**CSC505 HW4 Problem 1-3**

**Yiwei Li**

**Zirun Han**

1. Purpose: Understanding the structure of disjoint set union forests.

Do Exercise 21.3-4 on page 572.

You are asked to revise the implementation of disjoint sets as forests so that it supports a Print-Set(x) function, which prints all the elements in the set containing x. Runtime must be O(1) per element printed.

The naive solution – creating pointers that are reversals of the parent pointers – will not work for two reasons: (i) each node would have to keep track of all of its children, violating the condition that this be done with a single attribute for each node; and (ii) some elements would end up being printed more than once (the exercise does not explicitly forbid this, but we will require you to come up with a solution where each element is printed exactly once).

**Answer:**

1. To create Print-Set(x) function with runtime O(1) per element printed, we are going to implement a (circular singly) linked list to record of all the elements besides the disjoint set forest tree. Details as follows.

(1)While building the tree, we use Make-Set(x) first to create singleton sets for all the elements. Meanwhile, we are going to insert the name of each element x into the linked list and set pointer x.l (from the tree) to the element in that linked list. The running time is O(1). (2)The running time of Union(x,y) will keep the same. Because we didn’t need to update the pointer from all new added elements to the head. Linking up two lists takes constant time. (3)According the disjoint forest tree, Find-set(x) won’t change either.

As for the new function Print-set(x), we apply Find-set(x) to look for the representative (for example: m) for the set containing element x. Then we follow the m.l to the linked list, traversing all the elements in the linked list until we reached the same node where we started. Since we print all the elements in the set exactly one time, the running time of this operation Print-set(x) is O(1) per element.

2. Purpose: Understanding minimum spanning trees and edge contraction.

Do Problem 23-3 on pages 640-641 (bottleneck spanning tree).

You are asked to come up with an algorithm for a variant of the minimum spanning tree problem. Instead of minimizing the total weight of the edges in a spanning tree you are asked to minimize the maximum weight of an edge. The three parts ask you to (a) prove that any minimum spanning tree is also a bottleneck spanning tree; (b) come up with a linear time algorithm that determines (yes or no) whether there is a spanning tree whose maximum weight edge has weight b; and (c) give a linear time algorithm for the bottleneck spanning tree problem using (b) and an algorithm described in another problem (no need to give details of that one).

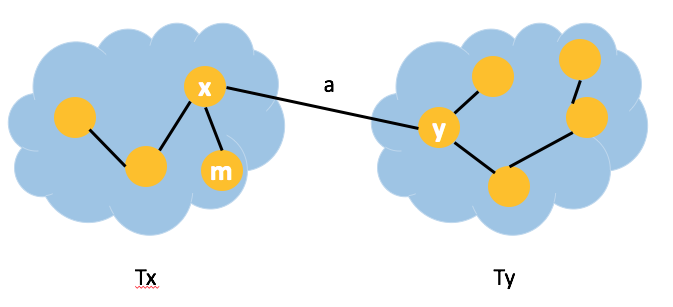
**Answer:**

2a. Consider we have a minimum spanning tree (T) for the graph G and the largest edge weight for T is w(x,y) = a. The value of bottleneck spanning tree for G is b.

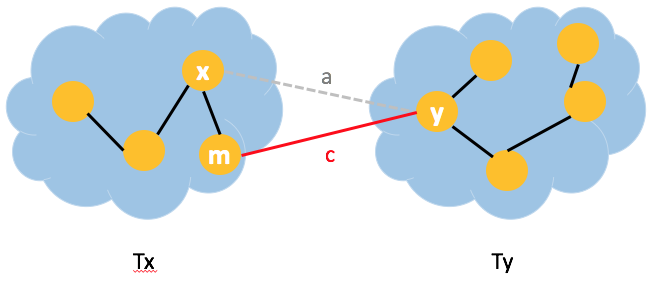
There are three possible results after comparing between a and b.

1. a<b. This won’t happen because the largest edge weight of bottleneck spanning tree is the smallest among all spanning trees.
2. a=b, which means T is the bottleneck spanning tree.
3. a>b. This won’t happen either. The explanation is listed as follows.

Since a = w(x,y), we are going to divide T into to sub-trees: Tx and Ty by ignoring the edge between x and y.



Assume the bottleneck spanning tree connect vertices x and y bypassing another vertex m instead directly connecting x and y together(figure in the next page). Since a is the minimum edge weight among the cut between Tx and Ty, a < c. Also c is in the bottleneck spanning tree with value b, so that c  b. Combing the two results: a< c . There is a conflict between our conclusion and the assumption: a >b. So that a> b won’t happen either.



In conclusion, a = b, so that the minimum spanning tree must be a bottleneck spanning tree.

2b. To decide if there is a spanning tree with the maximum weight edge b, we can simply see if there is a spanning tree of graph G or not after removing all the edges whose weight are larger than b. If the remaining graph of G cannot be connected together, we know that there isn’t a spanning tree with the maximum weight edge b. Vice versa.

