CS345 Theoretical Assignment 3 Submission

Naman Kumar Jaiswal

220687

namankj22@iitk.ac.in

Fall Semester 2024

Question 1: Bus-Station Connection Problem

We need to find a polynomial time algorithm for trying to connect buses to nearby stations given the following constraints and find if a possible connection exists.

Constraints:

- **Distance Constraint**: A bus can only be connected to a station if the distance between them does not exceed the specified range parameter r. Mathematically, for a bus b_i at position (x_{b_i}, y_{b_i}) and a station s_j at position (x_{s_j}, y_{s_j}):

$$\sqrt{(x_{b_i} - x_{s_j})^2 + (y_{b_i} - y_{s_j})^2} \leq r$$

- **Capacity Constraint**: No more than L buses can be connected to any single station. This means that each station can serve a limited number of buses simultaneously.
- **Single Connection Constraint**: One bus can only be mapped to a single station but a single station can be mapped to multiple buses.

These constraints are very similar to the Bipartite mapping problem.

Heuristic: Similarity to Bipartite Matching

The bus-station mapping problem can be viewed as a similar problem to the **bipartite matching** problem. In a typical bipartite matching scenario, we have two disjoint sets (buses and stations) and edges that connect them based on certain criteria. In our case, the additional distance and capacity constraints introduce complexities that can be handled using network flow techniques.

The key idea is to represent the connections between buses and stations as a flow network, where:

- Each bus is a node in one partition of the bipartite graph.
- Each station is a node in the other partition.
- An edge exists between a bus and a station if they are within the allowable distance r.

We know that the Bipartite Matching Problem can be solved by converting the problem to an equivalent Dual Network Flow Problem, which can further be solved using a polynomial time Edmonds-Karp algorithm.

Conversion to an equivalent Dual Network Flow Problem:

To convert the bus-station mapping problem into a network flow problem, we can construct a flow network as follows:

- **Graph**: Start the construction with an empty graph.
- **Vertices**:
  - Create a source node S.
  - Create a sink node T.
  - Represent each bus b_i as a vertex.
  - Represent each station s_j as a vertex.
- **Edges**:
  - From the source S to each bus b_i: Connect S to each bus with an edge of capacity 1 (since each bus can connect to only one station – constraint 3).

- From each station s_j to the sink T: Connect each station to T with an edge of capacity L (allowing up to L buses to connect to this station – constraint 2).
- From each bus b_i to each station s_j: Connect b_i to s_j with an edge of capacity 1 if the distance $d(b_i, s_j) \leq r$ (We must only connect a bus to a station if it is nearby enough – constraint 1).

This construction results in a flow network where finding the maximum flow will correspond to finding valid bus-station connections that respect the constraints.

Correctness: a Network Flow Solution is a Valid Bus-Station connection Solution

We represent the bus-station connection problem as a flow network:

- We create a **source node** S and connect it to each bus b_i with an edge of capacity 1, since each bus can only connect to one station.
- We create a **sink node** T and connect each station s_j to the sink with an edge of capacity L, as each station can accommodate up to L buses.
- For each bus b_i, we add an edge of capacity 1 to any station s_j if the Euclidean distance between them satisfies $d(b_i, s_j) \leq r$.

Proof

We now show that if the maximum flow in this network equals n, then a solution to the bus-station connection problem exists.

1. **Flow capacity matching**:
   - Each bus node b_i has an outgoing edge from the source with capacity 1, meaning that the bus can connect to at most one station.
   - Each station node s_j has an edge to the sink with capacity L, meaning that no more than L buses can be assigned to any one station.
2. **Flow value interpretation**:
   - The flow from S to T represents the number of bus-station connections.
   - If the total flow equals n, then each bus is connected to exactly one station, and the capacity constraints for each station are respected.
3. **Conclusion**: If the maximum flow in the network equals n, then every bus can be assigned to a station in a way that respects both the distance and capacity constraints. Therefore, a solution to the network flow problem guarantees a solution to the bus-station connection problem.

Correctness: a Bus-Station connection Solution is a Valid Network Flow Solution

Now, we assume that a valid solution to the bus-station connection problem exists. This means that:

- Each bus b_i is connected to exactly one station s_j such that $d(b_i, s_j) \leq r$.
- No station has more than L buses connected to it.

Proof

We now show that if such a solution exists, it corresponds to a valid flow in the network.

1. **Bus-Station Connection Setup**:
   - For each bus b_i, there exists a valid connection to some station s_j within distance r, meaning an edge exists in the network from b_i to s_j.
   - Each station s_j serves at most L buses, meaning that the capacity constraint on the edge from s_j to the sink T is respected.

2. **Flow Representation**:
   - We can map the bus-station connections to flow units. For each bus b_i connected to a station s_j, we assign a flow of 1 unit along the edge from S to b_i, from b_i to s_j, and from s_j to T.
   - Since each bus is connected to exactly one station, the flow capacity of 1 on each edge from S to b_i is respected.
   - Since no station is connected to more than L buses, the flow capacity on each edge from s_j to T is respected.

3. **Conclusion**: If a valid bus-station connection exists, we can construct a valid flow in the network. Therefore, solving the bus-station connection problem is equivalent to solving the network flow problem.

Conclusion: The Problems Are Dual

Since we have proven that:

1. A solution to the network flow problem guarantees a solution to the bus-station connection problem, and

2. A solution to the bus-station connection problem can be mapped to a valid network flow,

we conclude that the two problems are **dual** to each other. Solving one is equivalent to solving the other, as the constraints in the bus-station problem (distance and capacity) are mirrored in the corresponding flow network setup.

In conclusion the max flow in the equivalent network is same as the maximum Bus-Station Connection Solution. We will solve the network flow problem using the following logic/pseudocode.

Logic:

1. **Pre-Processing**: We will use the aforementioned conversion to the reach an equivalent Network Flow Problem.

2. **Edmonds-Karp Algorithm**: We will use the Edmonds-Karp algorithm to solve for the maximum flow across the network and report this flow solution as the solution of our original answer. If the flow value equals the number buses (i.e. n), we return true else we return false.

Pseudocode:

1. Pre-Processing:

```
Input: Set of buses B, |B| = n and stations S, |S| = m, load parameter L and
    distance parameter r.

Output: Transformed flow network G* which represents the connection problem.

function ConvertToNetwork(B,S,L,r):

```

```
    // Initialize an empty graph G*
    G* ← new Graph({},{})

    // Add the source, sink, buses and stations
    For each bus b_i in B:
        Add a vertex b_i in B
    For each station s_i in S:
        Add a vertex s_i in S
    Add a source vertex S
    Add a sink vertex T

    // Add Edges connecting the graph
    For each bus b_i in B:
        Add an edge from S to b_i of capacity 1
    For each station s_i in S:
        Add an edge from s_i to T of capacity L
    For each bus b_i in B:
        For each station s_i in S:
            If (x_{b_i} - x_{s_j})^2 + (y_{b_i} - y_{s_j})^2 ≤ r*r, then
                Add an edge from b_i to s_j of capacity 1

    Return G*
```

2. Edmonds-Karp algorithm to find the max flow across the network:

```
Input: Transformed flow network G*, source vertex s, sink vertex t.

Output: Maximum number of vertex-disjoint paths from s to t, which also is the
     max flow across the network.



function EdmondsKarp(G*, s, t):

    // Initialize total_flow = 0
    total_flow ← 0

    // Initialize flow values f(x, y) = 0 for all edges (x, y) in G* and
    residual capacities C(x, y) for residual graph G_f same as in orginal graph
    . f corresponds to flow network G* and c corresponds to residual graph G_f.
    G_f ← G*
    For each (x,y) ∈ E
        f(x,y) ← 0

    while ∃ an augmenting path from s to t in G_f:

        // Compute the Shortest Path P and BottleNeck capacity c'
        P ← shortest augmenting path P from s to t using BFS
        c' ← min{capacity of edges along P}

        For each edge (x, y) in P:

            If (x, y) is a forward edge in G*, then
                f(x, y) += c'

                If c(y, x) = 0 then
                    add edge (y, x) to G_f

                c(y, x) += c'
```

```
                c(x, y) -= c'

                If c(x,y) = 0 then
                    remove edge (x, y) to G_f

            Else if (x, y) is a backward edge in G*, then
                f(y, x) -= c'

                If c(y, x) = 0 then
                    add edge (x, y) to G_f

                c(y, x) += c'
                c(x, y) -= c'

                If c(x, y) = 0 then
                    remove edge (y, x) to G_f

        total_flow += c'
    Return total_flow
```

3. Check possible connection function, returns true if a bus-station connection exists else return false:

```
Input: Transformed flow network G*, number of buses n.

Output: Whether a valid bus-station connection exist or not

function checkFlow(G*, n):
    maxFlow ← EdmondsKarp(G*, S, T)

    if maxFlow = n:
        return True  # All buses can be connected to stations
    else:
        return False  # Some buses cannot be connected
```

Time Complexity Analysis

Given n buses, k stations, L load parameter and r range parameter, we go through the following steps
Preprocessing:

- There are five simple for loops in the preprocessing which run n, k, n, k and nk number of times respectively. Every iteration of any of the for loops take $O(1)$ time. Total it goes $O(2n + 2k + nk) \equiv O(nk)$ time.
- Thus, the preprocessing step takes $O(nk)$ time.

Edmonds-Karp Algorithm: For E edges and V vertices,

- *Breadth-First Search (BFS):* Each BFS takes $O(E)$ time, as it explores all the edges in the residual graph.
- *Augmenting Paths:* In the worst case, we find $O(V \cdot E)$ augmenting paths.
- *Total Time:* Since we perform $O(V \cdot E)$ BFS calls and each BFS takes $O(E)$ time, the total time complexity is $O(V \cdot E^2)$ time.
- *Total Time in n, k:* Since, the network has n+k+2 vertices and in the worst case n + nk + k edges, the time complexity is $O((n + k + 2) \cdot (n + nk + k)^2) \equiv O(n^2k^2(n + k))$ time complexity.

Summary: In total, the time complexity of the proposed algorithm is $O(n^2k^2(n + k))$ which is polynomial time.

Question 2: Node Disjoint Paths in a DiGraph

We need to find a polynomial time algorithm that runs over a Directed Graph G = (V, E) which given a source s and a sink t, computes the number of vertex disjoint paths from s to t. A path p_1 is said to be vertex disjoint from another path p_2 if they have no nodes in common (excluding the starts and the ends).

Heuristic:

1. We know that the "Edge Disjoint Paths" problem can be converted into a dual Network Flow Problem, where each possible edge-disjoint path can be represented as a unit flow in the network. To solve for Edge Disjoint Paths we required to add a constraint of edge capacity ≤ 1 for all edges. What this did was to make sure that if an edge is once considered for a path from s to t (flow through it increased to 1), it must not be considered again (flow ≤ 1).

2. A similar constraint can be thought of for the Vertex Disjoint Paths problem. If a vertex is chosen for a path, the flow through it increases by 1. To ensure all the possible paths are vertex disjoint, we must ensure the constrait on the vertices. The flow through a node must not exceed more than 1.

Conversion to an equivalent Dual Network Flow Problem:

We need to transform the vertex-disjoint path problem into a network flow problem, where the objective is to maximize the number of paths from s to t that do not share any vertices except s and t. We need to do the following steps for the same.

