CSE6140 Fall2016 Project - TSP

November 3, 2016

1 Overview

In Time Magazine's March 1981 issue, Dr. James Watson (the co-discoverer of the double-helix structure of DNA), is quoted to have said: "Let's put it this way: I wouldn't buy gene-splicing stock for my grandmother." Sorry, *James*, but your grandmother may well have a reason to be *cheesed off*.

A few (fictional) years after that (true) statement was made, Pokémon specialists (or *Pokéists*) went on an exhibition to a Guyana jungle and discovered a new Pokémon which they called Mew. While they also claimed that Mew gave birth to the legendary Mewtwo, our *Pokédex* (i.e. Pokémon dictionary) states that Mewtwo was "created by a scientist after years of horrific gene splicing and DNA engineering experiments" (talk about your *Pokénstine's Monster*). After years of torture, Mewtwo was able to escape to Cerulean Cave, before running off with hundreds of other Pokémons to non other than the United States of America.

You wake up early one morning to hear that Team Rocket, part of the team of evil scientists who experimented on Mewtwo, are out to recapture it and its friends. Realizing that Mewtwo can be very dangerous in the hands of evil, you set out to *catch'em all* as quickly as possible, before Team Rocket even has a chance.

Keeping in mind that Mom wants you back by lunchtime, you decide that you need to find the shortest tour such that you visit all Pokémon locations and end up back where you started, as quickly as possible. Hearing what you're about to do, your Pokémon-hating yet Algorithm-enthusiast sister Amrita chooses to help you out by giving you a crash course on TSPs. The rest of this document is narrated in Amrita's voice.

The Traveling Saleseman 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 real and random datasets (with, of course, real-world testing through your Poké-quest!).

2 Objectives

- Get hands-on experience in solving an intractable problem that is of practical importance
- Implement an exact branch-and-bound algorithm
- Implement approximate algorithms that run in reasonable time and provide high-quality solutions with 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

Your groups have been posted to T-Square under "Resources → Project → Project Groups.pdf".
If there are concerns with programming language compatibility, please email us with your team name and the concern. If you find somebody to swap with, then email us cc-ing both students that are swapping and explaining the reasons for the swap.

4 Background

We define the TSP problem as follows: given the x-y coordinates of N points on a 2D 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.

The cost function $c(u, v)$ is defined as the Euclidean distance between points u and v. Please see the TSPLIB documentation (which is included with the project, and `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf`) for details. The only difference between the definition

This version of the TSP problem is **metric**: all edge costs are symmetric and satisfy the triangle inequality. For more details about types of TSP, please refer to the lectures slides.

5 Algorithms

You will implement algorithms that fall into three categories:

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

1. **Exact algorithm using Branch-and-Bound**. Implement the Branch-and-Bound algorithm as seen in class. The slides of Oct 25th 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). 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 algorithm must have additional code that allows it to stop after running for T minutes, 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.

2. **Construction Heuristics with approximation guarantees**. Please implement the MST-APPROX seen in lecture and choose one more Construction heuristic (for a total of 2).

   - MST-APPROX: Implement the 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
   - SAVINGS HEURISTIC

3. **Local Search**. There are many variants of local search (LS) 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 Simulated Annealing 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 presented in slide 26 of September 30th lecture
    - Neighborhood - 3-opt exchange or more complex one
    - Perturbation using 4 exchange discussed in class
    - Simulated Annealing
    - Iterated Local Search
    - First-improvement vs Best-Improvement
    - Tabu Search

6 Data

You will run the algorithms you implement on real world datasets consisting of actual Pokestops in the Pokemon Go game. We have used data from several different cities and varied the instance sizes to vary the difficulty.

