CSIT 5500 Advanced Algorithm HW3

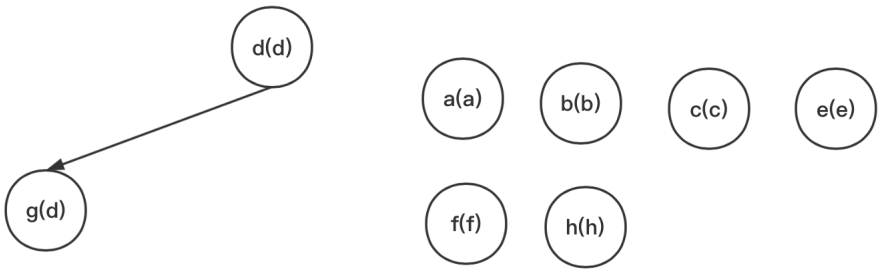
LIN Jialiang

Q1:

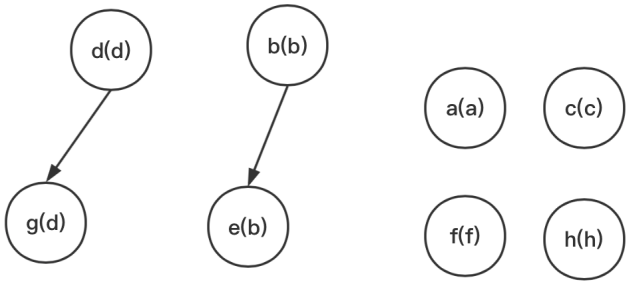
(a) (d,g) -> (b,e) -> (a,b) -> (a,e) -> (d,f) -> (f,h) -> (g,h) -> (c,b) -> (c,f) -> (e,g)

(b)

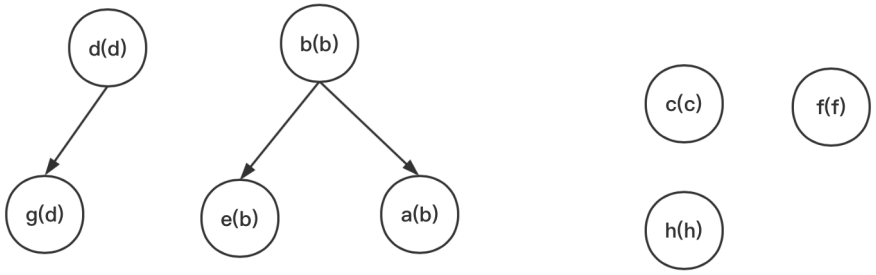
Process (d,g), the forest:



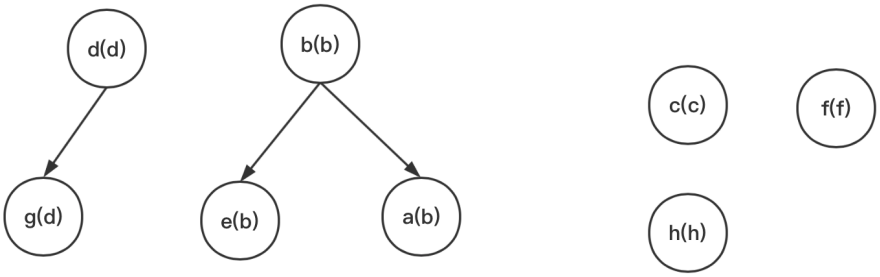
Process (b,e), the forest:



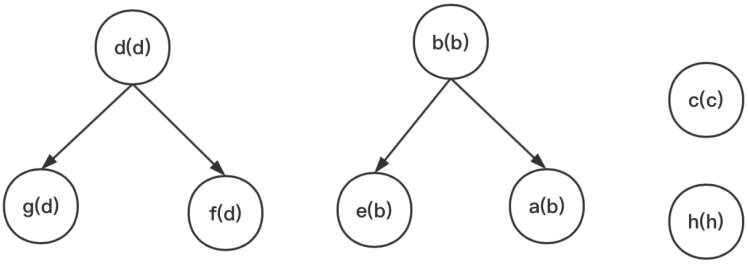
Process (a,b), the forest:



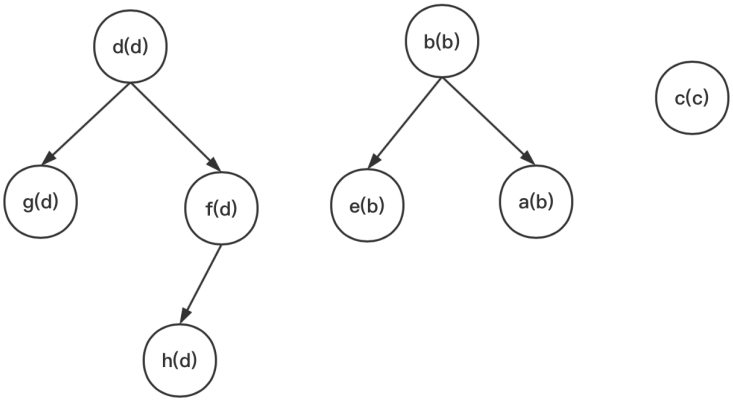
Process (a,e), the forest:



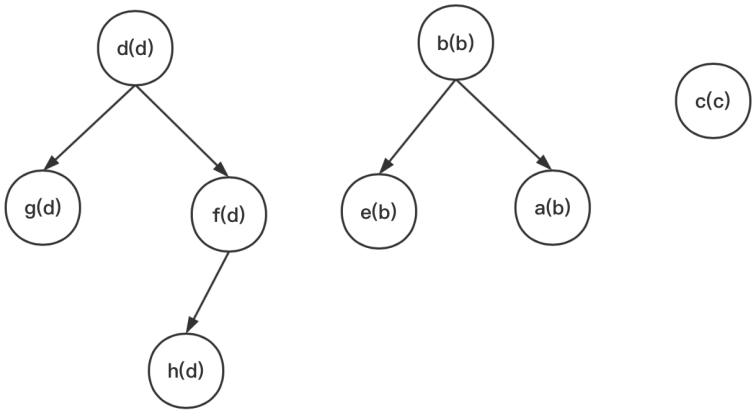
Process (d,f), the forest:



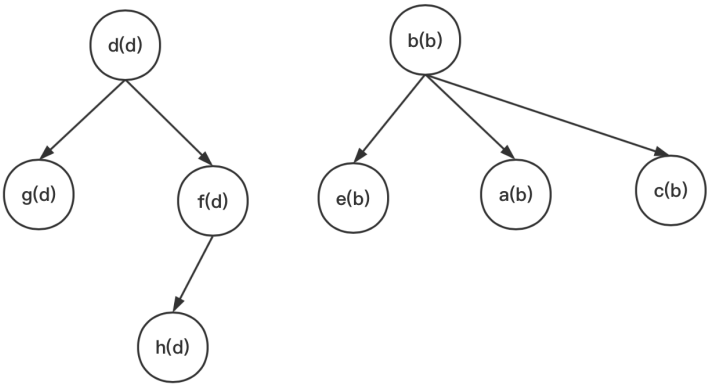
Process (f,h), the forest:



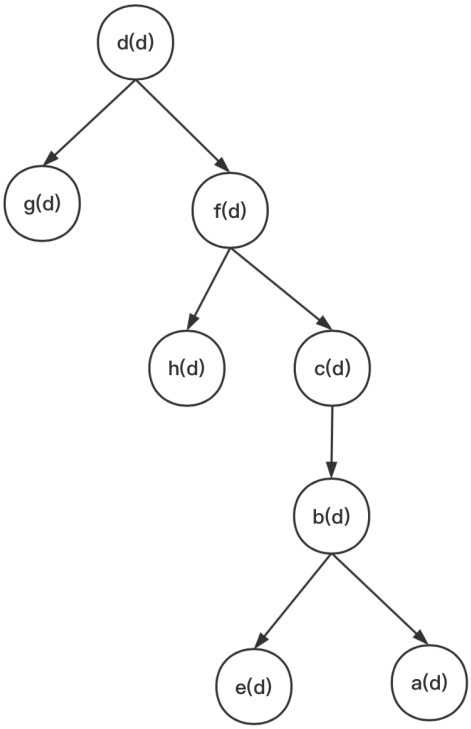
Process (g,h), the forest:



Process (c,b), the forest:



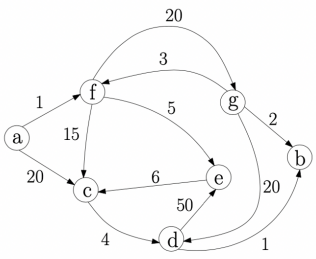
Process (c,f), the forest:



Q2:

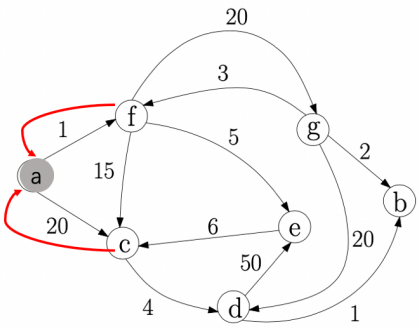
Initialization:

	a	b	c	d	e	f	g
D[]:	0	∞	∞	∞	∞	∞	∞
pred[]:	-	-	-	-	-	-	-



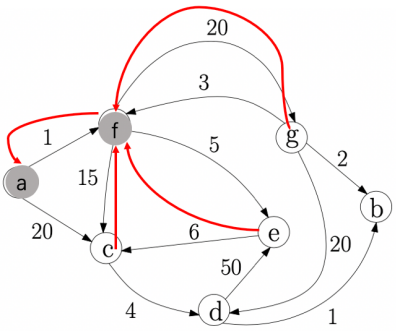
Process node a:

	a	b	c	d	e	f	g
D[]:	0	∞	20	∞	∞	1	∞
pred[]:	-	-	a	-	-	a	-



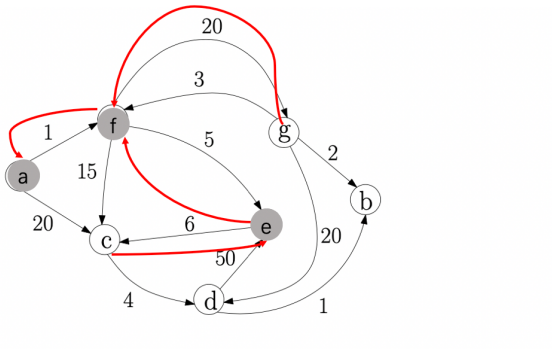
Process node f:

	a	b	c	d	e	f	g
D[]:	0	∞	16	∞	6	1	21
pred[]:	-	-	f	-	f	a	f



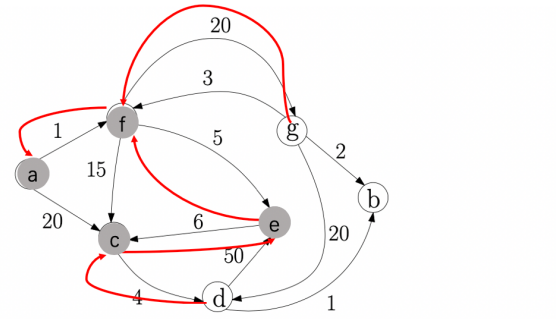
Process node e:

	a	b	c	d	e	f	g
D[]:	0	∞	12	∞	6	1	21
pred[]:	-	-	e	-	f	a	f



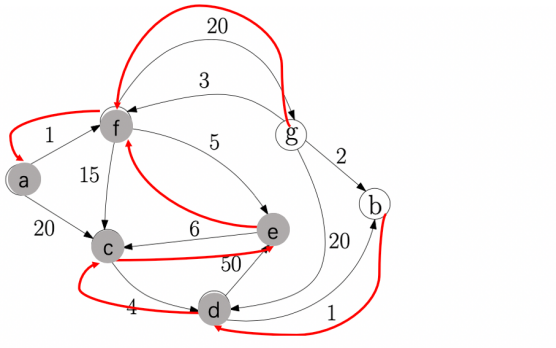
Process node c:

	a	b	c	d	e	f	g
D[]:	0	∞	12	16	6	1	21
pred[]:	-	-	e	c	f	a	f



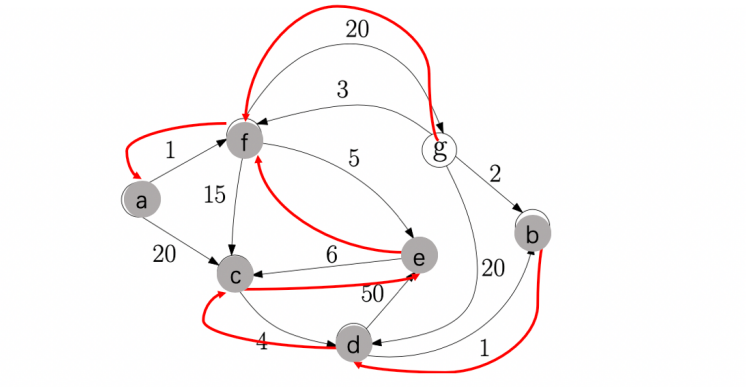
Process node d:

	a	b	c	d	e	f	g
D[]:	0	17	12	16	6	1	21
pred[]:	-	d	e	c	f	a	f



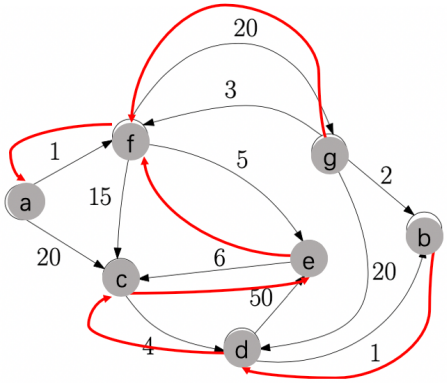
Process b:

	a	b	c	d	e	f	g
D[]:	0	17	12	16	6	1	21
pred[]:	-	d	e	c	f	a	f



Process g:

	a	b	c	d	e	f	g
D[]:	0	17	12	16	6	1	21
pred[]:	-	d	e	c	f	a	f



Q3:

Method 1: (apply Kruskal Algorithm, which is faster than Method 2)

```
function MinWidth(V,E,w):
mst = {};
for each vertex v in V do:
  make-set(v)
sort all edges (u,v) in E into List L;
for each (u,v) in sorted list L do:
  if find-set(u)!= find-set(v):
    mst = mst+{(u,v)}
    union-set(u,v)
return mst
```

In the above algorithm, three functions are illustrated as the following:

- Make-set(v): Create a new set whose only member is pointed by v.
- Find-set(v): Return a pointer to the set containing v.
- Union-set(u,v): Unites the dynamic sets, which contain u and v, into a single set.

For every node pair u and v, the path with minimum width between them is stored by the minimum spanning tree obtained in the above. Then, if we want to compute the path from given u to v, we can simply start from node u with BFS (Breadth-First-Search) to node v:

```
function MinWidthPath(mst,u,v):
BFS(mst,u,v)
```

- Running time analysis

Number of vertices: n;

Number of edges: m

```
function MinWidth(V,E,w):
mst = {};
for each vertex v in V do:          O(n)
  make-set(v)
sort all edges (u,v) in E into List L;      O(mlogm)
for each (u,v) in sorted list L do:
  if find-set(u)!= find-set(v):             O(m)*(find-set+union-set)
    mst = mst+{(u,v)}
    union-set(u,v)
return mst
```

```
function MinWidthPath(mst,u,v):
BFS(mst,u,v)                        O(n)
```

If we choose the implementation of disjoint-set data structure which uses union by rank and path compression, total time complexity of find-set and union-set is: $O(n+m)*\alpha(n)$

And $m \geq n-1$, $\alpha(n) = O(logn)$, thus $O(n+m)*\alpha(n) \rightarrow O(mlogn)$

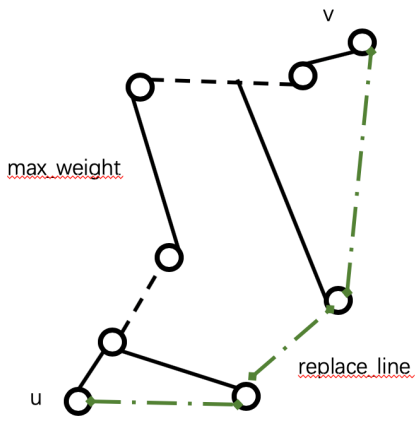
And $m \leq \frac{n(n-1)}{2}$, thus $O(mlogm) \rightarrow O(mlogn)$

Therefore, total running time is $O(mlogn)$

- Correctness analysis

The key is to prove the paths between every node pair in the minimum spanning tree have the smallest width.

Assume to the contrary that: for node pair (u,v), we have another path that is not in our mst (minimum spanning tree), and its width is smaller.



We denote this path as P, which is shown as green line in the above picture. And we can know that every weight in this path are smaller than the maximum weight of (u,v) path in our mst (shown as max_weight) according to the assumption.

Therefore, we definitely can find a replace line in path P to replace our max_weight line and still form a tree, whose weight sum should be smaller than our original mst.

But this is impossible. We apply Kruskal algorithm and the result mst should be the minimum spanning tree with smallest weight sum to connect all vertices. It is impossible to have a even smaller spanning tree.

So, the contrary assumption is wrong. There exists no another path with smaller width for every node pair (u,v). And our algorithm is correct.

Method 2: (apply similar idea of Dijkstra Algorithm)

According to the problem description, we need to find minimum-width paths for every pair of nodes. Therefore, we need set each node as the source node to find their minimum-width paths to other nodes.

And the array pred[:] for each source node stores the parent pointer thus saving the minimum-width path information.

