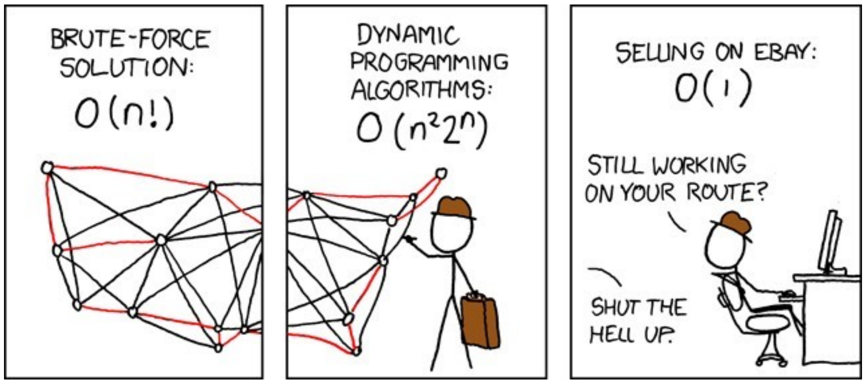
CS325 TSP Project



from: http://xkcd.com/399

This project is rather open-ended, and I hope you will have fun trying out ideas to solve a very hard problem, very classical problem called the Traveling Salesman problem (TSP).

You are given a set of n cities and, for each pair of cities c_1 and c_2, the distance between them $d(c_1, c_2)$. Your goal is to find an ordering (called a tour) of the cities so that the distance you travel is minimized. The distance you travel is the sum of the distance from the first city in your ordering to the second, plus the distance from the second city in your ordering to the third, and so on until you reach the last city, and then adding the distance from the last city to the first city. For example, say the cities are Chicago, New York, New Orleans and San Francisco. The total distance traveled visiting the cities in this order is:

$$d_{tour} = d_{Chicago,\ New\,York} + d_{New\,York,\ New\,Orleans} + d_{New\,Orleans,\ San\,Francisco} + d_{San\,Francisco,\ Chicago}$$

In this project, you will only need to consider the special case where the cities are locations in a 2D grid (given by their x and y coordinates) and the distance between two cities $c_1 = (x_1, y_1)$ and $c_2 = (x_2, y_2)$ is given by their Euclidean distance. So that we need not worry about floating point precision problems in computing the square-root (and so that everyone will get the same answer), we will always round this distance to the nearest integer. In other words, you will compute the distance between cities c_1 and c_2 as:

$$d(c_1, c_2) = [\ (\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}\ ]$$

Where [c] is the function that rounds the value c to the nearest integer.

For example, if three cities are given by the coordinates (0, 0), (1, 3), (3, 1), then a tour that visits these cities in this order has distance:

$$\begin{aligned} d_{tour} &= \left[\sqrt{1^2 + 3^2}\right] + \left[\sqrt{2^2 + (-2)^2}\right] + \left[\sqrt{(-3)^2 + (-1)^2}\right] \\ &= \left[\sqrt{10}\right] + \left[\sqrt{8}\right] + \left[\sqrt{10}\right] \\ &= [3.16227766\ldots] + [2.82842714\ldots] + [3.16227766\ldots] \\ &= 9 \end{aligned}$$

Project specification

Your group is to design and implement a method for finding the best tour you can. TSP is not a problem for which you will be able to easily find optimal solutions. It is that difficult. But your goal is to find the best solution you can in a certain time frame. You may want to start with some local search heuristics as described in DPV Section 9.3.1. (This chapter is attached to the program description in Blackboard)

Beyond that you may:

- Read anything about how to solve the traveling salesperson problem
  - Caveat: You must cite any resources you use.
- Use which ever programming language you want
  - Caveat 1: All group members must be comfortable with it.
  - Caveat 2: It must run on our server: *flip2.engr.oregonstate.edu*

You may not use:

- Existing implementations or subroutines for your submitted project
- Extensive libraries (if you are unsure, ask in the discussion forum for this project)
- Other people's code (other than that of your group members)

Your program must:

1. Accept problem instances and options on the command line so it can be run automatically with scripts.
2. Explicitly have a usage statement if any options are required (or recommended)
3. Name the output file as the input file's name with .tour appended.
   (e.g. for input **tsp_example_1.txt** output will be **tsp_example_1.txt.tour**)
4. Compile/Execute ***correctly*** and ***without debugging on my part*** on *flip2.engr.oregonstate.edu*
   If your program does not compile/run correctly, according to the program requirements and documentation **you provide**, you will receive no points.
   Yes, this means if you provide no execution documentation, you get no points.

Input specification:

1. A problem instance (a particular list for the input) will always be given to you as a text file.
   (Note: this means a text encoded file, not necessarily a file ending with .txt)

2. Each line defines a city and each line has three numbers separated by white space,
   (Note: be careful with this requirement, as I may format my input files to be easy for a human to read by using columns with white space padding the values)

   a. The first number is the city identifier,
   b. The second number is the city's x-coordinate,
   c. The third number is the city's y-coordinate.

Output specification:

1. You must output your solution into another text file with *n + 1* lines, where *n* is the number of cities.

2. The first line is the total length of the tour your program computes (d_{tour}).

3. The next *n* lines should contain the city identifiers in the order they are visited by your tour.
    a. Each city must be listed exactly once in this list.
    b. This is the certificate for your solution and will be checked
    If your certificates are not valid, you will not receive credit for them.

