CS 130B
DATA STRUCTURES AND ALGORITHMS
Programming Assignment
INDIVIDUAL WORK ON THIS ASSIGNMENT IS MANDATORY
Due Date: Tuesday May 31 at 8:00 AM
Remember: Points will be deducted for homework turned in after the Due Date.
Deadline: Wednesday June 1 at 8:00 AM
Remember: Projects will not be accepted after the Deadline
Preliminary Version

1 Introduction

Remember that you must work on this assignment INDIVIDUALLY.

For this assignment you will code in C++ several algorithms to provide several different solutions to the problem of finding *minimum-max weight (cost) spanning trees on k-weighted connected graphs MMWST*. Given a k-weighted connected graph G, consisting of a set of n vertices and a set of m edges each with a k-vector of (positive integer) weights (i.e., a vector with k weights). The problem is to find a minimum-max spanning tree T for G. I.e., a spanning tree for G such that $max\{W(0), W(1), \dots, W(k-1)\}$ is least possible amongst all possible spanning trees. For $0 \leq i < k$, $W(i) = \sum_{e \in T} w_e(i)$ and $w_e(i)$ is the i^{th} weight[1] of edge e. In Figure 1 you will find a 2-weighted graph H and two different minimum-max spanning trees for H. Note that if we take out the edge $\{4,5\}$ with weight $(2,6)$ from the minimum-max spanning tree in Figure 1(B) and add the edge $\{2,5)\}$ with weight $(3,6)$ one will end up with a spanning tree that it is not a minimum-max spanning tree for H.

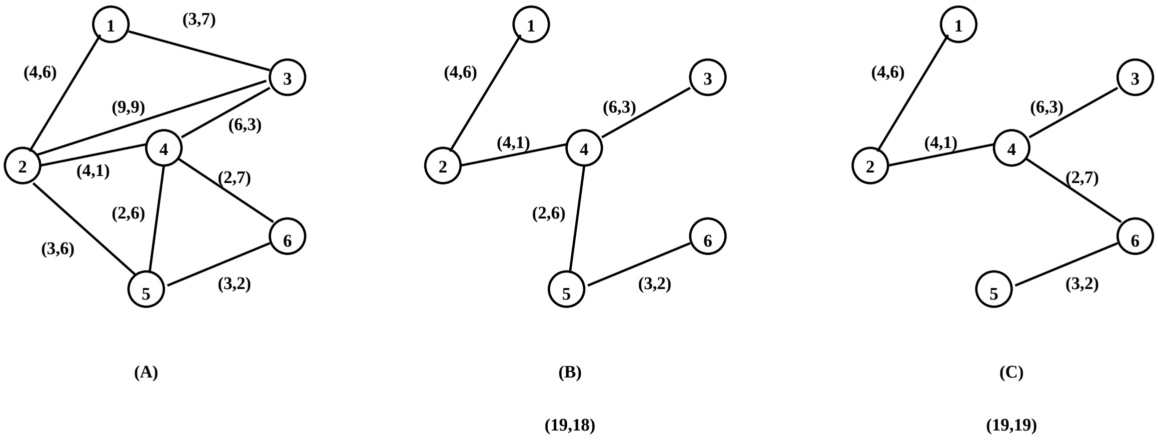

Figure 1: (A) 2-weighted graph H. (B) Set of edges S_1 that forms a minimum-max spanning tree for H with $W_{S_1} = (19,18)$ (C) Set of edges S_2 that forms a minimum-max spanning tree for H with $W_{S_2} = (19,19)$

The problem of finding a minimum-max spanning tree for a k-weighted graph is a computationally intractable problem (i.e. all know algorithms for its solution take exponential time with respect to the number of vertices in the graph) even for $k = 2$. In class we discussed several algorithms (Kruskal, Prim and Sollin) that construct optimal solutions when $k = 1$. The purpose of this assignment is to implement

[1] 0^{th} is the first weight of the vector, 1^{th} is the second weight, and so on.

