Yash Avinash Patole

Course: CS600

Homework 7

CWID: 10460520

Chapter 15

Exercise 15.6.16

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

Answer:

In order to get D(n^2) time complexity, we need to store the graph in adjacency matrix and store the edges between vertices in unsorted array. Here, n = |v| = number of vertices MST-PrimJarnik(G, S).

{

for each x ∈ vertices of G

[x key = ∞ distance from sources]

[x π = NIL VIA WHICH VERTEX]

Initializing arrays in above steps.

S.key = 0

A = V [Array A is filled with vertices and their distance from adjacent vertices & π]

While A is not 0

X = Extract min (A)

For each y ∈ vertices adjacent to x

If y ∈ A and weight (x, y) < y.key

y. π = x

y.key = weight(x, y)

Explaination:

The first for loop is executed |V| times, with each iteration doing O(t) time overate O. (v).

The while loop is executed until A is not present, which is |v| times. It performs the extraction of min operation in each iteration. To find the smallest element in an unsorted array, we need to make one selection pass, which takes O(v) time overall $v*v = O(v^2)$

Inner for loop is executed 2|E| times and also it takes |v| each time it checks adjacent of vertex in adjacency matrix. It checks adjacent matrix |v| times. Therefore, $2|E| +|v^2| \rightarrow O(E +|v^2|)$

Total time complexity =

$O(v) + O(v^2) + O(E+V^2)$

While worst case $E = O(v^2)$

$= V + V^2 + V^2 + V^2$

$= O(v^2)$

$= O(n^2)$ considering, |v| = n

Exercise 15.6.22

Answer:

We can use Kruskal's algorithm to generate a maximum spanning tree that maximizes the amount of money we can obtain from the CIA. Rather than sorting the edges in order of minimum weight, we should sort them in order of maximum weight.

Efficient Algorithm to maximize spanning tree in G is as follows,

cs600.py > main

```python
import os
import sys

"""Maximum Spanning Tree is the Minimum Spanning Tree with negative weights."""

def main():
    inFile = input("\nEnter input file name: ")
    if not os.path.isfile(inFile):
        print ('File not found.Exiting...')
        sys.exit(1)

    #Reading file
    inp = open(inFile, 'r')
    table = {}
    tree = []
    weight = 0
    for line in inp.readlines():
        temp = line.split(' ')
        weight = 0 - int(temp[2])
        if weight != 0:
            table[tuple(temp[:2])] = weight

    weight = kruskal(table, tree)
    print("The maximum spanning tree:")
    print (tree)
    print ("The maximum benefit is: %d" % weight)

```

```python
def valid(key, tree):
    """In the new edge 'key,' it creates a temporary list for each vertex v,
    storing all the vertices with which v is associated in the tree's edges.
    It will check for any common vertex after creating lists for both edges in the edge.
    If there is a shared vertex, this new edge will form a circuit in the tree.
    If no common vertex is found, returns true; otherwise, returns false."""
    temp1 = []
    temp2 = []
    for k in tree:#for each edge in tree
        if key[0] in list(k):#if vertex v is present in the edge
            for i in list(k):#for each vertex in that edge
                if i != key[0]:#storing the other vertex in list temp1
                    temp1.append(i)
        #similarly for another vertex in the new edge
        if key[1] in list(k):
            for i in list(k):
                if i != key[1]:
                    temp2.append(i)
        #Checking for common vertex in the lists
        for i in temp1:
            for j in temp2:
                if i == j:
                    return False
        return True
```

```
def kruskal(table, tree):
    #Using kruskal's algorithm to find minimum spanning tree
    i = 0
    weight = 0
    v = int(input("\nNo. of vertices: "))
    #Sorting and finding edges for spanning tree
    for key, value in sorted(table.items()):
        print value
        if valid(key, tree):
            tree.append(key)
            weight += (0 - value) #Calculating weight
            i += 1
        if i == (v - 1):#Exit condition - if tree has n-1 edges
            break
    return weight


if __name__ == "__main__":
    main()
```

The time complexity will be O (Elog V) for the graph G with E edges and V vertices.

Exercise 15.6.25

Answer:

long long MinimumspanTree (int x 1)

// sets the priority Queue

priority queue<PII, vector<PII>,

greater<PII> > QVa1;

// Declare variables

int YI;

long long MinTree = 0

PII per;

// call the make _ pair ( )

```
QVa1. push (ma ke_pair (0 , xl));
/ / while checks empty ( )
while ( ! QVa1. empty ( )
{
        per = QVa1. pop ( ) ;
        x1 = per . second;
        if (marked [x 1] == true)
        continue;
        MinTree += per. first;
        marked [XI] = true;
        for (int it = 0; it < adj [x 1]. size ();++it)
        {
                Y1 = adj [XIJ (it] . second;
                if (marked [YI] false)
                QVaI . push (adj [XIJ [it]) ;
        }
        Return MinTree;
}
```

Explanation:

We replace the edges in T with edge e one by one in decreasing order of weight and try to create a new path in the tree, thus creating T. Iterating over all the edges takes O(n) time, and creating a new path takes O(m) time, so this can be done in O(n + m) time.

