Distributed Minimum Spanning Tree

Overview

This is a distributed program that implements a distributed Minimum Spanning Tree solver in DistAlgo.

Problem Description

A spanning tree is defined as a tree which is a sub-graph of a given graph and connects all the nodes in the graph. The graph must be a *connected* and *undirected* graph. A Minimum Spanning

Tree is a spanning tree whose sum of weights of the edges is the lowest of all possible spanning trees for the given graph.

Same graph, multiple minimum spanning trees

If a graph has multiple edges *with the same weight*, then the graph could have several MSTs. This situation can be obviated by constructing a graph whose edges have unique weights.

Algorithms

Sequential Solvers

There are two commonly used algorithms for finding the MST of a graph *sequentially*: *Prim's algorithm* and *Kruskal's algorithm*. To better understand the problem and get a feel for it, I wrote a Kruskal's solver which can found at `Kruskal.py`.

Distributed Solvers

Here the nodes of the graph are nodes in a distributed system; i.e. a set of computers/processes where each computer represents a node of the graph and the edge between two nodes of the graph respresent a *communication interlink* between two processes. Nodes (processes) without edges between are *not* allowed to communicate with each other.

The best known algorithm that solves this problem is the GHS algorithm of R. G. Gallager, P. A. Humblet and P. M. Spira. According to Wikipedia, there also is a parallelization of Prim's sequential algorithm by Nobri et al. The GHS algorithm could be considered the ***state of art*** for the distributed MST problem. I have implemented the GHS algorithm using DistAlgo, a seuperset of Python enhanced for distributed programming by Annie Liu, Bo Lin et al. from Stony Brook University.

A pre-condition for the GHS aglorithm is that the MST be unique, i.e. that all edges have *unique* weights.

Papers on GHS

I found two papers online that describe GHS. One is the original from 1983, by Gallager, Humlbet & Spira. The other is an enhanced version of the original (with better graphics, typesetting, explanation, etc.) prepared by Guy Flysher and Amir Rubinshtein. I followed the second one while creating my implementation in DistAlgo.

I've posted PDFs of both papers (found online) in this GitHub repo under the `papers` directory. Links to them are below:

- The Original paper by Gallager, Humblet and Spira from 1983.
- The enhanced version of the original prepared by Guy Flysher and Amir Rubinshtein. (*I recommend this one*)

High-level Explanation of the GHS Algorithm

The algorithm hinges on the idea of a "fragment of the MST". It relies on this property of MSTs:

- *If F is a fragment of an MST M, then joining the node of the other end of the minimum weight "outgoing" edge will yield yet another fragment of M.* Here, "outgoing" is defined as an edge which connects (any node in) the fragment to a node that is *not* part of the fragment.

The algorithm initially starts out by assigning *each node to a fragment of its own*. It then proceeds to to follow a set of steps to ***merge the fragments*** *over and over again*, until there is only **one** fragment left. The final fragment is equal to the MST for the graph. Additionally, each fragment has a property called its *level* which determines what kind of merge process occurs between two fragments. There are two kinds of merges: *absorptions* and *level-augmenting merges*.

The steps followed by the aglorithm are:

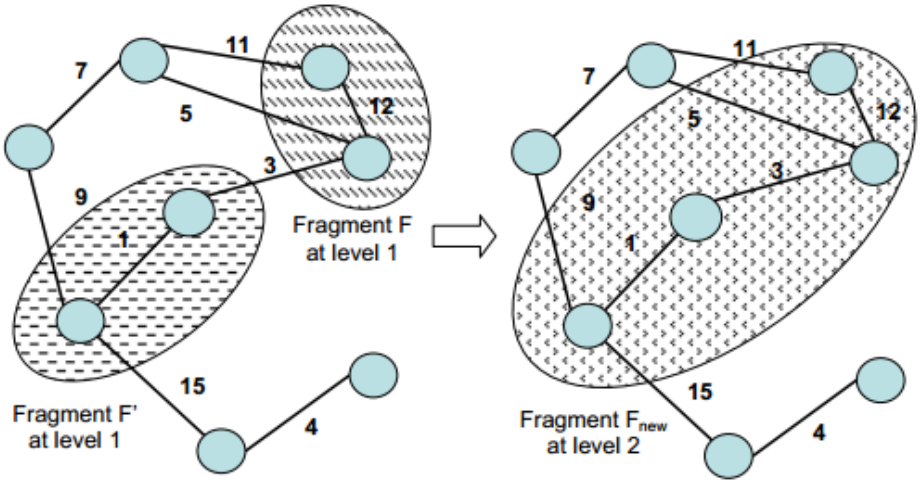
0. Initially, all fragments are at level 0 and contain just the one node. Additionally, all nodes are *sleeping* at first.
1. A node has to be woken up before it can do anything. A nicety of the GHS algorithm is that there are no restriction whatsoever on the wakeup process. One may opt, if necessary, to wake up all nodes immediately; or alternatively wake up just one single node. In the course of the algorithm, all nodes will eventually be owken up. Other operations in the algorithm result in other nodes waking up.
2. Every fragment finds its *minimum weight outgoing edge* and sends a **Connect** request over it containing the `level` of the fragment it belongs to.
3. Every time a fragment sends a *Connect* to another fragment, it enters a "waiting" state (denoted by `FOUND` in the code.) The fragment then, waits *until is either gets absorbed by or merges with the other fragment*.
4. When a fragment **receives** a Connect, the conditions that determine whether it should *absorb* or *merge* with the requesting fragment are as follows:
    - If the fragment it received the *Connect* request from is of a lower level, then that fragment gets *absorbed immediately*.
    - If the fragment it received the *Connect* request from is of a level *equal* to its own or *higher*, two things can happen:
        - If the fragment receving the Connect *has also* sent a Connect to the other fragment, they ***merge***.
        - In any other case, the fragment *simply does not reply* and *waits for the situation to change*. The way the algorithm works, a merge or absorption will occur eventually.
5. The termination case for the algorithm is, when a fragment is ***unable*** *to find a minimum weight outgoing edge*. This case means that it is the *only fragment left.* Therefore it must be the complete MST.
6. Finally two important thing to note are:
    - Fragments of level-1 and above, are controlled by a pair of nodes called the ***core nodes***. These nodes are first formed during the initial level-0 Connect exchanges. Nodes sending _Connect_s to each other form a level-1 fragment.

- Fragments are identified by the *weight* of the edge between the core nodes. Since, all edges have to be unique as pre-requisite for the GHS algorithm, the edge weight can be used to uniquely identify a fragment.

There is a lot more to inner working of the algorithms, such as how exactly mergers and absorptions occur, how the minimum weight outgoing edge is found, etc. However these details would be too low-level to be discussed in this document.

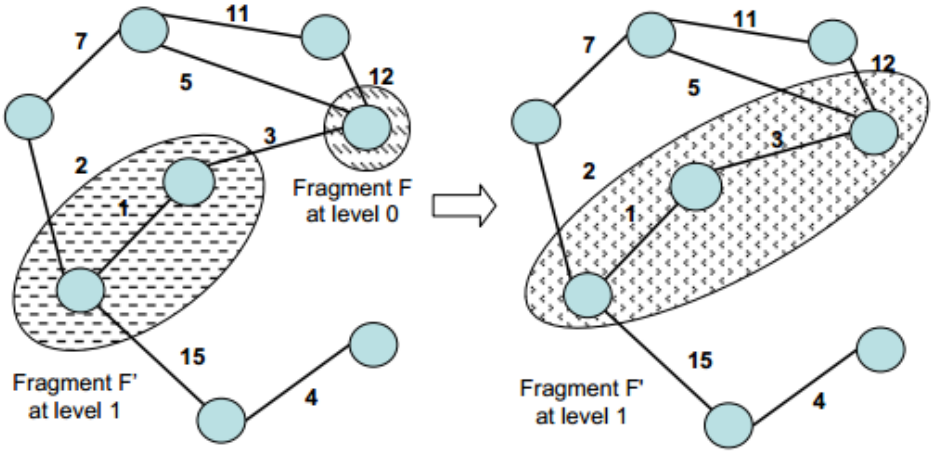
The following diagram (from Guy Flysher and Amir Rubinshtein's version of the GHS paper) illustrates the fragment *absorption* and *merge* processes:

- Case 1: If two fragments have the same *level* L and the same minimum weight outgoing edge, they combine to form a new fragment at level $(L+1)$.



The shared minimum weight outgoing edge is called the new fragment's *core* and its adjacent nodes are called the *core nodes*.

- Case 2: If fragment F is at level L and fragment F' at level $L'>L$ is at the other end of F's minimum outgoing edge, then fragment F is absorbed by F' (the level remains L').



The rule of thumb (The logic behind it will be explained in section 3.7) is: A low level component is never kept waiting.

Pseudocode

The pseudocode for GHS provided in the paper is quite low-level. I tried, but wasn't able to directly translate it to DistAlgo. The main problem I ran into it was that, DistAlgo does not provide a way (or I don't know of a way) to manipulate the local message queue directly. The pseudocode uses that

ability to "delay" processing a message, by putting it back in the end of the queue. My implementation differs in this aspect and also other aspects that make it more high-level and easier to understand (but less efficient.)

I largely followed the pseudocode as a guide, rather than following it directly. For my implementation I tried, to the maximal extent possible, to *work out the lower level details myself, while simply following the high-level details* of the algirithm as described and shown in the diagram above. The benefit of following the high-level explanation was that, I was able to keep the *big picture* in my head (all at once.) I found that impossible to do with the pseudocode (or with the low-level description, which contains a lot of minute details that are intricately connected with each other.) I found I was able to understand part of it at a time, but to hold the whole thing in my head at once was impossible.

The following is an image of the pseudocode for GHS extracted from Guy Flysher and Amir Rubinshtein's version of the paper:

The Algorithm (As Executed at Each Node)

(1) Response to spontaneous awakening (can occur only at a node in the sleeping state)
execute procedure *wakeup*

(2) procedure *wakeup*
begin let m be adjacent edge of minimum weight;
SE (m) ← *Branch;*
LN ← 0;
SN ← *Found;*
Find-count ← 0;
send *Connect(O)* on edge m
end

(3) Response to receipt of *Connect(L)* on edge j
begin if *SN* = *Sleeping* then execute procedure *wakeup;*
if L < *LN*
then begin *SE (j)* ← *Branch;*
send *Initiate(LN, FN, SN)* on edge j;
if *SN* = *Find* **then**
find-count ← *find-count* + *1*
end
else if *SE(j)* = *Basic*
then place received message on end of queue
else send *Initiate(LN + 1, w(j), Find)* on edge j
end