several *variations* of Kruskal, Prim and Sollin algorithms to generate sub-optimal solutions to this more general problem. *The goal is for you to practice C++ and learn more about it, rather than to make you an expert in C++, and to evaluate experimentally the performance of several algorithms.*

A k-weighted graph G consists of n vertices and m edges. Each edge e in G will have k weights represented by $w_e(0), w_e(1), \ldots, w_e(k-1)$. For the graph in Figure 1(A) n is 6, m is 8 and k is 2. The weights of the edges are: edge $(1,2)$ has weight $(4,6)$, edge $(1,3)$ has weight $(3,7)$, edge $(4,5)$ has weight $(2,6)$, and so on.

The spanning tree S_1 given in Figure 1(B) has $(W(0), W(1))$ equal to (19,18) and S_2 in Figure 1(C) has $(W(0), W(1))$ equal to (19,19). Both have the objective function value of 19 as $max\{19,18\} = max\{19,19\}$ and one can show that both of these two trees are minimum-max spanning trees for H.

2 Notation and Definitions

Definition (min, max, lexicographic order): For any edge $e = \{i,j\}$ let $min(e)$ be the smallest of i and j, and $max(e)$ be the largest of i and j. Let e and f be two different edges in a graph. Edge $e = \{i,j\}$ is said to be in *lexicographic order* before edge $f = \{p,q\}$ iff $min(e) < min(f)$, or if $min(e) = min(f)$ then $max(e) < max(f)$.

Example: Note that edge $\{7,3\}$ is in lexicographic order before edge $\{4,5\}$. Also, edge $\{3,2\}$ is in lexicographic order before edge $\{2,4\}$.

Definition (Forest of Trees): A set S of edges is said to be a *forest of trees* if there are no cycles in S.

Example: In addition to the sets S_1 and S_2 defined before, we define the following sets of edges all of which form a forest of trees for the graph given in Figure 1(A). Let $S_3 = \{\{1,2\},\{1,3\},\{4,5\}\}$, $S_4 = \{\}$, $S_5 = \{\{1,2\},\{2,5\},\{4,6\},\{3,4\}\}$, and $S_6 = \{\{1,2\},\{1,3\},\{3,4\},\{4,5\},\{4,6\}\}$.

Definition (k-weighted vector for forest S): Let S be a forest of trees. The vector $W_S = (W_S(0), W_S(1), \ldots, W_S(k-1))$ is called the *k-weighted vector for forest S* iff, $W_S(i) = \sum_{e \in S} w_e(i)$ for $0 \le i < k$.

Definition (Max): $Max(W_S)$ is defined as $max\{W_S(0), W_S(1), \ldots, W_S(k-1)\}$

Example: For the previously defined sets $W_{S_1} = (19,18)$, $Max(W_{S_1}) = 19$, $W_{S_2} = (19,19)$, $Max(W_{S_2}) = 19$, $W_{S_3} = (9,19)$, $Max(W_{S_3}) = 19$, $W_{S_4} = (0,0)$, $Max(W_{S_4}) = 0$, $W_{S_5} = (15,22)$, $Max(W_{S_5}) = 22$, $W_{S_6} = (17,29)$ and $Max(W_{S_6}) = 29$.

Definition (objective function value (ofv)): The *objective function value (ofv)* of forest S is simply $Max(W_S)$ and it is denoted by $ofv(S)$.

Example: Clearly, $ofv(W_{S_1}) = 19$, $ofv(W_{S_2}) = 19$, $ofv(W_{S_3}) = 19$, $ofv(W_{S_4}) = 0$, $ofv(W_{S_5}) = 22$, and $ofv(W_{S_6}) = 29$.

Definition (better ofv, equal ofv): Let S and T be two forests of trees for G. Forest S is said to have a *better ofv* than forest R if $ofv(W_S) < ofv(W_R)$ (i.e. $max(W_S) < max(W_R)$). Forest S is said to have *equal ofv* than forest R if $ofv(W_S) = ofv(W_R)$ (i.e., $max(W_S) = max(W_R)$).

