CS345: Design and Analysis of Algorithms

Assignment 3

Submitted by: Aditi Khandelia, 220061

Question 1.

There are n buses in a town and k stations. The position of each bus is specified by its (x, y) coordinates in the plane. Also, the position of each station is specified by its (x, y) coordinates.
For each bus, we wish to connect it to exactly one of the k stations. Our choice of connections is constrained in the following ways. There is a range-parameter r and a capacity parameter L. A bus can only be connected to a station that is within distance r, and no more than L buses can be connected to any single station.
Your goal is to design a polynomial time algorithm for the following problem. Given the position of a set of buses and a set of stations, as well as the range and load parameters, decide whether every bus can be connected simultaneously to a station, subject to the range and load conditions in the previous paragraph.

Answer.

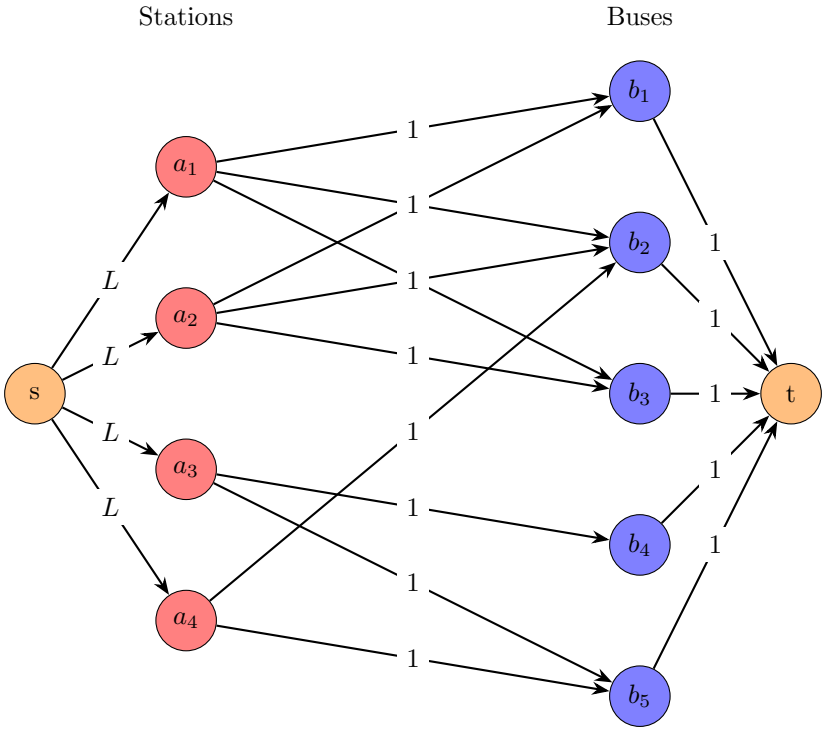
This problem can be solved using Maximum Flow calculation for Flow Networks.

Approach.

We can determine the stations that every bus can be connected to by calculating the Euclidean distance between the station and the bus, since the coordinates of buses as well as stations are available. If the Euclidean distance is less than or equal to r, the station can be connected to the bus.

The given problem can now be modelled as a Flow Network problem :

1. There will be edges between the stations and buses whose distance, as calculated above, is $<= r$. All of these edges will have edge capacity 1. This enforces the constraints of range-parameter and that every bus may be serviced by a single station.
2. There will be a super-source vertex connected to every station, with edge-capacity L. This enforces the constraint that every station can service at most L buses.
3. Since the problem still has multiple sinks, we can convert it to a single-sink problem by adding a super-sink, with edges from every bus. These edges will have a capacity of 1 since each bus should be serviced by one station.



The flow network above is a conventional flow network, for which max flow may be calculated.

Theorem.
A flow of F exists in the flow network iff F buses can be connected to a station.

Lemma 1.
If a flow of value F exists in the flow network, F buses may be connected to a station.

Proof of Lemma 1.
Let the flow of value F be f in the above flow-network. This can be used to construct the required connections of buses and stations.
Since all the edge capacities are integers(cosnidering the parameter L to be an integer, if not, it can be rounded down to the nearest integer without impacting the result), the flow is integral.
Now consider the cut of super-source and the set of stations. Let this cut be A. Since the flow is F,

$$f_{out}(A) - f_{in}(A) = F$$

Now, there will be no edges from $\bar{A}$ to A. Thus, $f_{in} = 0$ and $f_{out} = A$.
Additionally, all the edges between A and $\bar{A}$ begin at a station and end at a bus, and have edge capacities 1. Thus, the no. of edges between stations and buses is F.
Now, each bus is connected to the super-sink with an edge of capacity 1. Thus, each bus can be connected to atmost one station. Each station is connected to the super-source with an edge of capacity L, implying that each sink vertex is connected to atmost L bus-vertices.
Thus, we may connect F buses to stations, while satisfying the given constraints.