(4) Response to receipt of *Initiate* (L, F, S) on edge j
begin *LN* ← *L; FN* ← *F; SN* ← *S; in-branch* ← *j;*
best-edge ← *nil; best-wt* ← ∞ *;*
for all i ≠ j such that *SE(i)* = *Branch*
do begin send *Initiate(L, F, S)* on edge i;
if *S* = *Find* then *find-count* ← *find-count* + *1*
end;
if *S* = *Find* then execute procedure *test*

```
        end

(5) procedure test
        if there are adjacent edges in the state Basic
                then begin test-edge ← the minimum-weight adjacent edge in state
        Basic;
                                send Test(LN, FN) on test-edge
                        end
        else begin test-edge ← nil; execute procedure report end

(6) Response to receipt of Test(L, F) on edge j
        begin if SN = Sleeping then execute procedure wakeup;
                if L > LN then place received message on end of queue
                 else if F ≠ FN then send Accept on edge j
                         else begin if SE (j) = Basic then SE (j) ← Rejected;
                                        if test-edge ≠ j then send Reject on edge j
                                            else execute procedure test
                                end
                                end

(7) Response to receipt of Accept on edge j
        begin test-edge ← nil;
                if w(j) < best-wt
                   then begin best-edge ← j; best-wt ← w(j) end;
                execute procedure report
        end

(8) Response to receipt of Reject on edge j
        begin if SE (j) = Basic then SE (j) ← Rejected;
                execute procedure test
        end

(9) procedure report
        if find-count = 0 and test-edge = nil
          then begin SN ← Found;
                        send Report(best-wt) on in-branch
        end

(10) Response to receipt of Report(w) on edge j
        if j ≠ in-branch
          then begin find-count ← find-count - 1
                        if w < best-wt then begin best-wt ← w; best-edge ← j end;
                        execute procedure report
                end
          else if SN = Find then place received message on end of queue
                else if w > best-wt
                        then execute procedure change-core
                        else if w = best-wt = ∞ then halt

(11) procedure change-core
        if SE (best-edge) = Branch
                then send Change-core on best-edge
        else begin send Connect(LN) on best-edge;
                    SE (best-edge) ← Branch
        End
```

(12) Response to receipt of *Change-core*
execute procedure *change-core*

Implementation

The implementation is in DistAlgo 0.2. There are two algorithms: Kruskal's in `Kruskal.py` and GHS in `MST.dis.py`. I wrote Kruskal's just an exercise early on to get a better understanding of MSTs (and in an attempt to come up with my own distirbuted algorithm.) The relevant program in contained in its entirety in the file `MST.dis.py` in this directory. The `.py` was added to obtain syntax highlighting in GitHub. It is pointed to by the symlink `MST.dis`. `MST.dis` can be run using DistAlgo by typing: `python3 -m distalgo.runtime MST.dis`. Alternatively, there's a script `run.py` which does the same thing.

Usage

Both the GHS algorithm (in `MST.dis.py`) and `Kruskal.py` import the module `tools.py` which provides a common set of services like handling optargs, constructing the graph from a *graph file* and visualizing the solution using `matplotlib`. The graph file is a simple text file which lists all the edges in the graph in CSV-style, except without the commas. If no graph file is passed, then `graph-2` is used by default. `tools.py` builds from the graph file, a NetworkX graph object representing it.

The available arguments and purposes can be displayed by passing the `-h` argument to display the help message:

```
usage: MST.dis [-h] [-v] [-b BACKEND] [-o OUTPUT] [graph]

Finds the Minimum Spanning Tree (MST) of a given graph.

positional arguments:
  graph                 File listing the edges of a graph line-by-line in the
                        following style: "A B 2", where "A" and "B" are node
                        names and "2" is the weight of the edge connecting
                        them.

optional arguments:
  -h, --help            show this help message and exit
  -v, --visualize       Visualize the graph and its solution using matplotlib,
                        with the branches of the MST marked in thick blue.
  -b BACKEND, --backend BACKEND
                        Interactive GUI backend to be used by matplotlib for
                        visualization. Potential options are: GTK, GTKAgg,
                        GTKCairo, FltkAgg, MacOSX, QtAgg, Qt4Agg, TkAgg, WX,
                        WXAgg, CocoaAgg, GTK3Cairo, GTK3Agg. Default value is
                        Qt4Agg.
  -o OUTPUT, --output OUTPUT
```

```
                        File to write the solution (MST edge list) to. By
                        default it written to the file `sol`.
```

The Spark process

The `Spark` process is another process that is present in my implementation in addition to the node processes. It's job solely bureaucratic: to start the algorithm by waking up atleast one node, and to finish up the algorithm and present the results.

It accomplishes starting the algorithm by simply picking a node at random and sending it a `Wakeup` message. As an interesting side note, the GHS algorithm allows *any number of nodes to be woken up **spontaneously***. This aspect of the GHS algorithm can be easily tested by un-commenting two lines of code in Spark's main() function. The two lines send `Wakeup` messages to all nodes rather one random node.

Finally, output collection starts after `Spark` receives a `Finished` message from one of the two core nodes. Subsequently, it sends a `QueryBranches` message to every node process, and they all reply with the edges they know to be branches. (No single node knows all the branches of the MST, only which of its edges are branches.) The `Spark` processes collects these results and combines them in a set, then `output`s the solution using DistAlgo, and finally passes it on to `toos.py` to write it down to a file and/or visualize it using `matplotlib`.

