**Chapter 15**

**C-15.2** Suppose *G* is a weighted, connected, undirected, simple graph and *e* is a largest weight

edge in *G*. Prove or disprove the claim that there is no minimum spanning tree of *G* that contains *e*.

**Answer:**

If G is weighted, connected, undirected, simple graph, then the largest weight edge must belong to its minimum spanning tree. Hence the claim is false and the claim that there is no minimum spanning tree of G that contains e is also false.

Let’s prove:

Let the Statement be: Suppose S be the minimum spanning tree that has edge e

We consider a new graph S that is created by adding edge e to the graph. S has cycle that consist of edge e

Now to find the minimum spanning tree, the edge that has the maximum weight in the cycle is excluded. Here e has the maximum weight. Edge e is removed and the resulting spanning tree be S‘’ (prime) so S ‘’ prime is left than weight S because the removed edge has more weight than the edge that is added. Since here S’’ prime has minimum weight than s, S’’ prime is the minimum spanning tree, which is the contradiction the statement mentioned above. Therefor minimum spanning tree does not contain edge e.

**A-15.2** Suppose you are given a diagram of a telephone network, which is a graph *G* whose vertices represent switching centers, and whose edges represent communication lines between two centers. The edges are marked by their bandwidth, that is, the maximum speed, in bits per second, that information can be transmitted along that communication line. The bandwidth of a path in*G* is the bandwidth of its lowest-bandwidth edge. Give an algorithm that, given a diagram and two switching centers *a* and *b*, will output the maximum bandwidth of a path between *a* and *b*. What is the running time of your algorithm?

**Answer:**

This can be accomplished by modifying Dijkstra’s algorithm. Here, 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 bandwidth 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)}}

The Algorithm is maximumBandwidth (G, a, b):

Here the input is the weighted graph G and two distinguished vertices a & b, and the output is maximum Bandwidth 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.removeMaximumElement()

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, 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.

**A-15.6** Suppose you have *n* rooms that you would like to connect in a communication network in one of the dormitories of Flash University. You have modeled the problem using a connected, undirected graph, *G*, where each of the *n* vertices in *G* is a room and each of the *m* edges in *G* is a possible connection that you can form by running a cable between the rooms represented by the end vertices of that edge. In this case, however, there are only two kinds of cables that you may possibly use, a 12-foot cable, which costs $10 and is sufficient to connect some pairs of rooms, and a 50-foot cable, which costs $30 and can be used to connect pairs of rooms that are farther apart. Describe an algorithm for finding a minimum-cost spanning tree for *G* in *O* (*n* + *m*) time.

**Answer:**

Here Prim-Jarnik algorithm is used where the input is a weighted connected graph G with n vertices and m edges and the output is a minimum spanning tree T for G.

Pick any vertex v of G

D[v] ← +∞

Initialize T ← ∅

Initialize a priority queue 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

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

**Chapter 16**

**R-16.2** Answer the following questions on the flow network *N* and flow *f* shown in

Figure 16.6a:

*•* What are the forward and backward edges of augmenting path *π*?

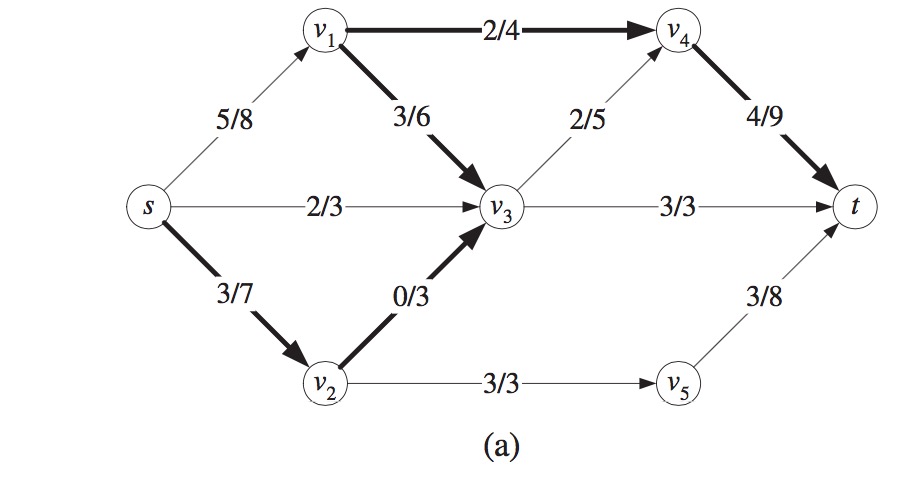
*•* How many augmenting paths are there with respect to flow *f*? For each