Lemma 2.
If F buses can be connected to a station, we can create a flow f of value F in the given flow network.

Proof of Lemma 2.
This lemma can be proven by construction of the flow f.

The following is done:
$\forall i \in n$ and $\forall j \in k$, do the following if b_i is connected to the station a_j:

1. Increase the flow between vertices b_i and a_j by 1.
2. Increase the flow between vertices b_i and t by 1.
3. Increase the flow between vertices a_j and s by 1.

Now consider the path $s - a_j - b_i - t$. Conservation constraint is satisfied for each of the nodes b_i and a_j. Additionally, the capacity constraint is satisfied as well since :

1. the bus b_i could've been connected to one station only. Thus, before a_j, the edge between t and b_i had not been reduced.
2. the station a_j can be connected to atmost L buses, implying that edge between the source s and a_j has edge of capacity atleast 1 before this iteration.
3. since a bus can only be a connected to a station within a distance r, there will be an edge of capacity 1 between the vertices a_j and b_i.

Thus, this path adds a value of 1 to the total flow. Since F buses are connected, this process will be repeated F times.
We have thus constructed a legitimate flow f of value F in the flow network.

The proof of the theorem is thus complete since we have proven both sides of the argument.
Now, Edmonds-Karp algorithm can be used to find the maximum flow in the flow network. If this flow has the value n, we can connect each bus to a station(by lemma 1). Additionally, since at most n buses can be connected to the stations, if such a connection is possible, it will appear as the max. flow(by lemma 2).

Pseudocode.
Following is the pseudocode for the aforementioned algorithm :

1. Create a vertex for super-source, for each station, for each bus and for the super-sink.
2. Add an edge of capacity L from the super-source to the stations.
3. Add an edge of capacity 1 from the buses to the super-sink.

4. For each pair of bus and station, calculate the euclidean distance. If the bus is within r of the station, add an edge of capacity 1 from the station to the bus.

5. Run Edmonds-Karp algorithm on the resultant flow-network to calculate the maximum flow. Suppose this is F.Return *true* if $F = n$.

Algorithm.

Algorithm 1 Bus-Station Matching Algorithm

```
function BusStationMatching(stations, buses, L, r)
    G ← CreateFlowNetwork()
    source ← AddNode(G, "super-source")
    sink ← AddNode(G, "super-sink")
    for each s in stations do
        AddNode(G, s)
        AddEdge(G, source, s, L)
    for each b in buses do
        AddNode(G, b)
        AddEdge(G, b, sink, 1)
    for each s in stations do
        for each b in buses do
            if EuclideanDistance(s, b) ≤ r then
                AddEdge(G, s, b, 1)
    F ← EdmondsKarp(G, source, sink)
    if F = |buses| then return true
    elsereturn false
```

Algorithm 2 Euclidean Distance Calculation

```
function EuclideanDistance(point1, point2)
    dx ← point1.x − point2.x
    dy ← point1.y − point2.y
    distance ← √(dx² + dy²) return distance
```

Algorithm 3 Edmonds-Karp Algorithm

```
function EdmondsKarp(G, source, sink)
    flow ← 0
    while true do
        path, pathFlow ← BFS(G, source, sink)
        if pathFlow = 0 then
            break
        flow ← flow + pathFlow
        for each edge (u, v) in path do
            G[u][v].flow ← G[u][v].flow + pathFlow
            G[v][u].flow ← G[v][u].flow − pathFlow
    return flow
function BFS(G, source, sink)
    Initialize all nodes as not visited
    queue ← empty queue
    queue.enqueue(source)
    parent ← array to store path
    while queue is not empty do
        u ← queue.dequeue()
        for each neighbor v of u in G do
            if v is not visited and residual capacity of (u, v) > 0 then
                queue.enqueue(v)
                parent[v] ← u
                if v = sink then
                    path ← reconstruct path using parent
                    pathFlow ← minimum residual capacity along path return path, pathFlow
    return empty path, 0
```

Time Complexity Analysis.

1. Creation of the flow network G takes time $O(|stations| * |buses|)$ since we need to calculate the Euclidean distance between each station and bus(which takes $O(1)$ time and there are $O(|stations| * |buses|)$ pairs. We also need $O(|stations|) = O(|stations| * |buses|)$ time to add edges between the super-source and the stations, and $O(|buses|) = O(|stations| * |buses|)$ time to add edges between the buses and super-sink.

