Learning outcomes. *Gain deeper understanding of shortest paths algorithms and graph algorithm implementation; design experiments to test hypotheses about algorithm behavior; write a compelling comparative analysis based on experimental results.*

Overview

You will implement and compare the performance of two algorithms for the all-pairs shortest paths problem. I will still require implementations in Java, C, or C++. This time Python is not a good option because efficient access to two-dimensional arrays is required.

You have (or will have) learned two ways of finding shortest paths between all pairs of vertices for undirected[1] graphs with positive edge weights.

- *n* iterations of Dijkstra's single source algorithm, one iteration per start node
- the Floyd-Warshall algorithm

You will implement each of these algorithms and do experiments reporting both runtime and number of edge-weight comparisons. Both algorithms have logic that requires comparing weights of two edges. In the case of Dijkstra's, these comparisons take place both when deciding whether an edge improves distance to a node[2] *and* when a node's position in the priority queue is updated. You will probably need to implement your own heap to make this possible (and you may also need to do so because neither the Java priority queue nor the C++ STL support a decreaseKey operation). The part of the Floyd-Warshall algorithm that compares edge weights is obvious.

Specifications

Your programs must run from the command line and read input from standard input in **gph** format, as described here.

```
c comment line 1
...
c comment line k
g number_of_nodes number_of_edges
--------------- (not part of the input)
OPTIONAL node position info (for conversion to graphml)
n v_1 x_1 y_1
...
n v_n x_n y_n
------------
ALWAYS
e source_1 target_1 weight_1
```

[1] Undirected rather than directed because the graph generators I will provide may not yield interesting results if edges are interpreted as directed, i.e., there may be pairs with no paths between them.

[2] An alternate implementation suggested to me by Dr. Sheehy simply puts edges on the priority queue, obviating the need for a decreaseKey operation; see **sheehy_sp.alg** in Galant under **Algorithms.**

```
    ...
    e source_m target_m weight_m

v_1 through v_n are node numbers, typically 1 through n
x_i, y_i are x and y coordinates of v_i
```

Output will have the following format.

```
    number_of_nodes
    d_1_1
    ...
    d_1_n
    -1
    d_2_1
    ...
    d_2_n
    -1
    ...
    -1
    d_n_1
    ...
    d_n_n
d_i_j is the length of the shortest path from node i to node j
```

So every line contains a single integer - all weights are integers and there are exactly n^2 lines. Lines `d_i_i` will be 0 and the output matrix will be symmetric, i.e., `d_i_j = d_j_i`. The `-1`'s are there to make the output more readable when debugging. There is not one at the very end. Because of the output size you will want to suppress it when experimenting with larger graphs by sending it to `/dev/null` on the command line.

The files `dual_8_12-normalized.gph` and `dual_8_12.out` give the input and output, respectively, for a sample graph. I have also included the corresponding graphml file so that you can visualize Dijkstra's algorithm on it.

Additional output giving statistics about the run should be sent to standard error. Format of the additional output is …

```
runtime     SECONDS
comparisons NUMBER_OF_COMPARISONS
```

The runtime should *not* include the time it takes to read the input or produce the output. All three programming languages have functions/methods that facilitate access to runtime during execution. All runtimes should be at least 1/10 of a second and should be rounded to the nearest 1/10 of a second.

Your code should be well documented with

- comments that clearly indicate the relationship between each part of the algorithm and the segment of code that implements it
- compact methods/functions with names that clearly identify what they do

- descriptions of any parameters and/or return values
- descriptive variable names

Project report

A critical part of this assignment is a written report. The report should include the following elements.

Implementation description. An account of details that were necessary for the implementation of the algorithms using linked lists in your chosen programming language. Focus on details that are not obvious from the high-level description of the algorithms. Keep your descriptions as language-independent as possible; linked-lists are universal data structures – there is no need to belabor the details of, for example, what specific Java methods you used. What is much more important is when and how you needed to traverse a list, relink the cells of the list, etc.

Experimental design. Explain what hypotheses you wanted to test and how you decided to test them. Describe the inputs you used (and explain why you used those inputs). Be sure that the set of inputs includes a variety of sizes and categories and some large enough to have runtimes of more than a minute for the slowest algorithm. Describe how many trials you performed for each size/type of input.

Experimental results. Use charts and tables to effectively illustrate your experimental results. Provide an analysis of your results. What did they tell you about average case behavior of the algorithms? Did you observe anything unexpected?

Conclusions and future work. Summarize your results and make some suggestions about future experiments that you believe are worth pursuing and why.

