CS14 Winter 2016 Lab 5
Minimum Spanning Tree (MST) Algorithm for Undirected Graphs

February 25, 2016

PART 1 DUE: Submit by iLearn by 11:59pm Wednesday, November 25, 2015
LATE DEADLINE (50% penalty) 11:59pm **Black Friday**, November 27, 2015

PART 2 DUE: Submit by iLearn by 11:59pm Wednesday, December 2, 2015
LATE DEADLINE (50% penalty) 11:59pm Thursday, December 3, 2015

1 Introduction

In this assigment, you will be implementing a solution to the minimum spanning tree (MST) problem on an undirected graph, G, with $|V|$ vertices in $V = \{0, 1, 2, \ldots\}$, $|E|$ edges in $E = \{a, b, c, \ldots\}$, and each edge $e \in E$ has a non-negative "weight", $W(e)$, representing the cost of including that edge in the final MST.

The textbook includes detailed descriptions for two different MST algorithms:

- Prim's algorithm (section 9.5.1) chooses a *root* vertex and expands a *single* minimum-weight *fragment* outwards from the root, by adding its *minimum-weight outgoing edge* (MOE), repeatedly, until the fragment has expanded to include every vertex.
- Kruskal's algorithm (section 9.5.2) starts from *every* vertex being the root of its own fragment, then repeatedly finds the *globally-smallest MOE* among the current set of fragments, and adds this edge to the MST until all the fragments have merged into one complete MST.

Obviously, you will be required to implement something different! *Yet Another MST Algorithm* (YAMA), is a greatly-simplified version of the distributed MST algorithms that were an important research topic more than a decade ago.[1] Just like Prim's and Kruskal's algorithms, YAMA is a "greedy" algorithm which uses a sequence of local decisions to add new edges to the final solution, one at a time, and never changes its mind to reject a previously-chosen edge based on what it learns about the problem in the future. Here is a basic summary of how YAMA works:

- Like Kruskal's algorithm, YAMA starts with every vertex being the root of its own fragment.
- Like Prim's algorithm, a fragment always knows its current MOE.
- Each *iteration* of YAMA consists of the following sequence of steps:
  1. YAMA chooses one of the fragments as the *leader* for this iteration and tells it to add its current MOE to the MST.
  2. By adding its MOE to the MST, the *leader* fragment establishes a new connection to a *follower* fragment. These two fragments are now merged together, as described below in sections 2.2–2.4.
  3. The merger at step 2 reduces the number of fragments by one, and increases the number of edges added to the MST by one. If there is only one fragment remaining (or equivalently, the number of edges added to the MST is now $|V| - 1$), then the algorithm stops.

[1] Look at *this paper* if you wish to see more about distributed MST algorithms. However, reading the paper is *not* required, and doing so will *not* make it easier for you to solve this assignment!

It is interesting to note that YAMA includes both existing MST algorithms as special cases of the selection rule at step 1:

- If YAMA always chooses the *same* fragment (the one that contains a specific "root" vertex), then it acts like Prim's algorithm.
- If YAMA always chooses a fragment that currently has the *globally-smallest MOE*, then it acts like Kruskal's algorithm.

In the case of a "real" distributed MST algorithm, the selection at step 1 is carried out randomly, and in parallel — which makes the algorithm faster but makes it difficult for a fragment to determine its own MOE while its connections to other fragments are changing! To control the chaos, distributed MST algorithms often include some features that make it easier/faster for smaller fragments to make connections than larger fragments. We will add a similar feature to YAMA by maintaining a FIFO queue of the fragments for making the selection at step 1. (More on this below in the next section.)

2 Implementation Details

Your program will be veritable "Swiss Army Knife" of data structures and algorithms, including a vector of graph edges, a vector-of-vector of adjacent vertices, a disjoint set representation for the union/find algorithm, a vector of priority queues, and a single FIFO queue.

2.1 Adjacency list representation of G

For each vertex in V you must store its name and a vector of adjacent edge numbers. Note that many references (including Prof. Shelton's lecture slides) use a vector of adjacent *vertices* (not edges), but trust me, a vector of edges will make your life much easier! For each edge in E you must store its name, its weight, and the vertex numbers for its two endpoints. I recommend creating a vector class for each object type, although separate vectors for each attribute will also work.

2.2 Fragment membership

Follow the suggested method from the textbook (and lecture slides) for implementing Kruskal's algorithm by using the union/find algorithm described in Chapter 8 to associate each vertex with the fragment that it currently belongs to. You can find a good C++ implementation of this algorithm here at *github.com*

It is important to note that the *same integers* from 0 to $|V| - 1$ can be used *interchangeably* to identify individual vertices, fragments, and the subset IDs maintained by the union/find algorithm. This is a lucky break, which can save a lot of programming effort later on as we explain the details of the algorithm. More specifically our algorithm will take advantage of the following result:

> **Observation 1:** Whenever subset ID x is still being used, then for vertex x the query $find(x)$ must return subset ID x.

This result is easy to prove by induction, assuming (i) we initialize the union/find algorithm by assigning each vertex v to subset ID v, and (ii) whenever we execute $merge(u, v)$ the new ID assigned to the merged subset must be one of the two previous values, $find(u)$ or $find(v)$. The design of the union/find algorithm guarantees that condition (ii) will always be true. However, the choice of which of those two subset IDs will be applied to the merged subset is chosen by the algorithm to maximize its own internal efficiency, and is *not* something we can control. Thus we have:

> **Observation 2:** When two fragments are merged at step 2, we cannot predict whether the "surviving" subset ID came from the leader or follower. Instead, we must apply the test from Observation 1 to identify the survivor.

2.3 MOE finding using priority queues

At all times, YAMA requires each remaining fragment to keep track of its MOE. Your program will accomplish this by maintaining separate priority queues for every surviving fragment, each filled with a list of candiate edges in smallest-weight first order. Initially, each vertex v is the root (and only element) of a separate fragment, so the priority queue for fragment v will be initialized by adding each edge adjacent to vertex v.

Whenever two fragments are merged at step 2, their respective priority queues must be combined. Using Observations 1 and 2, you can determine which fragment identifier "survived" the merge and which one "died". Take all the elements from the "dead" queue and insert them into the "surviving" queue — taking care to discard any of these edges for which both endpoints now belong to the same fragment. Note that this *edge cleanup* test only applies to edges being transferred from a "dead" to a "surviving" priority queue; no such edge cleanup is applied to the edges already in the "surviving" priority queue.

2.4 FIFO queue for choosing fragments at step 1

Use the STL queue to maintain a FIFO queue of fragments, to be chosen at step 1. In this case, the data type for the FIFO queue is simply an integer, which represents the subset ID for some fragment.

The motivation for this FIFO-queue application is to provide YAMA with a mechanism for scheduling the mergers in "round-robin" order. This is not quite the same as "smallest fragment first", but it does guarantee that every surviving fragment has its nth turn at leading a merger before any fragment has an $(n+1)$st turn.

- **Initially** all the integers between 0 and $|V|-1$ are pushed onto the FIFO queue, since each vertex is a distinct fragment.
- **At the beginning of each iteration** YAMA uses the FIFO queue to choose one fragment p to act as the leader. Unfortunately, the process isn't quite as easy as just popping the front element from the FIFO queue. This is because p may have participated as the follower in a merge led by another fragment, q say, while p was waiting its turn in the queue. Since only one of the two participating fragments survives each merger, fragment p may now be "dead" with the contents of its priority queue already transferred to q. Thus, to avoid the "dead" fragment problem, YAMA step 1 consists of loop with Observation 1 as its exit condition:

  while ($find(p) \neq p$) discard p and replace it by the front of queue

- **At the end of each iteration** the contents of the FIFO queue may be updated, depending on which of the two subset IDs survived the merger. Let us continue with the notation that fragment p was the leader for this iteration.
  - If $find(p) == p$ at the end of the iteration, then the lead fragment p survived the merger, so we push its ID p onto the back of the FIFO queue.
  - Conversely if $find(p) \neq p$ at the end of the iteration, then fragment p must have "died" during the merger so *we do not update the FIFO queue.*

These three update rules give us the following invariant condition on the contents of the FIFO queue:

> **Observation 3:** At the start of every iteration, the FIFO queue contains the subset IDs for every "surviving" fragment, and possibly some additional subset IDs that belong to "dead" fragments.

Once again, this result can easily be proven by induction.

- By construction, it was true initially, when the FIFO queue is first created.
- As we progress through each interation, discarding "dead" fragments during the selection process has no effect on the invariant condition.
- Choosing a fragment p as the leader for this iteration does remove one "surviving" fragment from the FIFO queue, but this happens during the middle of an iteration where the invariant does not apply.

- If leader p does not survive its merger with follower q, then the invariant condition is immediately true, and we are free to discard p without changing the invariant. (Note that follower q must survive if p doesn't, but q is already in the FIFO queue by the inductive assumption.)

- If leader p does survive its merger with follower q, then it will be pushed back into the FIFO queue at the end of the iteration, restoring the invariant condition. (In this case, follower q, which is still in the FIFO queue by the inductive assumption, has been converted from "surviving" to "dead" during this iteration, so we don't care whether it is in the FIFO queue.)

3 A Detailed Example

In this section, we show the step-by-step trace of how YAMA finds the MST for *a sample graph with 7 vertices and 9 edges*.

Figure (a) shows the initial graph. Each vertex has a different color, to show that it belongs to a different fragment (i.e., red for vertex/fragment 0, green for vertex/fragment 1, etc.), and each edge is shown in black to indicate that it has not yet been added to any fragment. Each fragment has an associated priority queue, which includes a list of outgoing edges ordered by weight. The edges currently stored in each priority queue are marked by a star whose color matches the color of its associated fragment. For example, edges a, b, c are all marked with red stars because they are adjacent to (red) vertex 0, while edges a, d are marked with green stars because they are adjacent to (green) vertex 1.

Figure (b) shows how the graph has been updated after completing the first iteration. Since we initialized the FIFO queue to the sequence $(0, 1, 2, 3, 4, 5, 6)$, YAMA will choose fragment 0 (currently just vertex 0) to lead the first iteration. In this case, lead fragment 0 chose edge e as its MOE because e has the lowest weight among the three edges a, b, c in priority queue 0 (labeled with red stars in Figure (a)). Since edge e is adjacent to vertex 0 (in fragment 0) and vertex 2 (in fragment 2), 2 will be the follower fragment for this iteration. First, leader 0 adds edge e to the MST (as indicated by its color change from black to red). Then leader 1 carries out a merger with follower 2. In this example, we assume that subset ID 0 survives the execution of $merge(0, 2)$, so all the resources from "dead" fragment 2 must be transferred to "surviving" fragment 0 by:

- changing the color of all vertices and edges belong to "dead" fragment 2 from blue to red, indicating their transfer to fragment 0;

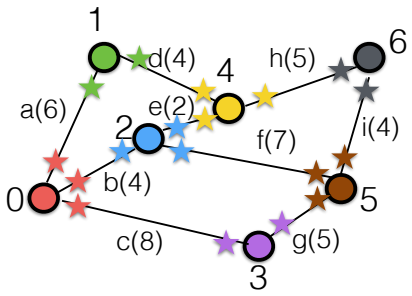
- changing the color of vertex label 2 from black to red, indicating that subset ID 2 is no longer valid, and from now on $find(2)$ returns 0 (red); and

- transferring the contents of priority queue 2 to priority queue 0, indicated by the color change from blue to red for the stars on edges e and f.