The algorithm will first traverse all the edges and removing the edges with weights larger than b, which takes (E). Then we could run BFS or DFS to figure out if the remaining vertices are all connected together. The running time is O(V+E) for this procedure. So the total running time for determine if there is a spanning tree with the maximum weight edge b is O(2E+V) (linear running time).

2c. To come up a linear time algorithm for returning bottleneck spanning tree, we leverage the result from 2b and MST-Reduce from 23-2. From the previous lecture, the selection algorithm takes linear running time.

Step1: Using the selection algorithm, we get the median value x of all the edge weights with O(E) running time.

Step 2: We’ll remove all the edges with weights larger than x by applying 2b algorithm, which takes O(2E+V).

Step 3: There will be two results after applying 2b algorithm with x.

1. If there is a spanning tree with value at x, we’ll repeat Step 1 and 2 on the remaining tree structure. The number of edges in the remaining structure is monotonically decreasing. So that repeating step 1 and 2 will cost O(E/2) + O(2E+V) = O(2.5E+V) time (still linear running time).
2. If there isn’t any spanning tree with value at x, we are going to contract all the remaining edges. According to 23-2a and 23-2e, we run k phases of MST-Reduced followed by MST-Prim to get the minimum spanning tree, also bottleneck spanning tree, within O(ElglgV) time.

Or

1. If there is a spanning tree with value at x, we’ll repeat Step 1 and 2 on the remaining tree structure. In total, it takes O(E/2) + O(2E+V) time.
2. If there isn’t any spanning tree with value at x, we’ll resume the previous tree structure. Repeat Step 1 to get the median value y from range x to maximum edge weight, which takes O(E/2). And apply Step 2 by new median value y, which takes O(2E+V) time.

Running Step 3 until we find some value m is the maximum edge weight of a spanning tree while m-1 isn’t the maximum edge weight of a spanning tree. Practically, Step 3 is repeating step 1 and 2 on the tree structure. The total time will be (O(E) + O(2E+V)) + (O(E/2) + O(2E+V)) +(O(E/4) + O(2E+V)) … + (O(1) + O(2E+V)) = O(2E) + (lgE)O(2E + V) (applying geometric series for O(E) + O(E/2) + … +O(1))

3. Purpose: Understanding Dijkstra’s shortest paths algorithm.

Suppose you have a weighted, undirected graph G with positive edge weights and a start vertex s. Come up with an algorithm that runs as fast as Dijkstra’s and assigns a label usp[u] to every vertex u in G, so that usp[u] is true if and only if there is a unique shortest path from s to u. By definition usp[s] is true. Be sure to prove both the correctness and time bound of your algorithm.

To get full credit, you need to (a) explain in English how your algorithm works; (b) give pseudocode (this may involve a modification of Dijkstra’s algorithm); (c) illustrate how your algorithm works on a small (at most five vertices) but nontrivial (has cases where labels change) example; and (d) give a correctness proof (similar to that for Dijkstra’s algorithm). Proof of the time bound should be simple if you did everything else correctly, but you need at least a sentence or two to address it.

**Answer:**

3a. To come up with an algorithm that runs as fast as Dijkstra’s and also keeps record of uniqueness property for the shortest path, we just need to make some modification based on the original Dijkstra’s algorithm.

When we run original Dijkstra’s algorithm, we relax all the outgoing edges from min node which in the priority queue. As for the relaxation, we compare the distance of vertex when go through the min node to the distance record for this vertex. If pass by min node create a shorter path, we update the vertex distance as min node’s distance plus the edge weight from min node to this vertex and also update the vertex predecessor as this min node. If we cannot see a shorter path, we don’t change anything.

To keep track of the uniqueness property, we just need to evaluate one more thing. If pass by min node create a same distance path, we’ll set the uniqueness of the shortest path as false. In another word, if vertex distance is the same as min node’s distance plus the edge weight from min node to this vertex, we’ll set usp[vertex] = false. Because there is a second route from source to go to this vertex, so the shortest path is no longer unique. In addition, once one vertex’s path uniqueness is false, all the vertices’ path uniqueness if their shortest paths go through this vertex (a.k.a this one vertex is the predecessor of all other vertices). This means that the false uniqueness can be inherited from the predecessor.

3b. The pseducode was listed as below. Original code was colored as black. The modification part was colored as orange and the comment was colored as green.