2. The time complexity of Edmonds-Karp algorithm is $O(m^2 * n)$ where m = no. of edges in the graph $= O(|stations| * |buses|)$ and n = no of vertices in the graph $= O(|stations| + |buses|)$

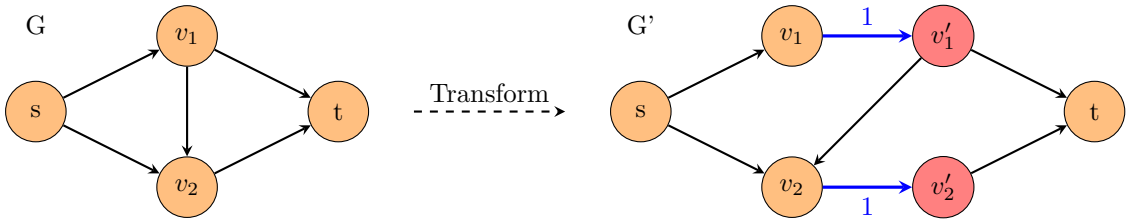
$$\text{Time complexity} = O(|stations| * |buses|) + O((|stations| * buses|)^2 * (|stations| + |buses|) = O((|stations| * buses|)^2 * (|stations| + |buses|)$$

Note:
There exist several other algorithms that may be used to find the maximum flow in polynomial time, including some with lower time complexities than Edmonds-Karp(like Orlin's Algorithm).

Question 2.
Let $G = (V, E)$ be a directed graph on n vertices and m edges. There are two vertices s, t in V . Two paths from s and t are said to be vertex disjoint if they do not share any vertex except s and t. Design a polynomial time algorithm to compute the maximum number of vertex disjoint paths from s to t.

Answer.
This problem may be solved using a modification of maximum no. of edge disjoint paths between two vertices s and t.
Approach
We will construct a new graph $G' = (V', E')$ for the given graph G. The construction proceeds as follows:

1. Let $V' = V$ and $E' = E$
2. $\forall v \in V | \{s, t\}$:
   (a) add a vertex v' to V'
   (b) add an edge of capacity 1 from v to v'
   (c) $\forall z$ such that $(v, z) \in E$:
      i. add (v', z) to E'
      ii. remove (v, z) from E'



Theorem.
There exist n edge-disjoint paths in G' iff there are n vertex-disjoint paths in G.

Lemma 1.
If there are n edge-disjoint paths in G', there are n vertex-disjoint paths in G.

Proof of Lemma 1.
Consider any edge-disjoint path in G'. Let this path be P.

If the path P contains v_i, it must use the edge $v_i - v_i'$ since this is the only outgoing edge that can connect v_i to t.
If some other path P' were to contain v_i, it will have to use $v_i - v_i'$ as well, which is not possible since P and P' are edge-disjoint.

If the path P contains v_i', it must use the edge $v_i - v_i'$ since this is the only incoming edge that can connect v_i' to s.
If some other path P' were to contain v_i', it will have to use $v_i - v_i'$ as well, which is not possible since P and P' are edge-disjoint.

Thus, the edge-disjoint paths in G' are vertex disjoint as well. For any pair of vertices $v_i - v_i'$ in a path P in G', we can replace it with v_i in G, to obtain the corresponding path in G.
We can transform the n edge-disjoint paths in G', to n vertex-disjoint paths in G.

Lemma 2.
If there are n vertex-disjoint paths in G, there are n edge-disjoint paths in G'.

Proof of Lemma 2.
Consider any path P in G. Let this path be $s - v_i - v_j - t$. Consider the corresponding path P' in G' : $s - v_i - v_i' - vj - v_j' - t$. This path is always possible since for the outgoing edge $v_i - v_j$ in G, we have the set of edges $v_i - vi' - v_j$. We will transform all the n vertex-disjoint paths in G to n corresponding paths in G'. We can prove that this set of paths is vertex-disjoint in G'.

Consider any vertex v_i is in path S' in G'. Then it belongs to corresponding path S in G. If possible, let $v_i \in R'$ in G' such that $S' \neq R'$. Then v_i is in corresponding path R in G such that $S \neq R$. But this would result in a common vertex, v_i, in two vertex-disjoint paths, which is not possible. Thus, v_i can belong to a single path in S' in G'.

Consider any vertex v_i' is in path S' in G'. Then it contains v_i which will belong to corresponding path S in G. If possible, let v_i' is in path R' in G' such that $S' \neq R'$. Then v_i is in corresponding path R in G such that $S \neq R$. But this would result in a common vertex, v_i, in two vertex-disjoint paths, which is not possible. Thus, v_i' can belong to a single path in S' in G'.