Example instances: We have provided you with three example instances, attached to the program description in Blackboard:

- **tsp_example_[*].txt** Input files (which meet the input specs of the program)
- **tsp_example_[*].txt.tour** Example outputs corresponding to these three input cases.
  You should use the output instances to test that you are computing distances correctly. Our best tour lengths for test cases 1, 2 and 3 are 108159, 2579, and 1573084, respectively. Clearly, these do not match the values in the tour files. You should use these values to judge how good your algorithm is.

Testing:

A testing procedure **tsp-verifier.py** is also attached to the program description in Blackboard. We will this program to verify your solutions. Usage to test a solution to a corresponding input is:
(NOTE: requires **TSPAllVisited.py**):

```
python tsp-verifier.py inputfilename solutionfilename
```

You should test that your outputs are correct.

Graded test instances:

On the night the assignment is due (Friday of Week 10, 11:59pm), we will make available a number of test instances in Blackboard. Within 48 hours of these being posted, you are required to submit an output file for each test instance according to the output specification and corresponding to each of these test instances as input files. These input files will be named as such: **test-input-[*].txt**, where [*] will be replaced with the number of the test instance. Your group's solutions should be submitted to the proper TEACH subdirectory by a single member of the group (and no files should be submitted by other team members) by the same Sunday, 11:59pm.

Aside from these test instances you must finish and submit your project Report and code submission as well (Friday 11:59pm, Week 10). These will not be accepted late.

Project report

You will submit a project report. The project report may only be up to 2 pages in length with formatting:

- No less than 10pt font
- No less than 0.5” margins
- No larger than 8.5” x 11.0” digital paper

In this report you must include

1. The group number and names of all group members.
2. Detailed descriptions of the ideas behind your algorithm.
3. Pseudo code for your algorithm.
4. Possible improvements you’d make, if you were to continue.

Remember to be succinct and have all group members read/revise any drafts to ensure that it both (1) makes sense and (2) is complete.

Competition

We will also hold a “competition”. The competition will require your program to find the best solution possible to one or more instances within a fixed amount of time (e.g. 5 minutes).

Termination procedure: The termination procedure **watch.py** is attached in Blackboard. Two examples of how to write code that will work with this procedure are also given: **simpleprimes.py** and **primes.py**.

Example usages:

```
# tries to sleep for 400 seconds, but will be terminated at the end of
# 300 seconds (5 minutes)
$ ./watch.py /bin/sleep 400

# Finding prime numbers: a simple program that keeps writing the latest result
# to a file primes.txt - somewhat inefficient in that it writes the latest
# results each time a result is computed.
$ ./watch.py ./simpleprime.py

# Finding prime numbers; continues calculations until a sigterm is received
# at which point, the primes.py writes a file last_prime.txt cleaner,
# and more efficient.
$ ./watch.py ./primes.py
```

Competition structure:

There will be $\log_2(classSize) - 1$ required rounds and an approximately equal number of TSP tests. Everyone will compete in the first round on the first test. The best ½ of the projects will progress to the second round, the best ¼ teams to the third round and so on through the last round (where the final round will be between two projects). Ties will be broken by performance in earlier rounds. More than the number of teams given may progress to future rounds at the discretion of the judge. The competition instances will be of size $2^{\text{roundCount}} *$ 100 city points (100, 200, 400, 800, 1600, 3200, ...).

Grading Rubric

- 30% of your project grade will be determined by your solutions to the test instances.
    - You will be judged on how close your tour length is to that of the best possible solution. However, you will not be told what the optimal tour length is for the test instances.
    - Our current proposed formula for this is (<my best tour> / <your best tour>) * 1.2.
        - Yes, this means that you can then earn up to 120% of this section!
    - Remember that doing well on this section (you have about 48 hours to run your code on the tests) does not necessarily mean you will do well in the competition, because of the short duration of the competition tests.

- 50% of your project grade will be determined by your project report.
    - You will be judged on clarity and creativity of your explanation for your algorithm and implementation.

- 20% of your project grade will be determined by your participation in the competition.
    - This will be awarded for full participation in finding a working, but not necessarily very short, solution to each competition test case. There will be a penalty for each test case that your program cannot find a solution for in the allotted test time, and more if it cannot find a solution in a reasonable time, determined by the time we have available to allow programs to run on our servers. Only after they fail at 5 minutes will they be offloaded to other hardware to try a longer time, to keep the environment as similar as possible between tests).
    - The benefit for winning through the rounds will be the right to brag to your fellow classmates that your program was more efficient or more robust than theirs.

Check List

1. _____ Does your program correctly compute tour lengths for simple cases?

2. _____ Does your program read input files and options all from the command line?

3. _____ Does your program meet the output specification?

4. _____ Did you check that you produce solutions that verify correctly?
   (verification is easier after all, right?)

5. _____ Did you find solutions to the test instances by the final due date, which is 48 hours after the test cases are initially revealed?

6. _____ Does your code compile/run without issue, according to your documentation, on *flip2.engr.oregonstate.edu* ?

7. _____ Have you checked the announcements on Blackboard for any updates before submission?

8. _____ Have you (**after** checking the above requirements) submitted your project report, your solutions to the test cases, your source code for your program, and any comments you wish to include to TEACH and Blackboard?