Example: The forest in Figure 1B has ofv equal to the one in Figure 1C. The forest of trees given by S_3 has better ofv than the one for S_4. The ofv of S_3 is better than the one for S_5 which in turn is better than the one for S_6.

Definition (possible addition edge): Let S be a forest of trees for graph G and W_S be the k-weighted vector for forest S. We say that edge $e \in G$ is a *possible addition* to S if adding e to S does not form a cycle, i.e., $S \cup \{e\}$ does not have a cycle.

Example: Some possible addition edges for S_3 (with respect of the graph in Figure 1(A) are edges $\{2,3\}$, $\{2,4\}$, $\{2,5\}$, $\{4,6\}$, and $\{3,6\}$, but edge $\{2,3\}$ is NOT a possible addition edge. For set S_4 any of the edges in the graph (Figure 1(A)) are possible additions.

Definition (better possible addition): Let S be a forest of trees for graph G and W_S be the k-weighted vector for forest S. Let e and f be two edges in G each of which are possible additions to S. Let $S(e)$ be $S \cup \{e\}$ and $S(f)$ be $S \cup \{f\}$. Then edge e is said to be a *better possible addition* to S than edge f if either $S(e)$ is better than $S(f)$, or $S(e)$ is equal to $S(f)$ but edge e is in lexicographic order before edge f.

Example: For set S_3 edge $\{2,4\}$ is a better possible addition than edge $\{3,4\}$. Also, for set S_3 edge $\{3,4\}$ is a better possible addition than edge $\{4,6\}$.

Definition (best possible addition): Let S be a forest of trees for graph G and W_S be the k-weighted vector for forest S. The possible addition edge e is the best possible addition for S if for every possible addition edge f in S, edge e is a better possible addition than f.

Example: For set S_3 edge $\{2,4\}$ is the best possible addition for S_3.

Definition (cycle forming edge, cycle breaking edge): Let S be a forest of trees for graph G and W_S be the k-weighted vector for forest S. Edge $e \in G$ is a *cycle forming edge* for S iff $S \cup \{e\}$ has once cycle. An edge f in S is said to be a *cycle breaking* edge in $S \cup \{e\}$ iff deleting edge f from $S \cup \{e\}$ results in a forest of trees. The resulting forest is $S \cup \{e\} \setminus \{f\}$

Example: Edge $e = \{1,3\}$ is a cycle forming edge for S_1 and S_2, but not for S_3. For S_6, edges $\{2,3\}$, $\{2,4\}$, $\{2,5\}$, and $\{5,6\}$ are cycle forming edges.

Definition (valid replacement edge): A *valid replacement edge* for the cycle forming edge $e \in G$ for the forest of trees S for graph G is the cycle breaking edge f such that $ofv(S \cup \{e\} \setminus \{f\}) < ofv(S)$. If no such edge exists, then thee is no valid replacement edge for cycle forming edge e.

Example: For none of the cycle forming edges in $S_1, S_2, \ldots, S_5$ there is a valid replacement edge. For S_6 the cycle forming edge $\{2,4\}$ has the following valid replacement edges: $\{1,2\}$, $\{1,3\}$, and $\{3,4\}$. But cycle forming edge $\{2,5\}$s in S_6 does not have any valid replacement edges.

Definition (best possible replacement edge): The *best possible replacement edge* for the cycle forming edge $e \in G$ for the forest of trees S for graph G is a valid replacement edge f such that every other valid replacement edge f' for e either $ofv(S \cup \{e\} \setminus \{f\}) < ofv(S \cup \{e\} \setminus \{f'\})$ or $ofv(S \cup \{e\} \setminus \{f\}) = ofv(S \cup \{e\} \setminus \{f'\})$ and edge f is in lexicographic order before edge f'.

Example: The best possible replacement edge for cycle forming edge $\{5,6\}$ in S_6 is the replacement edge $\{4,6\}$. The one for cycle forming edge $\{2,4\}$ in S_6 is replacement edge $\{1,3\}$

Definition (best resulting ofv value): The *best resulting ofv value* for the cycle forming edge $e \in G$ is $ofv(S \cup \{e\} \setminus \{f\})$, where edge f is the best possible replacement edge for the cycle forming edge $e \in G$ for the forest of trees S.