Here is the pseudo code algorithm:

```
// input: adjacent list AL
// output: a matrix P whose row stores the min width path for each node s in AL as the
source node
function MinWidth(AL):
initialize matrix P;
for node s in AL do:
  initialize list E: E[s]=0, otherwise E[:]=∞ ;
  initialize array pred: pred[:]=null;
  while E.isempty()==false do:
    scan list E and delete minimum E[u];
    for edge (u,v) in AL do:
      if E[v] > max(E[u],w(u,v)):
        E[v]=max(E[u],w(u,v));
        pred[v]=u;
  P[s] = pred[:];
return P
```

- Running time analysis:

Number of vertices: n;

Number of edges: m

```
// input: adjacent list AL
// output: a matrix P whose row stores the min width path for each node s in AL as the
source node
function MinWidth(AL):
initialize matrix P;
for node s in AL do:                                   O(n)
  initialize list E: E[s]=0, otherwise E[:]=∞ ;
  initialize array pred: pred[:]=null;
  while E.isempty()==false do:                         O(n)
    scan list E and delete minimum E[u];               ──→ O(n)
    for edge (u,v) in AL do:                           ──→ Totally O(m)
      if E[v] > max(E[u],w(u,v)):
        E[v]=max(E[u],w(u,v));                         O(1)
        pred[v]=u;
  P[s] = pred[:];
return P
```

In other words, the running time is $O(n) * O(n^2 + m)$

And $m \leq \frac{n(n-1)}{2}$,

So the total running time is $O(n^3)$

- Correctness analysis

The key is to prove:

when node u is removed, aka, minimum E[u] is removed, E[u] is the actual minimum width value for node pair (s, u).

we denote the actual minimum width value for node pair (s,u) as d(s,u), and here is the proof:

1. Base case:

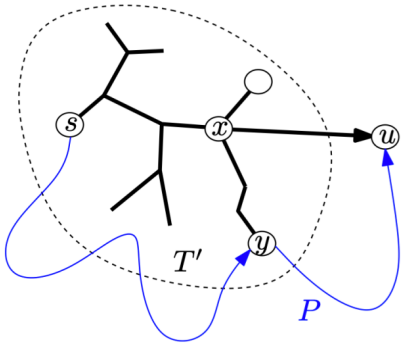
   Source node s is removed. Since E[s]=0, E[s]=d(s,s) trivially.

2. Induction:

   Induction Assumption: E[x] = d(s,x) for every vertex x has been removed so far.

   Induction Step: Now we consider node u from those nodes not processed yet, and E[u]=max(E[x], w(x,u)) for some vertex x which has been processed so far.

   Notice that the estimation E[u] is supposed: E[u] >= d(s,u), and here we assume to the contrary that E[u]>d(s,u). Therefore, there must be another actual minimum-width path P from s to u, and we let the last vertex which has been processed to be vertex y:



   Then I am going to prove the contrary assumption, aka, E[u]>d(s,u) is always wrong, thus proving E[u] is always: E[u]=d(s,u).

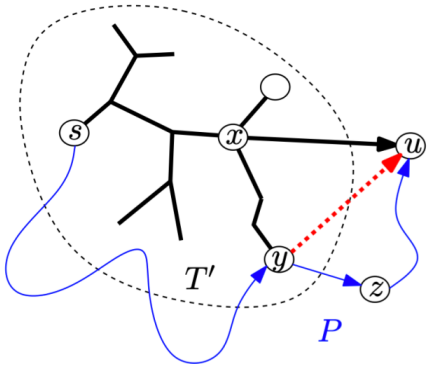
   Case 1: (red line)

   There exists a direct link between node y and node u.

   In this case, d(s,u)=max(E[y], w(y,u)). And E[u] > d(s,u), in order words:

$$E[u] = \max(E[x], w(x,u)) \quad > \quad d(s,u) = \max(E[y], w(y,u)).$$

   which is impossible according to the algorithm: if we correctly update the nodes that has been processed, E[u] should be the smaller one, aka, max(E[y], w(y,u))



   Case 2: (blue line)

   There exists a node z not processed yet in path P. Node y link to the node z.

In this case, we can know that:

max(E[y], w(y,z)) <= d(s,u) due to the definition of width of path.

d(s,u) < E[u] due to the contrary assumption

E[z] <= max(E[y], w(y,z)) according to the algorithm

Combine the above three inequality, we get:

$$E[z] < E[u]$$

which is impossible according to the algorithm: since we always look at the unprocessed nodes with minimum E[:] values, we should then consider node z before node u.

All two possible cases are impossible, leading that the contrary assumption is impossible. Therefore, E[u]=d(s,u).

When node u is removed, aka, minimum E[u] is removed, E[u] is the actual minimum width value for node pair (s, u).