Chapter 16

Exercise 16.7.19

Answer:

So, in the residual graph G=(V,E), we will try to find the path with the highest resi dual capacity to every vertex in V from source s.

In the same way that d[v] in the Dijkstra algorithm indicates the shortest weighted path from s to v, d[v] in this case indicates the largest residual capacity in the path from s to v.

So we'll set d[v] = 0 and d[s] = INFINITY, and the updating equation will be d[v] = min(d[u], c(u,v)) if d[v] = min(d[u], w(u,v) if d[v] = min(d[u], w(u,v) if d[v] = m in(d[u], w(u,v). As a result, we will store all nodes in the priority queue Q, just like in the Dijkstra method, and the vertex with the greatest value d[v] will have the highest priority, while p[v] indicates the parent node of v in the path from s to v.

Consider the flow network N, which contains n vertices and m edges.

- The residual capacity of an edge (u,v) on an augmenting path is the amount of flow that can be increased on that edge (u,v) without breaching the capacity limitation. Increasing the flow of the network by adding residual capabilities to edges along the augmenting path.
- To discover the enhancing road with the greatest residual capacity (i.e., the path with the most flow or fattest path), the residual capacities must be minimized.
- If cm is the minimum of the residual capacities of the edges along the augmenting path p, then cm is the maximum or largest residual capacity that may be increased on each edge in the augmenting path.
- An algorithm similar to Dijkstra's method will assist us in determining the minimum residual capacity. Dijkstra's algorithm, in general, determines the shortest path. That is, the path's edge lengths must be as short as possible.
- Our modified Dijkstra's algorithm discovers the path's edges with the lowest residual capacity (cmin). On each edge of the augmenting path, cmin will be the greatest or largest residual capacity that can be raised.

The algorithm is as follows:

Algorithm Dijkstra-LRCPath(G,s)

Input: Graph G=(V,E) and s belongs to V. Where I Vl=n , IEI =m and each edge has a non-negative edge capacities.

Output: An augmenting path p with largest residual capacity.

```
for each v belongs to V-{s}
        v.d= INFINITY
s.d=0
while Q is not empty
        for all neighbours v of u:
                if v.d< min{u.d, cÅu,v)} //edge (u,v)
                        v.d min{ u.d, cÅu,v)}
                        u.pred=v
```

Running time explained as follows:

- The improved version takes the same amount of time as Dijkstra's algorithm.
- The algorithm performs n insert operations to insert n vertices into the queue, n extract-min operations, and m reduce-key operations. If a binary heap is used as the data structure, each of these operations requires O(log n) time.

Hence, the total running time of this algorithm mentioned will be O((n+m) log n).

Exercise 16.7.30

Answer:

Similarly, to before, we will establish a set of vertices P of size n where each vertex represents a pet and a set R of residents of Irvine of size n where each vertex represents a citizen of Irvine city.

Create s and t nodes to represent the source and target nodes, respectively.

To ensure that a pet is assigned to only one person, establish an edge from s to every vertex in P with capacity 1, ensuring that no more than one unit flow passes from a pet vertex and so a pet is assigned to only one citizen.

To ensure that no citizen adopts more than three pets, establish an edge from every vertex in set R to vertex t with a capacity of three units to ensure that no citizen adopts more than three pets. To ensure that every citizen adopts a pet that he or she likes, we will make a change. Previously, we were constructing an edge from a pet to a citizen if the citizen liked the pet, but now we must ensure that each citizen gets

