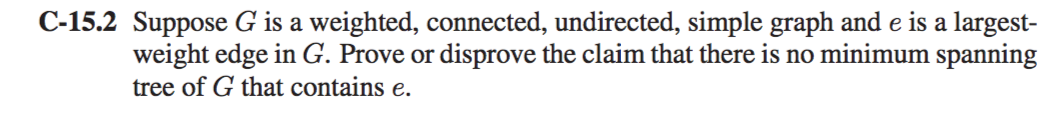
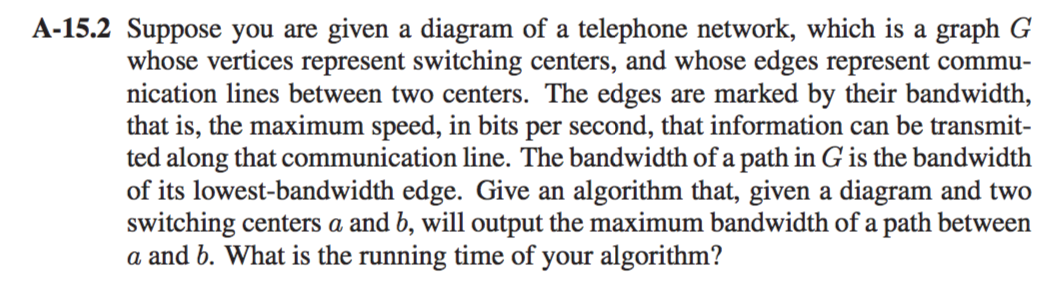
CS 600 Homework 8 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 11/1/2017



If G is already a tree (weighted | connected | undirected | simple graph), then the largest-weight edge must belong to its minimum spanning tree. Hence, we say that the claim is false & the claim that there is no minimum spanning tree of G that contains e is also false.



This can be Solved by modifying Dijkstra’s algorithm. Instead of representing the shortest path from a to u, the label D[u] represents the maximum bandwidth of any path from a to u. The maximum bandwith for path from a through u to a vertex z adjacent to u is min{D[u], w((u, z))} so that the relaxation step updates

D[z] to max{D[z], min{D[u], w((u, z))}}.

**algorithm maxBandwith (G,a,b):**

input: Weighted graph G and two distinguished vertices a and b

output: Maximum bandwith over all paths between a and b

initialize D[a] ← ∞ and D[u] ← 0 for each vertex u != a in G

let a priority queue Q contain all the vertices of G using the D labels as keys

while Q is not empty **do**

u ← Q.removeMaxElement()

if u = b then

return D[u]

else

for each vertex z adjacent to u such that z is in Q **do**

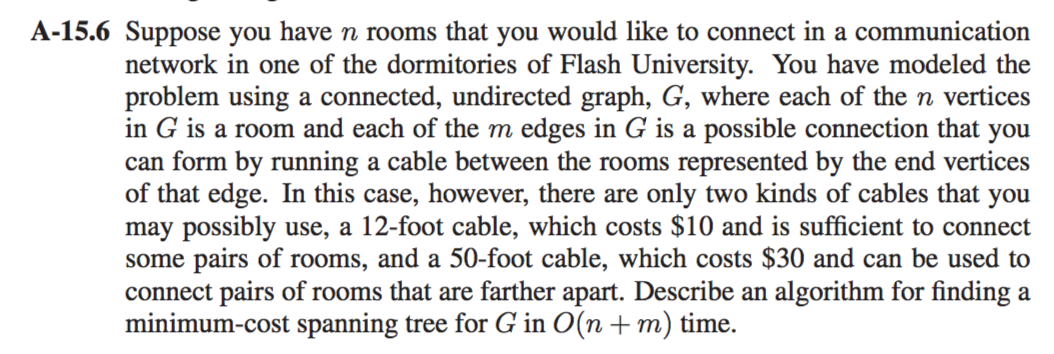
d ← min{D[u], w((u, z))}

if d>D[z] then

D[z] ← d

change the key value of z in Q to D[z]

Using an adjacency-list representation for the graph (chosen because of the need for efficient adjacent Vertices calls hidden in the for loop), the running time is the same as the running time of Dijkstra’s algorithm — O((n + m) log n) if the priority queue is implemented as a heap and O(n2) if the priority queue is implemented as an unsorted sequence.



1] We use the Prim-Jarn´ık algorithm with implementation of the priority queue, Q.

2] We implement the priority queue as two doubly linked lists, one for the items with cost 10 and the other for items with cost 30.

3] Insertions and updates can therefore be done in O(1) time, as can the removal of an item with smallest key.