Pseducode:

DijkstraWithUniqueness()

Set all vertex costs to infinite, path\_uniqueness as true and predecessors to undefined

Q = queue of all vertices, ordered by distance

s.distance = 0 #initial the distance of the source as zero

usp[s] = true #initial the uniqueness of source to be true

while Q is non-empty:

u = remove lowest-distance vertex from Q

for each u -> j edge

if j.distance > u.distance + edge\_weight(u,j)

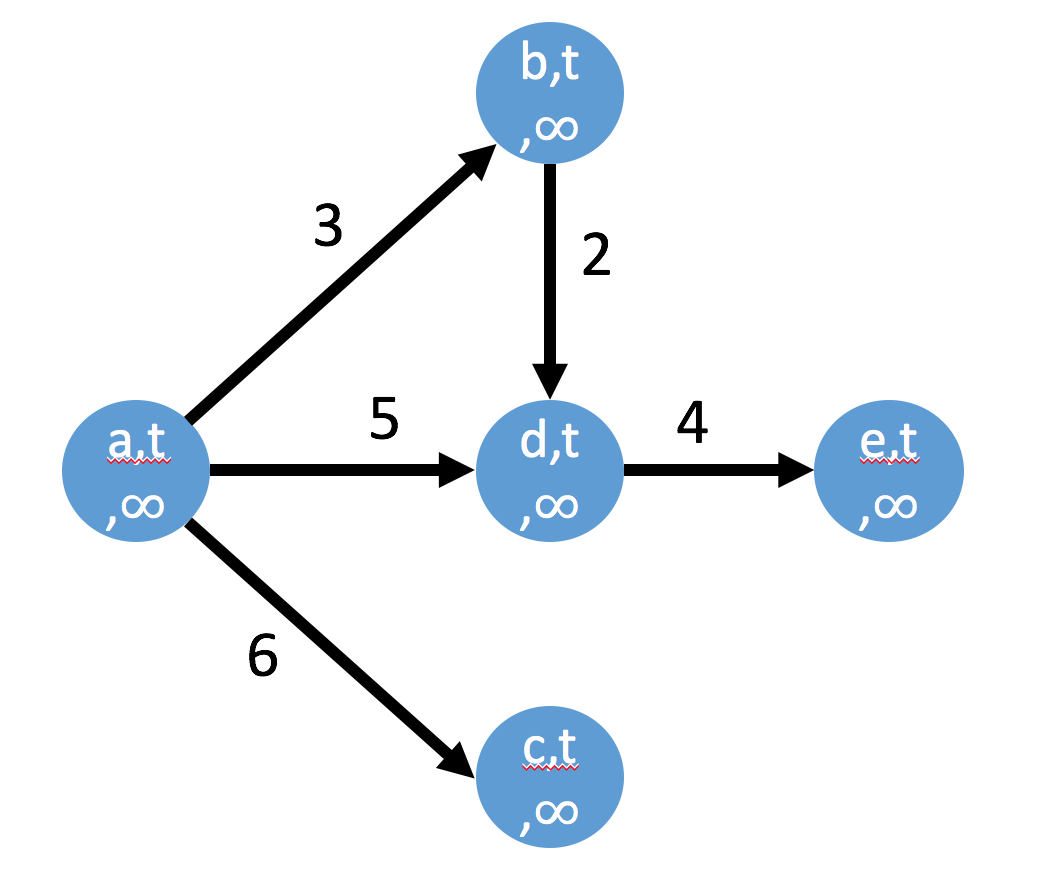
j.distance = u.distance + edge\_weight(u,j)

j.predecessor = u

usp[j] = usp[u] #j inherite the uniqueness from it predecessor

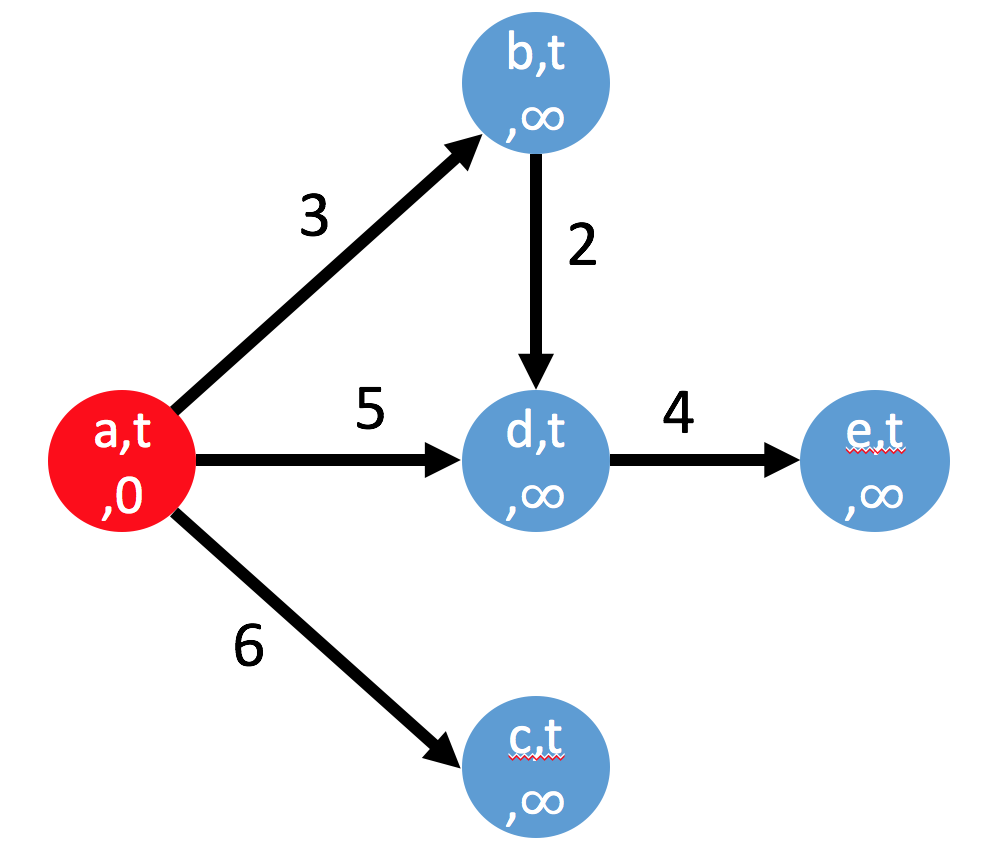
if j.distance = u.distance + edge\_weight(u,j) # there is another shortest path to j