Running

When `MST.dis` is run (by typing `python3 -m distalgo.runtime MST.dis` or using `run.py`), the program produces a long list of output messages, each emenating from the processes representing the nodes describing operations that occured at the node. A truncated example of the output is shown below:

```
[2012-10-11 17:18:47,434]runtime:INFO: Creating instances of Node..
[2012-10-11 17:18:47,458]runtime:INFO: 13 instances of Node created.
[2012-10-11 17:18:47,464]runtime:INFO: Creating instances of Spark..
[2012-10-11 17:18:47,465]runtime:INFO: 1 instances of Spark created.
[2012-10-11 17:18:47,472]runtime:INFO: Starting procs...
[2012-10-11 17:18:47,473]runtime:INFO: Starting procs...
[2012-10-11 17:18:47,475]Node(F):INFO: Received spontaneous Wakeup from: Spark
[2012-10-11 17:18:47,475]Node(F):INFO: F is waking up!
[2012-10-11 17:18:47,477]Node(E):INFO: Received Connect(0) from: F
[2012-10-11 17:18:47,477]Node(E):INFO: E is waking up!
[2012-10-11 17:18:47,478]Node(F):INFO: Received Connect(0) from: E
[2012-10-11 17:18:47,478]Node(E):INFO: E merging with F
[2012-10-11 17:18:47,479]Node(F):INFO: F merging with E
[2012-10-11 17:18:47,480]Node(F):INFO: Received Initiate(1, 1, 'Find') from: E
[2012-10-11 17:18:47,480]Node(E):INFO: Received Initiate(1, 1, 'Find') from: F
[2012-10-11 17:18:47,481]Node(F):INFO: F has sent Test() to A
[2012-10-11 17:18:47,481]Node(E):INFO: E has sent Test() to H
[2012-10-11 17:18:47,481]Node(A):INFO: Received Test(1, 1) from: F
[2012-10-11 17:18:47,482]Node(A):INFO: A is waking up!
```

```
[2012-10-11 17:18:47,482]Node(H):INFO: Received Test(1, 1) from: E
[2012-10-11 17:18:47,482]Node(H):INFO: H is waking up!
...
[2012-10-11 17:18:47,529]Node(M):INFO: M merging with L
[2012-10-11 17:18:47,530]Node(L):INFO: L merging with M
....
[2012-10-11 17:18:47,537]Node(K):INFO: K has sent Test() to J
[2012-10-11 17:18:47,537]Node(J):INFO: Received Test(1, 20) from: K
[2012-10-11 17:18:47,538]Node(J):INFO: J sent Accept() to K
[2012-10-11 17:18:47,538]Node(K):INFO: Outgoing Neighbor K -> J @ 23 [find_count =
[2012-10-11 17:18:47,538]Node(K):INFO: Least weight(23) outgoing edge from K: K ->
[2012-10-11 17:18:47,539]Node(L):INFO: Received [K, J] @ 23 from K [find_count = 0
[2012-10-11 17:18:47,539]Node(L):INFO: Least weight(23) outgoing edge from L: L ->
[2012-10-11 17:18:47,540]Node(L):INFO: Least weight(23) outgoing edge of fragment(
[2012-10-11 17:18:47,540]Node(M):INFO: Received [L, K, J] @ 23 from other core nod
[2012-10-11 17:18:47,541]Node(K):INFO: Fragment (20) ------------ Sending Connect(
[2012-10-11 17:18:47,541]Node(J):INFO: Received Connect(1) from: K
[2012-10-11 17:18:47,542]Node(J):INFO: J absorbing K
[2012-10-11 17:18:47,542]Node(J):INFO: J has sent FIND to branch K
[2012-10-11 17:18:47,542]Node(K):INFO: Received Initiate(2, 10, 'Find') from: J
[2012-10-11 17:18:47,543]Node(K):INFO: K has sent FIND to branch L
[2012-10-11 17:18:47,543]Node(L):INFO: Received Initiate(2, 10, 'Find') from: K
[2012-10-11 17:18:47,543]Node(L):INFO: L has sent FIND to branch M
[2012-10-11 17:18:47,544]Node(J):INFO: J Received Reject() from: K
[2012-10-11 17:18:47,544]Node(M):INFO: Received Initiate(2, 10, 'Find') from: L
[2012-10-11 17:18:47,544]Node(M):INFO: Least weight(999999999) outgoing edge from
[2012-10-11 17:18:47,543]Node(K):INFO: K sent Reject() to J
[2012-10-11 17:18:47,545]Node(L):INFO: Received None @ 999999999 from M [find_coun
[2012-10-11 17:18:47,545]Node(L):INFO: Least weight(999999999) outgoing edge from
[2012-10-11 17:18:47,546]Node(K):INFO: Received None @ 999999999 from L [find_coun
[2012-10-11 17:18:47,546]Node(K):INFO: Least weight(999999999) outgoing edge from
[2012-10-11 17:18:47,547]Node(J):INFO: Received None @ 999999999 from K [find_coun
[2012-10-11 17:18:47,547]Node(J):INFO: Least weight(999999999) outgoing edge from
[2012-10-11 17:18:47,548]Node(I):INFO: Received None @ 999999999 from J [find_coun
[2012-10-11 17:18:47,548]Node(I):INFO: Least weight(999999999) outgoing edge from
[2012-10-11 17:18:47,549]Node(I):INFO: ----- NO MORE OUTGOING EDGES -----
[2012-10-11 17:18:47,549]Node(E):INFO: Received None @ 999999999 from other core n
[2012-10-11 17:18:47,549]Node(E):INFO: ----- NO MORE OUTGOING EDGES -----
[2012-10-11 17:18:47,575]Spark(Spark):INFO: Solution: (A, B), (A, F), (C, D), (D,
[2012-10-11 17:18:47,592]runtime:INFO: ***** Statistics *****
* Total procs: 14

[2012-10-11 17:18:47,593]runtime:INFO: Terminating...
```

The `999999999` that can be seen conspicuously towards the end is a constant representing *infinity*. The "infinite" weight outgoing edge is used to denote the case where there are no outoging edges. When the two core nodes of the final fragment have received replies from all their branches indicating there are ***no more*** outoging edges, it sends a message to the special process `Spark`.

`Spark` immediately *presents* the solution, and sends a message to all the nodes indicating the completion of the algorithm. Then the program terminates. `Spark` presents the solution, by first using DistAlgo's `output` and printing the solution; and second by drawing the graph using

NetworkX and `matplotlib` if the user has opted to visualize the solution (using the `-v` option). The visualized graph denotes the branches of the MST using *thick blue edges*.

