EECS 281 - Fall 2014 Project 4

2014: EECS Drone Delivery v1.1.1

Due Wednesday, December 10, 2014 at 11:55 PM

Overview

Amazon.com in recent times announced their new delivery method, Amazon Prime Air, where Amazon allows 30 min delivery to a customer's doorstep by means of a flying drone. You are joining a company called EECS in the new drone tech space to compete with Amazon and its drones. EECS will be providing clients with their own drone and routing technology for quick delivery on campuses, either corporate or educational, amongst the campus facilities.

There are two types of drones at EECS:

Drone Type I is a drone that moves ten times as fast as the drones from their Amazon counterpart. However, these drones require drone director beacon installations along the path. The cost of these installations are directly proportional to the distance of the desired path.

Drone Type II is a regular drone similar to their Amazon counterpart. The drone's energy usage is directly proportional to the distance it travels.

There are currently three types of clients (A, B, and C) that are interested in your drones, and your task will vary based on the client type. This is described further in detail below.

To be clear, these scenarios are separate: your program will create a plan for one *or* the other, but not both in the same run (although you may find that the algorithms from one scenario may help with another)**.**

Project Goals

- Understand and implement MST algorithms. Be able to pick the best graph representation for a given problem (Client Type A & B)
- Understand and implement a Branch and Bound algorithm. Develop a fast and effective bounding algorithm. (Client Type C)
- Use of gnuplot for visualization (Optional, but helpful!)

Distance:

We represent the path between two points as a sequence of intermediate points. For simplicity, the distance between each pair of points should be calculated using Euclidean distance. You should represent your distances as doubles. Please be very careful with rounding errors in your distance calculations; the autograder will allow a 0.01% margin of error to account for rounding, but you should avoid rounding at intermediate steps. Calculate the distance with the following formula:

$$d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2},$$

which is a conceptual distance for comparison purpose only, indicating that roads are not actually going along the line segment between two points. The comparison of d values just serves to help with route selection.

Command Line Input

Your program, `DroneRouting`, should take the following case-sensitive command line options:

- `-c, --clientType CLIENT_TYPE`

This command line option must be specified, if it is not, print a useful error message to standard error (`cerr`) and `exit(1)`. `CLIENT_TYPE` is a required argument. `CLIENT_TYPE` must be one of `A`, `B`, or `C`. The `CLIENT_TYPE` corresponds to which problem `DroneRouting` solves (and therefore what it outputs).

- `-h, --help`

Print a short description of this program and its arguments and `exit(0)`.

Valid examples of how to execute the program:

```
DroneRouting --clientType A    (OK) Must type input by hand, no < redirect.
DroneRouting -h < inputFile    (OK) -h happens before we realize there's no -c.
DroneRouting -c C < inputFile  (OK)
DroneRouting -c BLAH           (BAD)
```

We will not be specifically error-checking your command-line handling, but we expect that your program conforms with the default behavior of getopt_long. Incorrect command-line handling may lead to a variety of difficult-to-diagnose problems.

Client Type A — Sparse Path MST

Task

Client Type A companies are companies that, for whatever reason, want to deliver things as speedily as possible and want to implement Drone Type I as their delivery method while minimizing beacon installation costs. Client Type A companies are located on areas with more diverse terrain where beacons are unable to be installed amongst certain paths between campus facilities. Your task is to find the most cost efficient installation of beacons for the drones to reach any facility at the company from any other facility. This is in essence an MST problem in the form of beacon installation costs. Note that it is possible for a set of facilities to be unreachable by another set of facilities, in which case your algorithm should generate a MST forest so EECS drones can still fly amongst any given subset of facilities.

Input Format

On startups, your program, `DroneRouting`, reads from standard input (`cin`) a description of an abstracted layout of the campus on a cartesian plane. The line indicates how many campus facilities exist on the map:

```
Facilities: <num_facilities>
```

Followed by `<num_facilities>` lines indicating the integer coordinates of every facility on the map in the following format:

```
<X_coord> <Y_coord>
```

Coordinates may be negative. You may assume that both X_coord and Y_coord will fit into an integer. You may also assume the distances between any two points will fit into a double. This applies to all coordinates in this project, not just part A.