usp[j] = false # the shortest path to j is no longer unique

3c. Start from the following graph.

a is our source.

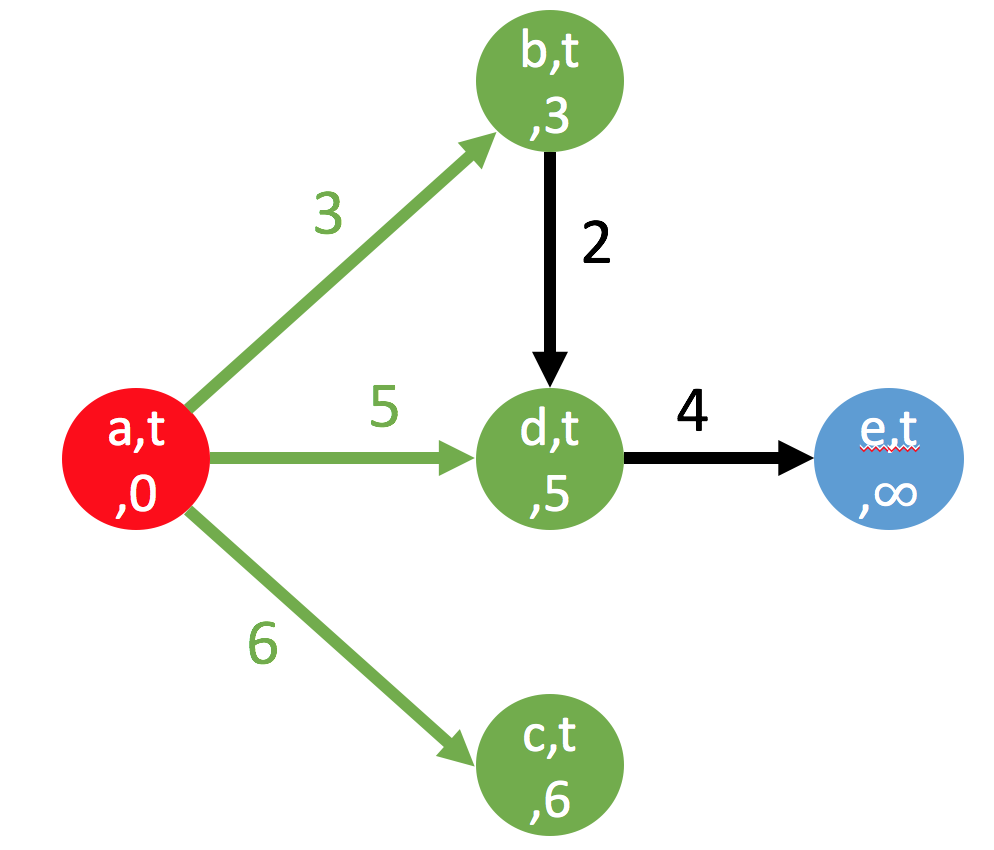
Initialize a’s distance as zero



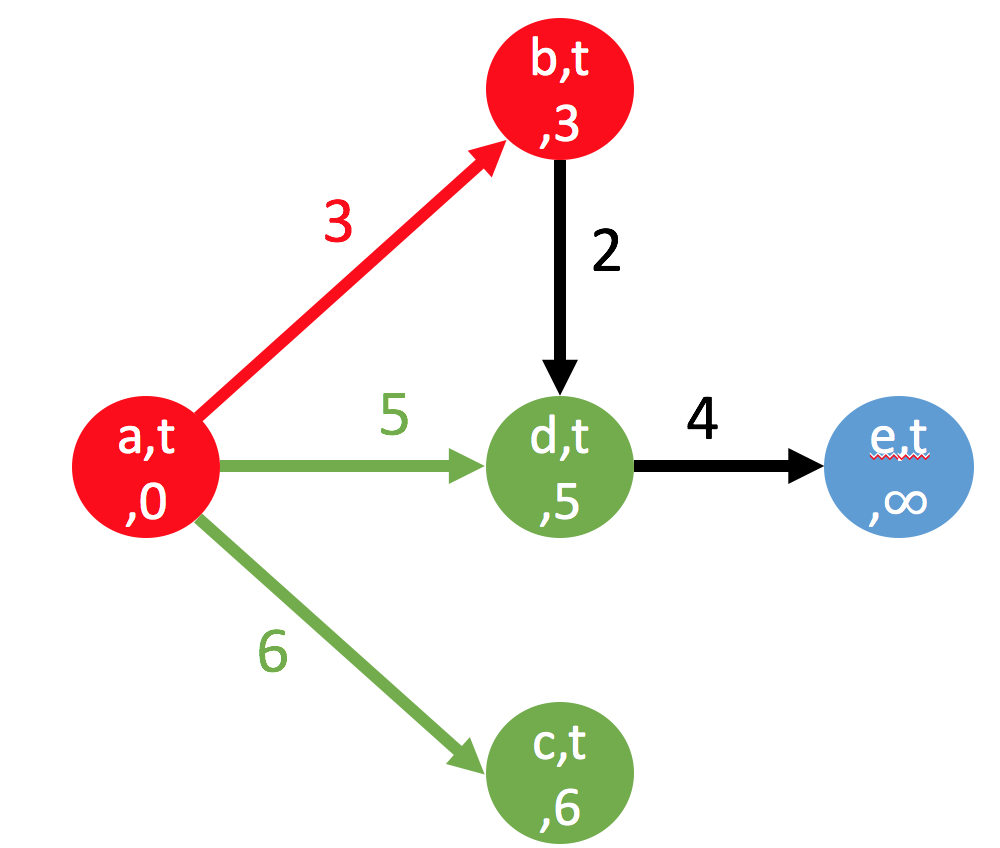