such path, list the sequence of vertices of the path and the residual capacity

of the path.

*•* What is the value of a maximum flow in *N*?

**Answer:**



1. The forward edges are (s, v2), (v2, v3), (v1, v4) and (v4, t)

The backward edge is (v1, v3)

1. 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: six possible augmenting paths:

(S, v1, v4, t)

(S, v1, v3, v4, t)

(S, v3, v4, t)

(S, v2, v3, v4, t)

(S, v2, v3, v1, v4, t)

1. The augmenting paths (S, v1, v4, t) has a residual capacity of 2.

The augmenting paths (S, v1, v3, v4, t) has a residual capacity of 3.

Hence, these two paths add a total of 2 +3 = 5 to the flow from v4 to t, which is also apparently maximum that can run through. Therefore, there is no augmenting paths and the maximum flow is 15.

**C-16.7** Give an algorithm that determines, in *O* (*n* + *m*) time, whether a graph with *n*

vertices and *m* edges are bipartite.

**Answer:**

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V where every edge (u, v) either connects a vertex from U to V or vice versa. So, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. 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 is a modified DFS that colors the graph using 2 colors. Whenever a back-edge, forward-edge or cross-edge is encountered, the algorithm checks whether 2-coloring still holds.

Function graph-coloring(G) where input is Graph G and the 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) = false

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

**A-16.2** The city of Irvine, California, allows for residents to own a maximum of three dogs per household without a breeder’s license. Imagine you are running an online pet adoption website for the city, as in the previous exercise, but now for *n* Irvine residents and *m* puppies. Describe an efficient algorithm for assigning puppies to residents that provides for the maximum number of puppy adoptions possible while satisfying the constraints that each resident will only adopt puppies that he or she likes and that no resident can adopt more than three puppies.

**Answer:**

Given bipartite graph G = (A U B, E) direct the edges from a to be. New vertices s and t are added and an edge from s to every vertex in A is added. An Edge from every vertex in B to t is also added. All the capacities are made 1 and maximum network flow problem on the graph G’ Prime is solved.

So, first input and output are defined. Input is form of Edmonds Matrix which is a 2D array with M rows and N columns, Owners and dogs (M and N). The value Graph [i][j] is 1 if ith owner is interested in jth dog, otherwise 0. Output is maximum number of people that can get a max 3 dogs. A simple method to implement this is to create a matrix that represents adjacency matrix representation of a directed graph with M + N + 2 vertices. The fordFulkerson() is called for the matrix. This implementation required O ((M+N) \* (M+N)) extra space. That extra space can be reduced, and code can be simplified because the graph is bipartite, and capacity of every edge is either 0 or 1. Here the DFS traversal to find an owner for a dog is done. Bipartite Matching Problem Bpm() is called for every owner. It is a DFS based function that tries all possibilities to assign a dog to the owner. In this method one by one all dogs are tried to an owner ‘u’ is interested until a dog is found.

So, if the dog is not assigned to anybody, it is assigned to the owner and return true. If a dog is assigned to somebody else say x, then recursively it is checked whether x can be assigned some other dog. TO make sure that x does not get the same dog again, the dog v is marked before recursive call. If x can get another dog we change the owner for dog v and return true. We can use array MaxR[0…n-1]. If Bmp() is true then it just means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBpm().