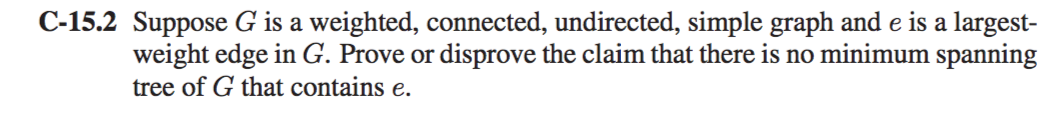
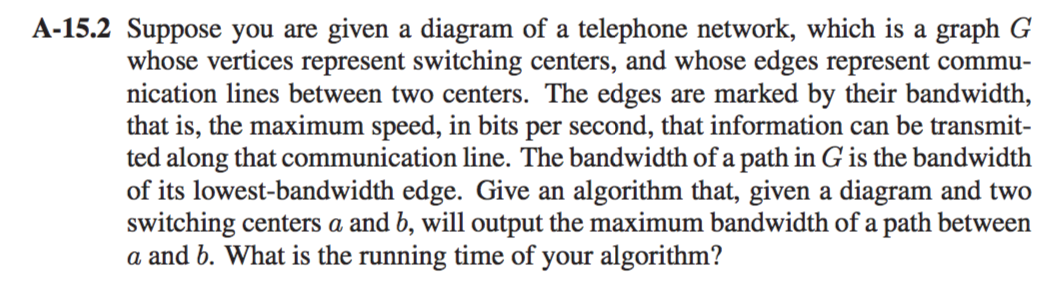
Homework 7

Atiq Patel CWID: 10432883



**Solution:** If the largest weight edge e in G is the one and only edge then, it would be present in every minimum spanning tree of G. Thus, the claim given is a false statement.

Consider a Minimum Spanning Tree, suppose there exists a spanning tree T’ such that the largest edge weight is smaller than the largest edge weight in T. Call the corresponding edges, e’ and e respectively. Remove e from T. This breaks T into two connected components. There must exist an edge e’’ in T’ such that it connects these components.



**Solution:** This is a maximization problem that can be solved using Djikstras algorithm

Consider a priority queue PQ where the priorities are given to the max value.

**Algorithm MaxB(G,A,B):**

**Input:** A weighted graph G with vertices V wherein there is vertex a and b.

**Output:** The shortest path from a to b.

ForEach Vertex(u) != a in G

Dist[a] 🡸 +∞

Dist[u] 🡸 0

While (PQ!= empty) **do**

A 🡸 PQ.removeMin() // removes the vertex with max value

**If** A ==B **do**

Return D[A]

**Else do**

**ForEach** neighbor of vertex(v) to vertex(u) in PQ **do**

Tot 🡸 Dist[A] + length(A,B)

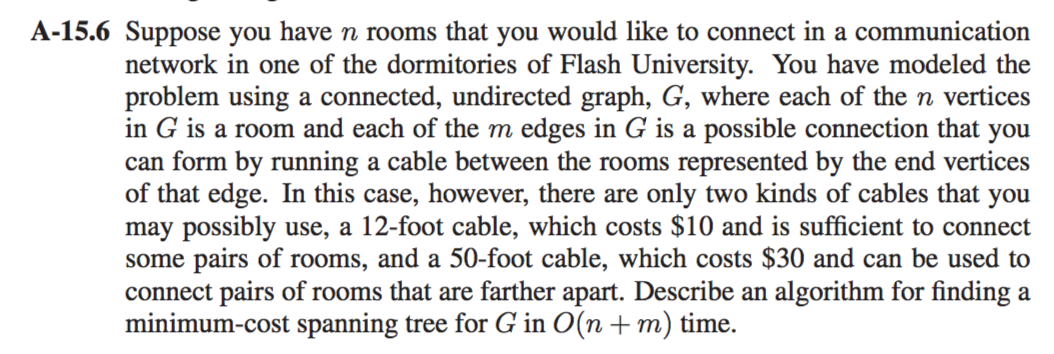
**If** tot< Dist[A] **do**

Dist[B] 🡸tot

//Alter the value of c in PQ to Dist[C]

Return Dist[A,B]

To reduce the space complexity we can make the use of adjaceny list. The run time for this algorithm depends on how we implement the algorithm , in our case it would be **O((n+m)log n)** wherein n +m are the edges of the graph if it is implemented using Binary Heap.



**Solution:** We will make use of Prim-Jarnik algorithm that finds a minimum spanning tree for a connected weighted undirected graph.

**Algorithm PJMST(G):**

**Input:** A graph G with Vertices V with n number of vertices and m edges

**Output:** A minimum spanning tree for the graph G.

Pick a random vertex inside G

Dist[v] 🡸0

**ForEach** Vertex(u) != Vertex(v) in G

Dist[u] 🡸 +∞

Initialize T 🡸 ∅.