Relax all vertices where a->vertices

Update b, d, c’s distances

b, c, d’s uniqueness is the same as their predecessor



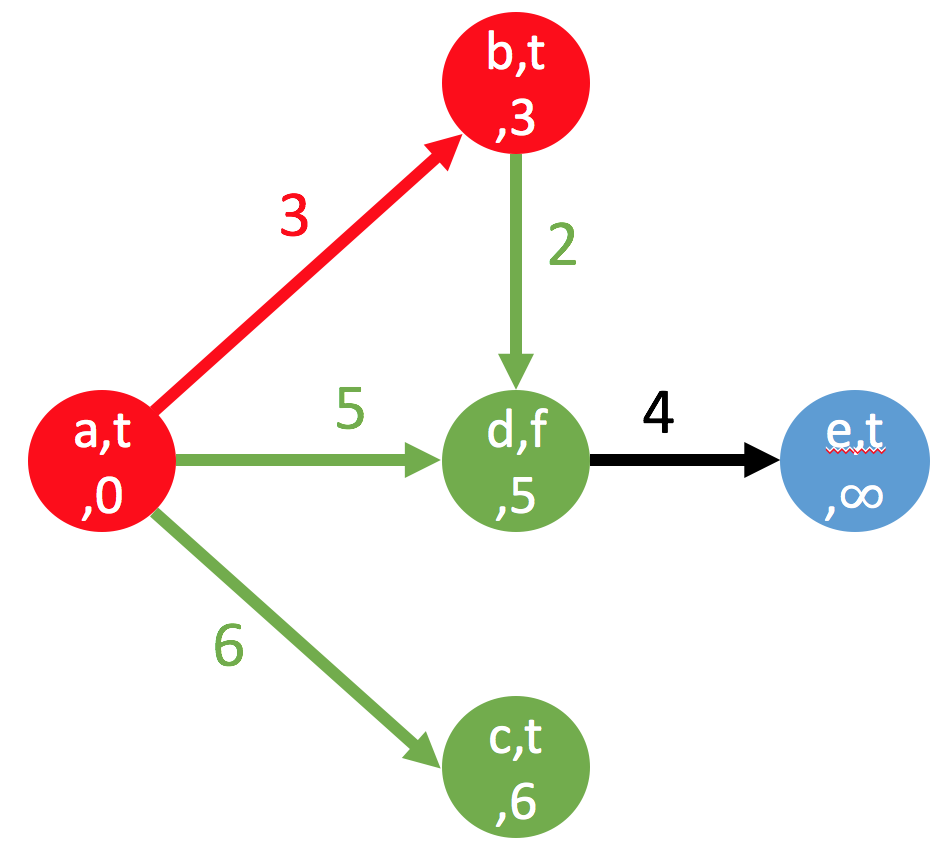
Remove the minimum node from the queue --- b