Note that each line automatically indicates which facility the coordinates are for. The first pair of coordinates is for facility # `0`, the second pair of coordinates is for facility # `1`, and the `n`th pair of coordinates is for facility # `n-1`. The next line will indicate how many paths exist amongst the facilities:

```
Paths: <num_paths>
```

Followed by `<num_paths>` lines in the following format:

```
<lower_facility_number> <higher_facility_number>
```

where the pair of facility numbers indicate the existence of a path between two facilities. Note that you will not need to check for edges that are indicated twice.

Example Input Format

```
Facilities: 5
0 -3
1 5
10 12
9 8
-1 -1
Paths: 5
0 1
0 3
2 4
2 3
1 4
```

Output Format

For output, you should print the total distance of the MST you generate on the first line. This distance is the sum of the distances of all paths in your MST. You should then print all paths in the MST, where each path is a line that contains a pair of facility numbers describing a path on the campus beacon installation map from the first facility to the second facility (please make sure the first facility has a lower integral value than the second facility). All output should be printed to standard output (cout).

The output should be of the format:

```
<total_distance>
<lower_facility_number> <higher_facility_number>
<lower_facility_number> <higher_facility_number>
...
```

For example, given a particular input file (**not** the one above), a valid output for Client Type A could be:

```
22.03
0 3
0 2
1 2
```

Note: You should also always print the pairs of facilities that describe a path such that the one on the left has a smaller integer value than the one on the right. In other words:

`1 2`
is possible valid output, while
`2 1`
is not.

The absolute ordering of the facility pairs *does not matter*. We will take your output and figure out if you have all appropriate edges.

Client Type B — Connected Coordinates MST

Task

Client Type B companies are the same as Client Type A companies but the location of their companies are located on beacon friendly terrain, therefore all facilities are connected by paths to all other facilities. Your task for Client Type B is therefore the same as Client Type A except the input will be different.

Input Format

The input format for client type B is in the following format:

```
Facilities: <num_facilities>
```

Followed by `<num_facilities>` lines indicating the coordinates of every facility on the map in the following format:

```
<X_coord> <Y_coord>
```

You may assume that every facility is connected to every other facility with an edge weight equal the the Euclidean distance between them.

Sample

```
Facilities: 6
1 2
4 20
-3 10
5 50
2 10
5 5
```

Client Type C — TSP

Task

Client Type C companies are interested in simply implementing Drone Type II, but they would like to have the drones be programmed to make a circuit through the entire campus, stopping at all campus facilities. Your task for Client Type C is to find the most efficient way for a drone to reach all the campus facilities and return to home base upon completion. This is in essence a Traveling Salesman Problem (TSP).

Input Format

The input format is the exact same as that for Client Type B. Also like Client Type B, **you may assume that every facility is connected to every other facility.**

Output Format

You should begin your output by printing the overall path length of your round trip tour on the first line, which is a double written out following the rule in Appendix A. On the next line, output the facility numbers in the order in which you visit them. The initial location should be the first facility (0) and the last should be the location directly before returning to home base (don't print 0 again at the end). The facility numbers should be separated by a single space. After the last location index, the file should end with a newline character (with no space between the last location and the newline). All output should be printed to standard output (`cout`).

Sample I/O

Input:

```
Facilities: 5
-4 3
0 5
7 -2
3 8
2 1
```

Output:

```
31.64
0 4 2 3 1
```

OR

```
31.64
0 1 3 2 4
```

Branch & Bound

To find an optimal tour, start by implementing the brute force method of exhaustive enumeration that evaluates every tour and picks the smallest tour. Once you have the brute force enumeration correct, you can speed up your program by bounding.

Clever optimizations (identifying hopeless branches of search early) can make it *a hundred times* faster. **Remember that there is a tradeoff between the time it takes to run your bounding function and how many branches that bound lets you prune.**

Libraries and Restrictions

We highly encourage the use of the STL for this project with the exception of three prohibited features: The C++11 regular expressions (RegEx) library (whose implementation in gcc 4.7 is unreliable), shared_ptr, and the thread/atomics libraries (which spoil runtime measurements). Do not use other non-STL libraries (e.g., boost, pthreads, etc).

Testing and Debugging

Part of this project is to prepare several test cases that will expose defects in buggy solutions - your own or someone else's. As this should be useful for testing and debugging your programs, **we strongly recommend** that you **first** try to catch a few of our intentionally-buggy solutions with your test cases, before completing your solution. The autograder will also tell you if one of your own test cases exposes bugs in your solution.

Each test case should consist of an input file. When we run your test cases on one of intentionally-buggy project solutions, we compare the output to that of a correct project solution. If the outputs differ, the test case is said to expose that bug.

Test cases should be named `test-n-CLIENT_TYPE.txt` where 0 < n <= 10. The autograder's buggy solutions will run your test cases for the specified `CLIENT_TYPE`.

Your test cases may have no more than 20 lines in any one file. You may submit up to 10 test cases (though it is possible to get full credit with fewer test cases). Note that the tests the autograder runs on your solution are **NOT** limited to 20 lines in a file, your solution should not impose any size limits (as long as sufficient system memory is available).

Submitting to the Autograder

Do all of your work (with all needed files, as well as test cases) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- You have deleted all .o files and your executable(s). Typing '`make clean`' should accomplish this.
- Your makefile is called Makefile. Typing '`make -R -r`' builds your code without errors and generates an executable file called `DroneRouting`. (Note that the command-line options -R and -r disable automatic build rules, which will not work on the autograder).
- Your Makefile specifies that you are compiling with the gcc optimization option `-O3`. This is extremely important for getting all of the performance points, as `-O3` can speed up code by an order of magnitude.
- Your test case files are named `test-n-CLIENT_TYPE.txt` and no other project file names begin with test. Up to 10 test cases may be submitted.
- The total size of your program and test cases does not exceed 2MB.
- You don't have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (i.e. the .git folder used by git source code management).
- Your code compiles and runs correctly using version 4.7.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 4.7.0 running on Linux (students using other compilers and OS did observe incompatibilities). **Note: To compile with g++ version 4.7.0 on CAEN you must put the following at the top of your Makefile:**

```
PATH := /usr/um/gcc-4.7.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-4.7.0/lib64
LD_RUN_PATH := /usr/um/gcc-4.7.0/lib64
```

Do not cut and paste the above lines from this PDF project spec file, it will NOT work. PDF files use non-ASCII characters for hyphens.

Turn in all of the following files:
- All your .h and or .cpp files for the project
- Your Makefile
- Your test case files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. In this directory, run
`dos2unix -U *; tar czf ./submit.tar.gz *.cpp *.h Makefile test-*.txt`
This will prepare a suitable file in your working directory.

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. **Note that when the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and

kept track of) between them. You may submit up to four times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). We will count only your last submission for your grade. Part of the programming skill is knowing when you are done (when you have achieved your task and have no bugs); this is reflected in this grading policy. We realize that it is possible for you to score higher with earlier submissions to the autograder; however this will have no bearing on your grade. We strongly recommend that you use some form of revision control (ie: SVN, GIT, etc) and that you 'commit' your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and CTools regarding the use of version control.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to note if the autograder shows that one of your own test cases exposes a bug in your solution (at the bottom).

Grading

90 points -- Your grade will be derived from correctness and performance (runtime). This will be determined by the autograder. On this project we expect a much broader spread of runtimes than on previous projects. Therefore, we may adjust the runtime-sensitive component of the score for several days, but will freeze it a few days before the deadline. As with all projects, the test cases used for the final grading are likely to be different.

10 points -- Test case coverage (effectiveness at exposing buggy solutions).

We also reserve the right to deduct up to 5 points for bad programming style, code that is unnecessarily duplicated, etc.

Refer to the Project 1 spec for details about what constitutes good/bad style.