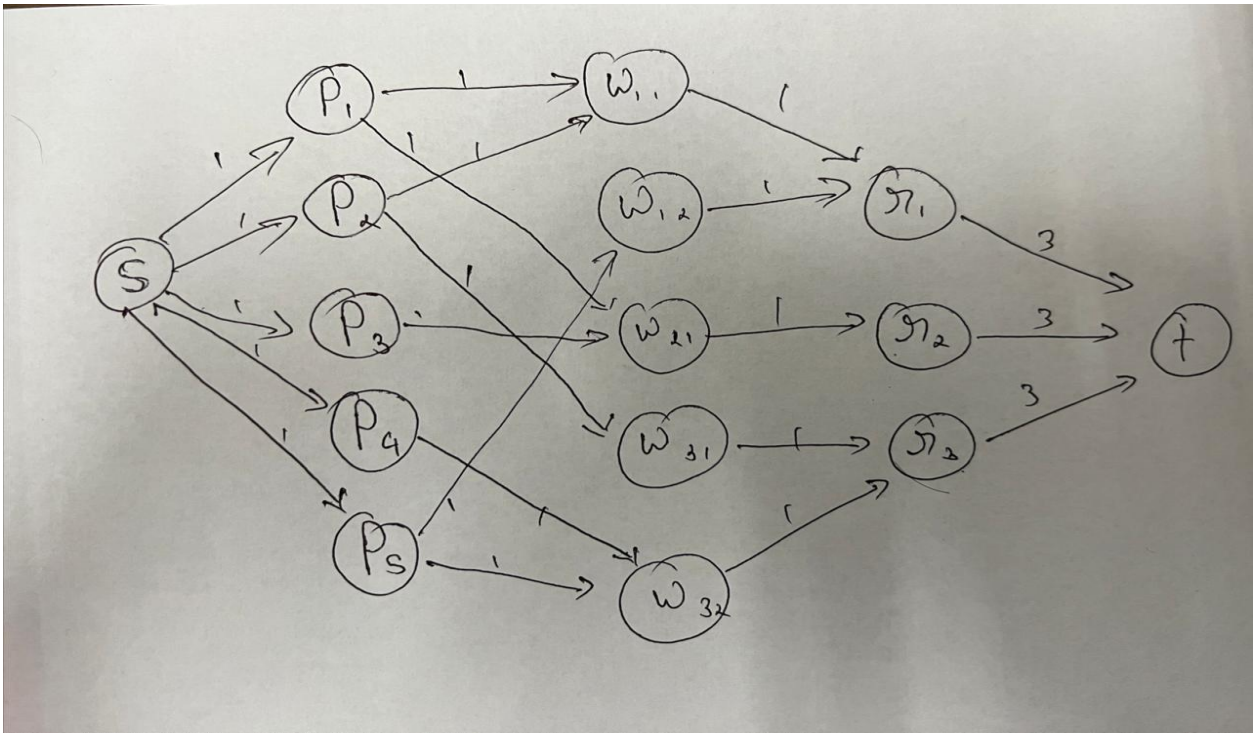
at least one pet of a specific species. Instead of building an edge (u,v) from pet u to citizen v, we will create sv number of intermediary nodes for each citizen v, where sv is the number of pet species that citizen v likes.

Thus, if citizen v likes 5 different species of pets, we will create 5 intermediate nodes for citizen v and 5 vertices w1,w2,w3,w4, and w5 for citizen v, where each vertex from w1 to w5 denotes different species of dogs liked by v and there will be an each from w1 to w5 towards vertex v with capacity 1 unit to ensure that citizen v does not receive more than 1 dog of particular species. So, there will be a capacity 1 unit edge from vertex u in set P to vertex w if pet u is in species w and u is liked by citizen v.

If pet u is in species w and preferred by resident v, there will be an edge of limit 1 unit from vertex u in set P to vertex w.

So, for each resident v, if he has a pet u from one of the animal groups w, we will build a limit 1 unit edge (u,w) and a limit 1 unit edge (w,v).

Let us look at an example. There could be 5 pets pl, p2, p3, p4 and p5, with p1,p2,p3 belonging to one animal group and p4,p5 belonging to another. Furthermore, suppose there are three residents r1,r2, and r3, and r1 likes p1,p2,p5, r2 likes pl, p3, and r3 likes p2, p4, p5, at that time the stream organize is as follows.



Exercise 16.7.34

Answer:

Before making any other computations, Floyd-Warshall algorithms should be employed to determine the shortest distance between each location and every other site. Once the shortest distance matrix has been produced, the next step is to discover the permutation of sites that results in the shortest total distance traveled by all of the limos. This can be performed by the use of a dynamic programming approach, in which dp[i][j] specifies the minimum cost to dispatch limo I to position j, assuming that all earlier locations were allocated a limo in the previous iteration.

These are the base instances, which are represented by I = O or j = O, respectively, indicating that either no locations have been assigned yet or that there are no limos left to assign.

Following shows the explanation of the respective solution:

- Using a Floyd-Warshall method, the algorithm should first calculate the shortest distance between each location and every other location. Once the shortest distance matrix has been obtained, the next step is to identify the permutation of locations that minimizes the total distance travelled by all n limousines.
- This can be accomplished through the use of dynamic programming, where dp[i][j] represents the minimum cost to dispatch limo I to location j given that all preceding locations have been allocated a limo. When I = O or j = O, it signifies that either no locations have been assigned yet or no limos are left to assign.
- In both of these circumstances, the cost is simply infinite because the assignment cannot be completed. Otherwise, repeat through each k j and calculate as min(dp[i - 1] [k] + dist[k][j], dp[i][j]). This means that one must either assign limo I to location j or not; if he does, he must include the cost of dispatching all previous locations up until k (including k), as well as the cost of traveling from k directly to j; otherwise, if he does not assign anything to position j, the total cost remains unchanged from what it was previously when only considering position j - 1.
- Choose the option that results in the lowest overall trip costs for this specific iteration/comparison and save it in THE MEMORY table for future use/reference.

- Return dp [n-1] [n-1] after completely computing this 2D array, which corresponds to the ideal option for assigning n places with n available automobiles.

Finally, we can use the Dijkstra or Bellman-Ford Algorithms to find the shortest paths from each limousine to each customer to the nearest airport. Assuming that each client requires his or her own limo, we can dispatch the limos in such a way that d_{limo}, customer + $d_{customer,\ airport}$ are all kept to a minimum for all n limousines.