,3,2] is just the rotated version of [1,3,2,4]).

Hint: there is a simple trick to generate 2-neighbours quickly. See if reversing some substring of the tour list gives you something useful.

Use `python3 hillclimb.py -h` to get familiar with different command line options that you can use. After making changes to `generate2optNeighbours(tour)` in `hillclimb.py`, use the following command to test the function.

- `python3 hillclimb.py --file data/tsp4.tsp --task 1`

This call should return [[4, 3, 1, 2], [4, 1, 2, 3]] or some variation involving rotations and mirror images. Also run tests with other instances, if you would like to make sure your code is correct. Also try to ensure that you **DO NOT** have mirrored or rotated version of the same sequence as they can affect the performance of your search by adding unnecessary complexity. Note that the number of such unique 2 neighbours is given by ${}^{n}C_{2}$-n. You can verify your implementation by checking whether the number of neighbours generated satisfy this formula.

Generating your own TSP instance

You can generate your own instance and use it for printing the 2-neighbours, using this command.

- `python3 hillclimb.py --cities 'number of cities' --r1seed 'random seed for generating city distances' --r2seed 'random seed for tour generation' --task 1`

Note that the '--r1seed' random seed will be used to generate random coordinates for the given number of cities, and the '--r2seed' random seed will be used for creating a random initial tour. For example,

```
python3 hillclimb.py --cities 10 --r1seed 2 --r2seed 3 --task 1
```

will generate the 'tsp10' file for 10 cities with random coordinates in your current directory and will use it to generate 2-neighbours. If you are using an existing file, then there is no need to pass the '--cities' and '--r1seed' options. Just use this command.

- `python3 hillclimb.py --file 'filename' --r2seed 'seed value' --task 1`

These command line options for generating new TSP instances or using existing ones are common to all tasks.

For submission, run this command.

- `python3 hillclimb.py --file data/tsp10.tsp --r2seed 1 --task 1`

This command will generate **`2optNeighbours.txt`** in your directory, which you need to submit for Task 1.

Task 2: Implement hill climbing (1 mark)

You will implementing the function `hillclimbFull(initial_tour)`, which must return two things:

- a list of tour lengths, one for each tour visited by hill climbing, until a local minimum is reached, and
- the final tour found (that is, the local minimum).

For example, `hillclimbFull([1, 2, 3, 4, 5])` on some instance might return something like [12.2, 11, 7.7, 5.6, 5.5, 3.8], [2, 3, 1, 5, 4].

As an intermediate step, you might find it useful to implement a **`hillclimbUtil(Tour)`** function that undertakes a single step of hill climbing: that is, takes as input a tour, generates all its 2-neighbours, and chooses the minimum length one. You will use the function that you have implemented in Task 1 to generate 2-neighbours. (In other tasks, too, feel free to modularise your code by creating your own functions.)

By running

- `python3 hillclimb.py --file data/berlin52.tsp --r2seed 1 --task 2`

you can generate **task2.png**, which will be a graph of tour lengths against the number of hill climbing iterations. You can compare this with the **berlin_task2.png** already present in your directory to verify the correctness of your graph.

For submission, run

- `python3 hillclimb.py --submit --task 2`

This generates **task2_submit.png** which will be used for evaluation. This experiment runs your hill climbing code on **st70.tsp**, and uses 5 different *random* tour initialisations. The graph is plotted for each of the initialisations.

The choice of initial tour before applying hill climbing can make a significant difference in the final tour found. In the next two tasks, we will investigate different approaches to pick a 'good' initial tour.

Task 3: Nearest Neighbour (1 mark)

Under this approach, given a starting node A, a tour is constructed in the following manner. You find the node closest to A, say B. Then you find the node closest to node B, say which is C. This process is continued until all the nodes are covered.

The last edge of the tour is from the last node found to the starting node.

`In this task you will be implementing the function` **`nearestNeighbourTour(start_city)`** and returns the tour found out by applying this method. This function will be called within **`hillclimbWithNearestNeighbour`**, which will generate the graph just as it was done in Task 2.

Use command line option '--start_city' to give the start city. The command to test Task 3 is this.

- python3 hillclimb.py --file data/berlin52.tsp --start_city 1 --task 3

This should generate **task3.png**. You can compare this graph with **berlin_task3.png** present in your directory to verify correctness. To prepare a graph for submission, run this command.

