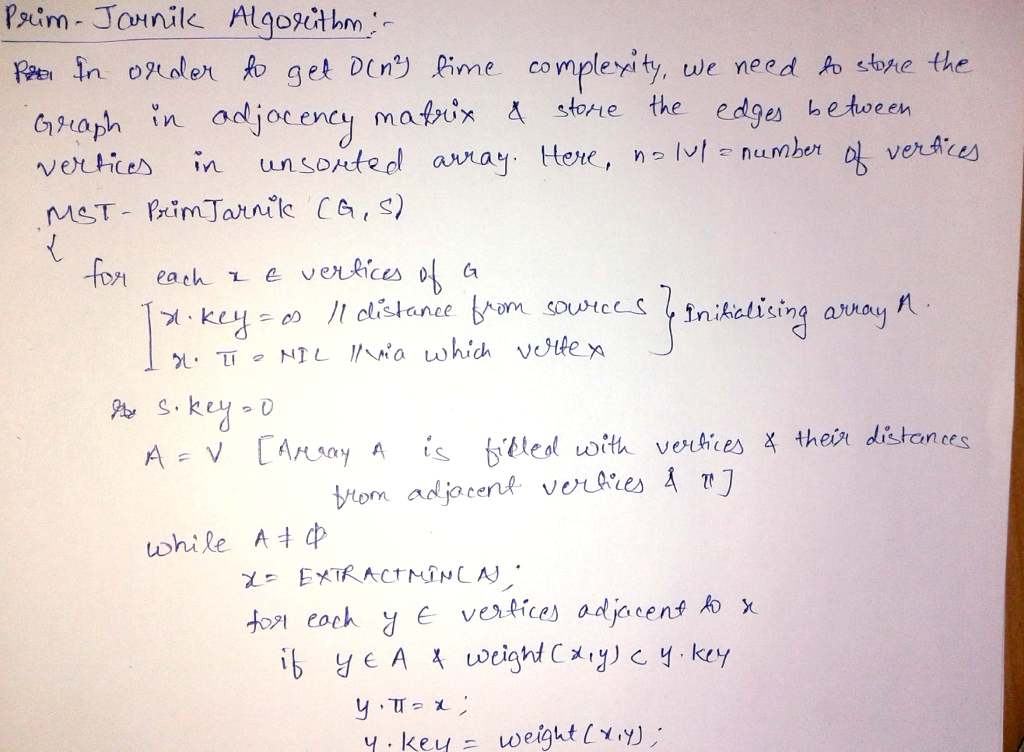
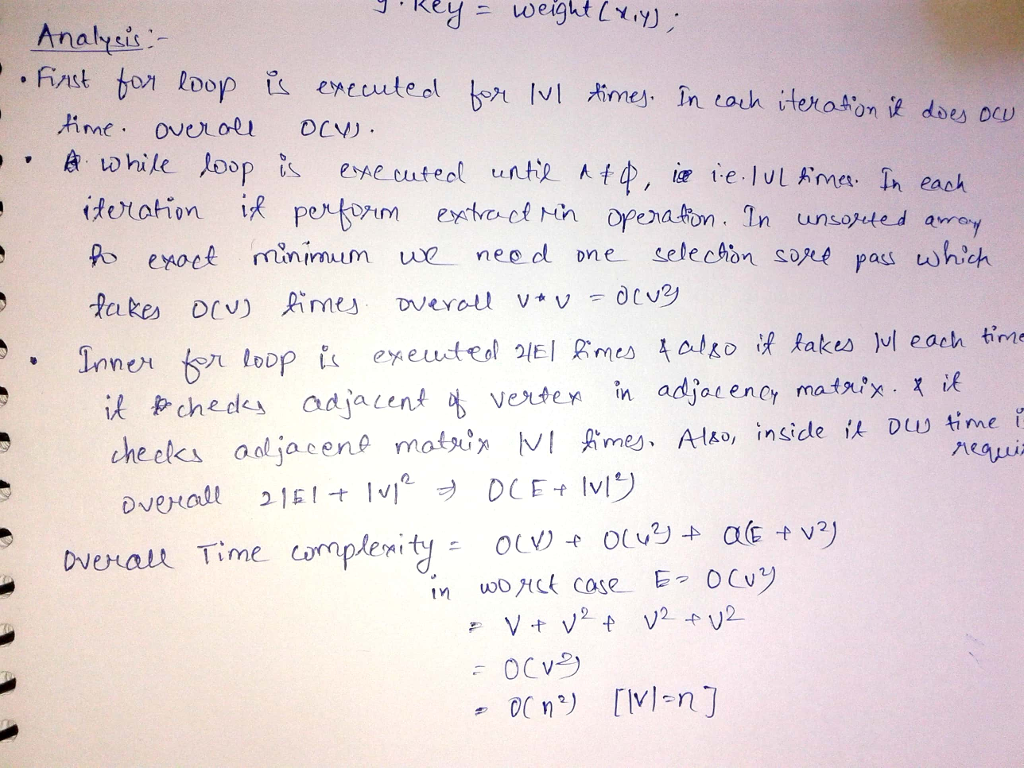