Hints and Advice

Visualizing the input and output maps

It will be difficult to get this project right without visualizing your MSTs and TSP tours. We will be providing a visualizer tool for the input and output graphs in this project at Resources/Project 4/ on ctools. Following is some description on how you could use them:

You can run the visualizer on your personal computer (Mac or Ubuntu) or on the CAEN machines. There is a one-time setup needed to install the environment necessary in either case. On your personal computers, this one-time setup will install certain packages and would

ask for our root permissions to do so. On CAEN, there is no such requirement, but you still have to perform the one-time setup by providing with your uniqname. The CAEN setup will also automatically create a webspace for you -- your personal website will be hosted at: http://www-personal.umich.edu/~uniqname/

This webspace will be useful for you to visualize figures and plots that you create through the provided commandline tools on CAEN -- this is provided in order to give a fluid visualization pipeline that doesn't involve copying of files through ssh.

Setup on personal computer: Mac (one-time requirement)

```
$bash setup_mac.sh
```

Setup on personal computer: Ubuntu (one-time requirement)

```
$bash setup_ubuntu.sh
```

Setup on personal computer: Windows (one-time requirement)

```
Please refer to README_win.txt
```

Setup on CAEN (one-time requirement)

```
$bash setup_caen.sh <uniqname>
```

where you should type in the uniqname ignoring the characters <>

Once setup is finished, you can use the following commands to run the visualizer on your input and/or output maps.

Running the Visualizer

You can run the visualizer with just the input map (or) with both the input map as well as the output you generate in your program, whether MST/TSP. Running the visualizer will generate visualizations accordingly. The visualizer tool will automatically determine the specific mode of operation (clients A/B/C or MST/TSP etc.) for both inputs and outputs, so you do not have to be concerned about it.

On your personal computers (in order words, local machines), the visualizer will attempt to automatically open the visualizations for you, using the default applications installed. On CAEN machines, the visualizations will be saved to your personal website which you can then open in any browser to visualize. Specifically, these visualizations in CAEN for the input and output maps would be saved at:
http://www-personal.umich.edu/~uniqname/input.png
http://www-personal.umich.edu/~uniqname/output.png

Running the Visualizer locally (Mac or Ubuntu)

The following would visualize both input and output maps:

```
$bash viz_local.sh <input_file> <output_file>
```

Example: `$bash viz_local.sh input-1.txt output-1.txt`

The following would visualize only the input map:
`$bash viz_local.sh <input_file>`
Example: `$bash viz_local.sh input-1.txt`

Running the Visualizer locally (Windows)
`Please refer to README_win.txt`

Running the Visualizer on CAEN
The following would visualize both input and output maps:
`$bash viz_caen.sh <input_file> <output_file>`
Example: `$bash viz_caen.sh input-1.txt output-1.txt`

The following would visualize only the input map:
`$bash viz_caen.sh <input_file>`
Example: `$bash viz_caen.sh input-1.txt`

These visualizations generated in CAEN would be saved at:
http://www-personal.umich.edu/~uniqname/input.png
http://www-personal.umich.edu/~uniqname/output.png
which you can access using any web browser.

Tip: One of the first visual checks you can perform: if your TSP tour self-intersects, then it's not optimal (why not? can this idea be used for optimization algorithms?).

Valgrind and other tips

Running your code locally in **valgrind** can help you find and remove undefined (buggy) behavior and memory leaks from your code. This can save you from losing points in the final run when you mistakenly believe your code to be correct.

It is extremely helpful to compile your code with the following gcc options: -Wall -Wextra -Wvla -Wconversion -pedantic. This way the compiler can warn you about poor style and parts of your code that may result in unintended/undefined behavior.

Make sure that you are using **getopt_long** for handling command-line options.

Appendix A - Outputting Doubles

In order to ensure that your output is within the tolerable margins of error for the autograder to correctly grade your program you **must** run the following lines of code before you output anything to cout. We highly recommend that you simply put them at the top of your main function so that you don't forget about them.

```
cout << std::setprecision(2); //Always show 2 decimal places
cout << std::fixed; //Disable scientific notation for large numbers
```

You will need to `#include <iomanip>` to use this code.

Appendix B - Overlap between parts

Note that parts A & B **share the same output format,** and parts B & C **share the same input format.** This is on purpose - make sure to write functions to avoid duplicating code! You may also find that can write many helper functions that can be used in multiple parts of the project.

However, despite the fact that both parts A & B are doing the same thing (finding a minimum spanning tree), you will not be able to reuse all of your code for the two parts.

Part A deals with **sparse** graphs. That is, the number of edges **E** will be far smaller than **V^2** (the number of vertices squared).

Part B deals with connected, **dense** graphs.

In both graph representation and MST-finding algorithm, you will need to take into account the differences between the two modes. Also note that in order for us to create test cases large enough to measure your program performance, **V** will be **very very large.**

This is most significant when trying to store an adjacency matrix. Because V is so large, **it will not be possible to store a full adjacency matrix.** To give you a rough idea, lets say we have 1,000 vertices. In a fully connected graph, this would require roughly 500,000 entries in an adjacency matrix. If each edge stores 3 integers (12 bytes), then this adjacency matrix would require **4.8 megabytes** of RAM! We will be testing your program with graphs larger than 1,000 vertices.

One suggestion would be to write two separate graph classes for the two input modes. Writing a graph class could help with isolating separate logic and make debugging easier.

Appendix C - Brute Force Enumeration

Finding the optimal TSP solution is an NP-hard problem. As of now, there is no known polynomial solution to the problem other than enumerating all possible paths. This is **V!** where **V** is the number of vertices in the graph.

Here is some code to get you started on enumerating all possible paths and will also prove helpful later when you try to apply bounds.

```
void permute(vector<int>& path, queue<int>& unvisited) {
  if(unvisited.empty()) // complete path!
    print(path);
  for(int i = 0; i < unvisited.size(); ++i) {
    path.push_back(unvisited.front());
    unvisited.pop();
    permute(path, unvisited);
    unvisited.push(path.back());
    path.pop_back();
  }
}
```

If we call permute with `path = {}` and `unvisited = {0, 1, 2}`, we will get the following output:

0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0

This permute() function can be really confusing at first, but it is very efficient and very easy to adapt for branch and bound. While std::next_permutation() does the same thing, it is a little tricker to use when pruning.

If you are confused about the details of this function and how to prune with it, it will be covered in more detail in discussion.

Appendix D - Branch & Bound

Once you have a simple brute-force enumeration finished for part C, it's time to speed up your code by bounding. Bounding is covered in more detail in lecture, but here are a couple of things to keep in mind.

Upper Bounds

An upper bound is an **overestimation** of the optimal solution in a set of solutions. This means that at all times, the following property should hold:

upper_bound >= optimal_solution

Rather than starting your branch and bound with a random cycle, you may be able to speed up your program by starting with an upper bound. There are many algorithms designed to give a better-than-random (but not optimal) cycle that can be used for an upper bound.

One of the simplest is the **nearest neighbor** algorithm. This is a greedy algorithm that starts with a random vertex, and simply adds the closest yet unvisited vertex to the cycle until all vertices are visited. While far from the best graph approximation algorithm, nearest neighbor should be sufficient in this project as an initial upper bound. We do not recommended spending large amounts of time researching other upper bounds unless the rest of the project is already finished and you want to fine-optimize your part C, because the accuracy of an upper bound is not as influential in performance as the lower bound.

Lower Bounds

A lower bound is an **underestimation** of the best possible cycle in a given set of cycles. This means that the following must hold for a valid lower bound:

lower_bound <= optimal_solution

Because of this property, we know we can prune if:

upper_bound < lower_bound.

When the lower bound is computed to be all possible cycles that exist in the current "branch" of enumeration.

There are many different lower bounds that exist in the TSP problem. It is up to you to determine which one is most appropriate for the time and memory performance requirements of the autograder. For both upper and lower bounds, there always a trade-off between accuracy and performance.

Accuracy vs Performance

The more accurate a bound is, the closer it is to an optimal solution, and the more accurately/frequently your program can prune redundant solutions. However, just because a bound is accurate does not make it good - bounds must be **faster** than enumerating all possible solutions in order for them to be of any use.

You may find through experimentation that a faster, yet less accurate lower bound may lead to better performance.