The datasets can be downloaded from T-Square: Resources → Project → Data.zip

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

The first five lines include some information about the dataset. For example, 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
...
...
...
EOF
```

The remaining line are space delimited and contain the node ID, x value, and y value of each node in the dataset. The x and y values are given as latitude and longitude points that have been multiplied by $1e6$. The file will end with a single line that only contains the string "EOF". As the EDGE_WEIGHT_TYPE is always EUC_2D you should use Euclidean distance, as described in the TSPLIB documentation, as the cost function between points. It is important to read the TSPLIB documentation because the distance values you use should be rounded in the way that is given in that document.

If you want to get the actual latitude and longitude values back (e.g. to map in GIS software) you should divide the given x and y values by $1e6$. Note: all cost calculations should be done with the given x and y points, NOT the latitude and longitude versions.

7 Code

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 code, as well as how to run your executable.

Your executable should take as input i) the filename of a dataset and ii) the cut-off time (in seconds) iii) the method to use, iv) a random seed. If it is run with the same 4 input parameters, your code should produce the same output. The executable must conform with the following format:

$exec -inst < filename > -alg\ [BnB|MSTApprox|Heur|LS1|LS2]$
$-time < cutoff\_in\_secods > -seed < random\_seed >$

Any run of your executable with any four inputs (filename, cut-off time, method and seed) must produce two types of output files:

1. Solution files:
   - File name: $< instance > \_ < method > \_ < cutoff > \_ < randSeed > *.sol$, e.g. $Atlanta\_BnB\_600.sol$, $Atlanta\_LS1\_600\_4.sol$ Note that as in the example above, randSeed is only applicable when the method of choice is randomized (e.g. local search). When the method is deterministic (e.g. branch-and-bound), randSeed is omitted from the solution file's name.
   - File format:
     (a) line 1: quality of best solution found (integer)
     (b) lines 2 through $n + 2$: a line for each edge in the tour, formatted as: $u\ v\ c(u, v)$

     Note: You need to include the edge from the last node in the tour back to your initial node. Each node in the graph will be present twice in this edgelist.
2. Solution trace files:
   - File name: $< instance > \_ < method > \_ < cutoff > \_ < randSeed > *.trace$, e.g. $Atlanta\_BnB\_600.trace$, $Atlanta\_LS1\_600\_4.trace$. Note that randSeed is used as in the solution files.
   - File format: each line has two values (comma-separated):
     (a) A timestamp in seconds (double)
     (b) 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

8 Output

You should run all the algorithms you have implemented on all the instances we provide, and submit the output files generated by your executable, as explained in the Code section.

P.S.: DO NOT SUBMIT THE INPUT DATA FILES.

9 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 MVC algorithms as seen below. For all algorithms report the relative error with respect to the optimum solution quality provided (soon) to you in the MVC instance files. Relative error ($RelErr$) is computed as $(Alg - OPT)/OPT$. Round

time and $RelErr$ to two significant digits beyond the decimal. For local search algorithms, your results for each cell should be the average of some number (at least 10) of runs with different random seeds for that dataset. You will fill in average time (seconds) and average vertex cover size. You can also report in any other information you feel is interesting.

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

The next three evaluation plots are applicable to local search algorithms only, and need only be presented for the instances XX and YY. All the information you need to produce these plots is in your solution trace files.

1. Qualified Runtime for various solution qualities (QRTDs): A plot similar to those in Lecture on Empirical Evalutation. 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 VC of size 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 in slide 24.

2. 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.

3. 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 online box plot generators such as `http://boxplot.tyerslab.com/` to produce the plot automatically from your data.

10 Report

1. Formatting

   You will use the format of the Association for Computing Machinery (ACM) Proceedings to write your report.

   For users of:

   - Word: download the template from `http://www.acm.org/sigs/publications/pubform.doc`.
   - LaTeX: download the 'Option 2: LaTex2e - Tighter Alternate style' Style File V2.5 (.CLS file) and template Sample File V2.0 (.TEX file) from `http://www.acm.org/sigs/publications/proceedings-templates#aL2`.

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, 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 drive from the results of your approximation algorithm and how far is it from the true optimum? How about from your branch-and-bound?
- Discussion: a comparative analysis of how different algorithms perform with respect to your evaluation criteria, or expected time complexity, etc.
- Conclusion

11 Deliverables

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

1) Each group must submit a Report: a PDF file of the report, titled

$$group\_ < group\_id > - < gtusername > -Report.pdf$$

following the guidelines in section Report.

2) Each student should submit the same ZIP or RAR file, titled:

$$group\_ < group\_id > - < gtusername > .zip$$

Example: $group\_F - smith3.zip$

The file must have the following files/folders:

2.1 Code: a folder named 'code' that contains all your code, the executable and a $README - group\_ < group\_id > .txt$ file, as explained in section Code.

2.2 Output: a folder named 'output' that contains all output files, as explained in section Code.

3) Each student should submit an evaluation of the team. For each team member (including self) include a score from 0 to 10, outline the contributions that the member did to the project, and justification for the score.