EECS 281 - Winter 2014 Project 4

2014: A Base Station Plan v1.0

Due Tuesday April 22nd 11:55pm

Overview

Welcome to the base station construction plan around the Great Smoky Mountains National Park! As a fresh member of the construction team to the park, you are thinking of best ways of visiting base stations in the National Park and its surrounding area.

Your first task in Part A is to plan roads connecting our base stations efficiently. Then in Part B & C, you will design a trip to all base stations and check for construction process of them. Since you intend to accomplish the trip in only one day, you expect to minimize the travel distance from the starting point and back to it before midnight.

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 mode help with another mode)**.**

Project Goals

- Understand and implement MST algorithms. Be able to determine whether Prim's or Kruskal's is more efficient for a particular scenario.
- Understand and implement a Branch and Bound algorithm. Develop a fast and effective bounding algorithm.
- Explore various heuristic approaches to achieving a nearly-optimal solution as fast as possible (Part C).
- Use of gnuplot for visualization

Map Input

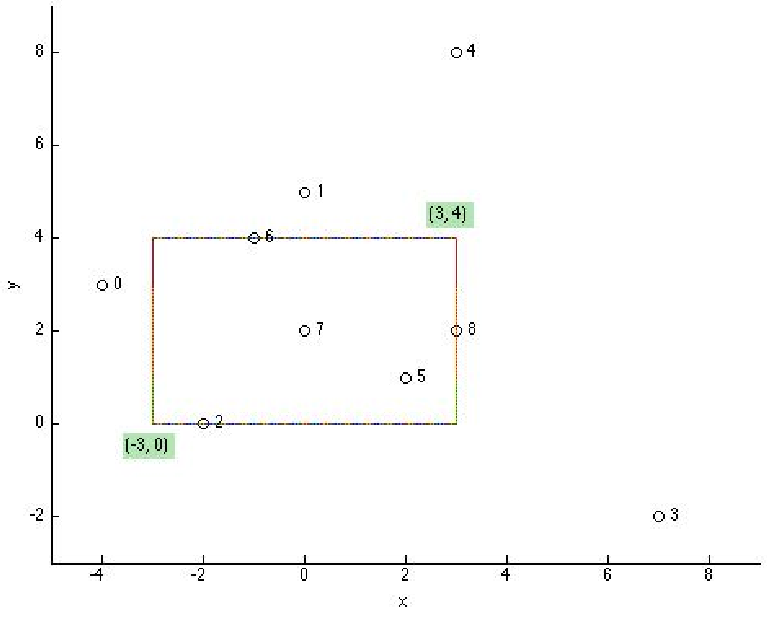
On startup, your program, `base`, reads input from standard input (cin) describing boundary of the National Park and a list of locations of base stations (some are located at entrances of the National Park on its boundary and others can be located in and out of the park). The National Park and its surroundings are mapped on a grid. The rectangular-shaped virtual boundary is described by its lower-left and upper-right corners. Then you will be given a list of `M` locations with associated integer coordinates (`x,y`). The locations are ordered (the first coordinate you read corresponds to location 0, the second coordinate to location 1), and are indexed. The traveller always starts at the 0-th location.

Formally, the input will be given by 4 integer numbers on the first line, namely '`lx ly rx ry`', representing x and y coordinates of lower-left corner and x and y coordinates of upper-right corner of the boundary of national park, where `lx < rx` and `ly < ry` are guaranteed. This is followed by '`M`', the number of locations by itself on a line, followed by a list of x, y coordinates in the form: '`x y`' (without the apostrophes). You may assume that the input will always be well-formed integer numbers (it will always conform to this format and you do not need to check for errors). There may be blank lines at the end of the file but nowhere else.

Sample input:

```
-3 0 3 4
9
-4 3
0 5
-2 0
7 -2
3 8
2 1
-1 4
0 2
3 2
```

The above sample can be visualized as follows, where numbers shown right to circles are the position indices and coordinates with green background color are positions of lower-left corner and upper-right corner of the boundary.



Note: There are many ways to represent this configuration internally in your program, and this will affect your runtime. Choose your data structures wisely!

Movement Rules

Distance:
We represent the path between points as a sequence of 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 you 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.

You are **not** allowed to wrap around the edges of the world (you **cannot** go above the top of the map to arrive at the bottom).

The boundary of the National Park:
There is a rectangular-shaped boundary for the National Park that is defined by two corner points as described above. The boundary separates the map into two parts, in and out of the National Parks. Base stations are possible to be located in both parts.