**Algorithm PrimJarn ́ıkMST(G):**  
Input: A weighted connected graph G with n vertices and m edges   
Output: A minimum spanning tree T for G

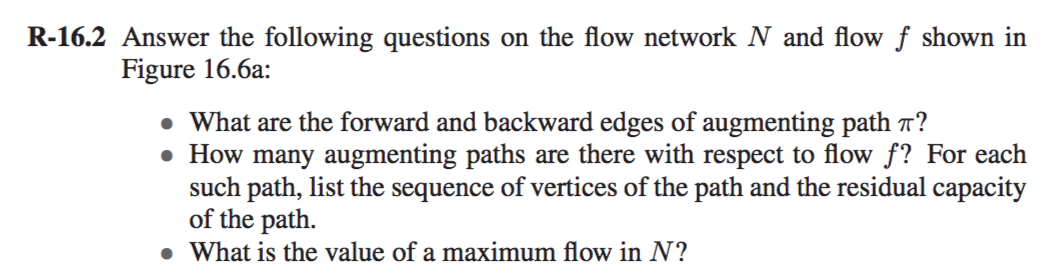
Pick any vertex v of G   
D[v] ← 0   
**for** each vertex u ̸= v **do**   
 D[u] ← +∞   
Initialize T ← ∅.   
Initialize a priority queue Q with an item ((u, null), D[u]) for each vertex u, where (u, null) is the element and D[u] is the key.   
**while** Q is not empty **do**

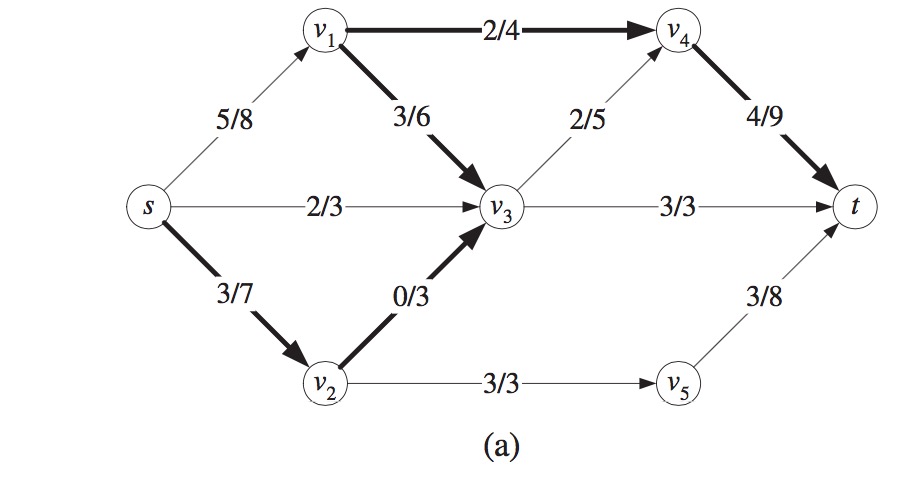
(u, e) ← Q.removeMin()   
Add vertex u and edge e to T.   
**for** each vertex z adjacent to u such that z is in Q **do**

// 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

Therefore, the running time of the algorithm is O(n + m).





a] The forward edges are (s, v2), (v2 ,v3), (v1,v4), and (v4,t).   
 The backward edge is (v1, v3).

b] The paths (V3, t) and (V2, V5, t) are at full capacity, so the only way to possibly increase the flow is through (V4, t). So, we can form augmenting paths using edges with capacity we have.

There are six **(POSSIBLE)** augmenting paths:

(S, V1, V4, t)

(S, V1, V3, V4, t)

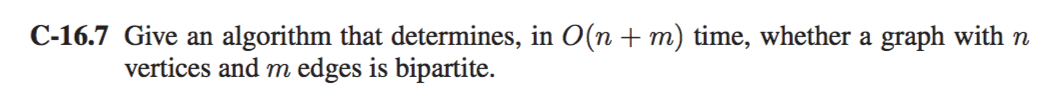
(S, V3, V4, t)

(S, V3, V1, V4, t)

(S, V2, V3, V4, t)

(S, V2, V3, V1, V4, t)

3] The augmenting paths (S, V1, V4, t) has a residual capacity of 2.   
 The augmenting path (S, V2, V3, V4, t) has a residual capacity of 3.   
 Hence these two paths add a total of 5 to the flow from V4 to t, which is also apparently the maximum that can run through.   
  
Therefore, there is no augmenting paths and the maximum flow is 15.



A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

To get a linear time algorithm to determine whether a graph is bipartite. The property says that an undirected graph is bipartite if it can be colored by two colors. The algorithm we present is a modified DFS that colors the graph using 2 colors. Whenever an back-edge, forward-edge or cross-edge is encountered, the algorithm checks whether 2-coloring still holds.

**function graph-coloring(G)**

**Input**: Graph G Output: returns true if the graph is bipartite, false otherwise

for all v ∈ V:

visited(v)= false color(v) = GREY

while ∃s ∈ V : visited(s) = f alse

visited(s) = true

color(s) = WHITE

S = [s] (stack containing v)

while S is not empty

u = pop(S)

for all edges (u,v) ∈ E:

if visited(v) = false:

visited[v] = true

push(S,v)

if color(v) = GREY

if color(u) = BLACK:

color(v) = WHITE

if color(u) = WHITE:

color(v) = BLACK

else if color(v) = WHITE:

if color(u) 6= BLACK:

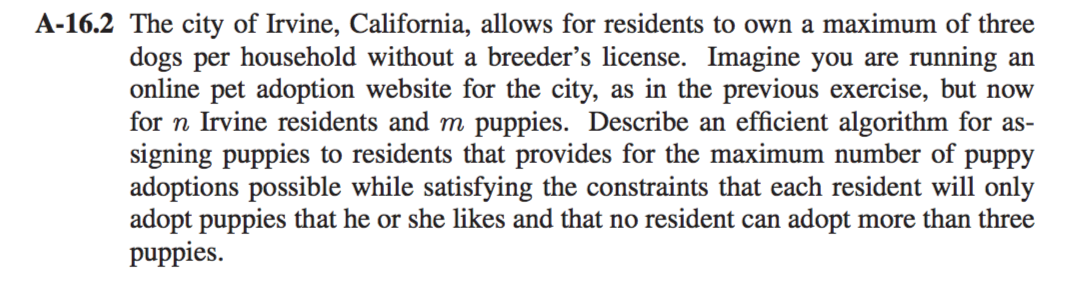
return false

else if color(v) = BLACK:

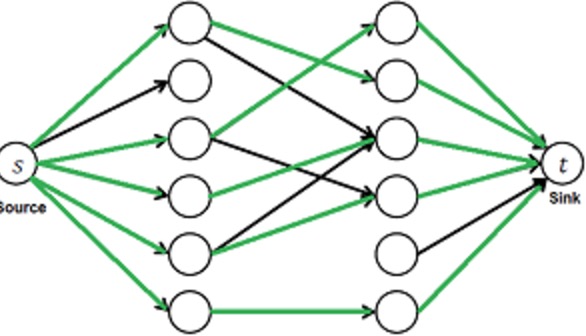
if color(u) 6= WHITE:

return false

return true



Solution:

This is a maximum bipartite matching problem.  


Owners Dogs

A] Given bipartite graph G = (A ∪ B, E), direct the edges from A to B.

B] Add new vertices s and t.

C] Add an edge from s to every vertex in A.

D] Add an edge from every vertex in B to t

E] Make all the capacities 1.

F] Solve maximum network flow problem on this new graph G’

1. We first define input and output forms. The Input is in form of [Edmonds matrix](http://en.wikipedia.org/wiki/Edmonds_matrix) which is a 2D array with M rows (for M Owners) and N columns (for N Dogs).   
   The value Graph[i][j] is 1 if i’th owner is interested in j’th dog, otherwise 0.
2. Output is number maximum number of people that can get at max 3 dogs.
3. A simple way to implement this is to create a matrix that represents [adjacency matrix representation](http://www.geeksforgeeks.org/graph-and-its-representations/)of a directed graph with M+N+2 vertices. Call the [fordFulkerson()](http://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) for the matrix. This implementation requires O((M+N)\*(M+N)) extra space.
4. Extra space can be reduced and code can be simplified using the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a owner for an dog (similar to augmenting path in Ford-Fulkerson). We call bpm() for every owner, bpm() is the DFS based function that tries all possibilities to assign a dog to the owner.
5. In bpm(), we one by one try all dogs that an owner ‘u’ is interested in until we find a dog.
6. For every dog we try, we do following:  
   If a dog is not assigned to anybody, we simply assign it to the owner and return true. If a dog is assigned to somebody else say x, then we recursively check whether x can be assigned some other dog. To make sure that x doesn’t get the same dog again, we mark the dog ‘v’ as seen before we make recursive call for x. If x can get other dog, we change the owner for dog ‘v’ and return true. We use an array maxR[0..N-1] that stores the owners assigned to different dogs.
7. If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

**Note: This problem can also be solved by BFS.**

**Short Explanation:**

1] Begin from some random vertex and apply BFS.  
2] The graph is not bipartite if there are any non-tree edges joining vertices on the very same level.  
3] After performing this, if there is an unvisited vertex, v, we repeat this algorithm with v.  
4] We keep repeating this process until we have determined that the graph is not bipartite or if we have visited all its vertices.  
5] If we have visited all its vertices and found no odd-length cycles, then the graph is bipartite.

Because we are performing a BFS on each connected component, the running time is   
**O(n + m)**.