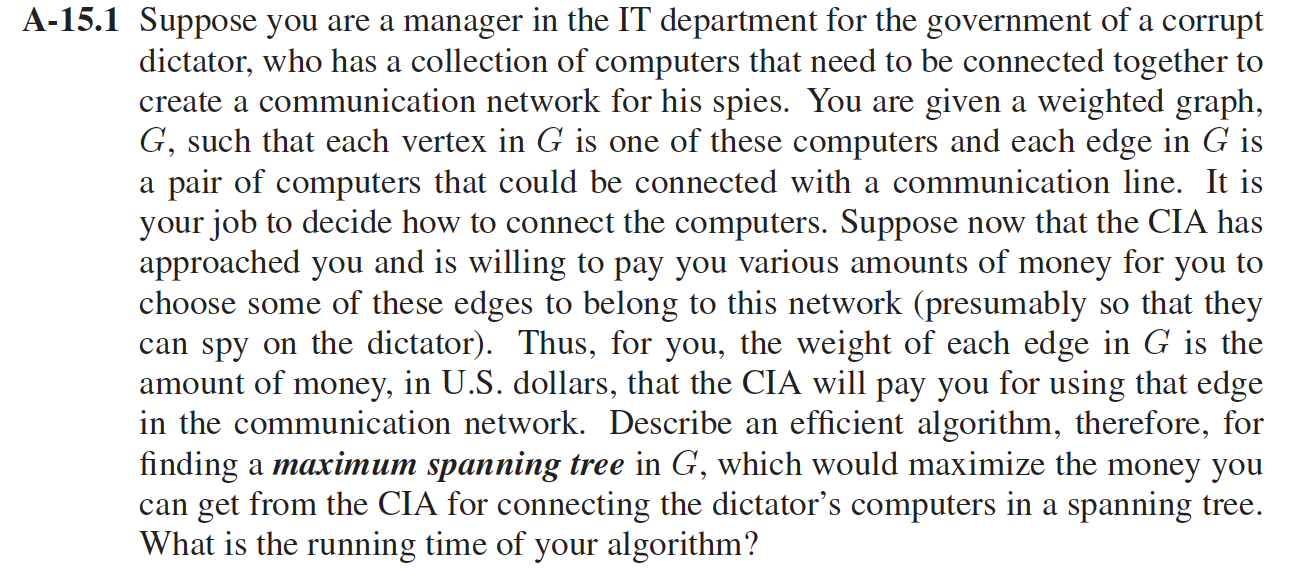
Chapter 15 Exercises: C-15.6,