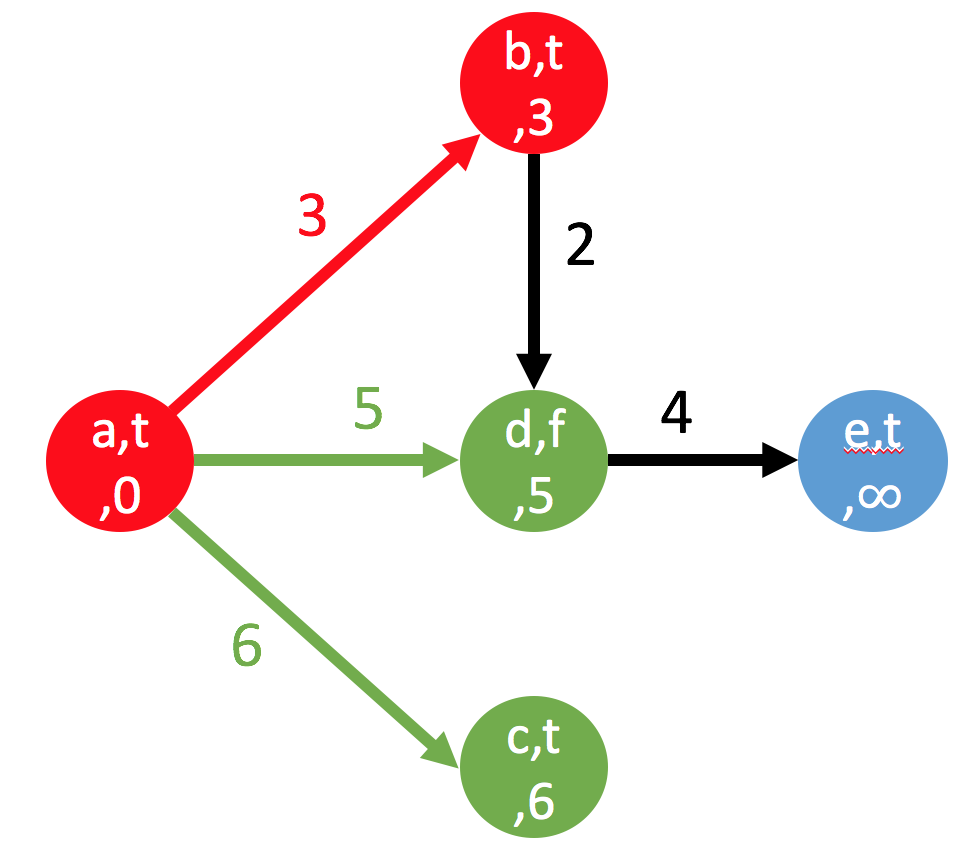


Relax all vertices where b->vertices

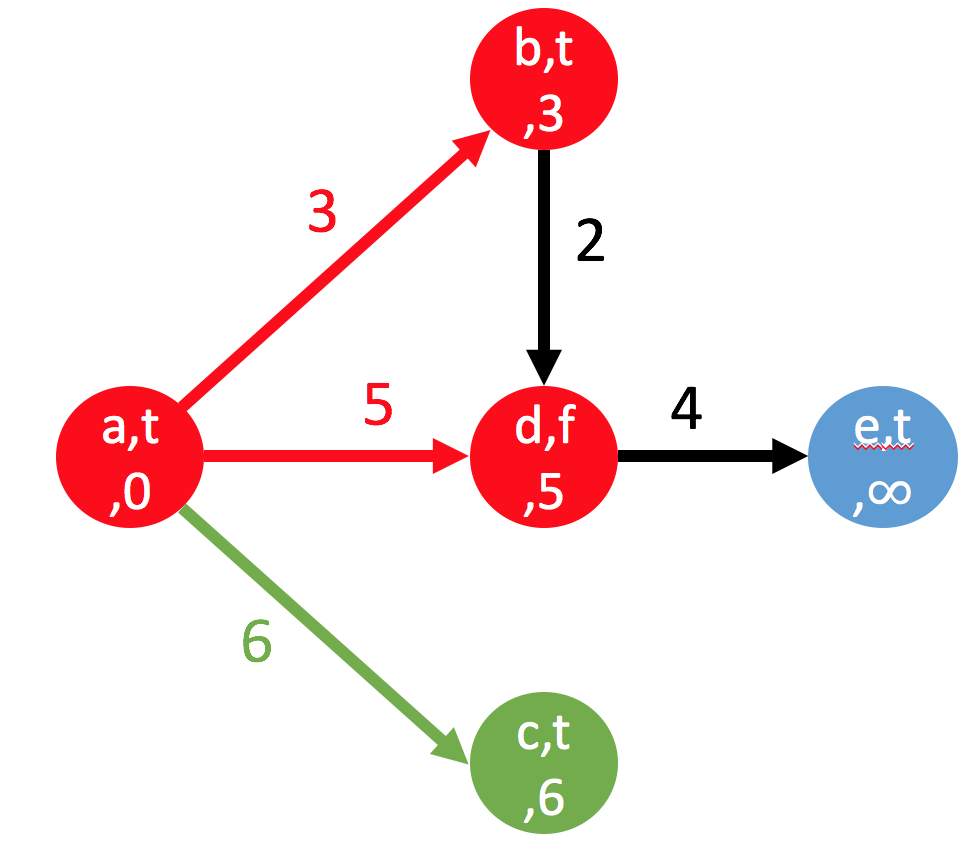
Compare d’s distance and b’s distance + w(b,d) (there was already a route to d)