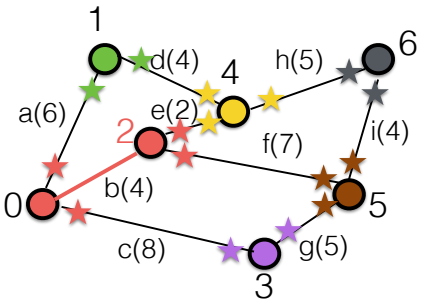
Notice that the stars on both ends of edge b have disappeared: the red star was lost when b was removed from priority queue 0 to become its MOE, and the blue star was lost because b was dropped by the *edge cleanup test* during the priority queue transfer. Since we assume that lead fragment 0 survived the merger, at the end of this iteration the updated FIFO queue will contain the sequence $(1, 2, 3, 4, 5, 6, 0)$.

Figure (c) shows how the graph has been updated after completing the second iteration, where lead fragment 1 chooses edge d as its MOE because d has the lowest weight among the two edges a, d in priority queue 1 (labeled with green stars in Figure (b)). Leader 1 adds edge d to the MST, then carries out a merger with follower 4. In this example, we assume that subset ID 1 survives the execution of $merge(1, 4)$ so all the resources from "dead" fragment 4 are transferred to "surviving" fragment 1, and at the end of this iteration the updated FIFO queue will contain the sequence $(2, 3, 4, 5, 6, 0, 1)$.

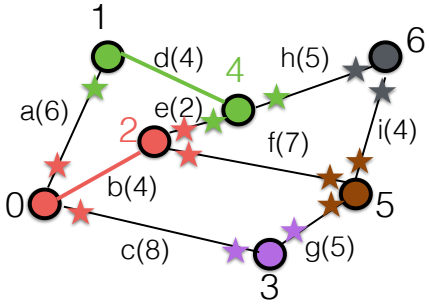
Figure (d) shows how the graph has been updated after completing the third iteration. Notice that 2 was at the front of the FIFO queue at the end of the second iteration, but must be rejected at step 1 because $find(2)$ returns 0. Thus the next lead fragment must be 3, which chooses edge g as its MOE because g has the lowest weight among the two edges c, g in priority queue 3 (labeled with purple stars in Figure (c)). Leader 3 adds edge g to the MST, then carries out a merger with follower 5. In this example, we assume that subset ID 3 survives the execution of $merge(3, 5)$ so all the resources from "dead" fragment 5 are transferred



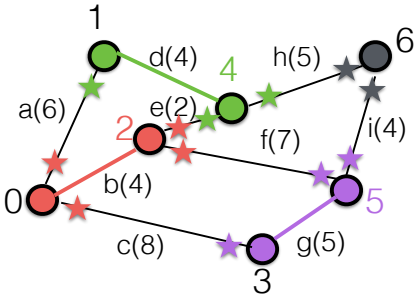
(a) Initial graph.



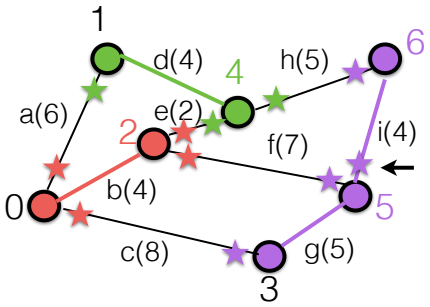
(b) After iteration 1: 0 adds edge b to MST.



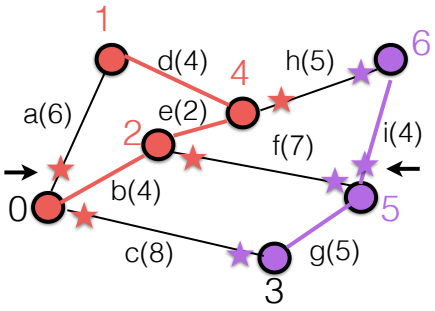
(c) After iteration 2: 1 adds edge d to MST.