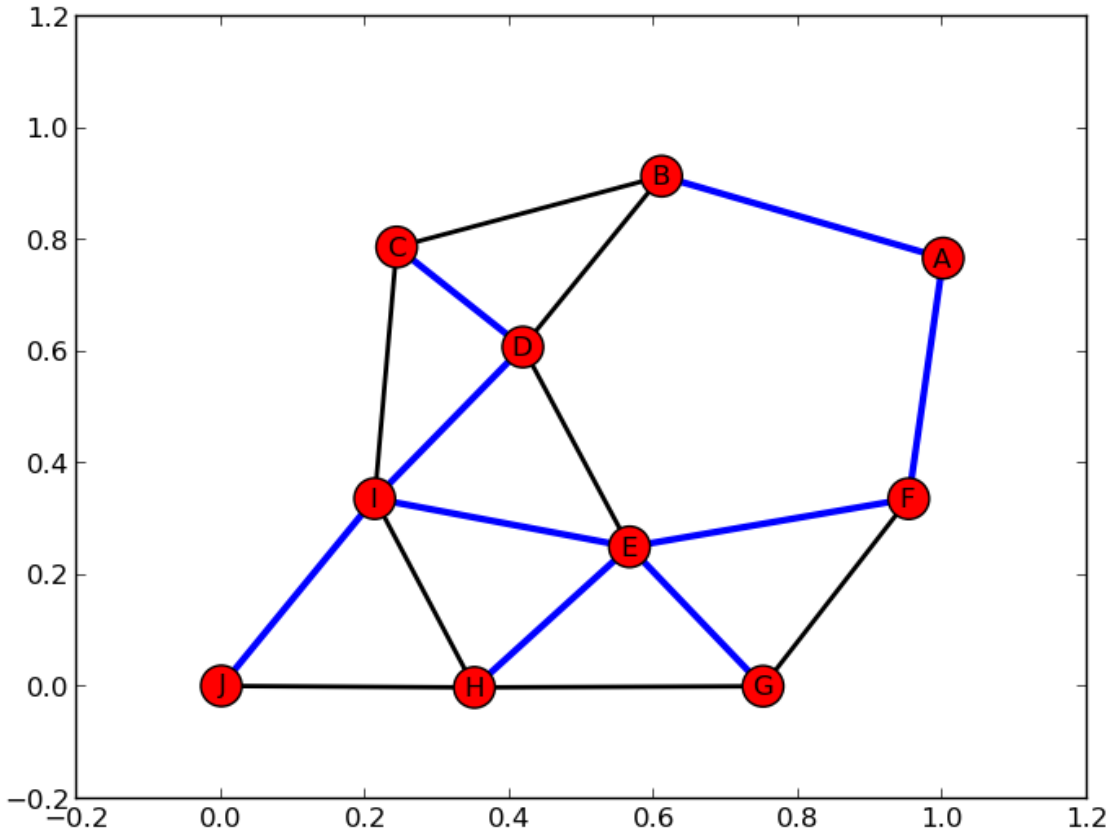
Test Cases

The following are a series of test cases run on the algorithm. The main test cases are enclosed in the files: `graph-1`, `graph-2` and `graph-3`. There is an easy way to verify the solution: Run `Kruskal.py` against the same graph. Kruskal is quite clever; it verifies its *own* solution against NetworkX's MST solver so there's no way it could be wrong.

Graph 1

For my initial test case which I used for much of the developed of my algorithm, I wanted to use a graph that was in particular designed to test this algorithm. So I looked online for web pages discussing Minimum Spanning Trees, and I finally came across one that had a graph with all unique weights: http://cgm.cs.mcgill.ca/~hagha/topic28/topic28.html

The following `matplotlib` diagram was generated representing the solution to `graph-1`:



The thick blue edges denote the branches of the MST (Minimum Spanning Tree).

What Happens

Some of the key GHS events that occur while solving `graph-1` are:

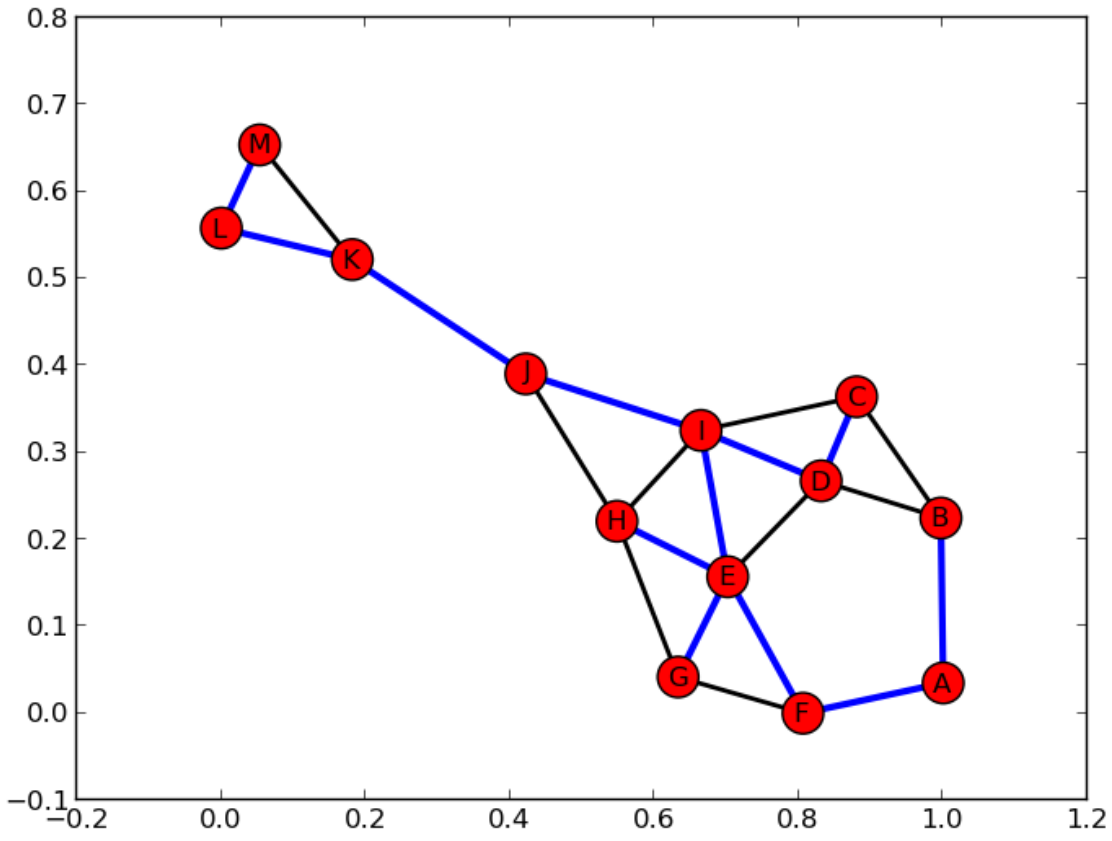
- `E` and `F` from a **Level-1** *fragment*.
- `I` and `J` from a **Level-1** *fragment*.
- Both *level-1 fragments* proceed to absorb every other node in the graph.
- In the end, these 2 fragments merge across `I` and `E` forming a *new **Level-2** fragment* with `I` and `E` as the core nodes.