There are also base stations located at entrances of the National Park on its boundary. You can tell whether base station is at an entrance by checking if it is on borders of the rectangle.

In Part A, you may travel directed between any pair of base stations in the same part of the map separated by the boundary, or directly between a base station in or out of the Park and one at an entrance. You can also go directly from one station at the entrance to another one also at the entrance. However, there is no way that you travel between a base station in the Park and one out of the Park since you cannot fly across the boundary.

When in the round tour mode (Part B or C), you are capable of flying! It just means that traveling directly between any two base stations on the map is allowed. No boundaries any more. Just ignore it!

Command Line Input

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

● `-m, --mode MODE`
This command line option must be specified, if it is not, print a useful error message to standard error (cerr) and exit(1). `MODE` is a required argument. Set the program mode to `MODE`. `MODE` must be one of `MST`, `OPTTSP`, or `FASTTSP`. The `MODE` corresponds to what algorithm `base` runs (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:

```
base --mode MST                 (OK)  must type input by hand; no < redirect
base -h < inputFile.txt         (OK, -h happens before we realize there's no -m)
base -m OPTTSP < inputFile.blah (OK)
base -m BLAH                    (BAD)
```

(Remember that when we redirect input, it does not affect the command line. Redirecting input just sends the file contents to cin. You should not have to modify your code to allow this to work, the operating system will handle it.)

We will not be specifically error-checking your command-line handling, however 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.

Algorithms

Your program should run **one and only one** of the following modes at runtime depending on the `mode` option for that particular program call. We divide it into parts for your convenience, though *you may find that elements and algorithms of some parts can help with others*.

Part A - Base station road planning (MST mode)

When run in the `MST` mode, your program should return a network of roads of the minimal total distance which connects all the base stations, which should be a minimal spanning tree with vertices = base locations, and edges = the roads you plan to build (edge weight = Euclidean distance). You may use either Prim's or Kruskal's to do this, though one of them will prove faster. Hint: Unless you want to implement both and compare, think about the nature of the graph (how many vertices and edges does it have?). You are free to adapt code from the lecture slides to fit this project, but you will want to carefully think about the data structures *necessary* to do each part (storing unnecessary data can slow you down). Your program must always generate one valid MST for each input.

Output Format

For `MST` mode, you should print the total weight of the MST you generate by itself on the first line; this weight is the sum of the weights of all edges in your MST (which would be Euclidean distances). To be clear, the weight double should be formatted as specified in Appendix A. You should then print all edges in the MST. All output should be printed to standard output (cout).

The output should be of the format:

```
totalWeight
node node
node node
...
```

where the nodes are the location indices (ordered by input order) corresponding to the vertices of the MST. Each line that contains a pair of nodes describes an edge in the MST from the first node to the second. For example, given a particular input file (not the one above), your MST mode output might be:

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

You should also always print the pairs of vertices that describe an edge 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

```
2 1
```

is not.

If it is not possible to construct an MST for all base stations (if there are stations on either side of the boundary but no stations at an entrance on the boundary), your program should print the single-line message “Cannot construct MST” to cerr and exit(1).

Parts B & C - Round Trip Plan (OPTTSP & FASTTSP)

Your task is to find a lowest fuel cost path from the first base station, after reaching base stations, back to it. This is basically a Traveling Salesman Problem (TSP).

For OPTTSP mode, you must find an **optimal** solution to the TSP (the actual minimum distance necessary). For FASTTSP mode, you do **not** need to produce an optimal result, but your solution should be close to optimal. Because your FASTTSP algorithm does not need to be perfectly optimal, we expect it to run much faster. More on the differences between OPTTSP and FASTTSP modes will be discussed later in the spec.

For **both** methods:

You starts from the first base station, the 0-th location in input.

You visit each base station exactly once (there's no point in returning to a place already sampled), except when it returns to the first base station.

The total fuel cost of the route is defined same as the total distances travelled (sum of Euclidean distance of all traversed edges).

Your program must print the indices of the sampling locations in an order such that the total length of this route plan is as small as possible. More details below.

Output Format (both Parts B and C)
You should begin your output by printing the overall fuel cost of your tour on the first line, which is a double written out following the rule in Appendix A. On the next line, output the location indices in the order in which you visit them. The initial location should be the home base (0) and the last should be the location directly before returning to home base. The location indices 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).

Below is output corresponding to the sample input.

