CS 600 A - Advanced Algorithms – Homework 7

Name: Deepali Nagwade | 20013393

Question 15.6.16: Show how to modify the Prim-Jarník algorithm to run in O(n2) time.

Answer: To modify the Prim-Jarník algorithm to run in O(n^2) time, we can use a technique called adjacency list representation. In this representation, each vertex in the graph has a list of all its adjacent vertices. This allows us to quickly find all the edges that are adjacent to a given vertex.

Prim-Jarník algorithm with adjacency list representation

Input: A graph G = (V, E), represented as an adjacency list

Output: A minimum spanning tree T of G

- Initialize T to be the empty set.
- Choose a starting vertex v in V and add it to T.
- While T does not contain all the vertices in V:
- For each vertex u in V that is not in T, find the minimum-weight edge (u, v) where v is a vertex in T.
- Add the edge (u, v) to T.
- Update the adjacency list of u to remove all edges to vertices that are already in T.
- Return T.

This algorithm works by iteratively adding the minimum-weight edge that connects a vertex outside the current tree to a vertex inside the tree. The adjacency list representation allows us to quickly find all the edges that are adjacent to a given vertex, which makes the algorithm very efficient.

Here is an example of how to use the modified Prim-Jarník algorithm to find the minimum spanning tree of the following graph:

```
A --- 1 --- B

|         |

3 --- 2 --- C
```

- Choose a starting vertex, say A. Add A to T.
- The minimum-weight edge that connects a vertex outside T to a vertex in T is (A, B). Add (A, B) to T.
- Update the adjacency list of B to remove the edge (B, C).
- The minimum-weight edge that connects a vertex outside T to a vertex in T is (B, C). Add (B, C) to T.
- T now contains all the vertices in the graph, so we stop the algorithm.

The minimum spanning tree of the graph is T = {(A, B), (B, C)}.

The time complexity of the modified Prim-Jarník algorithm is O(n^2). This is because we need to iterate over all the vertices in the graph and for each vertex, we need to find the minimum-weight edge that connects it to a vertex in the current tree. This can be done in O(n) time using the adjacency list representation. Therefore, the overall time complexity of the algorithm is O(n^2).

Question 15.6.22: Suppose you are a manager in the IT department for the government of a corrupt dictator, who has a collection of computers that need to be connected together to create a communication network for his spies. You are given a weighted graph, G, such that each vertex in G is one of these computers and each edge in G is a pair of computers that could be connected with a communication line. It is your job to decide how to connect the computers. Suppose now that the CIA has approached you and is willing to pay you various amounts of money for you to choose some of these edges to belong to this network (presumably so that they can spy on the dictator). Thus, for you, the weight of each edge in G is the amount of money, in U.S. dollars, that the CIA will pay you for using that edge in the communication network. Implement an efficient algorithm, therefore, for finding a maximum spanning tree in G, which would maximize the money you can get from the CIA for connecting the dictator's computers in a spanning tree. What is the running time of your algorithm?

Answer: To solve the problem of finding a maximum spanning tree in a weighted graph where each edge represents the amount of money you can receive, you can modify Kruskal's algorithm to maximize the total amount of money while still ensuring that the selected edges form a spanning tree. Here's an efficient algorithm for this purpose:

- Sort the edges in decreasing order of their weights (the money the CIA is willing to pay).
- Initialize an empty graph that will represent the maximum spanning tree.
- Initialize a disjoint-set data structure to keep track of connected components.
- For each edge in the sorted list of edges, if adding the edge to the current maximum spanning tree doesn't create a cycle, add it to the maximum spanning tree and update the disjoint-set data structure.
- Continue this process until the maximum spanning tree has (V - 1) edges, where V is the number of vertices in the graph.

Here is the pseudocode.

```
Input: Weighted graph G represented as a list of edges (u, v, weight)
Sort the edges of G in decreasing order of weight
Initialize an empty list MST to store the maximum spanning tree
Create a Disjoint-Set data structure DSU
for each vertex v in G:
   DSU.make_set(v)  // Initialize a set for each vertex
while MST has fewer than V - 1 edges, where V is the number of vertices in the graph:
   (u, v, weight) = next edge from the sorted list of edges
   if DSU.find(u) != DSU.find(v):
      // Adding this edge to MST won't create a cycle
      MST.add((u, v, weight))
      DSU.union(u, v)
Output MST  // MST is the maximum spanning tree
```

Question 15.6.25: Imagine that you just joined a company, GT&T, which set up its computer network a year ago to link together its n offices spread across the globe. You have reviewed the work done at that time, and you note that they modeled their network as a connected, undirected graph, G, with n vertices, one for each office, and m edges, one for each possible connection. Furthermore, you note that they gave a weight, w(e), for each edge in G that was equal to the annual rent that it costs to use that edge for communication purposes, and then they computed a minimum spanning tree, T, for G, to decide which of the m edges in G to lease. Suppose now that it is time renew the leases for connecting the vertices in G and you notice that the rent for one of the connections not used in T has gone down. That is, the weight, w(e), of an edge in G that is not in T has been reduced. Describe an O(n + m)-time algorithm to update T to find a new minimum spanning, T' , for G given the change in weight for the edge e.

Answer: Algorithm

```
Input : Graph G, MST T, e' (edge with reduced weight not in T)
1. Add e' to T
2. Find the only cycle in T
3. Find the maximum weight edge in T (e)
4. T' <- T - {e}
5. Return T'
```

Since T is a tree, adding another edge to T will create exactly one cycle in T.

Due to cycle property of minimum spanning trees, the maximum weight edge of a cycle cannot appear in any minimum spanning tree of the graph.

Therefore, the algorithm first adds edge e' to T to create a graph with one cycle. Then find the maximum weight edge in T and remove that edge from T. The resulting graph will be connected and will have exactly n-1 edges (as one edge is added, and one is removed from original minimum spanning tree). Thus T' is the updated minimum spanning tree of the graph as all other edge weights are the same as in original graph.

Time complexity:

Steps 1,4, 5 take constant time. Finding the cycle (step 2) takes O(n+m) time and finding the max weight edge in updated T will take O(n) time.

Therefore, overall time taken = **O(n+m)**

Question 16.7.19: Let N be a flow network with n vertices and m edges. Show how to compute an augmenting path with the largest residual capacity in O((n + m) log n) time.

Answer: We will follow the approach analogous to Dijkstra algorithm. So here in the residual graph G=(V, E), from source vertex s, we will try to find the path of largest residual capacity to every vertex in V from source s.

Just like in Dijkstra algorithm, d[v] indicates minimum weighted path from s to v, here d[v] will indicate largest residual capacity in path from s to v. So we will initialize d[v] = 0 and d[s] = INFINITY and the updation equation will be if d[v] < min (d[u] , c(u,v)) then d[v] = min(d[u] , w(u,v)) . Hence we will store all the nodes in priority queue Q just like in Dijkstra algorithm and the vertex having highest value d[v] will have highest priority and p[v] indicate parent node of v in the path from s to v.

```
Find_Path(G=(V,E), s)

1. For every vertex v != s in V: - Set d[v] = 0; d[s] = INFINITY

2. Add all the vertices in priority queue Q.

3. While Q.notEmpty():-

4...........u = Q.delete() //Delete the highest priority node from Q

5...........For every vertex v adjacent to u :-

6...................if d[v] < min( d[u] , w(u,v) )

7............................d[v] = min( d[u], w(u,v) )

8............................p[v] = u;

9............................Modify_priority(Q, v) //modify the priority of v in Q

10. Return
```

Thus above algorithm will compute augment path from s to every vertex and hence d[v] will be equal to maximum residual capacity path from s to v. Since we have performed the algorithm just analogous to Dijkstra algorithm, so its time complexity will be O((|V|+|E|)log |V|) = O((n+m) log n).

Question 16.7.30: Consider the previous exercise but suppose the city of Irvine, California, changed its dog-owning ordinance so that it still allows for residents to own a maximum of three dogs per household, but now restricts each resident to own at most one dog of any given breed, such as poodle, terrier, or golden retriever. 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, that no resident can adopt more than three puppies, and that no resident will adopt more than one dog of any given breed.

Answer: we will create a set of vertices P of size n where each vertices represent a pet and create a set R of residents of Irvine of size n where each vertex represents a citizen of Irvine city.

And create s and t node representing source and target node respectively.

Now to ensure that a pet is assigned to only one person, create an edge from s to every vertex in P with capacity 1, which ensure that no more than one unit flow will pass from a pet vertex and hence a pet will be assigned to at most 1 citizen.

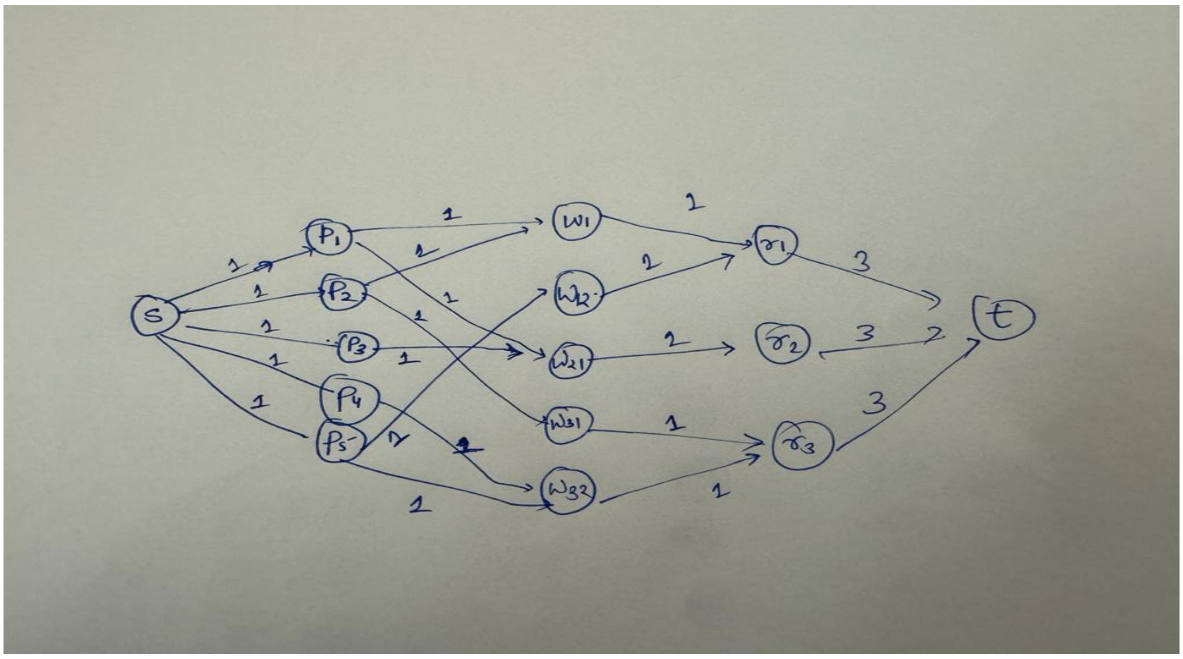
To ensure that each citizen adopt at most 3 pets, create an edge from every vertices in set R to vertex t of capacity 3 units which ensure that not no citizen will adopt more than 3 pets.

Now to ensure that every citizen adopt a pet which he or she likes, earlier we were creating an edge from pet to a citizen if citizen like the pet, but now we have to ensure that each citizen get at most pet of a particular species, we will do a modification. Instead of creating an edge (u,v) from pet u to citizen v, here for each citizen v, we will create s_v number of intermediate nodes where s_v is number of species of pets for which citizen v has liking for. Hence if citizen v likes 5 different species of pets then we will create 5 intermediate nodes for citizen v and we will create 5 vertices w1,w2,w3,w4 and w5 for citizen v, where each vertex from w1 to w5 denote different species of dogs like by v and there will be an each from each of 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 an edge of capacity 1 unit from vertex u in set P to vertex w, if pet u is in species w as well as u is liked by citizen v.

So for each citizen v, if he likes a pet u of species w then we will create an edge (u,w) of capacity 1 unit and an edge (w,v) of capacity 1 unit.

Now with all the construction is done, we will push max-flow from s to t assign maximum number of pets to each citizen.

Let’s take an example. Let there be 5 pets p1, p2, p3 , p4 and p5 where {p1,p2,p3} belongs to one species and {p4,p5} belongs to another species. And let there be 3 citizen r1, r2 and r3 where r1 likes {p1,p2,p5}, r2 likes {p1, p3} and r3 likes {p2, p4, p5}, then below is the flow-network.



Question 16.7.34: A limousine company must process pickup requests every day, for taking customers from their various homes to the local airport. Suppose this company receives pickup requests from n locations and there are n limos available, where the distance of limoi to locationj is given by a number, dij. Describe an efficient algorithm for computing a dispatchment of the n limos to the n pickup location that minimizes the total distance traveled by all the limos.

Answer: The problem can be efficiently solved using the Hungarian algorithm (also known as the Kuhn-Munkres algorithm). The Hungarian algorithm finds the optimal assignment of n limos to n pickup locations to minimize the total distance traveled by all the limos.

Here's a step-by-step description of the algorithm:

- Create a cost matrix C, where C[i][j] represents the cost (distance in your case) of assigning limo i to pickup location j. Ensure that the dimensions of the matrix match the number of limos and pickup locations (n x n).
- Transform the cost matrix into a square matrix by adding rows or columns with zero costs if necessary.
- Find the smallest element in each row of the cost matrix and subtract it from all elements in the corresponding row. This step is known as row reduction. Similarly, find the smallest element in each column and subtract it from all elements in the corresponding column. This step is called column reduction.
- Use the minimum number of lines (rows and columns) to cover all the zeros in the reduced matrix. The minimum number of lines required is n.
- If the number of lines equals n, you have found an optimal assignment. You can read the assignment from the zero entries in the matrix. If not, proceed to the next step.
- Find the smallest uncovered element in the matrix (let's call it "min_uncovered"). Subtract "min_uncovered" from all uncovered elements, and add it to all the intersections of the lines (rows and columns) that cover exactly zero elements. This step is known as the "relabel" step.
- Repeat steps 4 to 6 until you find an optimal assignment with n lines covering the zeros.

The Hungarian algorithm guarantees an optimal assignment that minimizes the total distance traveled by the limos.

The time complexity of the Hungarian algorithm is O(n^3), which is efficient for small to moderate-sized problem instances. If you need to handle larger instances, there are more efficient algorithms with lower time complexity, such as the "Jonker-Volgenant" algorithm, which has a time complexity of O(n^3) in the worst case but often performs faster in practice.