Graph 2

I added three more nodes: `L`, `K` and `M` to the previous graph for increased complexity and to test other aspects of the GHS algortithm. The added steps that occur after adding these three new nodes are:

- `L`, `K` and `M` form a **Level-1** *fragment* of their own with `L` and `M` as the *core nodes.*
- The `L`, `K` and `M` fragment gets absorbed by the *Level-2* fragment with core nodes `I` and `E`.

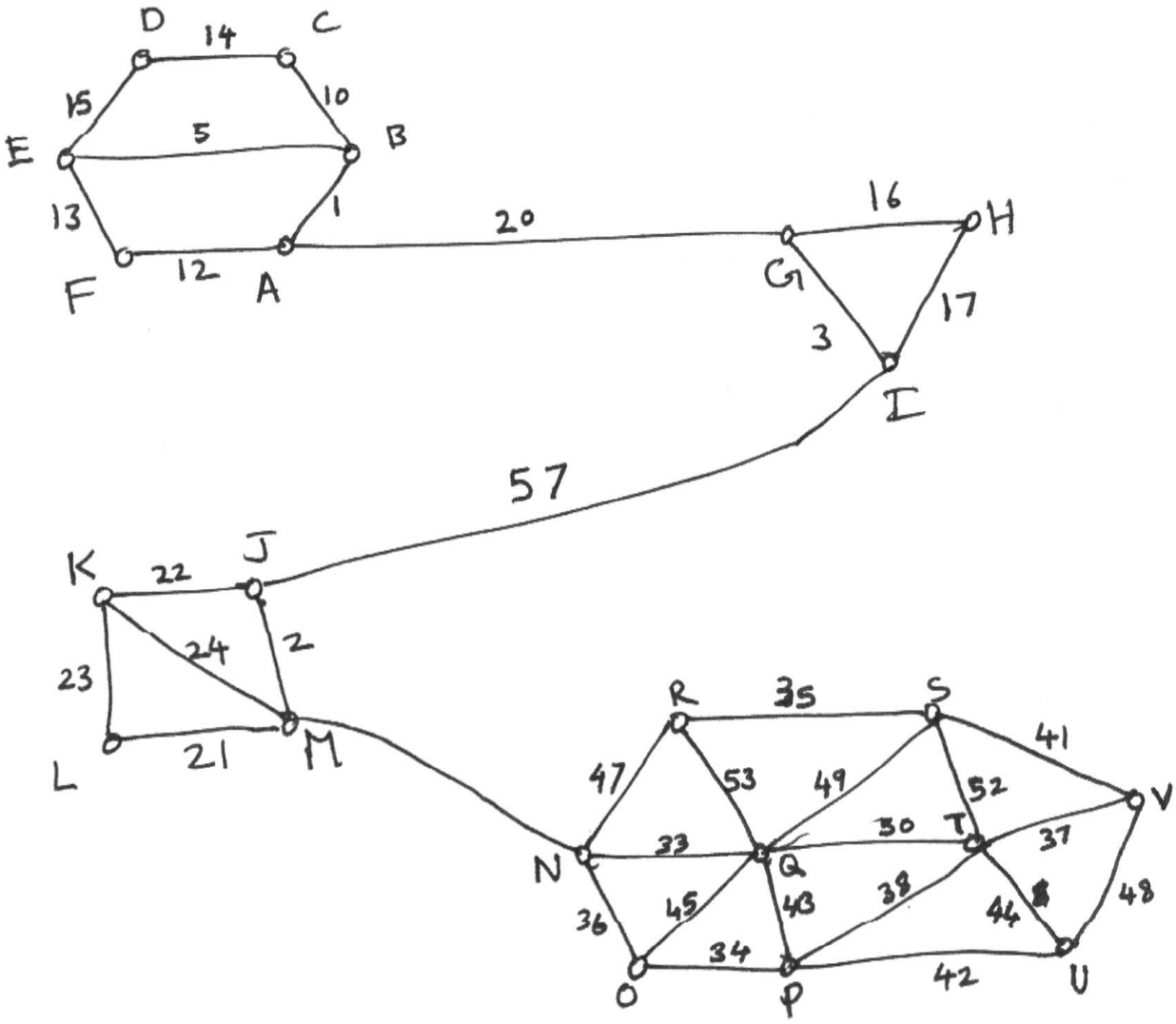
The solution to `graph-2` is depicted below:



As always, *the thick blue edges* denote the **branches** of the MST.