Graph Transformation: Splitting Nodes

To enforce vertex disjointness, we modify the graph as follows:

- **Intermediate Edges:** For each vertex $v \in V \backslash \{s, t\}$, split it into two nodes: v_{in} and v_{out}. The split vertices v_{in} and v_{out} replace the original vertex v from the graph. If G had n vertices, the new graph will have $2n - 2$ vertices.

- **Split Edges:** Add an edge from v_{in} to v_{out} with a capacity of 1. This ensures that at most one path can pass through vertex v, as only one unit of flow can go through the edge $v_{\text{in}} \to v_{\text{out}}$.

- **Source and Sink:** Vertices s and t remain unchanged.

- **Original Edges:** For each edge $(u, v) \in E$ in the original graph, create a new edge from u_{out} to v_{in} with capacity 1 in the transformed graph. If u is the source, then $u_{\text{out}} = u$. If v is the source, then $v_{\text{out}} = v$.

Constraints of the Flow Network

- **Capacity constraints:**
  - Each edge has a capacity of 1, enforcing that at most one path can use any given edge. If two paths are vertex disjoint, they are also edge disjoint. This can be easily argued as an edge joins two vertices, if the set of vertices do not contain a common value, the paths can not have a common edge.
  - The edge from v_{in} to v_{out} also has a capacity of 1, enforcing vertex-disjointness by limiting the flow through any vertex v to at most 1.

- **Flow conservation:** At each node (except s and t), the incoming flow must equal the outgoing flow (standard in flow networks).

- **Objective:** Maximize the flow from s to t. The value of the maximum flow will correspond to the number of vertex-disjoint paths from s to t.

Correctness: a Network Flow Solution is a Valid Vertex-Disjoint Paths Solution

A valid solution to the network flow problem corresponds to a set of vertex-disjoint paths in the original graph for the following reasons:

- The capacity constraints ensure that no edge is used by more than one path, enforcing edge disjointness.
- The transformation that splits each vertex v into v_{in} and v_{out} ensures that no path can use the same vertex more than once. Since the capacity of the edge $v_{\text{in}} \rightarrow v_{\text{out}}$ is 1, at most one path can pass through any vertex v, enforcing vertex disjointness.
- A unit flow represents the edge/vertex has been counted and no flow represents that the edge/vertex has not been counted.

Thus, each unit of flow from s to t in the transformed network corresponds to one vertex-disjoint path from s to t in the original graph. The value of max flow is the maximum number of vertex disjoint paths from s to t.

Correctness: a Vertex-Disjoint Paths Solution is a Valid Network Flow Solution

Conversely, if there exists a set of k vertex-disjoint paths from s to t in the original graph, we can construct a valid flow in the transformed network as follows:

- For each vertex-disjoint path, direct one unit of flow along the corresponding edges in the transformed network.
- Since the paths are vertex-disjoint, no two paths will pass through the same vertex, and thus, the flow through each edge and split vertex in the transformed graph will respect the capacity constraints.

Hence, a valid vertex-disjoint paths solution corresponds to a valid flow in the transformed network, with the flow value equal to the number of vertex-disjoint paths.

Duality of the Problems

The two problems are **dual** to each other in the following sense:

- The vertex-disjoint paths problem can be reduced to a network flow problem.
- The solution to the network flow problem directly gives the solution to the vertex-disjoint paths problem.
- Conversely, a solution to the vertex-disjoint paths problem corresponds to a valid flow in the network flow problem.

This duality arises because the flow in the network captures the essence of vertex disjointness via the splitting of nodes and the use of capacity constraints, making the network flow problem a suitable tool for solving the vertex-disjoint paths problem.
In conclusion the max flow in the equivalent network is same as the maximum number of vertex disjoint paths. We will solve the network flow problem using the following logic/pseudocode.

Logic:

1. **Pre-Processing**: We will use the aforementioned conversion to the reach an equivalent Network Flow Problem.
2. **Edmonds-Karp Algorithm**: We will use the Edmonds-Karp algorithm to solve for the maximum flow across the network and report this solution as the solution of our original answer.