Recommended Reading (see the ProgramOne folder).

- *How to Present a Paper on Experimental Work with Algorithms*, Catherine McGeogh and Berhard Moret, *SIGACT News,* November 1999.
- *A Theoretician's Guide to the Experimental Analysis of Algorithms,* David S. Johnson, in *Fifth and Sixth DIMACS Implementation Challenges,* American Mathematical Society.
- *Tables and Charts*, from lecture notes for a course on Experimental Algorithmics.

What to submit

Using the submission mechanism for this assignment, ***one chosen team member*** should submit a zip archive that unzips into a directory called `P1-team` (where `team` is the name of the team) containing

- all of your source code
- a `README.txt` file with simple instructions for compiling and running your programs
- a `run_apsp.sh` script that takes either `d` or `fw` as a command line argument, for Dijkstra or Floyd-Warshall, respectively.
- your project report as a pdf file

- a list of names and email addresses of all your team members including yourself in a file called `collaborators.csv`. Format is one line per team member, each line has the form `name_of_person,email_of_person`

Most of this is very similar to what you submitted for the first project.

Input Graphs

The repository https://github.ncsu.edu/mfms/MST has a collection of graph generators in the src-generate directory. It also has a `gph2graphml.py` script (in the scripts directory) that will convert a graph from the gph format required for input to your program to the graphml format used by Galant. If the gph file includes node positions, these will be honored during translation (with appropriate scaling). If not, they are assigned randomly. The script has a large number of command-line arguments that allow you to specify, for example, the size of the window (in pixels). Use the `-h` option to find out what these are. As far as I know, all generators (I should really say both generators - the only ones you should use are `nextgen` and `random_delaunay.py`) produce graphs whose nodes are numbered consecutively 1 through *n*, where *n* is the number of nodes.

Also included in the repository, in the src folder, is a collection of implementations of minimum spanning tree algorithms in C. The files `graph.h` and `graph.c` have pretty much everything you need to read a graph and store it in adjacency list format. If you do this in C++ or Java, the memory allocation will be simpler and you may use C++ STL vectors or Java ArrayLists for the adjacency lists. I did not write this code, so I cannot vouch for its quality, but I know that it is efficient. We did experiments with graphs having up to 32 million edges. The limitation we ran into was not runtime (less than five minutes for all the algorithms if I recall correctly), but memory. That is no longer likely to be an issue.

For the adjacency matrix representation you are on your own, but that should be simple in all three languages. A warning about geometric graphs: *the number of edges reported at the beginning of the input file is just an estimate; it is the number of edges requested by the user, but the generator needs to determine a distance threshold for the edges, which may not, in general, give the exact number. Also, a geometric graph may not end up being connected* - the program reports the number of connected components to stderr along with other information. You may want to use the `generate.sh` script, a wrapper for `nextgen`; it performs three useful tasks: (i) corrects the number of edges reported on the `g` line for geometric graphs; (ii) allows you to specify how many instances you want and keeps trying (within reason) until you get that many (some geometric graphs may not be connected; these will be rejected); and (iii) sends output to files with standardized names. Run it with no command-line arguments to get a usage message. See also the instructions file - `instructions.pdf`.

Division of Labor

I highly recommend that you ***not*** take the approach of assigning each person a set of tasks and then combining your efforts at the end. The most successful teamwork results if you cooperate closely on all tasks. Use pair programming during the implementation phase. Programming can easily be split into low-level functionality (methods that create lists from

input, traverse lists, relink cells, etc.) and algorithm architecture (recursive calls, insertion, merging). Make heavy use of all communication platforms you feel comfortable with.

Discuss the experimental design and the main points of the report together. Certainly it makes sense to delegate a discrete task such as creating a single chart or table to one individual, but allow each person to do at least one pass through the text.

Above all, use the project as an opportunity to get to know each other and (I hope) enjoy working together.

Rubric

Rubric item	*points*	*comments*
completeness	`5`	are all the required elements present in the archive
compilation and execution	`15/alg = 30`	do the programs compile without error on a university vcl server running Linux and execute successfully on a variety of input instances
code and comments	`10`	is the code easy to read and easy to connect with the algorithm it implements
experimental design	`20`	are there coherent and well-considered hypotheses; is the set of instances chosen thoughtfully
charts and tables	`15`	are charts and tables compelling and easy to interpret; you should have learned some best practices by now
analysis and conclusions	`20`	are the conclusions interesting, i.e., do they go beyond merely reporting what is easily seen in the charts and tables