Graph 3

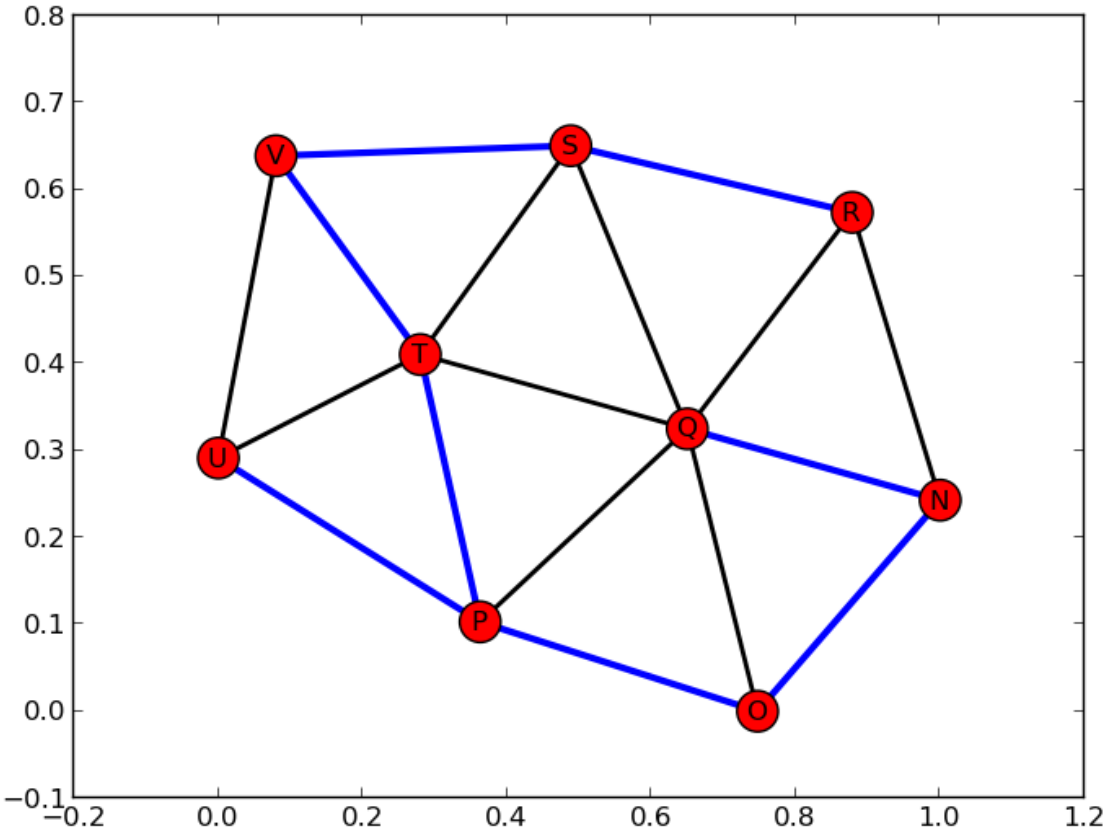
This is a much more complex test case than the previous ones. I drew a sketch of it on paper:



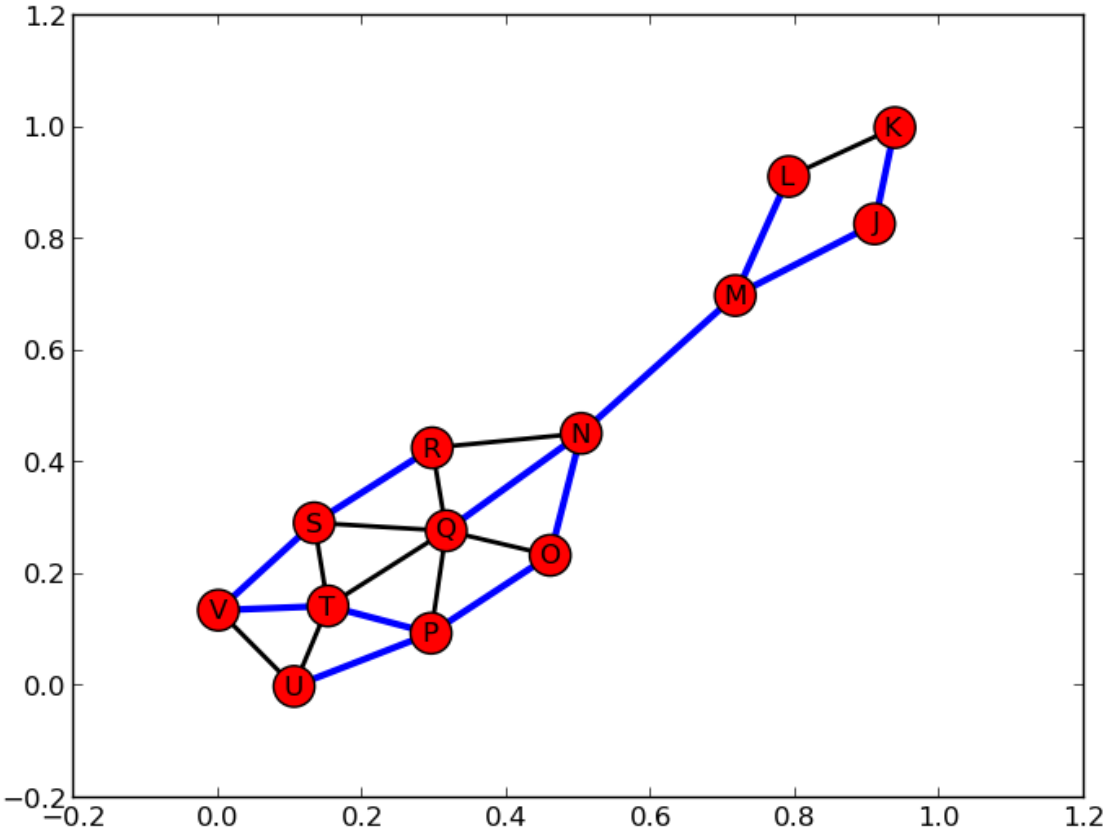
The edge list for it can be found in the file `graph-3`. The output produced by `MST.dis` and `Kruskal.py` can be found in `graph-3-ouptput.txt` (in this directory.)