Example: The best resulting ofv value is $(18, 24)$ for cycle forming edge $\{5, 6\}$ in S_6, where as it is $(18, 23)$ for cycle forming edge $\{2, 4\}$ in S_6.

Definition (best possible cycle forming edge): The *best possible cycle forming edge* e in S is the cycle forming edge e with best possible replacement edge f such that for every other cycle forming edge e', either the best resulting ofv value for e is less than the one for e', or it is equal, but e is in lexicographic order before edge e'. Remember that the best resulting ofv value for e is $ofv(S \cup \{e\} \setminus \{f\})$ and the one for e' is $ofv(S \cup \{e'\} \setminus \{f'\})$. Example: The best possible cycle forming edge for S_6 is edge $\{2, 4\}$.

3 The Methods

In this section we explain the greedy methods which are variations of the methods we have discussed in class for the case when k is 1.

You need to implement the following methods: Kruskal with Iterative Improvement (KILI), Prim with Iterative Local Improvement (PILI), Sollin with Iterative Local Improvement (SILI), Fixed Start with Iterative Local Improvement (FILI), and Your Own Method (YILI).

3.0.1 Kruskal with Iterative Local Improvement (KILI)

```
global $n, m, k$
global Adjacency List Representation of the input graph $G$
KILI();
  // Find a spanning tree of the n-vertex edge $k$-weighted (undirected) connected graph $G$.
  // $K$: Set of edges in spanning tree to be constructed.
    $K \leftarrow \emptyset$;
    Let $W(0) \leftarrow W(1) \leftarrow \ldots \leftarrow W(k-1) \leftarrow 0$ for set $K$;
    Let $E$ be the set of edges in $G$;
    while $| K | \neq n - 1$ do
      Let $e = \{u, v\} \in E$ be the best possible addition to $K$
        $E \leftarrow E \setminus \{u, v\}$
        $K \leftarrow K \cup \{u, v\}$
        // Local Iterative Improvement
        while there is a cycle forming edge in $G$ for $K$ with a replacement edge do
          Let $e$ be the best possible cycle forming edge for $K$;
          Let $f$ be the best possible replacement edge for $e$;
          $K \leftarrow K \cup \{e\} \setminus \{f\}$
        endwhile
    endwhile
  // K is a spanning tree constructed by KILI.
end
```

3.0.2 Prim with Iterative Local Improvement (PILI)

```
global $n, m, k$
global Adjacency List Representation of the input graph $G$
PILI();
  // Find a spanning tree of the n-vertex edge $k$-weighted (undirected) connected graph $G$.
  // $P$: Set of edges in spanning tree to be constructed.
    $P \leftarrow \emptyset$;
    Let $W(0) \leftarrow W(1) \leftarrow \ldots \leftarrow W(k-1) \leftarrow 0$ for set $P$;
  // Vp: Set of vertices in the tree
  // Remember that the vertices are labeled $1, 2, \ldots, n$
    $V_p \leftarrow \{1\}$
    Let $E$ be the set of edges in $G$;
    while $\mid K \mid \neq n-1$ do
      Let $e = \{u, v\} \in E$ be the best possible addition to $P$ when considering only
        as possible additions those edges with one vertex in $V_p$ and the other one $V \setminus V_p$
        $E \leftarrow E \setminus \{u, v\}$
        $P \leftarrow P \cup \{u, v\}$
        // Local Iterative Improvement
        while there is a cycle forming edge in $G$ for $P$ with a replacement edge do
          Let $e$ be the best possible cycle forming edge for $P$;
          Let $f$ be the best possible replacement edge for $e$;
          $P \leftarrow P \cup \{e\} \setminus \{f\}$
        endwhile
    endwhile
  // P is a spanning tree constructed by PILI.
end
```