- `python3 hillclimb.py --submit --task 3`

After executing this command you should find **task3_submit.png** in your directory, which shows the graph of 'tour length vs. hill climbng'. This experiment is based on the **st70.tsp** instance, and uses 5 different starting cities for the nearest neighbour algorithm. The difference from Task 2 is that the initial tour is chosen by the nearest neighbour construction, rather than at random. The graph is plotted for each of this initialisations.

Task 4: Euclidean Approximation (2 marks)

There's a well-known approximation algorithm to quickly find a route for Euclidean TSP that is guaranteed to be no more than twice as long as the optimal tour. This tour can be taken as an initial tour for hillclimb, and further improved. Here's the process to generate the initial tour.

1. Generate the complete graph for the nodes.
2. Construct a minimum spanning tree for the graph using Prim's or Kruskal's algorithm.
3. Do a preorder traversal starting from the initial node.

Part 2 is already done for you; you are only supposed to solve parts 1 and 3. For Part 1, you will be implementing a part of the **`euclideanTour`** function. An empty list called **edgeList** is initialized for you at the start of **euclideanTour**. You have to add all the edges of the complete instance graph to edgeList. Each element in **edgeList** (that is, an **edge**) consists of a list **[x,y,dist(x,y)]**, where **x** and **y** are indices of cities, and **dist(x,y)** is the euclidean distance between them. For instance, a partial **edgeList** might look like [[1, 2, 2.4], [1, 3, 4.3]]. For constructing the **edgeList**, you can use given global variables, such as are **`nodeDict`** and cities. You can also use the **`getDistance`** to get the distance between two cities.

Using your **edgeList** set, **Kruskal's algorithm** is applied to find a Minimum Spanning Tree. You can treat this portion as a blackbox for the purpose of this assignment. The Minimum Spanning Tree is stored as a list of `edges` in a list **mst**. An `edge`

here is just [x,y] where x and y are two indices of cities. The list **`mst`** is to be used to generate a legal tour.

For Part 3, you will implement the **`finalOrder`** function. This function takes `mst` and `start_city` as input. The `start_city` is the parameter passed to the `euclideanTour` function. **`finalOrder`** should return the preorder traversal (a list of city indices) starting from **`start_city`**. This preorder traversal is your final Euclidean tour.

Your function hillclimbing function will be called in **`hillclimbWithEuclideanMST`** where the initial tour is the one obtained from euclideanTour. Use command line option '--start_city' to give the start city. The command to test Task 4 is as follows.

- python3 hillclimb.py --file data/berlin52.tsp --start_city 1 --task 4

This should generate task4.png. You can compare this with the berlin_task4.png already present in your directory to verify your graph is correct.

To prepare your graph, run this command.

- `python3 hillclimb.py --submit --task 4`

After executing this command you should find task4_submit.png file which shows the graph of 'tour length vs. hill climbing iterations'. This graph runs your hill climbing code on st70.tsp. The difference from Task 3 is that initial tour is chosen by Euclidean approximation instead of random and is run on only one city.

Comparing the different intialisation methods

Compare the graphs **task2_submit.png** , **task3_submit.png** and **task4_submit.png** and write down your observations in **observations.txt**. Comment on which initialisation method is better and why that is the case.

Task 5: Generate 3-opt neighbours (2 marks)

In task 2 you have implemented 2-opt neighbours. There is yet another method by which we can generate neighbours. Where instead of replacing two non-adjacent edges we will be replacing 3 non-adjacent edges to form new neighbours. For a given cyclic graph the number of non-adjacent edges is given by $^{n}C_{3}-n(n-3)$. For each triplet of such edges, 4 different permutations exist, which result in 4 different neighbours.

Thus for each tour, $(^{n}C_{3}-n(n-3))*4$ neighbours will exist.

For example if you have a given tour as [1, 3, 4, 6, 5, 2]. Then the 3-opt neighbours of this tour are as follows -

[[1, 4, 3, 5, 6, 2], [1, 6, 5, 3, 4, 2], [1, 6, 5, 4, 3, 2], [1, 5, 6, 3, 4, 2], [1, 3, 6, 4, 2, 5], [1, 3, 5, 2, 4, 6], [1, 3, 5, 2, 6, 4], [1, 3, 2, 5, 4, 6]].

