Assignment 9: Spanning Trees
CS 758/858, Fall 2024
Due at **11:30pm on Mon Oct 28**

Implementation

The skeleton code on the course web page is the start of a program for generating minimum spanning trees of planar graphs. You may implement any graph representation and spanning tree algorithm you wish.

The test harness can either generate a random graph or generate a graph based on a PGM image. As detailed below, the harness is also capable of visualizing the resulting spanning tree, which for an image gives an interesting rendering effect.

Your Program

The input to your program will consist of the number of vertices, followed by an assignment of x and y coordinates to those vertices. Edge weights are calculated by the skeleton code as the Euclidean distance between the endpoints, and edges that are 'short enough' are included in the graph. The output will be a list of edges, represented by their start and end vertices, and the cost of the minimum spanning tree. Vertices are identified by integers according to their order in the input, starting from 0.

So for the input:

```
4
0 0
1 0
0 1
1 1
```

We would expect to see output like:

```
0 1
0 2
1 3
3.
```

`tree <points-file> <max-distance>` runs your program on the specified points file where all points that are at most `max-distance` away from one another are translated into vertices that have an edge between them. Since the input graphs are constrained to be within a unit-square, $0 <$ `max-distance` < 1.5.

Please note that some of the more interesting input graphs are quite large (a few hundred million vertices), so be careful when running your program. Make liberal use of `ulimit` or you'll end up thrashing agate (or whatever machine you happen to be on). If you find that a graph is too difficult to solve, try turning down the `max-distance` parameter. This will reduce the number of edges and make the resulting graph much smaller.

Harness

`span-harness` is the harness program. It's capable of converting portable gray maps (`.pgm` files) into graphs for the `tree` program, converting the output of itself and `tree` into a postscript file, and of testing your program on random input graphs.

`-b <filename>` sets the binary, default is `./tree`

`-o <filename>` sets an image output file. If you have trouble viewing postscript files, the program `ps2pdf` will convert to PDF.

`-t <filename>` sets a tree file (the output of `tree` on a points file) to be used to generate an image.

`-p <filename>` sets a points file (the output of `span-harness` on a pgm file) to be used to generate an image.

`--scale <float>` sets the size of the output image (in inches). The output images are square.

`--seed <int>` sets the random seed (useful for trying the same input many times)

`-m <int>` Maximum random graph size to consider

`-s <int>` Step size for considering many random graphs

`-v <int>` verbosity level

`-d <float>` Sets maximum distance (default 0.1)

Please note that the harness does not completely validate your output when running on random graphs. It simply makes sure that the edges you print out cover all vertices. If you want to see that you've computed the correct minimum spanning tree, compare the cost of your spanning tree to that of the reference for a given graph.

Examples

```
./span-harness -m 10000 -s 1000
```

will run `./tree <filename> 0.1` for 10 randomly generated graphs of size 1,000 through 10,000 stepping by 1,000's.

```
./span-harness -i ruml.pgm
```

converts a gray map of Wheeler's head into a set of points (stored in `ruml`).

```
./tree ruml 0.1 > ruml_tree
```

Solves the spanning tree of Wheeler's head and puts it into a file called `ruml_tree`.

```
./span-harness -p ruml -t ruml_tree -o ruml.ps
```

converts the points file `ruml` and the tree file `ruml_tree` into a post-script image stored in `ruml.ps`.

```
convert foo.jpg bar.pgm
```

converts a jpeg file into the corresponding portable gray map. Requires imagemagik be installed (which it is on `agate`).

Written Problems

1. Briefly list any parts of your program which are not fully working. Include transcripts or plots showing the successes or failures. Is there anything else that we should know when evaluating your implementation work?
2. Exercise 21.1–2 in CLRS.
3. Exercise 21.1–9 in CLRS.
4. (Those in 858 only) Problem 21–3 in CLRS.
5. What suggestions do you have for improving this assignment in the future?

Submission

Electronically submit using the script on agate (eg, `~cs758/scripts/sub758 9-undergrad your-asn9-dir`).

Evaluation

In addition to correctness, your work will be evaluated on clarity and efficiency. Tentative breakdown: For 758: 6 implementation, 4 for written problems. For 858: 4 implementation, 6 for written problems.