EECS 281 – Winter 2017
Programming Project 4:
Lukefahr | Paoletti 2017

Due April 18th, 2017

Overview

It's election season, and you've been selected as the manager for Senator Lukefahr and General Paoletti's presidential campaign. In order to triumph in the race to the White House, you must strategically plan Senator Lukefahr's campaign trail to maximize his votes without exceeding his campaign budget. Your task is to find the optimal states for him to campaign in and plan his route to get to them.

Senator Lukefahr and his running mate, General Paoletti, are in a tight race with their competitors:

Brofessor Baid / Monsieur Marino
Lord Luum / Her Highness Hua
Sudo "The Root Node" Sagar / Potentate PotatoBot

There will be three modes for the project: `CAMPAIGN`, `MST`, and `PATH`. Each will have separate input formats and behavior and are described individually below.

Note: These scenarios are separate; your program will run exactly one of them based on the command passed to it. So you can consider implementing solutions and unit testing the three parts independently!

Project Goals

- Understand and implement branch and bound algorithms. Develop fast and effective bounding algorithms.
- Explore various heuristic approaches to achieving a nearly-optimal solution much faster than optimal solutions.

Version 2017-03-27
Authored by: Ish Baid, Anna Dai, Noah Klimisch, Andrew Marino, Aary Sagar
Current Version by: Luum Habtemariam and Jiaxi Wu

Input Format

There are three different input formats for this project, one for each mode. The input formats for each of these modes are described in their respective sections.

On startup, your program, `election`, will read input from standard input (`cin`) describing the locations or campaign details of states in the country. The very first line of input will give the number of states to consider. The rest of the input depends on which input mode is being used.

Command Line Input

Your program, `election`, will take the following case-sensitive command line options:

- `-m, --mode MODE`
  This command line option is required. If it is not given, print a useful error message to the standard error stream (`cerr`) and return from `main()` with the exit code 1 denoting failure. `MODE` sets the program mode to `MODE`. `MODE` must be one of `CAMPAIGN`, `MST`, or `PATH`.

Examples of command line arguments:

```
election --mode CAMPAIGN                    (OK, but input would be typed by hand)
election -m CAMPAIGN < inputFile.txt        (OK, input comes from file)
election -m PATH < inputFile.txt                (OK, PATH  mode)
election -m < inputFile.txt                     (INVALID, no mode  specified)
```

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.

Part A: Campaign Planning (`CAMPAIGN` mode)

Program Description

In the U.S. Presidential election, each state is allocated a fixed amount of votes in the Electoral College. Within each state, the vote is winner-take-all[1], so the candidate who receives a plurality of votes within a given state gets all of the Electoral College votes allocated to that state[2].

You will be given a list of states that are key to the success of his campaign, with the number of votes that each state will provide on Election day, should he choose to campaign there. However, you only have a finite amount of money allocated to campaigning! Luckily, your researchers have determined the exact amount of money that Senator Lukefahr needs to spend campaigning in each state in order to win the votes for that state. It is your job to select the states in which Sen. Lukefahr should campaign in order to maximize his total vote count.

Input Format

You will be given the number of states and the campaign's maximum budget `B`. You will then be given a list of `N` states. Each state is associated with a non-negative campaigning cost `C` and number of electoral votes for that state.

Formally, the input will be formatted as such: the first line will contain a single number denoting the number of states, `N`. On the next line is the total budget of Lukefahr's campaign (a double). This will be followed by a list of `N` {cost, value} data tuples in the form: `C V`. Each 2-tuple will be on its own line. Each `C` represents the campaign cost to win that state (a floating point number), and each `V` represents the state's value in electoral votes (an integer).

Note: For the the purposes of this part of the project, we will assume that there is no bound on the number of states that can be read in. This means that a test case that contains greater than 50 states is still a valid test case.

Sample input:

```
Number of states (N): 7
Total budget (B): $3458.5
$1177.44 131
$1054.44 35
$1252.66 105
$946.021 136
$1298.03 120
$1051.04 120
```

[1] https://en.wikipedia.org/wiki/First-past-the-post_voting

[2] Except for Nebraska and Maine, which allocate their votes in the Electoral College at the district level. For the purposes of the project, ignore exceptions like these.

```
$1071.83 65
```

Error checking on input list
To insure the input data is well-formed, you must check that the input conforms to the following constraints:

- `N` is a non-negative integer value (you may assume it will be an integer).
- `C` is a positive decimal value. I.e. `C` > 0 .
- `V` is a positive integer value. I.e. 0 < `V` (you may assume it will be an integer).