This proves that the set paths in G' is vertex-disjoint. Since the paths are vertex disjoint, they will necessarily be edge-disjoint as well.
Hence, if there are n vertex-disjoint paths in G, we can get n edge-disjoint paths in G'.

The two lemmas together prove the theorem, which can be used to find the maximum no. of vertex disjoint paths. If we can find the maximum no. of edge disjoint paths in G', we will be able to create the same number of vertex-disjoint paths in G(by lemma 1). This will be the maximum no. of vertex-disjoint paths, since if the no. were higher, there would be higher no. of edge-disjoint paths as well(by lemma 2).

Pseudocode.

1. The given graph $G = (V, E)$ is converted to a graph $G' = (V', E')$ by the following process :
   (a) Add s, t to V'
   (b) $\forall v_i \in V | \{s, t\}$, add v_i, v_i' to V'
   (c) $\forall (s, v_i) \in E$, add (s, v_i) to E'
   (d) $\forall (v_i, t) \in E$, add (v_i', t) to E'
   (e) $\forall (v_i, v_j) \in E, v_i \neq s, v_j \neq t$, add (v_i', v_j) to E'
   (f) $\forall v \in V$, add (v_i, v_i') to E'
2. Add weight 1 to each edge in E', and calculate the maximum flow in G' using Edmonds-Karp Algorithm. This will be the maximum no. of edge-disjoint paths in G'. Return this number as the maximum no. of vertex-disjoint of the original graph G.

Algorithm.

Algorithm 4 Find Maximum Number of Vertex-Disjoint Paths

```
1: function MaxVertexDisjointPaths(G(V, E), s, t)
2:     G' ← TransformGraph(G, s, t)
3:     maxFlow ← EdmondsKarp(G', s, t) return maxFlow
```

Algorithm 5 Transform Graph for Vertex-Disjoint Paths

```
1: function TransformGraph(G(V, E), s, t)
2:     G'(V', E') ← CreateEmptyGraph()
3:     AddNode(G', s)
4:     AddNode(G', t)
5:     for each v_i in V \ {s, t} do
6:         AddNode(G', v_i)
7:         AddNode(G', v_i')
8:         AddEdge(G', v_i, v_i', 1)
9:     for each (s, v_i) in E do
10:        AddEdge(G', s, v_i, 1)
11:    for each (v_i, t) in E do
12:        AddEdge(G', v_i', t, 1)
13:    for each (v_i, v_j) in E where v_i ≠ s and v_j ≠ t do
14:        AddEdge(G', v_i', v_j, 1)
       return G'
```

Algorithm 6 Edmonds-Karp Maximum Flow Algorithm

```
function EdmondsKarp(G, source, sink)
    flow ← 0
    while true do
        path, pathFlow ← BFS(G, source, sink)
        if pathFlow = 0 then
            break
        flow ← flow + pathFlow
        for each edge (u, v) in path do
            G[u][v].flow ← G[u][v].flow + pathFlow
            G[v][u].flow ← G[v][u].flow − pathFlow
    return flow
function BFS(G, source, sink)
    Initialize all nodes as not visited
    queue ← empty queue
    queue.enqueue(source)
    parent ← array to store path
    while queue is not empty do
        u ← queue.dequeue()
        for each neighbor v of u in G do
            if v is not visited and residual capacity of (u, v) > 0 then
                queue.enqueue(v)
                parent[v] ← u
                if v = sink then
                    path ← reconstruct path using parent
                    pathFlow ← minimum residual capacity along path return path, pathFlow
    return empty path, 0
```

Time Complexity Analysis.

1. The time complexity of forming the graph G' from G with m edges and n vertices :
   (a) $O(n)$ time is required to add an additional vertex $\forall v \in G$
   (b) $O(m)$ time is required to transfer the outgoing edges from (v_i, v_j) to (v_i', v_j)
   (c) $O(n)$ time is required to add the edges (v_i, v_i')

   Total time complexity $= O(n) + O(m) + O(n) = O(m + n)$
2. The new graph has $O(n + m)$ edges and $O(n + m)$ time is required to add the capacity 1 to each edge
3. Edmonds-Karp algorithm will run in time complexity $O(m^2 * n)$

$$\text{Total time complexity} = O(n + m) + O(n + m) + O(m^2 * n) = O(m^2 * n)$$

Note:
There exist several other algorithms that may be used to find the maximum flow in polynomial time, including some with lower time complexities than Edmonds-Karp(like Orlin's Algorithm).