They are the same. So the usp[d] = false





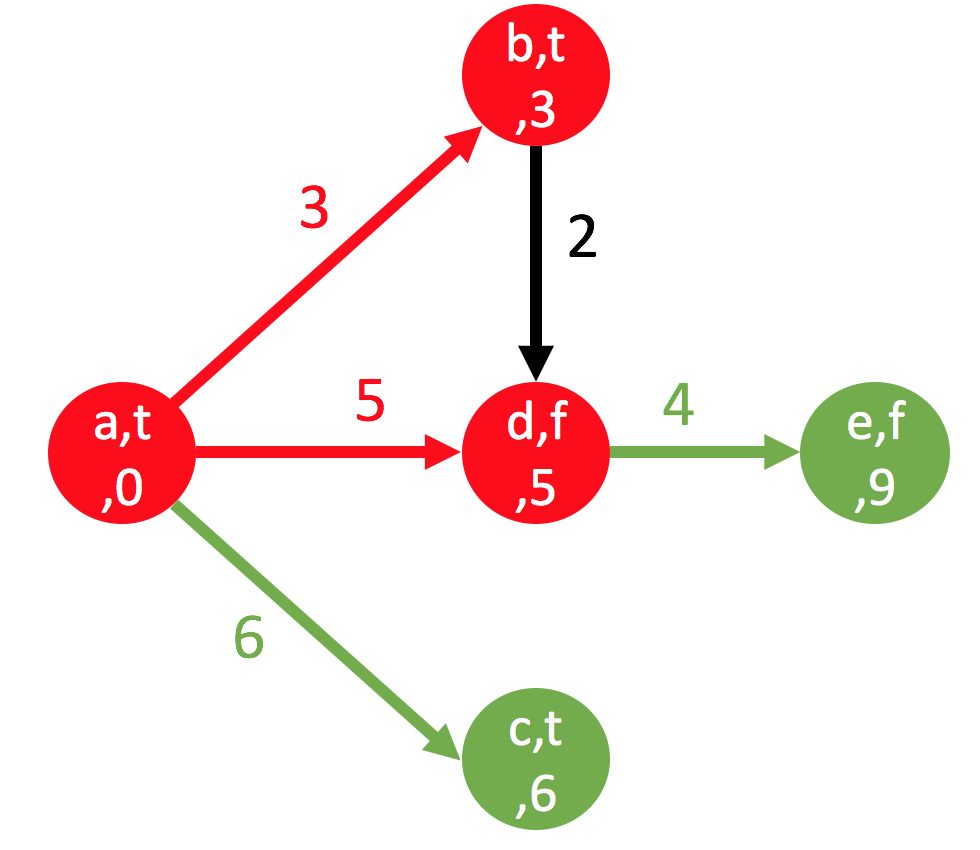
Remove the minimum node from the queue --- d



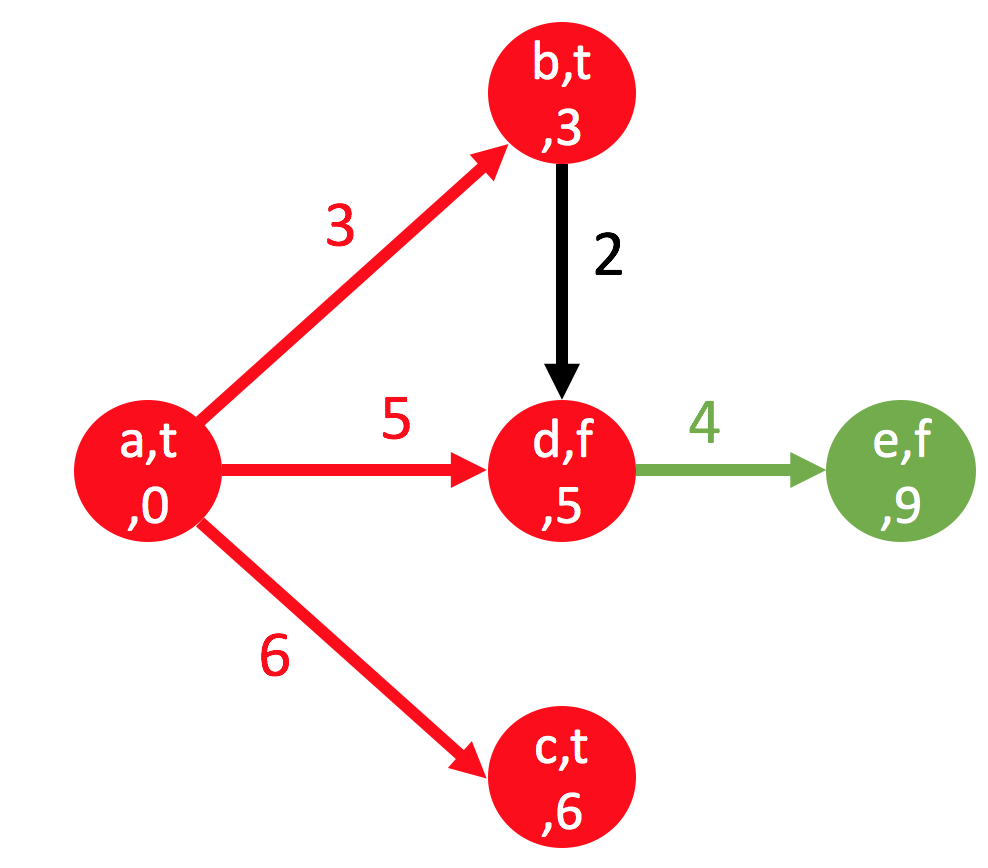
Relax vertex where d->vertex

Update e’s distance

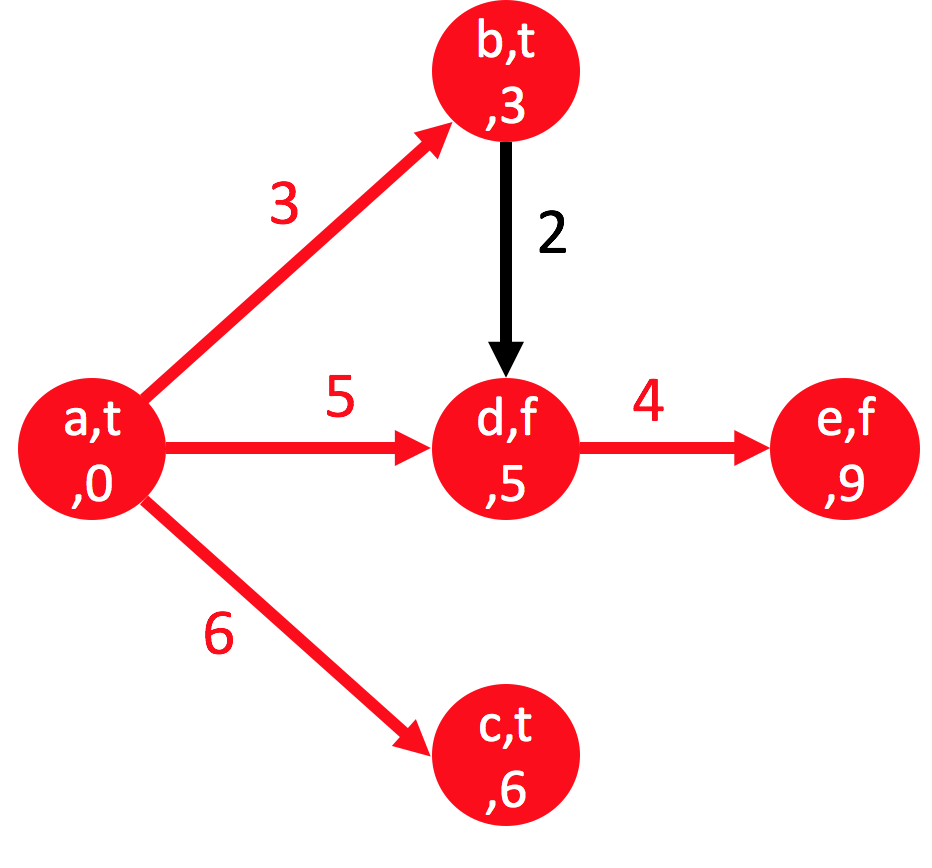
e’s uniqueness is the same as their predecessor ---d



Remove the minimum node from the queue --- c



Remove the minimum node from the queue --- e



3d. give a correctness proof (similar to that for Dijkstra’s algorithm). Proof of the time bound should be simple if you did everything else correctly, but you need at least a sentence or two to address it.

We’ve already learned the correctness proof of Dijkstra’s algorithm during the class. (The details are in the lectures slides. Briefly talking, assume u is the first vertex that goes into the tree with an incorrect shortest path p -> u. Maybe a shorter path from one of the vertex in the tree p’ -> u. This won’t happen because u would have received a new distance estimate (and predecessor edge) when edges out of p’ were relaxed.) The new algorithm is a modification based on the Dijkstra’s algorithm. We add new feature/object/field?? label to each vertex in the graph as the uniqueness property and another if statement to decide whether the newer considered path is the same as the previous discovered shortest path. This if statement and adding new label to each vertex will take constant time. So the modification won’t asymptotically change the running time of Dijkstra’s algorithm.