While writing the graph file `graph-3`, I ran the MST solver several times and obtained these intermediate results:

Intermediate Result 1

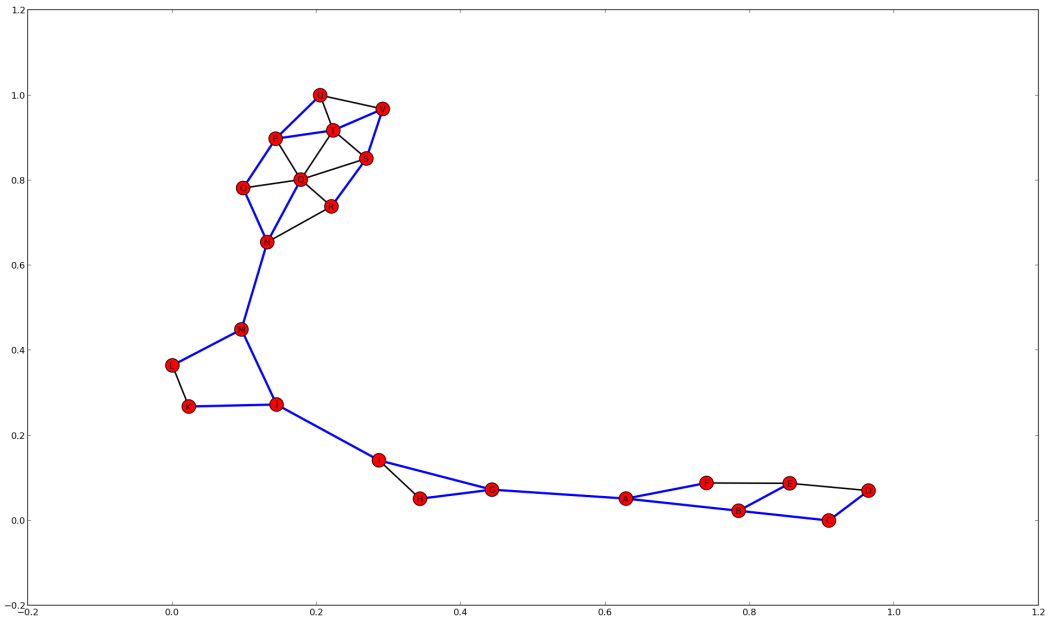


Intermediate Result 2



Final Solution

Click to enlarge:

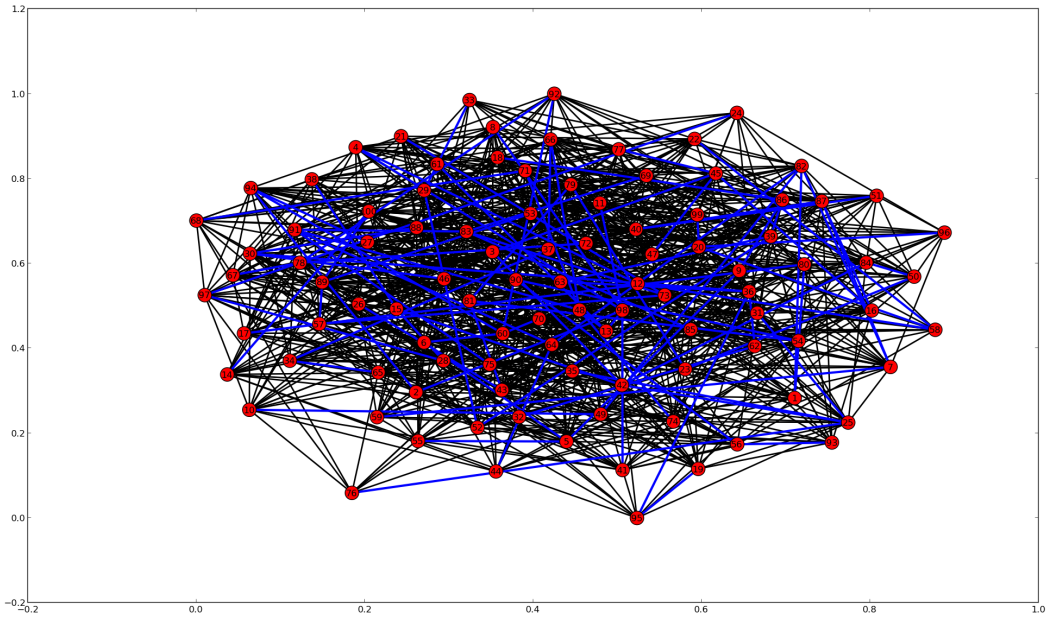


Once again, the *blue edges* denote the MST.

This third test case *tests merges between **Level-2** fragments*. The final fragment formed at the end of this test case is a **Level-3 fragment**. The `graph-3-ouptput.txt` file in this directory shows the output generated by `MST.dis` while solving this test case.

Random Graph with 100 nodes and 1000 edges

This random graph was generated using the random graph generator (`random_graph.py`) in the parent directory. A copy of the random graph used for this test case is in the file `1000edge-100node-graph` in this directory. The solution diagram, while not helpful per-se, is still shown below:



The output generated by the distributed MST while solving this problem is enclosed in the file `1000edge-100node-graph-output.txt`.