For this task you have to implement this generate3optNeighbours function in the hillclimb.py file. **Remember** you should avoid returning duplicate neighbours in the neighbours list (mirrored or rotated version of the same sequence) as they add unnecessary complexity to the hill climbing search algorithm.

The command to test Task 5 is as follows.

- `python3 hillclimb.py --cities 'numcities' --r1seed 2 --r2seed 3 --task 5`

Try it for different value of numcities and ensure that number of neighbours that you are generating matches the formula given above

For submission, run this command.

- `python3 hillclimb.py --file data/tsp10.tsp --r2seed 1 --task 5`

This command will generate **`3optNeighbours.txt`** in your directory, which you need to submit for Task 5.

Tasks 6, 7, and 8: Hill Climbing with 3optNeighbours + 2optNeighbours (3 marks)

Similar to tasks 2, 3 and 4 of this assignment, in these tasks we will compare the performance of the new neighbour generating method (3optNeighbours + 2optNeighbours) by generating the following plots.

Task 6

Use the following comand

- `python3 hillclimb.py --file data/berlin52.tsp --r2seed 1 --task 6`

to generate **task6.png**. In this task we perform hill Climbing using random initialisation. You can compare this with the **berlin_task6.png** already present in your directory to verify the correctness of your graph.

To prepare a graph for submission, run this command.

- `python3 hillclimb.py --submit --task 6`

This generates **task6_submit.png** which will be used for evaluation. This experiment runs your hill climbing code on st70.tsp, and uses 5 different random tour initialisations. The graph is plotted for each of the initialisations. Note that generation of this graph may take some time **(~2 hours)** so **DO NOT** wait until the deadline to generate this graph.

Visually inspect the two graphs **task2_submit.png** and **task6_submit.png** and write down your observations in **observations.txt**. Comment on which neighbour-generating method is better (if at all) and why?

Task 7

Use the following comand

- python3 hillclimb.py --file data/berlin52.tsp --start_city 1 --task 7

This will generate **task7.png**. You can compare this with the **berlin_task7.png** already present in your directory to verify your graph.

To prepare a graph for submission, run this command.

- `python3 hillclimb.py --submit --task 7`

After executing this command you should find **task7_submit.png** in your directory, which shows the graph of 'tour length vs. hill climbng'. This experiment is based on the st70.tsp instance, and uses 5 different starting cities for the nearest neighbour algorithm.

Compare the two graphs **task3_submit.png** and **task7_submit.png** and write down your observations in **observations.txt**. Comment your opinion on which neighbour generating method is better in this case and why?

Task 8

Use the following comand

- python3 hillclimb.py --file data/berlin52.tsp --start_city 1 --task 8

This generates **task8.png**. You can compare this graph with the **berlin_task8.png** already present in your directory to verify your graph.

To prepare a graph for submission, run this command.

- `python3 hillclimb.py --submit --task 8`

After executing this command you should find **task8_submit.png** file which shows the graph of 'tour length vs. hill climbing iterations'. This graph runs your hill climbing code on st70.tsp.

Now Compare the graphs **task4_submit.png** and **task8_submit.png** and write down your observations in **observations.txt**. Comment your opinion on which neighbour generating method is better in this case and why?

Submission

You are expected to work on this assignment by yourself. You may not consult with your classmates or anybody else about their solutions. You are also not to look at solutions to this assignment or related ones on the Internet. You are allowed to use resources on the Internet for programming (say to understand a particular command or a data structure), and also to understand concepts (so a Wikipedia page or someone's lecture notes or a textbook can certainly be consulted). However, you **must** list every resource you have consulted or used in a file named `references.txt`, explaining exactly how the resource was used. Failure to list all your sources will be considered an academic violation.

Place all files in which you have written code in or modified in a directory named `la8-rollno`, where `rollno` is your roll number (say 12345678). Tar and Gzip the directory to produce a single compressed file (say `la8-12345678.tar.gz`). It must contain the following files. (**NOTE** that your **observations.txt** must contain 4 different comparisions as mentioned in this assignment.)

- `hillclimb.py`
- 2OptNeighbours.txt
- 3OptNeighbours.txt
- task2_submit.png
- task3_submit.png
- task4_submit.png
- task6_submit.png
- task7_submit.png
- task8_submit.png
- observations.txt
- references.txt

Submit this compressed file on Moodle, under Lab Assignment 8.

Your submission will be evaluated based on these files, but we may also examine and run your code to verify correctness.