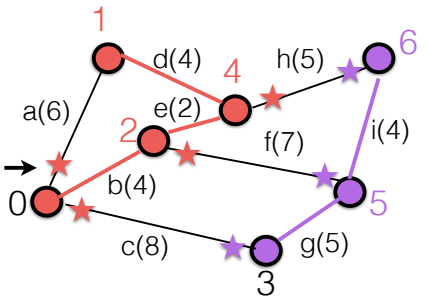
(d) After iteration 3: 3 adds edge g to MST.



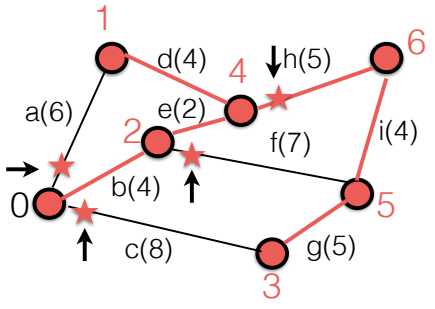
(e) After iteration 4: 6 adds edge i to MST.



(f) After iteration 5: 0 adds edge e to MST.



(g) Iteration 6: 3 rejects edge i.



(h) After iteration 6: 3 adds edge h to MST.

to "surviving" fragment 3, and at the end of this iteration the updated FIFO queue will contain the sequence $(4, 5, 6, 0, 1, 3)$.

Figure (e) shows how the graph has been updated after completing the fourth iteration. This time both 4 and 5 were at the front of the FIFO queue, but they were rejected at step 1 because $find(4)$ returned 1 and $find(5)$ returned 3. Thus 6 became the next lead fragment and chose edge i as its MOE because g has the lowest weight among the two edges h, i in priority queue 6 (labeled with grey stars in Figure (d)). Leader 6 adds edge i to the MST, then carries out a merger with follower 3. However, this time we assume that subset ID 3 survives the execution of $merge(6, 3)$ because it is the larger subset. As before, all the resources from "dead" fragment 6 are transferred to "surviving" fragment 3. However, notice that 3's copy of edge e remains in priority queue 3 (highlighted by the black arrow) even though it *stopped being an outgoing edge* for fragment 3 during this iteration. This happened because fragment 3 played the role of a *survivor* and hence did not have an opportunity to clean up this obsolete edge during the transfer of its priority queue to the other fragment. At the end of this iteration the updated FIFO queue will contain the sequence $(0, 1, 3)$.

Figure (f) shows how the graph has been updated after completing the fifth iteration, where lead fragment 0 chooses edge e as its MOE because e has the lowest weight among the four edges a, c, e, f in priority queue 0 (labeled with red stars in Figure (e)). Leader 0 adds edge e to the MST, then carries out a merger with follower 1. In this example, we assume that subset ID 0 survives the execution of $merge(0, 1)$ so all the resources from "dead" fragment 1 are transferred to "surviving" fragment 0. Similar to the situation in Figure (e), notice that 0's copy of edge a remains in priority queue 0 (highlighted by the black arrow) even though it stopped being an outgoing edge for fragment 0 during this iteration. At the end of this iteration the updated FIFO queue will contain the sequence $(1, 3, 0)$.

Figure (g) shows an intermediate result during the middle of the sixth iteration of the algorithm. First, we rejected fragment 1 from the front of the FIFO queue at step 1 because $find(1)$ returned 0. Next we selected 3 as the lead fragment because $find(3)$ returned 3. As usual, fragment 3 first chooses edge i as its MOE because it has the minimum weight among the four edges f, g, h, i in priority queue 3 (labeled by purple stars in Figure (f)). However, fragment 3 rejects i because it is not an outgoing edge, and reduces the contents of priority queue 1 to the three edges f, g, h (labeled by purple stars in Figure (g)).

Figure (h) shows the final graph after fragment 3 completes the sixth and final iteration of the algorithm. Leader 3 adds edge h to the MST, then carries out a merger with follower 0. In this example, we assume that subset ID 0 survives the execution of $merge(3, 0)$ because it is the larger fragment, and all the resources from "dead" fragment 3 are transferred to "surviving" fragment 0. Notice that priority queue 0 now contains four edges a, c, f, h — none of which are outgoing — and the updated FIFO queue contains the single value (0) even though we have already found the complete MST.

4 Program Input and Output

Your program should read the name of an input file from the command line. The input file has the following format.

Line 1: two integers, representing the number of vertices, $|V|$, and number of edges, $|E|$, in G.

The next $|V|$ lines, one line per vertex: the name of this vertex as a character string without whitespace or surrounding delimiters, such as: `Los_Angles` or `7`.

The next $|E|$ lines, one line per edge: the name of this edge as a character string without whitespace or surrounding delimiters, a single blank, a number representing the edge weight, a single blank, the name of one connected vertex as a character string without whitespace or surrounding delimiters, a single blank, the name of the other connected vertex as a character string without whitespace or surrounding delimiters.

During execution, the program generates *one* similarly-named output file with the original filename extension (if any) changed to "`.out`". For example, if the input file were `graph_ex.txt`, then the output file would be `graph_ex.out`.

The output file is a normal text file containing the following information.

Line 1: the string `Weight of MST:` followed by a single blank, then a number representing the sum of all edge weights included in the minimum spanning tree for this test case as determined by your program.

Line 2: the string `Edges in the MST:` followed by a single blank, then a list of edge names separated by blanks.

5 Simplifying the Problem into Partial Stages

It is important for you to turn in a working program, even if your program only handles a subset of the requirements for the assignment. Thus, I have identified a series of partial stages to help guide your progress towards the goal of completing the assignment.

5.1 Nearly Useless Tree Solver (NUTs)

- Generates: A spanning tree (almost certainly NOT minimum weight).
- Requires: Reading the input file to create the vector of graph edges and vector of edges with weights and adjacent vertices. Implement union/find algorithm.
- Does NOT require: priority queues, FIFO queue.

Initialize the union/find algorithm with the vertex numbers.

Iterate (once) through the vector of edges. For each edge e with adjacent vertices u and v: if $(find(u) \neq find(v))$ then add e to the NUT and execute $merge(u, v)$; otherwise do nothing. Stop when the NUT contains $|V| - 1$ edges.

You might find it interesting to run repeat the algorithm, but run through the edges in reverse order. Any edge that appears in both NUTs must be included in every spanning tree.

5.2 Kruskal's Algorithm

- Generates: A minimum spanning tree.
- Requires: Reading the input file to create the vector of graph edges and vector of edges with weights and adjacent vertices. Implement union/find algorithm. Maintain one priorty queue.
- Does NOT require: FIFO queue.

Same algorithm as NUTs, except for the following change. Create a single global priority queue for edges, sorted into increasing edge weight, and initialize it by inserting every edge from the entire graph.

For each iteration, use edge e from the front of the priority queue, instead of the next element from the edge vector.

5.3 Even's Algorithm

- Generates: A minimum spanning tree.
- Requires: Reading the input file to create the vector of graph edges and vector of edges with weights and adjacent vertices. Implement union/find algorithm. Maintain a vector of priorty queues.
- Does NOT require: FIFO queue.

Same algorithm as Kruskals algorithm, except for the following changes. Create a vector of priority queues for edges (one for each vertex), and initialize the priority queue for vertex v by inserting every edge adjacent to v. Choose a *root vertex*, say r.

For each iteration, always choose edge e from the front of priority queue r, instead of the global priority queue. For each edge e with adjacent vertices u and v: if $(find(u) \neq find(v))$ then

- add e to the MST
- execute $merge(u, v)$
- if $(find(u) \neq find(r))$ transfer contents of priority queue u to priority queue r; else transfer the contents of priority queue v to priority queue r

Note that the priority queue merger condition is slightly more complicated than YAMA because Even's root ID r might not "survive" the union/find mergers.