If you detect invalid input at any time during the program, print a helpful message to `cerr` and return from `main()` with the exit code 1.

Description
Your task is to select a subset of the states to campaign in such that you maximize your votes without exceeding your campaign budget.

For each state in the input list, you have been provided with the *exact* **cost** necessarily to campaign there and win the state, and thus all of the **votes** allocated to that state.

We say *exact* amount of cost, meaning that to devote any *less* budget to a state would result in you not winning that state; likewise, to devote *extra* is superfluous. Hence an optimal strategy would not spend any money on a state where doing so does not get you to win that state.

You can only win the votes for a given state at most once. Again, you should choose not to campaign in a state if you do not have sufficient funds to win all the electoral votes there.

In the case where two subset of states results in the same total number of votes, choose the subset of states with the lesser cost. Remember, your primary goal is to maximize the number of votes. The cost only comes into play should you have tie. In the case where the costs are also equal, choose the subset with the smaller first differing index.

Output Format
As output, your program will print the total number of votes that your campaign solution wins for Senator Lukefahr. Your program will then, on the subsequent lines, print the indices of the states in **increasing** order of the numeric value of the index, with each index listed on its own line. Here, the index of a state is given by the order that the states were listed in the original input list (0-indexed).

All output should be printed to standard output (i.e., `cout`).

For example, if given the input file above, your program should produce:

```
Expected number of votes on Election day: 387
Senator Lukefahr should campaign in the following states:
0
3
5
```

Your solution must be **optimal** for the given input of states. In other words, the set of states that you visit yields the greatest possible number of votes while remaining within your campaign's budget.

Part B: Route Information Gathering (`MST` mode)

Program Description

Senator Lukefahr and General Paoletti are designing their economic reform, starting with new trade agreements. One key piece of information needed to finish their plan is the minimum distance between every important trading outpost.

In this case, the trading posts and trading routes can be represented as a graph with posts as their vertices(nodes) and routes between them as the edges. The weight of an edge is given by the Euclidean distance between its endpoints, as defined by:

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Since there are potential routes between any two trading outposts, the graph is completely connected.

However, there's a complication with the trading routes: some trading posts are more difficult to ship out of than others. In these cases, the simple Euclidean distance doesn't provide an accurate metric of the weight between nodes. You will account for this by having the weights of all the edges incident with these vertices multiplied by a constant factor given in the input.

In this representation, the minimum distance between every trading outpost can be solved by constructing a minimum spanning tree of the trading posts.

Input Format

Formally, the input will be formatted as such: the first line will contain a single number denoting the number of trading posts, `N`. This will be followed by a list of `N` {x, y} data tuples in the form: `(x,y)`, representing the coordinates of that trading post. Each tuple will be on its own line. This will be followed by the amount of difficult posts, `M` (int), and the difficulty multiplier, `D` (positive floating point number), and the indices of the trading posts that are affected by it. These node indices, `idx` (integer), are in the range `[0,N)`. A difficult trading post will have all of the weights of edges incident to it multiplied by the difficulty multiplier.

Note: an edge that connects two difficult nodes will have its weight multiplied by the difficulty **twice**. Once by each one of the difficulty multipliers of the nodes it connects.

Sample input:

```
Number of states (N): 8
971 388
237 96
326 360
59 148
165 669
400 346
870 831
892 687
Difficult posts (M): 2
Difficulty (D): 1.7
0
5
```

Error checking on input list
To insure the input data is well-formed, you must check that the input conforms to the following constraints:

- `N` is a positive integer (you may assume it will be an integer).
- `D` is positive.
- `M` is a non-negative integer (you may assume it will be an integer).
- Each node in the difficulty list is an integer value in **[0, `N`)** (you may assume it will be an integer).

If you detect invalid input at any time during the program, print a helpful message to `cerr` and return from `main()` with the exit code 1.

Situations you don't need to verify
The following situations are guaranteed to be true

- There are exactly `N` coordinates of the type `x, y`
- There are exactly `M` difficult posts.