Show how to modify the Prim-Jarn.ık algorithm to run in O(n2) time.





A-15.1,



In order to find a maximum spanning tree, we should

1. Negate all edge weights and then apply the MST algorithm rule. That is , multiply the negative value(-1) to all edge weights.
2. Apply Kruskal’s algorithm to find the minum spanning tree.
3. The result of minimum spanning tree is the maximum spanning tree of the graph.

Algorithm:

//Initialize the vertex and edge for gragh:

Set of vertices vertex and set of edges ed

For edge in ed:

Edge.We=- edge.We

Mst = kruskals(vertex,ed)

//store the edge “ed” to “Max” variable

Max=ed

End loop until the minimum edges in minimum spanning tree “mst”

For edge in mst.edges:

Max.remove(edge)

Return the Max variable which contains the maximum edge weight.

End loop

Explanation:

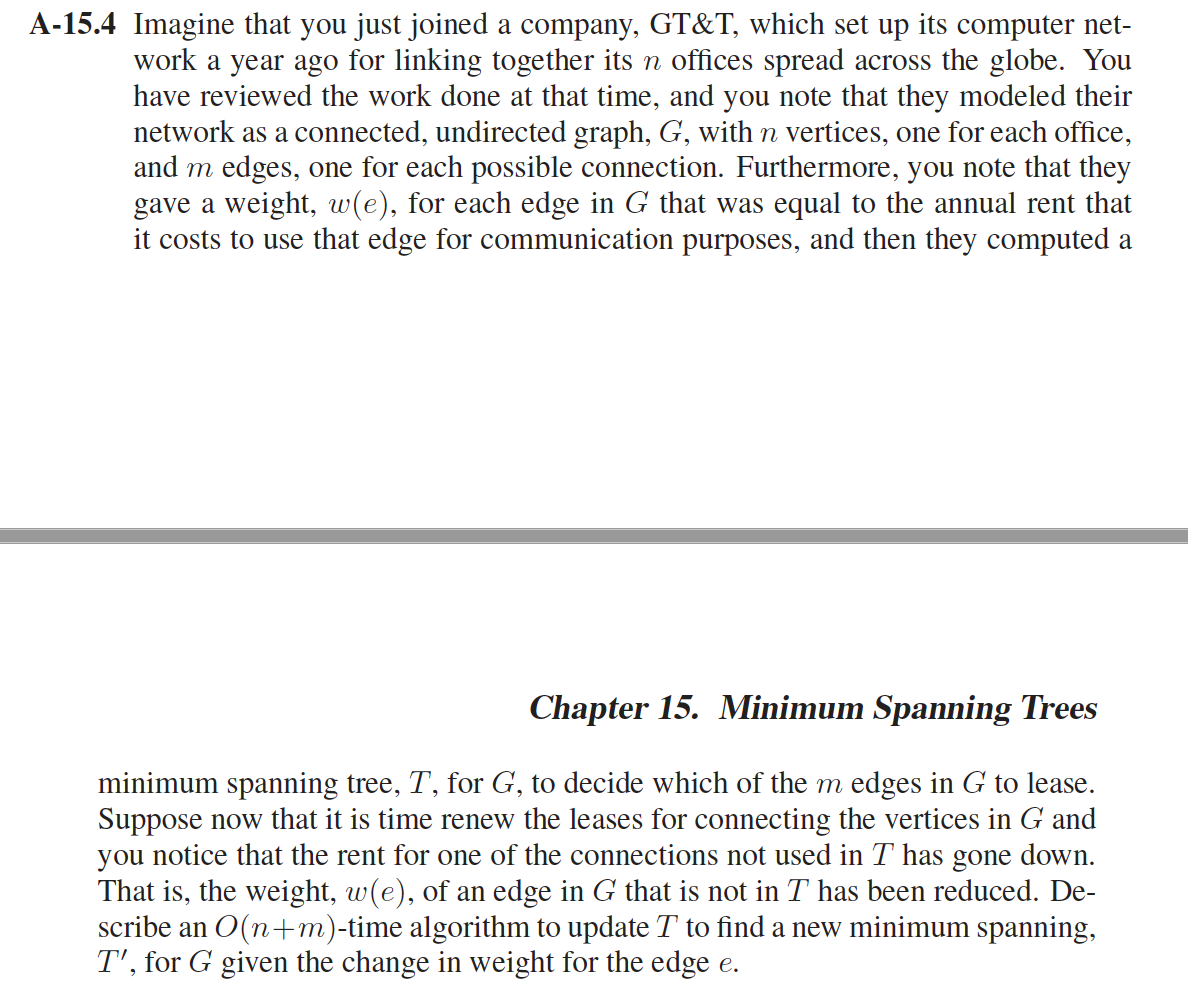
* G indicates the gragh without edge set.
* Edge set must contain the property “if any edges are added into gragh G then the graph must from the cycle with that edge” because ,all edges in graph are positive weights.
* Thus, the graph G contains the set number of edges with weights in order to get the minimum total edge weights for edge set and graph G contains the maximum total edge weights.

Complexity:

* Negating all the edge weights takes the running time of O(E).
* Use of Kruskal’s algorithm takes the running time of the O(ElogV).
* Finally, determining all edge weights that is not present in the minimum spanning tree takes the time of O(E). So ,it takes the running time of the O(ElogV).

Therefor the overall running time for maximum spanning tree is O(ElogV).

A-15.4

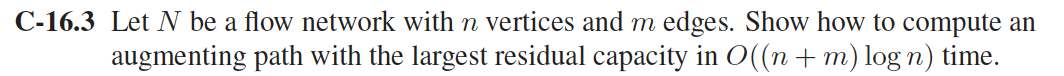


Let e = (u, v) and let Tu and Tv be the subtrees obtained by removing e.

By doing BFS (ignoring edge weights) from u and from v , we can determine which vertices are in Tu and which are in Tv in time O(m + n).

Assume we have marked each node with its membership. Now examine each edge, and keep the minimum weight edge e′ with one endpoint in Tu and the other in Tv. This can be done in O(m) time. The total time is thus O(m+n).

Chapter 16 Exercises: C-16.3,



Use the capacity-scaling method to choose augmenting path with highest residual capacity.

* Maintain the scaling parameter Δ.
* If capacity-scaling algorithm terminates, then f is a max-flow.

CAPACITY-SCALING(G, s, t, c)

FOREACH edge e ∈ E : f(e) ← 0.

Δ ← largest power of 2 ≤ C.

WHILE (Δ ≥ 1)

Gf (Δ) ← Δ-residual graph.

WHILE (there exists an augmenting path P in Gf (Δ))

f ← AUGMENT (f, c, P).

Update Gf (Δ).

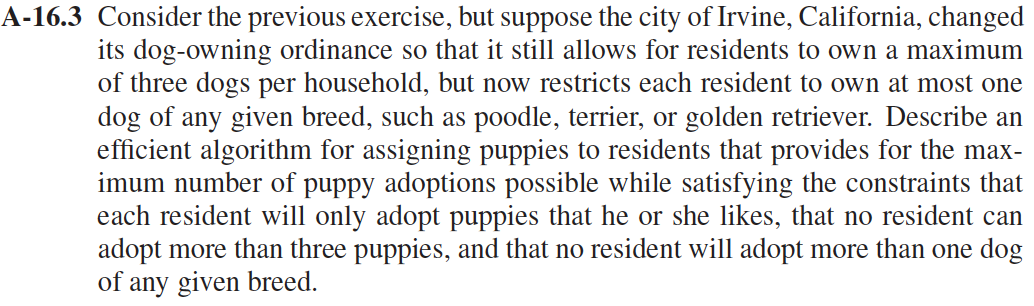
Δ ← Δ / 2.

RETURN f.

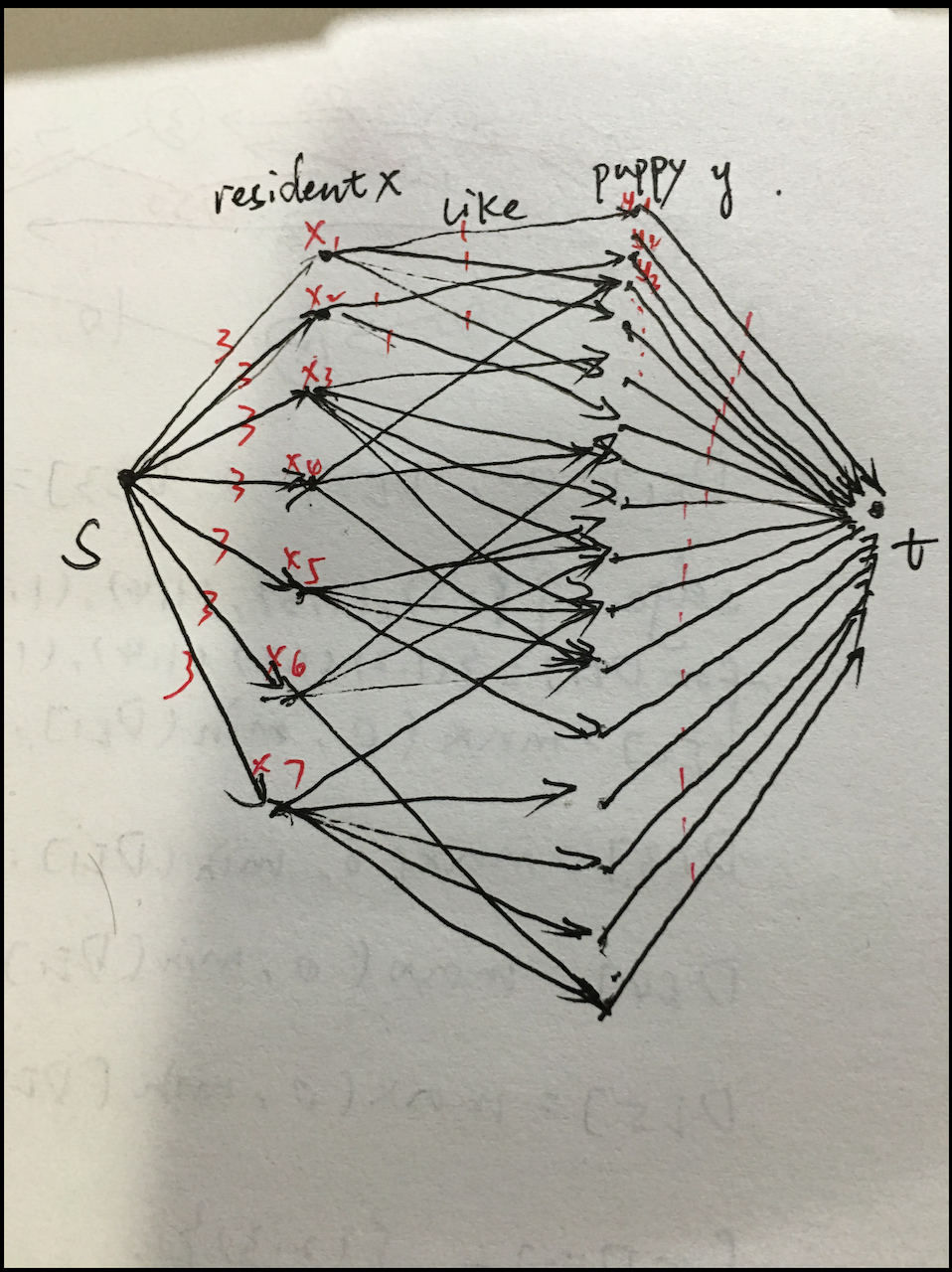
Explanation :