Initialize a priority queue Q with an item ((u, null), D[u]) for each vertex u, where (u, null) is the element and D[u] is the key.

While (PQ!= empty) **do**

(u,e) 🡸 PQ.removeMin()

Add vertex u and edge e to T. for each vertex z adjacent to u such that z is in Q do // perform the relaxation procedure on edge (u, z) if w((u, z)) < D[z] then D[z] ← w((u, z)) Change to (z,(u, z)) the element of vertex z in Q. Change to D[z] the key of vertex z in Q. return the tree T

**If** A ==B **do**

Return D[A]

**Else do**

**ForEach** neighbor of vertex(v) to vertex(u) in PQ **do**

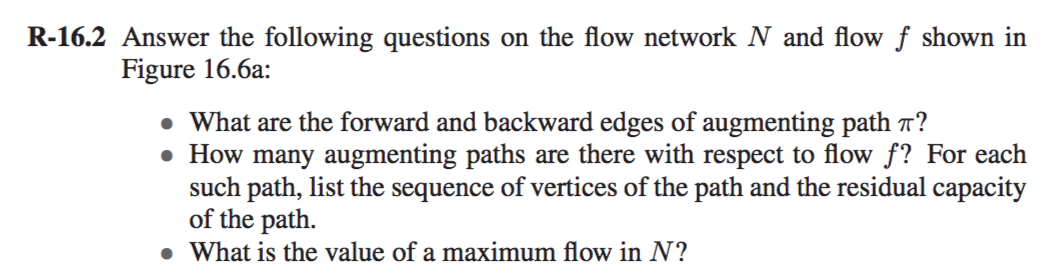
Tot 🡸 Dist[A] + length(A,B)

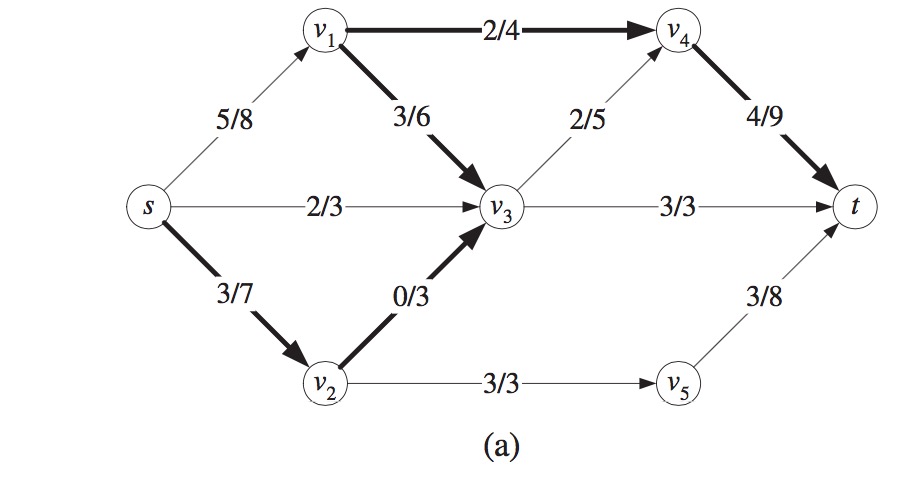
**If** tot< Dist[A] **do**

Dist[B] 🡸tot

//Alter the value of c in PQ to Dist[C]

Return Dist[A,B]





**Solution:**

1. The forward edges are as follows: -

[S,V2],[V2,V3],[V1,V4],[V4,T].

The backward edges are: -

[V1,V3]

1. There are two possible paths which can be at their full capacity namely [V3,T] and [V2,V5,T] and there exists a path to increase the flow which can be done by using the path [V4,T].

[S,V1,V3,V4,T]

[S,V1,V4,T]

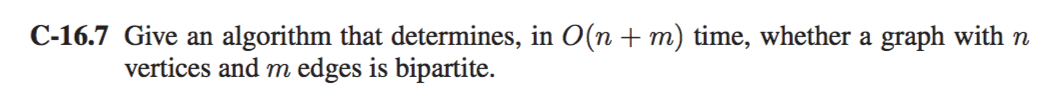
[S,V2,V3,V1,V4,T]

[S,V3,V4,T]

[S,V2,,V3,V4,T]

[S, V3,V4,T]

1. These paths [S,V1,V4,T] has a capacity of 2 and [S, V2, V3, V4, T] has a 3 capacity, They both have a total of 5 capacity but there are no augmenting path and the maximum flow would be 15.



**Solution:**

There are two ways to determine if a graph is Bipartite: -

1. A graph is bipartite if and only if it is colorable by 2 colors.
2. A graph is bipartite if it does not contain an odd cycle.

We perform BFS to the algorithm and color each vertices of the graphs with 2 colors red or blue. A vertex cannot have both the colors and then we check for the number of red and blue vertices if they are the same then the graph is bipartite.