Input:
```
-3 3 4 4
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
```

For Part B: OPTTSP

To find an optimal tour, you can start with the brute force method of exhaustive enumeration that evaluates every tour and picks a smallest tour. By structuring this enumeration in a clever way, you could determine that some branches of the search cannot lead to optimal solutions. For example, you could compute *lower bounds* on the length of any full tour that can be found in a given branch. If such a lower bound exceeds the cost of a full solution you have found previously, you can skip this branch as hopeless. If implemented correctly, such a **branch-and-bound** method should **always** produce an optimal solution. It will not scale as well as sorting or searching algorithms do for other problems, but it should be usable with a small number of locations. Clever optimizations (identifying hopeless branches of search early) can make it *a hundred times* faster. Drawing TSP tours on paper and solving small location configurations to optimality by hand should be very useful. **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.**

Given an input set of *N* locations defined by integer coordinates, produce an optimal tour using branch-and-bound algorithms. Your program should **always** produce the shortest possible tour as a solution, even if computing that solution is time-consuming. You will be given a 30-second cpu time limit. If your program does not produce a valid solution, it will fail the test case.

For Part C: FASTTSP

For large number of base stations it might be good enough to generate a good-enough round trip plan instead of an optimal one. You can use *heuristics* to find near-optimal tours. A heuristic is an algorithm that can produce a good answer that is not necessarily the best answer. For example, you can skip a branch speculatively rather than waiting to know for a fact that it can be skipped. There are many other simple heuristics, such as starting with a random tour and trying to improve it by small changes.

You should be able to produce a solution to the TSP within a 15-second cpu time limit that is as close as possible to the optimal tour length, **it does not need to be optimal**. In the best case, the produced tour length will equal to the optimal tour length.

You are allowed to use any combination of algorithms for this section that we have covered in class, including the MST algorithm you wrote for Part A and the branch and bound algorithm from Part B. (See Appendix A for suggestions)

You need to be creative when designing your algorithms for this section. You are free to implement any other algorithm you choose, so long as it meets the time and memory constraints. However, you should *not* use any advanced algorithms or formulas (such as Simulated Annealing, Genetic Algorithms and Tabu search - they are too slow) that are significantly different from what has been covered in class. Instead, creatively combine the algorithms that you already know and come up with concise optimizations.

Libraries and Restrictions

We highly encourage the use of the STL for this project with the exception of two prohibited features:
The C++11 regular expressions library (whose implementation in gcc 4.7 is unreliable) and the thread/atomics libraries (which spoil runtime measurements). Do not use other 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-MODE.txt` where 0 < n <= 10. The autograder's buggy solutions will run your test cases in the specified `MODE`.

Your test cases may have no more than 10 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 10 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`' shall accomplish this.
- Your makefile is called Makefile. Typing '`make -R -r`' builds your code without errors and generates an executable file called `base`. (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-MODE.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 strange 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.

Runtime Quality Tradeoffs

In this project there is no single correct answer (unlike previous projects). Accordingly, the grading of your problem will not be as simple as a 'diff', but will instead be a result of evaluating your output.

For Part C in particular, we expect to see greater variation in student output. Part C asks you to solve a hard problem, and with the given time constraints, we don't actually expect your output to be optimal for all cases. The quality of your solutions may even vary from case to case. We want you to quickly produce solutions that are close to optimal. This inevitably creates tradeoffs between solution optimality and runtime.

For grading, we will impose a uniform time budget of **15 seconds** for all Part C test cases. This budget does not depend on the size of the problems. This fixed budget means that you may find it useful to implement more than one algorithm or heuristic that you use in Part C and choose which one to invoke based on the input size. In other words, for smaller test cases, you can use a slower, but more thorough search algorithm.

Hints and Advice

It will be difficult to get this project right without visualizing your MSTs and TSP tours. We recommend that you use **gnuplot** (available on Linux systems) - your C++ program can produce a script for gnuplot (with point data embedded), but you can use any other tool, such as Excel to achieve similar results. We have provided a sample script for showing how to use gnuplot in the context of this project in Resources/Projects/Supplementary Material on ctools. One of the first visual checks you can perform perform: if your TSP tour self-intersects, then it's not optimal (why not? can this idea be used for optimization algorithms?).