Output Format
For `MST` mode, you should print the total weight of the minimum distances you generate by itself on a line; this weight is the sum of the weights (including multipliers based on difficulty) of all the edges in your graph. You should then print all trading posts that are connected by a minimum distance. All output should be printed to standard output (`cout`).

The output should be of the format:

```
Total distance: <weight>
Trade routes:
```

```
<node> <node>
<node> <node>
...
```

where `<node>` is the index corresponding to the vertices of the MST and a pair of nodes on a given line of the output describes an edge in the MST from the first node to the second. To be clear, the weight should be formatted as a double (with 2-decimal-point precision—see Appendix A), and the node numbers should be integer values when printed. For example, given the example input file above, your `MST` mode output might be:

```
Total distance: 2265.58
Trade routes:
0 7
6 7
2 5
2 4
1 3
2 7
1 2
```

You should also always print the pairs of vertices that describe an edge such that the index on the left has a smaller integer value than the index on the right. In other words:

```
1 2
```

is a possible valid edge of output, but

```
2 1
```

is not.

Part C: Route Planning (`PATH` mode)

Program Description

Senator Lukefahr and General Paoletti are getting ready for the Presidential primaries, so they have their team establish a list of all the states they wish to visit. However, Sen. Lukefahr hates flying, so you must compute a way for them to reach each state once and return home, while traveling the minimum distance.

As before, we can represent these states as nodes in a completely connected graph, where the weights of the edges are given by the Euclidean distance between them.

Input Format

The input format for `PATH` mode is the same as for `MST` mode. However, your Computer Security team has managed to hack Lord Luum's database, and found the best tour that he has computed. You know Lord

Luum didn't finish EECS 281, so there could be a better tour. This path of this tour will be given at the end of the input.

Sample input:

```
Number of states (N): 7
31 429
242 509
109 973
829 665
548 938
877 487
141 453
Luum path (L):
0 4 2 3 1 5 6
```

Sample input (Autograder spec example):

```
Number of states (N): 7
31 429
242 509
109 973
829 665
548 938
877 487
141 453
Luum path (L):
0 2 4 3 5 1 6
```

Note: Both these test cases yield the same output which is shown below

Error checking on input list
To insure the input data is well-formed, you must check that the input conforms to the following constraints:

- `N` is a non-negative integer (you may assume it will be an integer).
- Each node in the Luum path `L` is an integer in **[0, `N`)** (you may assume it will be an integer).
- The Luum path `L` visits every node exactly once (i.e. it is a valid, if non-optimal, solution).

If you detect invalid input at any time during the program, print a helpful message to `cerr` and return from `main()` with the exit code 1.

Situations you don't need to verify
The following situations are guaranteed to be true

- There are exactly `N` coordinates of the type `x, y`

Output Format
You should begin your output by printing the overall length of your tour on a line. On the next line, output the nodes in the order in which the senator and general travel. The initial node should be the starting location index and the last node should be the location number directly before returning back to the 0-th location. The nodes in your tour should be printed such that they are separated by an end line. All output should be printed to standard output (`cout`).

For example, if given the input file above, your program could produce:

```
The total campaign length is: 2429.55
Lukefahr should visit each state in the following order:
0
2
4
3
5
1
6
```

or

```
The total campaign length is: 2429.55
Lukefahr should visit each state in the following order:
0
6
1
5
3
4
2
```

Branch and Bound Algorithms

To find an optimal solution for parts A and C, you could start with the brute force method of exhaustive enumeration that evaluates every path and picks the best one. By structuring this enumeration in a clever way though, 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 path 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 moderate number of locations. Clever optimizations (identifying hopeless branches of search early) can make the search *a hundred times* faster. Drawing paths 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.**

For part A, you are given an input set of *N* states defined by a particular value (in number of votes), as well as a particular cost for campaigning there, and you must select the optimal subset of the states to maximize votes while not exceeding the maximum allowable cost (i.e. your budget). Your program should **always** select the subset giving maximum votes, even if computing that solution is time-consuming.

For part C, you are given an input set of *N* locations defined by integer coordinates, and you must produce an optimal tour. Your program should **always** produce the shortest possible path as a solution.