**Algorithm DetermineBP(G):**

**Input:** A graph G with Vertices V.

**Output:** The graph G is bipartite or not.

BFS(G) //Perform BFS to the graph G

**While** (G is not completely parsed) **do**

**If** Vertex(u) == source

Color it Red and put them in a list U

**ForEach** neighboring vertices of U **do**

Color it Blue and put it into list V

**ForEach** neighboring neighbors of U **do**

Color it Red and put it into the list U

//The whole graph is colored now we check for the graph is bipartite

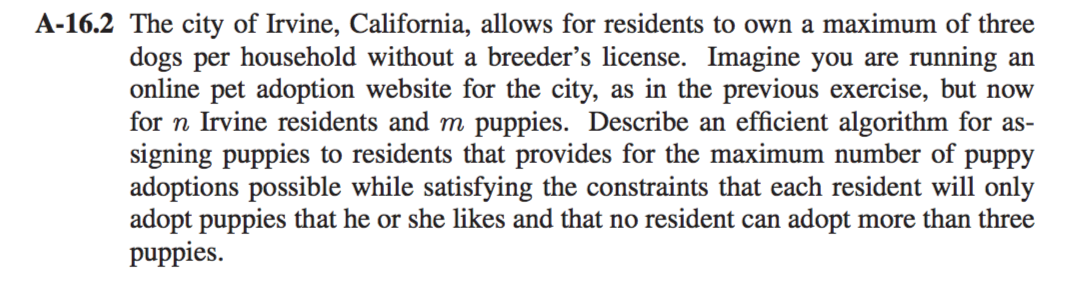
**if** (list(u).length == list(v).length) **do**

Display(The graph is Bipartite)

**Else do**

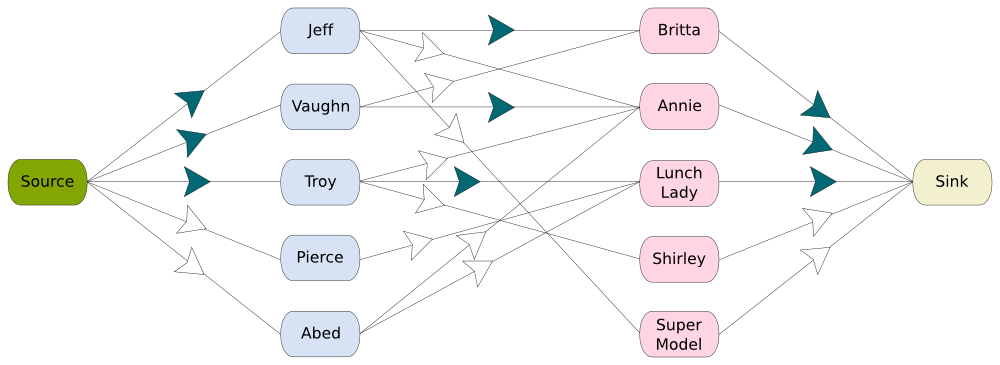
Display(The graph is not Bipartite)

Suppose the n vertices and m edges all the all algorithm has to do is to traverse the whole graph once so the run time for this algorithm would be **O (n+m)**.



**Solution:**

Consider a graph G which looks like a similar graph like this: -



Residents Puppies

We first define the source and we connect all the vertices of owners to the source vertex.

The input will be in the form of Edmonds matrix wherein it would be consisting of the n Irvine residents and m puppies. The matrix would contain like Sol[i][j] wherein ith is the owner would be 1 if the owner adopts the jth puppy or else 0. The output will be the number of residents that will get at maximum 3 puppies.

One of the ways we can try to implement this form is by using an Adjacent matrix representation of a directed graph which contains n+m+2 [2 because of source and sink] vertices and use the Ford Fulkerson algorithm for this matrix. This requires O((M+N)2) space.

Since the graph is bipartite and edge capacities are 1 or 0, The space can be reduced. We use graph traversals namely depth first search to first owners of the puppies which is as same as to augment a path.

We make use of the function bipartitmatching() which is a DFS function that tries all the possibilities to assign a puppy to a resident. The following is the algorithm

**Algorithm bipartitematching(boolean bpGraph[][], U, boolean seen[], int matchR[]):**

**Input:** A graph G with n residents and m puppies

**Output:** Assigned puppies to different residents and not more than 3 per resident

// **ForEach** puppy one by one

**for** v🡨0 to N **do**

         // If resident u is interested in puppy v and v is not visited

**if** (bpGraph[u][v] && !seen[v]) **do**

             // Mark v as visited

                 seen[v] = true

                  // If the puppy 'v' has not been assigned to a resident OR previously assigned resident for puppy v (which is matchR[v]) has an alternate puppy available. Since v is marked as visited in the above line, matchR[v] in the following recursive call will not get puppy 'v' again

**if** (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR)) **do**

                    matchR[v] = u

                    return true

**else do**

return false