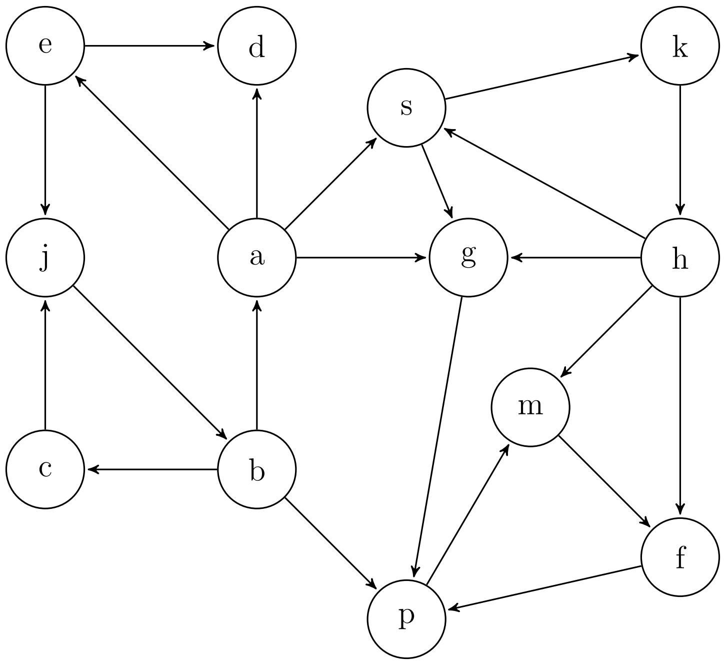
Justin Baraboo
Design and Analysis of Algorithms
Assignment 4

April 19, 2015

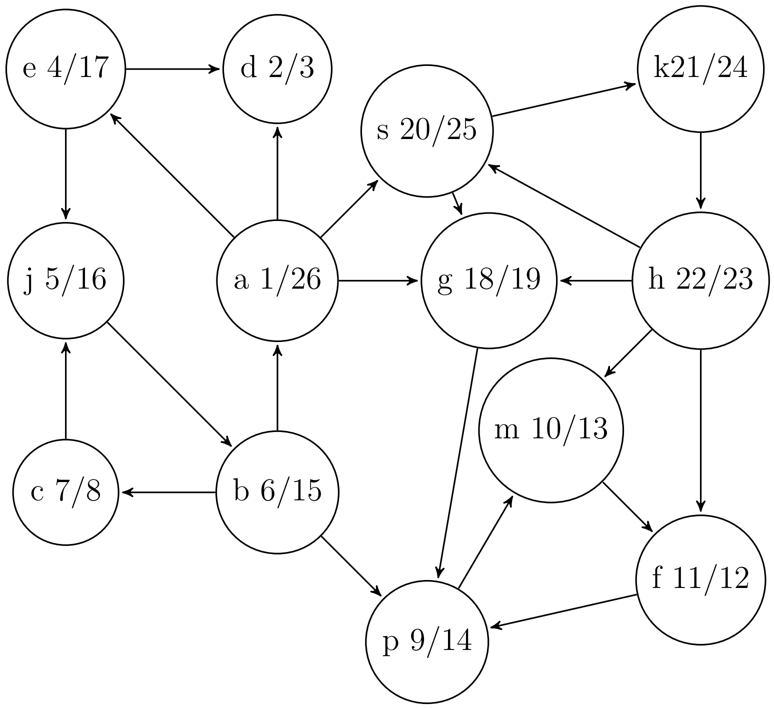
1 Problem 1

Conduct a DFS for the following graph.