The outer while loop repeats 1+ logC times. And there are at most m+n augmentations per scaling phase. So it can be implemented to run in O((m+n)logn) time.

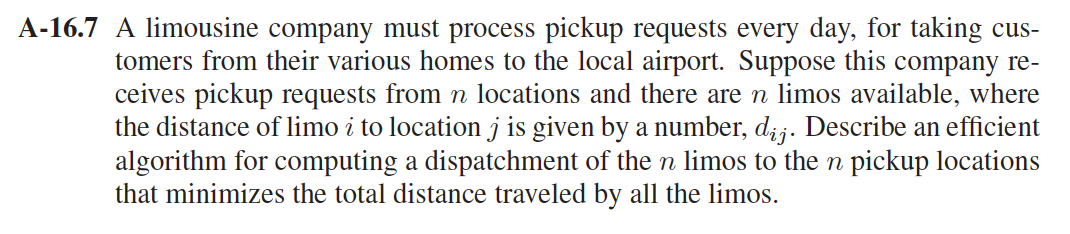
A-16.3,



As the Reduction to the Maximum Flow Problem showed in page 458 of the textbook, we can build a bipartite H. We can view residents as set X, puppies as set Y and “resident x want to adopt puppy y” as the edge between X and Y. And then we add a new source vertex s and sink vertex t. In addition, we insert a directed edge from s to each vertex in X, and a directed edge from each vertex in Y to t. Finally, we assign to every edge connected to s a capacity of 3 and every edge connected to t a capacity of 1. Finally, we use a maximum flow algorithm on this network. We can get the maximum puppies can be adopted and the which resident should adopt which puppy at the end.



A-16.7



We need to find the shortest path to each of the n locations of the limos from the given location, Hence this is the single source shortest path problem for computing a minimum cost flow.

We can view limos as set X, locations as set Y and “the distance of limo I to location j” as the edge between X and Y. And then we add a new source vertex s and sink vertex t. In addition, we insert a directed edge from s to each vertex in X, and a directed edge from each vertex in Y to t.

Hence we will use MinCostFlow Algorithm to this graph.

Algorithm MinCostFlow(N):

Input: Weighted flow network N = (G, c,w, s, t)

Output: A maximum flow with minimum cost f for N

for each edge e ∈ N do

f(e) ← 0

for each vertex v ∈ N do

d(v) ← 0

stop ← false

repeat

compute the weighted residual network Rf

for each edge (u, v) ∈ Rf do

w \_(u, v) ← w(u, v) + d(u) − d(v)

run Dijkstra’s algorithm on Rf using the weights w \_

for each vertex v ∈ N do

d(v) ←distance of v from s in Rf

if d(t) < +∞then

// π is an augmenting path with respect to f

// Compute the residual capacity Δf (π) of π

Δ ← +∞

for each edge e ∈ π do

if Δf (e) < Δ then

Δ ← Δf (e)

// PushΔ = Δf (π) units of flow along path π

for each edge e ∈ π do

if e is a forward edge then

f(e) ← f(e) + Δ

else

f(e) ← f(e) − Δ // e is a backward edge

else

stop ← true // f is a maximum flow of minimum cost

until stop

Complexity:

the running time of this minimum-cost maximum flow f with 2n+2 vertices and m edges can be computed in O(mlog2n) time