CSE6140/CX4140 Fall 2021 TSP Project

Assigned: October 23th, 2021

1 Overview

The Traveling Salesman Problem (TSP) arises in numerous applications such as vehicle routing, circuit board drilling, VLSI design, robot control, X-ray crystallography, machine scheduling and computational biology. In this project, you will attempt to solve the TSP using different algorithms, evaluating their theoretical and experimental complexities on both real and random datasets.

2 Objective

- Get hands-on experience solving an intractable problem that is of practical importance
- Implement an exact branch-and-bound algorithm
- Implement approximate algorithms that run in a reasonable time and provide high-quality solutions with concrete guarantees
- Implement heuristic algorithms (without approximation guarantees)
- Develop your ability to conduct empirical analysis of algorithm performance on datasets, and to understand the trade-offs between accuracy, speed, etc., across different algorithms
- Develop teamwork skills while working with other students

3 Groups

You will be in a group of up to 4 students. We encourage you to form your own groups, and use Piazza as a resource. Afterwards, please designate **one** member per group to complete the assignment "Group and programming language submission for project" (due 11/03 Wednesday 11:59pm) through Canvas:

In the text entry, put each student's last name and first name per line, followed by the programming language (Python or C++) your group will use. For example:

```
Zhang, Ziqi
Liu, Zefang
Xiao, Yiqiong
Pan, Xinhai
Python
```

4 Background

We define the TSP problem as follows: given the x-y coordinates of N points in the plane (i.e., vertices), and a cost function $c(u, v)$ defined for every pair of points (i.e., edge), find the shortest simple cycle that visits all N points. This version of the TSP problem is *metric*, i.e. all edge costs are symmetric and satisfy the triangle inequality.

5 Algorithms

You will implement 4 algorithms total, that fall into three category counts: 1 Exact, 1 Heuristics (with approximation guarantee), 2 Local Search:

1. exact, computing an optimal solution to the problem
2. construction heuristics, some of which have approximation guarantees
3. local search with no guarantees but usually much closer to optimal than construction heuristics.

In what follows, we present the high-level idea behind the algorithms you will implement.

- **Exact algorithm using Branch-and-Bound**. Implement the Branch-and-Bound algorithm as seen in class. We present several approaches to compute the lower bound function (please use either the 2 shortest edges, or the MST bounding functions, or something stronger you find in the literature) in class. Feel free to read up on what researchers have proposed for this problem. You may design any lower bound of your choice as long as it is indeed a lower bound.

  Since this algorithm is still of worst-case exponential complexity, your implementation must have additional code, similar to all of your other implementations, that allows it to stop after running after some amount of time and to return the current best solution that has been found so far. Clearly, for small datasets, this algorithm will most likely return an optimal solution, whereas it will fail to do so for larger datasets.

- **Construction Heuristics with approximation guarantees**. Please choose and implement one construction heuristic.

  MST-APPROX 2-approximation algorithm based on MST detailed in lecture

  FARTHEST-INSERTION insert vertex whose minimum distance to a vertex on the cycle is maximum

  RANDOM-INSERTION randomly select a vertex and insert vertex at position that gives minimum increase of tour length

  CLOSEST-INSERTION insert vertex closest to a vertex in the tour

  NEAREST NEIGHBOR find the closest vertex not yet visited and add it in the tour

  The description and approximation bounds of these algorithms can be found in literature [1, 2].

- **Local Search**. There are many variants of local search and you are free to select which one you want to implement. Please implement 2 types/variants of local search. They can be in different families of LS such as SA vs Genetic Algorithms vs Hill Climbing, or they can be in the same general family but should differ by the neighborhood they are using, or by the perturbation strategy, etc. They need to be different enough to observe qualitative differences in behavior. Here are some pointers:

- Neighborhood - 2-opt exchange
- Neighborhood - 3-opt exchange or more complex one
- Perturbation using 4 exchange
- Simulated Annealing
- Iterated Local Search
- First-improvement vs Best-Improvement

6 Data

You will run the algorithms you implement on some real-world datasets.

The datasets can be downloaded from Canvas as `DATA.zip`.

In all datasets, the N points represent specific locations in some city (e.g., Atlanta).

The first several lines include information about the dataset, including the data type (Euclidean).

For instance, the `Atlanta.tsp` file looks like this:

```
NAME: Atlanta
COMMENT: 20 locations in Atlanta
DIMENSION: 20
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 33665568.000000 -84411070.000000
2 33764940.000000 -84371819.000000
3 33770889.000000 -84358622.000000
...
```

Note that the node ID is a unique integer assigned to each vertex, the x and y coordinates may be real numbers, and the three values are separated by spaces.

To get distance between any two points, compute the Euclidean distance (L_2 norm), then round to the nearest integer. A double with 0.5 decimal value is rounded up.

6.1 Solutions

Optimal values for every instance are provided in the **solutions.csv** file, from the data zip. You will need these for evaluation metrics in the report. These should also serve as a sanity check for your implementations.

7 Code

The requirements of the code are summarized as below:

1. All your code files should include a top comment that explains what the given file does. Your algorithms should be well-commented and self-explanatory. Use a README file to explain the overall structure of your project.

2. You must have an executable we can call named **tsp_main/tsp_main.py**, that is:

    - **If you choose python, your code must conform with the following arguments / execution:**

    ```
    tsp_main.py -inst <filename>
            -alg [BnB | Approx | LS1 | LS2]
            -time <cutoff_in_seconds>
            [-seed <random_seed>]
    ```

    - **If you choose C++, you must provide the `Makefile` for compilation, and the generated executable must conform with the following arguments / execution:**

    ```
    tsp_main -inst <filename>
            -alg [BnB | Approx | LS1 | LS2]
            -time <cutoff_in_seconds>
            [-seed <random_seed>]
    ```

    And the arguments in the commands above are described as below:

    - `-alg`: the method to use, `BnB` is branch and bound, `Approx` is approximation algorithm, `LS1` is the first local search algorithm, and `LS2` is the second local search algorithm.
    - `-inst`: the filepath of a single input instance. Do not assume anything about what the path is, or hardcode this in any way.
    - `-time`: the cut-off time in seconds, your program must be able to produce the best solution and trace found within the specified time.
    - `-seed`: the random seed, this must be used to seed any random generator used for solving TSP.

      The arguments cannot be changed and if your submitted code does not compile to or result in an application that executes with these arguments, there may be a significant penalty.

3. If it is run with the same 4 input parameters, your code should produce the same output.

4. Any run of your executable with the three or four inputs (filename, cut-off time, method, and if applicable based on method, seed) must produce **two types of output files**:

    (a) Solution files:

    - File name: `<instance>_<method>_<cutoff>[_<random_seed>].sol`, e.g. `Atlanta_BnB_600.sol` or `Cincinnati_LS1_600_4.sol`.
      Note that as in the first example above, `random_seed` is only applicable when the method of choice is randomized (e.g., local search). When the method is deterministic (e.g., branch-and-bound), `random_seed` is omitted from the solution file's name.
    - File format:
        i. line 1: quality of best solution found
        ii. line 2: list of vertex IDs of the TSP tour (comma-separated and without spaces), **0-indexed**. Note this indexing is different from that of the input, which is 1-indexed.

Here an example solution file for Cincinnati:

```
277952
0,2,9,6,5,3,7,8,4,1
```

(b) Solution trace files:

- File name: `<instance>_<method>_<cutoff>[_<random_seed>].trace`, e.g. `Atlanta_BnB_600.trace` or `Cincinnati_LS1_600_4.trace`. Note that `random_seed` is used as in the solution files.
- File format: each line has two values (comma-separated):
  1. A timestamp in seconds (to two decimal places)
  2. Quality of the best found solution at that point in time (integer). Note that to produce these lines, you should record every time a new improved solution is found.
     Example:

     ```
     3.45, 102
     7.94, 95
     ```

5. If you have any questions, please contact the TAs well in advance of the submission deadline.

7.1 Language and Dependencies

Similar to past programming HW, you may use either Python 3 or C++. For Python, you may use any libraries included in a standard Python 3 distribution of Anaconda (`https://www.anaconda.com/distribution/`). For C++, you may use any of its standard library functions.

If you're not sure about any dependencies used in your code, mention these in the README. Better yet, ask ahead of time.

8 Report

The report will be a significant portion of your grade, and we strongly encourage you to put effort into producing a quality one.

8.1 Formatting

You will use the format of the Association for Computing Machinery (ACM) Proceedings to write your report. Please use the templates from `https://www.acm.org/publications/proceedings-template`.

8.2 Content

Your report should be written as if it were a research paper in submission to a conference or journal. A sample report outline looks like this:

- Introduction: a short summary of the problem, the approach and the results you have obtained.
- Problem definition: a formal definition of the problem.
- Related work: a short survey of existing work on the same problem, and important results in theory and practice.