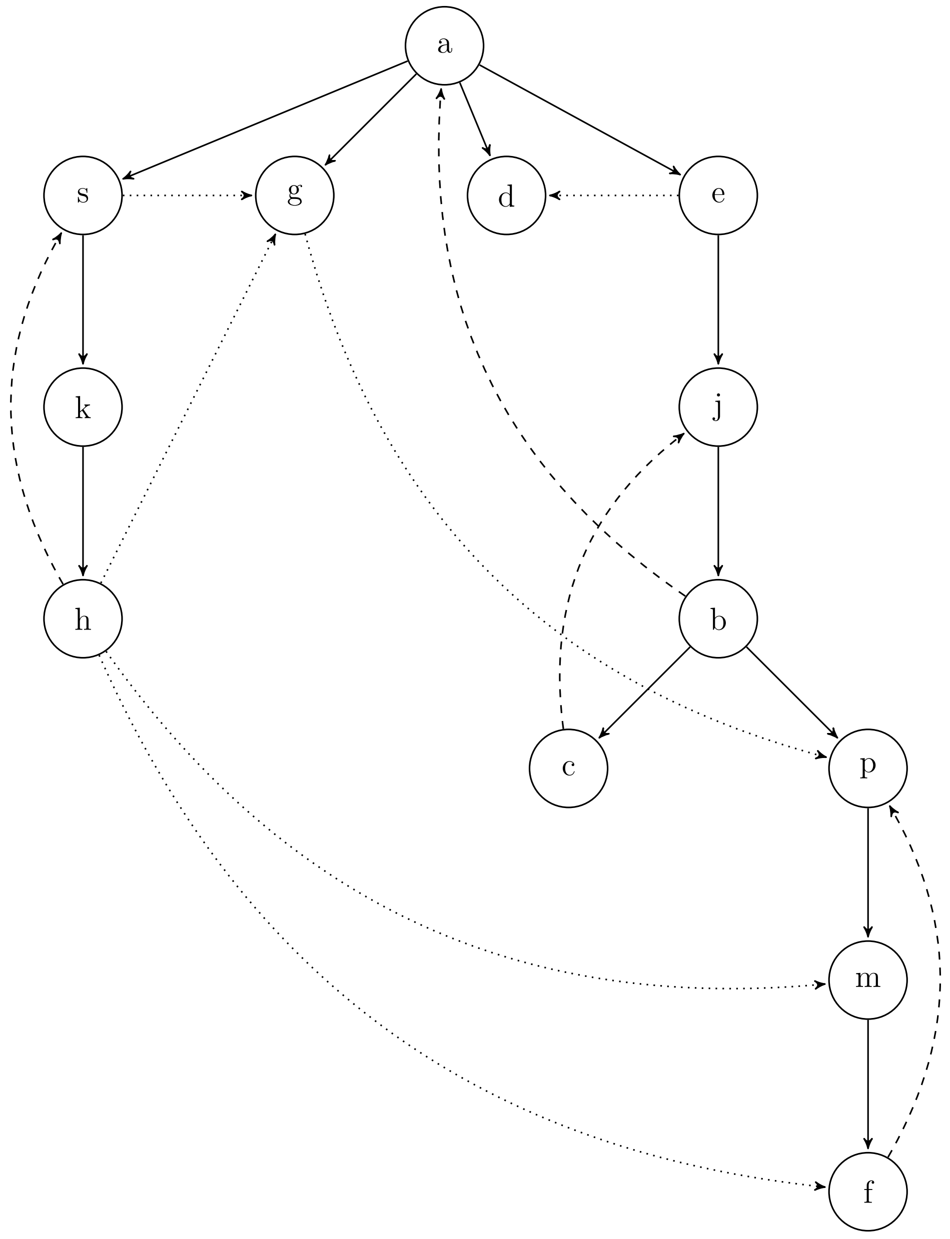
Solution For our DFS, we start at node a, we then go to node d which has no neighbors so we go to back to a.
We then go to e then from there to j, b, c where we have no node to go to. So we go back to b.

From there we go to p, m, f.Where we find no nodes to go to. We go back through our visited nodes finding they have no more nodes to visit until we reach back to a.
We then go to g find no node to visit and go back to a. We then go to s, then to k and then to h and we find no nodes to visit. We go back through and find no nodes to visit for any of our nodes. We then realize we have traversed the graph and are done.