Note that the branching and bounding characteristics are different for these different problems: in part A the problem is searching *subsets* (meaning that the order in which the states are selected does not affect the overall solution), whereas in part C the problem is searching *paths* (meaning that the order in which the locations are selected affects the solution). Your branching algorithm must be structured accordingly.

For these two parts, you will be given a 30-second CPU time limit to generate your solution. If your program does not produce a valid solution, it will fail the test case. Your solution will also be judged by time and space budgets as per previous projects.

Libraries and Restrictions

We highly encourage the use of the STL for this project. However, RegEx library (whose implementation in gcc 5.x is incomplete), smart pointer facilities, and thread/atomics libraries are prohibited. As always, the use of libraries not included in the C/C++ standard libraries (e.g. those in boost, etc.) is forbidden.

Testing and Debugging

Part of this project is to prepare several test files 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 recommend that you **first** try to catch a few of our intentionally-buggy solutions with your test files, before completing your solution. The autograder will also tell you if one of your own test files exposes bugs in your solution.

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

Test files should be named `test-n-MODE.txt` where 0 < `n` ≤ 10 and `MODE` is one of `{CAMPAIGN, MST, PATH}`. The autograder's buggy solutions will run your test files in the specified `MODE`.

Your test files may have no more than 10 vertices in any one file. You may submit up to 10 test files (though it is possible to get full credit with fewer test files). 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 files) 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 `election`. (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 files are named as described and no other project file names begin with test. Up to 10 test files may be submitted.
- The total size of your program and test files 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 (e.g., the .git folder used by git source code management).
- Your code compiles and runs correctly using version 5.1.0 of the g++ compiler on the CAEN servers. To compile with g++ version 5.1.0 on CAEN you **must** have the following at the top of your Makefile:

```
PATH := /usr/um/gcc-5.1.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-5.1.0/lib64
```

```
          LD_RUN_PATH := /usr/um/gcc-5.1.0/lib64
```

Turn in all of the following files:

- All your .h and or .cpp files for the project
- Your Makefile
- Your test 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. Alternatively, the sample makefile has a useful submit that will do this for you.

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 three 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 best submission for your grade.

We strongly recommend that you use some form of version 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 Canvas 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 files exposes a bug in your solution (at the bottom).

Grading

90 points — Your grade will be derived from correctness and performance (runtime). Details will be determined by the autograder.
10 points — Test file coverage (effectiveness at exposing buggy solutions).

You will be deducted 5 points if you have memory leaks. Make sure to run your code under valgrind before each submit. (This is also a good idea because it will let you know if you have undefined behavior, which will cause your code to crash on the autograder.)

Version 2017-03-27
Authored by: Ish Baid, Anna Dai, Noah Klimisch, Andrew Marino, Aary Sagar
Current Version by: Luum Habtemariam and Jiaxi Wu

Appendix A: Precision for Standard Output

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: Exceptions and RAII

The RAII (short for "Resource Acquisition is Initialization") pattern is a key component in most C++ programs and is used considerably in the C++ standard library. RAII is an object oriented programming idiom which helps to automate the process of acquiring a resource on construction and deallocating a resource on destruction.

For the purposes of this project you should remember that containers in the C++ standard library deallocate their resources on destruction. Calling `exit(1)` will not call the destructors for any containers (like `std::vector`) you may be using in your program, and as a result you would leak memory and fail leak checks on the autograder. This is where exceptions are useful. You should use exceptions in this project when executing your error checks, so when you detect an error you throw an exception such as `std::runtime_error`, catch it at the top level in `main()`, print to the standard error stream (`cerr`) then return a 1 denoting failure. So for example you can structure your code like so

```
int main() {

    try {
        // read in input
        read_in_input(); // throws an exception when an error is found!
        do_stuff();
    } catch(std::exception& exc) {
        // all resources (eg. memory) should have been released by now
        cerr << exc.what() << endl;
        return 1;
    }

    return 0;
}
```

Appendix C: Test Case Naming Convention

The test case names on the autograder are rather straightforward.

- INV tests: Testing error checking cases
- KNAP tests: Testing Part A
- MST tests: Testing Part B
- TSP tests: Testing Part C

Remember, when the autograders are turned on and accepting submissions, there will be an announcement on Piazza.