- Algorithms: a detailed description of each algorithm you have implemented, with pseudo-code, approximation guarantee (if any), time and space complexities, data structure used in implementation, etc. What are the potential strengths and weaknesses of each type of approach? Did you use any kind of automated tuning or configuration for your local search? Why and how you chose your local search approaches and their components? Please cite any sources of information that you used to inform your algorithm design.
- Empirical evaluation: a detailed description of your platform (CPU, RAM, language, compiler, etc.), experimental procedure, evaluation criteria and obtained results (plots, tables, etc.). What is the lower bound on the optimal solution quality that you can obtain from the results of your approximation algorithm and how far is it from the true optimum? How about from your branch-and-bound? Please also see Section Evaluation for tables and plots required to present the performance of your algorithms.
- Discussion: a comparative analysis of how different algorithms perform with respect to your evaluation criteria, or expected time complexity, etc.
- Conclusion

8.3 Evaluation

We now describe how you will use the outputs produced by your code in order to evaluate the performance of the algorithms.

1. Comprehensive Table: include a table with columns for each of your TSP algorithms as seen below. For all algorithms and all instances, report the time, your algorithms solution quality, and relative error with respect to the optimum solution quality (OPT) provided to you. Relative error ($RelErr$) is computed as $(Alg - OPT)/OPT$. Round time and $RelErr$ to two and four significant digits beyond the decimal, respectively. For local search algorithms, your results for each cell should be the average of some number (at least 10, include this number in your report) of runs with different random seeds of your choosing for that dataset. You will fill in average time (seconds) and average solution quality.

   For local search, you are free to choose the runtime cut-off for each algorithm, as long as it's large enough to produce interesting plots for local search. For branch-and-bound, please use a cutoff of 600 seconds.

   You can use multiple tables (eg. one table for one algorithm) instead of one table if needed for formatting.

	Branch and Bound			Etc. (other algorithms)		
Dataset	Time (s)	Sol.Qual.	RelErr	Time (s)	Sol.Qual.	RelErr
instance	3.26	3400	0.0021	...	....	...

2. Evaluation plots: the next three evaluation plots apply to local search algorithms only. These plots will be introduced during the lecture on 11/06. Choose two problem instances, with at least one of them should having more than 50 vertices. For each instance and each local search algorithm, present the following three plots:

(a) Qualified Runtime for various solution qualities (QRTDs): The x-axis is the run-time in seconds, and the y-axis is the fraction of your algorithm runs that have 'solved' the problem. Note that 'solve' here is w.r.t. to some relative solution quality q^*. For instance, for $q^* = 0.8\%$, a point on this plot with x value 5 seconds and y value 0.6 means that in 60% of your runs of this algorithm, you were able to obtain a solution quality at most the optimal size plus 0.8% of that. When you vary q^* for a few values, you obtain the points similar to those presented in class.

(b) Solution Quality Distributions for various run-times (SQDs): Instead of fixing the relative solution quality and varying the time, you now fix the time and vary the solution quality. The details are analogous to those of QRTDs.

(c) Box plots for running times: Since your local search algorithms are randomized, there will be some variation in their running times. You will use box plots, as described in the 'Theory' section of this blog post: `http://informationandvisualization.de/blog/box-plot`. Read the blog post carefully and understand the purpose of this type of plots. You can use a plot generator of your choice. Examples include online box plot generators such as `http://shiny.chemgrid.org/boxplotr/` or Python libraries like Matplotlib.

9 Deliverables

Failure to abide by the file naming and folder structure as detailed here may result in penalties.

One student from your group should submit the following:

1. A PDF file of the report following the guidelines in section 8.
2. A zip file of the following files/folders:
   - A folder named 'code' that contains all your code, the executable and a README file, as explained in section 7.
   - A folder named 'output' that contains all output files, as explained in section 7.

9.1 Team Evaluation

Each student will also individually submit an evaluation of their team members. This will be released as a later assignment.

10 Competition (Bonus)

If your group wishes, it may enter a competition for a small number of bonus points. Submissions that fail to follow the format instructions in Sec. 7 will be disregarded without exception. More details will be released on the competition later.

References

[1] B Golden, L Bodin, T Doyle, and W Stewart. Approximate traveling salesman algorithms. *Oper. Res.*, 28(3):694–711, 1980.

[2] Daniel J Rosenkrantz, Richard E Stearns, Lewis, II, and Philip M. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, September 1977.