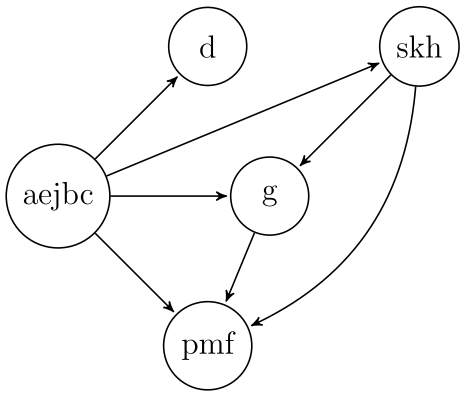
So our transversal is: $a, d, e, j, b, c, p, m, f, g, s, k, h$

Which gives the following tree:



Our back edges, dashed lines, are: $(h, s), (c, j), (f, p), (a, b)$.
We have no forward edges.
Our cross edges, dotted lines, are: $(e, d), (g, p), (h, f), (s, g), (h, g), (h, m)$.

The strongly connected components are: a, e, j, b, c and s, k, h and p, m, f and g and d.



2 Problem 2

A binary tree is called a full binary tree if any internal node has exactly two children. Let L be the set of leaves in a full binary tree T. Use mathematical induction on the height of the tree to prove the following formula: $\sum\limits_{x \in L} \frac{1}{2^{depth(x)}} = 1$.

Solution Assume that our tree is also complete. Then for a tree with n leaves, we have the height of the tree (and thus the depth to each leaf) as $\lg(n)$ giving:

$$\sum_{x \in L} \frac{1}{2^{depth(x)}} = \frac{n}{2^{\lg(n)}} = \frac{n}{n} = 1$$

So assume that our tree is full but not complete. We will use induction to show the relationship.
Base Case: $n = 1$. All we have is the root so:

$$\sum_{x \in L} \frac{1}{2^{depth(x)}} = \frac{1}{2^{\lg(1)}} = \frac{1}{2^0} = 1$$

Inductive Step: Now assume that $n > 1$ and we will show the relationship. Let r be the number of nodes in the last row and r' be the number of nodes in the (last - 1) row that are leaves. Consider the subtree consisting of the root to the parents of these nodes. This is a full tree with number of leaves equal to half of n and thus satisfies:

$$\sum_{x \in L_{Subtree}} \frac{1}{2^{depth(x)}} = 1$$

We will parition the left sum into two sums, those nodes whose children are in r, define their set as m, and those nodes whose children are in r', define

their set as m':
$$\sum_{x \in L_{Subtree}} \frac{1}{2^{depth(x)}} = \sum_{x \in m} \frac{1}{2^{depth(x)}} + \sum_{x \in m'} \frac{1}{2^{depth(x)}} = 1$$

Now, for each parent, we have 2 child nodes who are the leaves of the original tree which are one height greater than their parents, giving the formula for the original tree's leaves as:

$$\begin{aligned}\sum_{x \in L} \frac{1}{2^{depth(x)}} &= \sum_{x \in r} \frac{1}{2^{depth(x)}} + \sum_{x \in r'} \frac{1}{2^{depth(x)}} \\ &= \sum_{x \in m} \frac{2}{2^{depth(x)+1}} + \sum_{x \in m'} \frac{2}{2^{depth(x)+1}} \\ &= \sum_{x \in m} \frac{1}{2^{depth(x)}} + \sum_{x \in m'} \frac{1}{2^{depth(x)}} \\ &= 1\end{aligned}$$

3 Problem 3

Please write a program or a pseudo code that finds the successor for a given node x.

Solution For this algorithm we begin by checking to see if we have a right child. If we don't, and we are the root, then we are done. Else, we see which side of the tree we are on by comparing ourselves with the root.
If we are on the right side with no right child, we look up the tree, looking for the closest father/grandfather/ancestor whose key value is greater than our nodes, signifying that it is our successor. If we reach the root, we do not have a successor.
If we are on the left hand side, we do a similar thing, looking up the tree for the next key value whose value is greater than ours. Except this time we are guaranteed to have a successor as the root must greater than all the nodes on the left hand side.
If we do have a right child, we then check to see if our right child has a left child, if not, he is our successor. If so, we continue to check our left child for more left children until we reach a leaf. That node is our successor.

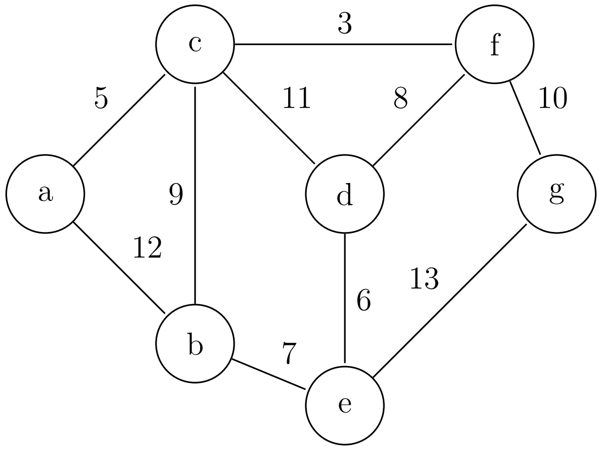
```
successor( tree t, node n)
{
   if( n.right() == nil && n.key() = t.root.key())
   {
      return nil \\no successor
   }
   else if( n.right() == nil && n.key() > t.root.key())
   {
      fatherR = n.father()
      while( fatherR.key != t.root.key())
      {
       if( fatherR.key() > n.key() )
       {
        return fatherR.key()
       }
       else
       {
        fatherR = fatherR.father()
       }
      }
       return nil \\no successor
   }
   else if( n.right() == nil && n.key() < t.root.key())
   {
      fatherL = n.father()
      while( true )
      {
       if( fatherL.key() > n.key() )
       {
        return fatherL.key()
       }
       else
       {
        fatherL = fatherL.father()
       }
      }
   }

   else
   {
      childR = n.right()
      if( childR.left() == nil)
      {
       return childR.key()
      }
      successor = childR.left()
      while( successor.left() != nil)
      {
       successor = successor.left()
      }
      return successor.key()
   }
}
```

---

4 Problem 4

Let G(V, E) be a weighted (undirected) graph G(V, E). The weight of an edge is called the width of the edge. Let P(u, v) be a path from vertex u to vertex v in graph G(V, E). The width of the path is defined to be the smallest weight among all edges on the path. (The edge with the smallest weight is called the bottle neck edge.) A path P(u, v) is called the widest if the width of the path is the largest among all paths from u to v. Please modify the Dijkstras algorithm to compute the widest path from a given vertex s to every other vertex. The correctness proof is not required.

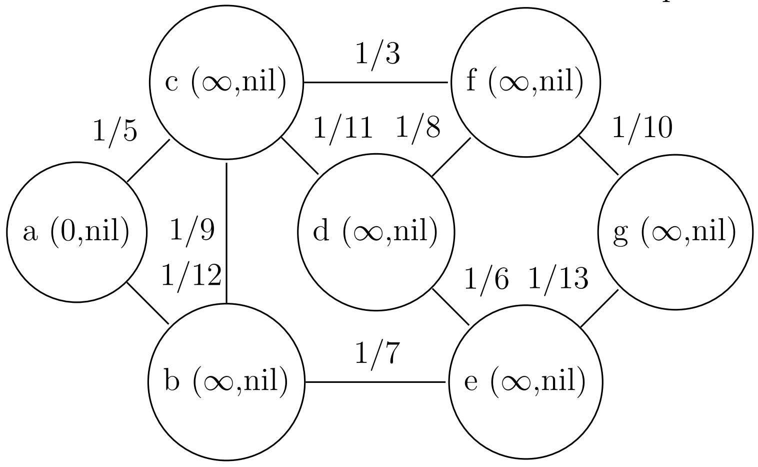


Solution To find the maximum width graph, we will employ Dijkstra's algorithm, but we will first take the inverse of all the edges weights. Since Dijkstra's finds the smallest path, with the edges inverted, we will find the largest.
Also, similar to Prim's algorithm, we only need to add the node with the shortest (which represents the longest) distance from the tree.

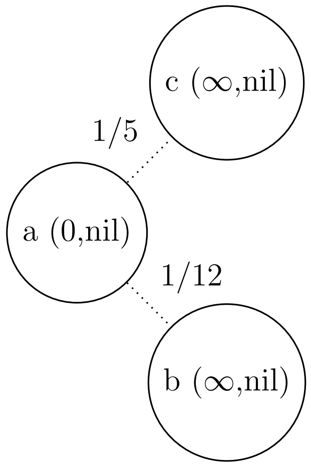
```
WidestPath( G, w, s)
{
 //invert each weight
 for each weight in w
 {
  weight = 1/weight
 }

 //then just do Dijkstra's as normal
 for each vertex in G
 {
  d[v] = infinity
  p[v] = nil
 }
 d[s] = 0
 S = empty
 Q = V[G]
 while Q nonempty
 {
  u = Extract-Min( Q )
  S = S U {u}
  for each vertex adjacent to u
  {
   if(d[v] > w(u,v))
   {
    d[v] = w(u,v)
   p[v] = u
   }
  }
 }
}
```

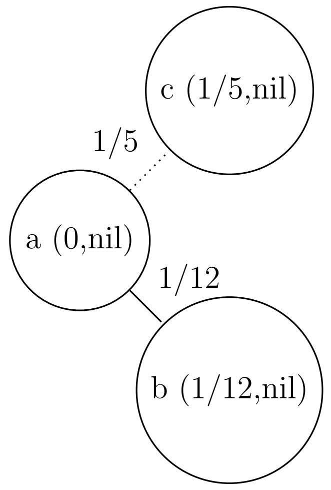
To print the path from s to v we merely have to recursively print out v, change v to it's father, and continue until we are at s which has not father. We will now show how this is done on the example. Our initialized graph:



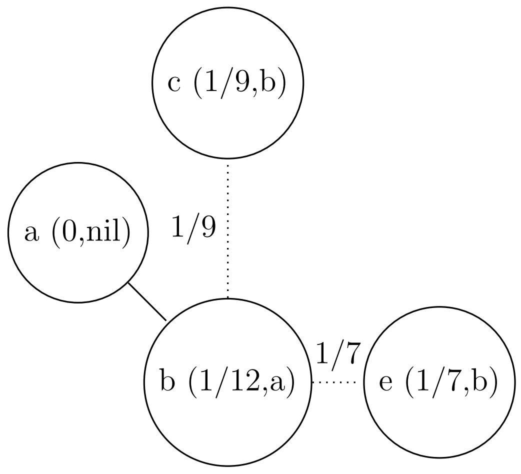
We start with a and look at it's neighbors, since they both have widths than their current values we update them.



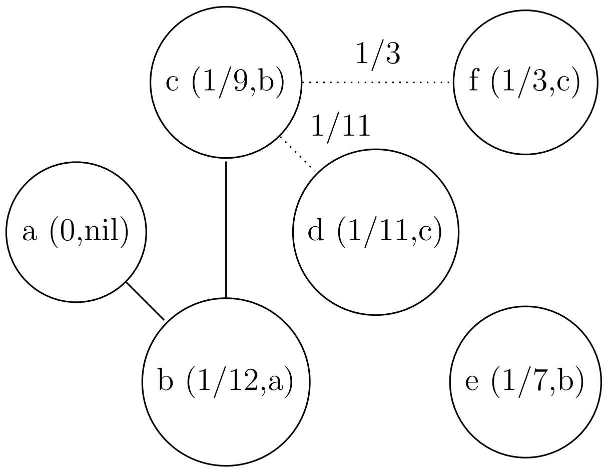
We now have 2 choices we could make, c or b. We choose b as it has the smallest width.



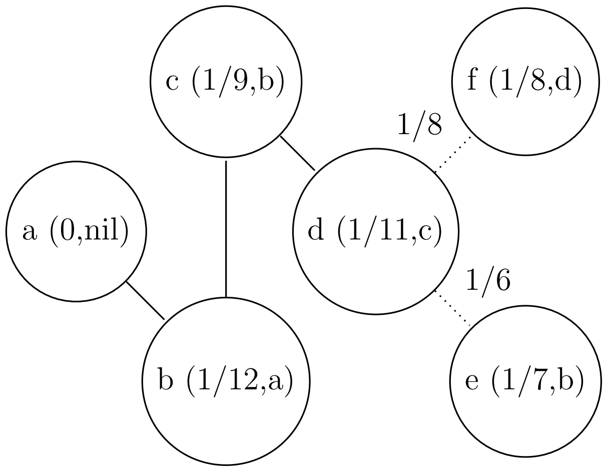
We continue, we look at the neighbors of b and update c as their edge is smaller, same with e.



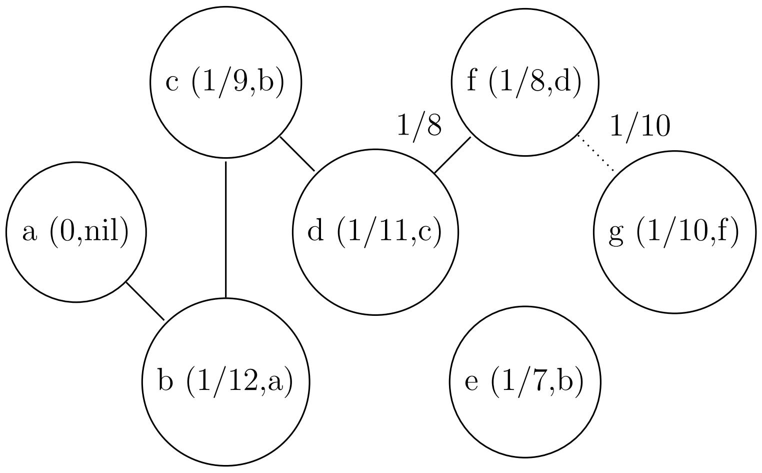
We choose c and look at it's neighbors and update them.



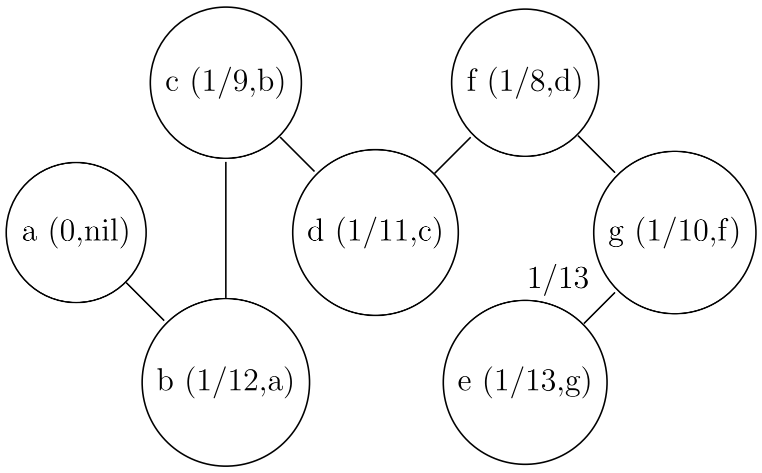
We choose d and update it's neighbors.



We chose f and update it's neighbors.



We choose g and update it's neighbor.



We choose e and we have no neighbors not visited and so are done.

5 Problem 5

Let T be a maximum spanning tree of graph G(V, E). Prove that any path P(u, v) in T is the widest path between u and v in G, where u, v can be any two vertices in G.

Solution Assume, by contradiction that P(u,v) is not the widest path from u to v. Then $\exists P'(u,v)$ such that $wid(P'(u,v)) > wid(P(u,v))$.
This means that $\exists (x,y) \in P(u,v)$ such that $weight(x,y) < (x',y') \ \forall \ (x',y') \in$

$P'(u,v)$.
There must also be an edge in $P'(u,v)$ that is not in our maximum spanning tree, else we would have two paths from u to v and thus a cycle and our tree wouldn't be a spanning tree.
We will construct a tree as follows:
Take our maximum spanning tree and remove (x,y). This will form two seperate trees.
We will rejoin the trees by selecting an edge from $P'(u,v)$ that joins them. This edge must exist as the path $P'(u,v)$ exists. This will also form a spanning tree as it connects the two disjoint trees. Call this edge (x',y').
So it follows that $\sum\limits_{weights} T_{new} > \sum\limits_{weights} T_{old}$ as they are exactly the same except for (x,y) in T_{old} and (x',y') in T_{new} and $(x,y) < (x',y')$.