3.0.3 Sollin with Iterative Local Improvement (SILI)

```
global $n, m, k$
global Adjacency List Representation of the input graph $G$
SILI();
  // Find a spanning tree of the n-vertex edge $k$-weighted (undirected) connected graph $G$.
  // $S$: Set of edges in spanning tree to be constructed.
    $S \leftarrow \emptyset$;
    Let $W(0) \leftarrow W(1) \leftarrow \ldots \leftarrow W(k-1) \leftarrow 0$ for set $S$;
    Let set $S_i \leftarrow \{i\}$ for $1 \leq i \leq n$
    while $\mid S \mid \neq n-1$ do
      Let $q$ be the number of $S_i$ sets //first iteration $q$ is equal to $n$
      for $i = 1$ to $q$ do
        Let $e_i = \{u_i, v_i\}$ be the best possible addition to $S$ when considering
          only as possible additions those edges with one vertex in $S_i$ and the
          other in $V \setminus S_i$ such that $u_i$ in $Si$ and $v_i$ in $V \setminus Si$
      endfor
      for $i = 1$ to $q$ do
        Add edge $e_i$ to $S$ if it is not in $S$ and it does not create a cycle in $S$
      endfor
      // Local Iterative Improvement
      while there is a cycle forming edge in $G$ for $S$ with a replacement edge do
        Let $e$ be the best possible cycle forming edge for $S$;
        Let $f$ be the best possible replacement edge for $e$;
        $S \leftarrow S \cup \{e\} \setminus \{f\}$
      endwhile
      // Redefinition of sets $S_1$, $S_2$, ...
      Let $S_1$ be the set of all nodes in the tree in $S$ that includes vertex 1
      $i \leftarrow 1$
      while $S_1 \cup S_2 \cup ...S_i$ does not include at least one vertex in $G$ do
        Let $j$ be the smallest integer such that vertex $j$ is not in $S_1 \cup S_2 \cup \ldots \cup S_i - 1$
        $i \leftarrow i + 1$;
        Let $S_i$ be the set of all nodes in the tree in $S$ that includes vertex $j$
      endwhile
      endwhile
  // S is a spanning tree constructed by SILI.
end
```

3.0.4 Fixed Start with Iterative Local Improvement (FILI)

For this method you may assume that the graph will always have the edges $\{1,2\},\{2,3\},\{3,4\},\ldots,\{n-1,n\}$. global n,m,k

```
global Adjacency List Representation of the input graph $G$
FILI();
  // Find a spanning tree of the n-vertex edge $k$-weighted (undirected) connected graph $G$.
  // $F$: Set of edges in spanning tree to be constructed.
    $F \leftarrow \{1,2\},\{2,3\},\{3,4\},\ldots,\{n-1,n\}$;
    Let $W(0) \leftarrow W(1) \leftarrow \ldots \leftarrow W(k-1)$ be the $W$ vector for set $F$;
        // Local Iterative Improvement
        while there is a cycle forming edge in $G$ for $F$ with a replacement edge do
          Let $e$ be the best possible cycle forming edge for $F$;
          Let $f$ be the best possible replacement edge for $e$;
          $F \leftarrow F \cup \{e\} \setminus \{f\}$
        endwhile
  // F is a spanning tree constructed by SILI.
end
```

3.0.5 Your Own Method (YILI)

You come up with your own method. The idea is to use some heuristics or trick that you feel will generate better solution than all the previous methods. Your method should be fast and should not take more than 10% than the slowest of the four methods. If it is slower than that, then there will be a penalty that will add a value o the *ofv* of the tree you generate.

4 Input

The first input line contains five positive integer values with the following meaning:

- *Method*: Integer in (1,2,3,4,5), where

  1: use KILI

  2: use PILI

  3: use SILI

  4: use FILI

  5: use YILI

- n: Number of vertices in the graph (the nodes are labeled $1, 2, \ldots, n$).
- m: Number of edges in the graph.
- k: Number of dimensions for the weight vector.
- h: Type of output (Integer in (1,2)).