Pseudocode:

1. Pre-Processing:

```
Input: Directed graph G with vertices V, |V| = n and edges E, |E| = m, source vertex s, sink vertex t. Source and Sink here are names, they need not be actual source and sink of the graph. Ref to example at: https://en.wikipedia.org/wiki/Edmonds-Karp_algorithm

Output: Transformed flow network G* where each vertex in V (except s, t) is split into two vertices and appropriate changes to the edges have been done.

function ConvertToNetwork(G,s,t):

    // Initialize an empty graph G*
    G* ← new Graph({},{})

    // Add the existing vertices and Split the intermediate vertices
    For each vertex v in V:
       If v ≠ s and v ≠ t then
          Split v into two vertices: v_in and v_out
          Add an edge from v_in to v_out in G* with capacity 1
       Else
          Add vertex v directly to G*
    For each edge (u, v) in E:
       If u ≠ s and u ≠ t then
          Add an edge from u_out to v_in in G* with capacity 1
       Else If u = s then
          Add an edge from u to v_in in G* with capacity 1
       Else
          Add an edge from u_out to v in G* with capacity 1
    Return G*
```

2. Edmonds-Karp algorithm to find the max flow across the network:

```
Input: Transformed flow network G*, source vertex s, sink vertex t.

Output: Maximum number of vertex-disjoint paths from s to t, which also is the max flow across the network.



function EdmondsKarp(G*, s, t):

    // Initialize total_flow = 0
    total_flow ← 0

    // Initialize flow values f(x, y) = 0 for all edges (x, y) in G* and residual capacities C(x, y) for residual graph G_f same as in orginal graph. f corresponds to flow network G* and c corresponds to residual graph G_f.
    G_f ← G*
    For each (x,y) ∈ E
        f(x,y) ← 0

    while ∃ an augmenting path from s to t in G_f:

        // Compute the Shortest Path P and BottleNeck capacity c'
        P ← shortest augmenting path P from s to t using BFS
        c' ← min{capacity of edges along P}
```

```

        For each edge (x, y) in P:

            If (x, y) is a forward edge in G*, then
                f(x, y) = f(x, y) + c'

                If c(y, x) = 0 then
                    add edge (y, x) to G_f

                c(y, x) += c'
                c(x, y) -= c'

                If c(x,y) = 0 then
                    remove edge (x, y) to G_f

            Else if (x, y) is a backward edge in G*, then
                f(y, x) = f(y, x) - c'

                If c(y, x) = 0 then
                    add edge (x, y) to G_f

                c(y, x) += c'
                c(x, y) -= c'

                If c(x, y) = 0 then
                    remove edge (y, x) to G_f

        total_flow += c'
    Return total_flow
```

Time Complexity Analysis

Given m edges and n vertices **Preprocessing:**

- *Graph Initialization:* Creating the residual graph involves processing both vertices and edges.
- *Vertices:* There are n vertices to process.
- *Edges:* Initializing flow values and capacities for each of the m edges.
- Thus, the preprocessing step takes $O(m + n)$ time.

Edmonds-Karp Algorithm:

- *Breadth-First Search (BFS):* Each BFS takes $O(|E|)$ time, as it explores all the edges in the residual graph.
- *Augmenting Paths:* In the worst case, we find $O(|V| \cdot |E|)$ augmenting paths.
- *Total Time:* Since we perform $O(|V| \cdot |E|)$ BFS calls and each BFS takes $O(|E|)$ time, the total time complexity is $O(|V| \cdot |E|^2)$.
- *Total Time in required parameters:* Since, here we have 2n - 2 vertices and m + n - 2 edges in the network, the total time complexity is $O(|2n - 2| \cdot |m + n - 2|^2) \equiv O(n(m + n)^2)$

Summary: The total time complexity is $O(n(n + m)^2$ which is polynomial time.