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

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
```

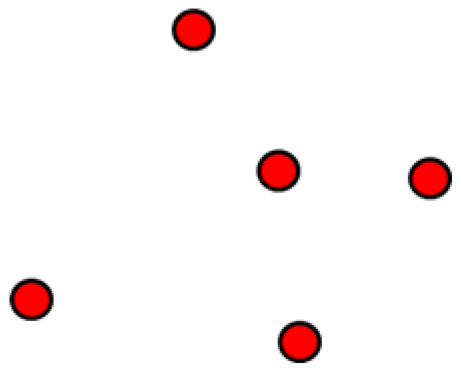
You will need to `#include <iomanip>` to use this code.

Appendix B

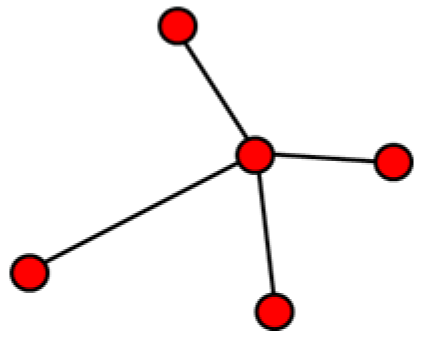
One possible heuristic is to construct a starting point for your TSP tour by following MST edges and skipping previously visited vertices. By using this technique correctly, you can find a tour that is guaranteed to be no more than twice the optimal solution's length (and use this "2x" check for debugging). You can then use this starting point to make adjustments and do better.

Corner Cutting algorithm illustrated

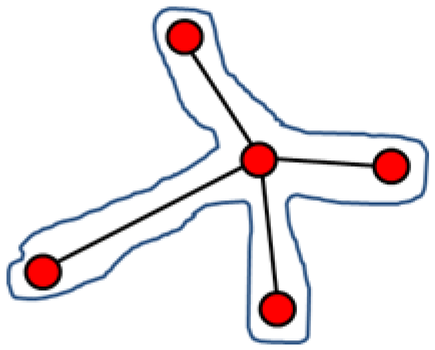
Suppose locations are distributed in an input map as shown below:



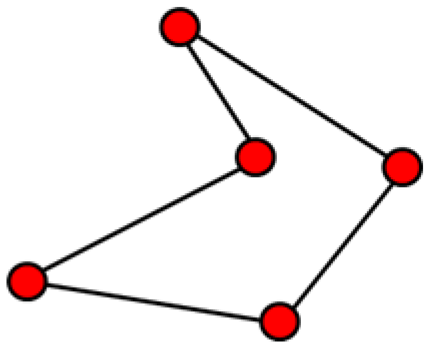
Below is an MST that would be formed from the above locations.



Below is a path taken by strictly following the edges of the MST.



However, by “cutting corners”, an effective TSP solution can be generated. This is possible because once a vertex is visited, it will not be visited again. In the above path that strictly follows the edges of the MST, the middle vertex is visited four times (visited after an outer vertex is visited). If the middle vertex is skipped after the first visit, a TSP tour shown below is achieved.



The reason we bring up this ‘twice-around-the-tree’ heuristic is to state the theorem that the resulting tours are no worse than 2x the MST cost.

The following is an explanation/proof sketch for the 2x bound:

If you perform a DFS traversal of a tree and return to the initial node, you have visited every edge exactly twice ("going in" and "returning"), so this gives you exactly double the cost/length of the MST (the fact that the tree is minimal is not important for the logic of the proof - this works for any tree). Since the tree is spanning, you have visited all landmarks. The only problem with this twice-around-the-tree self-intersecting path is that it's not a tour. It can be turned into a tour by taking shortcuts (i.e., taking shortcuts is important not only to reduce the length).

When considering other heuristics, note that:
1. The theorem allows you to upper-bound the cost of optimal TSP tours based on MST length.
2. The theorem has a constructive proof - a heuristic that always produces TSP tours that satisfy this upper bound[2].

As a consequence, your heuristic should also satisfy the 2x upper bound; if not, you can just implement the twice-around-the-tree heuristic. However, we do not require this because there are much better heuristics – ones that are faster and produce better results.

Appendix C

Be mindful of how you keep track of your runtime in Part C; calling system functions to check the time can be expensive and should be done sparingly. To give an analogy:
Let's say you are trying to make it to a deadline. So, as you work, you look at your watch every now and then. At some point, you realize that you are spending most of your time looking at the watch and making very little progress. So, you decide to throw the watch away to save time. Is this a good idea? No, because you won't be able to track time accurately. A good solution is to only look at the watch every few minutes (so that the majority of your time is spent on your work) and to make sure you are done a few minutes before the deadline.

---

[2] Such heuristics are also called approximation algorithms.