The next m input lines define the edges in the graph. The i^{th} line $(1 \leq i \leq n)$ contains the i^{th} edge. Each of these lines consists of $k + 2$ integers, the first two are the vertices the edge joins and the remaining k integer represent the (ordered) vector of weights for the edge. You may assume that edges $\{1, 2\}, \{2, 3\}, \{3, 4\}, \ldots, \{n - 1, n\}$ will always present in the graph. This implies the graph is connected. The edges in the graph are not necessarily ordered in any way. But the two vertices for every input edge $e\{i, j\}$ are such that $i < j$.

You may assume that the input is valid, i.e., there are no errors in the input. Note that all your structures must by allocated "DYNAMICALLY" so that the algorithm would work no matter what the input size is provided there is enough memory space. You may NOT use STL, especially the type `vector`, it is too slow for this application. Read the input with `cin` from the standard input. Sample input and output files will be added to the Web page later on.

5 Internal Representation

You may use any of the code in the text which is available through the link
http://www.cs.ucsb.edu/ teo/progs/ProgramIndex.htm The overall requirement is that all your structures must by allocated "DYNAMICALLY" so that the algorithm works no matter what the input size is provided there is enough space. Do NOT use STL, especially `vector`. Read the input with `cin`. You may use one dimensional arrays. You MUST store the graph by using adjacency lists. Also, use the "standard output" for writing your output and "standard input" for reading. When we test your program we will run your program as "`./executeit` < `input.1` > `output.1`", with several different input files we will be creating. (`executeit` is defined later on)

STOP HERE

6 Output

For all you methods when the Outp value is 1, your output should print all the edges (in lexicographic order) one per line. An edge is printed with two integers representing the two nodes it joins. The firt one should have value smaller than the second one. The at the end you will print the W vector. The k values separated by a blank in one line. The last line should start the the name of the method (KILI, PILI, ..., YILI) followed by a blank and the the objective function value.

When the Outp value is 2, just print the last two lines defined for Outp1.

7 Coding and Experimentation

Once your programs are working correctly, time it with the Linux `time` command (type `man time` for an explanation of the `time` command), and run the program with several INPUT FILES which you can create and YOU CAN SHARE WITH OTHER STUDENTS. When you turnin your program you must also turn in your test files. Try test files with n = 10, 30, 80, 500, 5000 as well as other values. The files may be created at random. Plot the actual running time (user time (the number followed by the letter "u")) for your algorithms for the different input files against the number of nodes and edges for the different methods. Note that there are many different experiments you can design. Just try a set of experiments that will provide some picture that is some sense is a "fair comparison". Try to determine experimentally which of the methods is fastest and compare the solutions generated (with respect to the objective function value) by the different algorithms. by the different methods. Make some recommendations as to when it would be more appropriate to use these different methods. Note that this is a general recommendation based on your experimentation. There are many ways one can make the comparisons. Choose any way and apply it consistently. The program must be in C++, you MUST use `makefiles`, and you MUST turn in electronically your code. The make file should compile your code and leave it in the file called `executeit`. Note that your code does not have to be C++ object oriented masterpiece. To grade it, we will save it in a directory and type "`make`", followed by "`./executeit < data.1 > output.1`", "`./executeit < data.2 > output.2` ", ..., where "data.1", "data.2", ... are files we will be creating, and the answers you program computes is stored in output.1, output.2, etc. After everyone turns in their project we will make these files available to you. Your program MUST work in the CSIL PCs running Linux. Note that your grade depends on how well you program works on the examples we will use to test your program. Some of these examples will have quite a few points.

Implement your procedures as efficient (fast) as possible, but don't go too far on this. Try to use a small amount of additional space, but do not try to reduce it to to the absolute minimum. You may NOT use any two dimensional arrays of size $\Omega(n)$ by $\Omega(n)$.

Partial credit will be given if not all of the code is working correctly. The partial credit will depend on how well your code works on the test files. You will be deducted points if your program takes too long. YOU MUST WORK ON THIS PROJECT INDIVIDUALLY.

8 TURNIN

Read the Web page for cs130b for instructions on how to turnin electronically your code.

GOOD LUCK