Assignment 4

Graph Algorithms

Name: Dipam Sen
Roll No: 24CS10059

---

1. An undirected graph $G = (V, E)$ is called **bipartite** if its vertex set V can be partitioned into two subsets V_1 and V_2, such that every edge in E has one endpoint in V_1 and the other in V_2. Equivalently, G is bipartite if its vertices can be colored using two colors so that no two adjacent vertices receive the same color.

Prove that an undirected graph G is bipartite if and only if it contains no cycle of odd length.

Sol. $\Rightarrow$: Let G be a bipartite graph. Assume that it contains a cycle of odd length, $\langle v_1, v_2, ..., v_{2k+1}, v_1 \rangle$. This means that $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), ..., (v_{2k+1}, v_1)\} \subseteq E$ are edges in the graph. Since the graph is bipartite, V can be partitioned into V_1 and V_2 such that all edges have one vertex from each subset.

WLOG, let $v_1 \in V_1$. Then, we have,

$$v_1 \in V_1 \Rightarrow v_2 \in V_2 \Rightarrow v_3 \in V_1 \Rightarrow ... \Rightarrow v_{2k} \in V_2 \Rightarrow v_{2k+1} \in V_1$$

But, $v_{2k+1} \in V_1$ and $v_1 \in V_1$ while $(v_{2k+1}, v_1) \in E$, which is a contradiction. Thus G cannot contain a cycle of odd length.

$\Leftarrow$: Let G be an undirected graph which does not contain any odd length cycles. Then, let us try to find a 2-coloring of the graph. Call an edge **well-colored**, if both its endpoints are differently colored. We need to show that there exists some coloring strategy by which every edge in the graph will become well-colored.

Perform depth first search on the graph, coloring each node such that two adjacent vertices will have different colors. This ensures that all edges which are a part of the DFS forest are well-colored.

Consider any non-tree edge (v_1, v_2). Note that this must necessarily be an edge connecting two vertices which are the part of the same DFS tree. Then, consider the (unique) path P from v_1 to v_2 in the DFS tree, having all well-colored edges. Now, there also exists the edge between v_1 and v_2, which makes $P + (v_2, v_1)$ a cycle. This cycle must have even length (by assumption), thus, in the tree path from v_1 to v_2 there must be an even number of intermediate vertices.

Since we know that the path from v_1 to v_2 is a well-colored path, and it has even number of intermediate vertices, it follows that v_1 and v_2 must be differently colored; using similar reasoning as above. Thus, (v_1, v_2) is a well-colored edge, which completes the proof.

---

2. Given an undirected graph $G = (V, E)$, design an algorithm to determine whether G is bipartite.

Sol. Perform a DFS on the graph. Arbitrarily color the first unvisited vertex, and hence for each edge (u, v)

- If v hasn't been visited yet, color it the opposite color of u.
- If v has already been visited, and its color is the same as that of u, then report G is not bipartite. Continue otherwise.

If we succeed in visiting all vertices, we report that G is bipartite.

Correctness

Success: Since we have successfully 2-colored the graph, G must be bipartite. (We have ensured all tree-edges and non-tree edges to be well-colored.)

Non-success: We break at a certain point when we find that by our coloring we have found a non well-colored edge, i.e. an edge with both vertices colored the same. Let this edge have endpoints v_1 and v_2. Consider the unique path P from v_1 to v_2 in the DFS tree. Note that this is a well-colored path (by our coloring strategy). Since v_1 and v_2 have the same color, this means there must be an odd number of intermediate vertices in this path. Consider the cycle $P + (v_2, v_1)$, this is a cycle of odd length, thus G is not bipartite (from 1).

Time Complexity: $O(|V| + |E|)$

---

3. Given an undirected graph $G = (V, E)$, define its **square** $G^2 = \left(V, E^2\right)$ such that $(u, v) \in E^2$ if and only if there exists a path in G from u to v with at most two edges. Design efficient algorithms to compute G^2 from G for both:

(a) Adjacency list representation

(b) Adjacency matrix representation

Analyze the time complexity of both algorithms—as functions of $|V|$, $|E|$ and Δ, where:

$$\Delta := \max_{v \in V} \deg^+(v), \qquad \deg^+(u) := \text{number of outgoing edges from } u$$

Which one is better between (a) and (b), and when? Justify.

Sol. The key observation is that G^2 will have an edge between u and v iff there exists a path of length 1 between them (an edge), or if there exists a path of length 2 $\langle u, k, v \rangle$ between them.

(a) Observe that the adjacency list of G^2 will contain the edge (u, v) if

(i) $(u, v) \in E$, or

(ii) $(u, k) \in E$ and $(k, v) \in E$ for some k

This leads to the following algorithm:

Algorithm 1: Find G^2 from Adjacency List

```
1  procedure FIND-SQUARE-LIST(G)
2      for u in V do                          ▷ 1
3          for v in adj[u] do
4              adj2[u] ← append v
5
6
7      for u in V do                          ▷ 2
8          for k in adj[u] do
9              for v in adj[k] do
10                 if u ≠ v then
11                     adj2[u] ← append v
12
13
14
15
16     return adj2
17
```

Time Complexity: $O\left(|E| + |V|\,\Delta^2\right) \equiv O(|E| + |E|\,\Delta) = O(|E|\,\Delta)$

(b) Let A be the adjacency matrix representation of G and A^* be the adjacency matrix representation of G^2.

Observe that $A^*[i, j] = 1$ iff at least one of the following is true:

(i) $A[i, j] = 1$

(ii) There exists k such that $A[i, k] = 1 \wedge A[k, j] = 1$

Equivalently, $A^*[i, j] = 0$ iff $A[i, j] = 0$ and $A[i, k] \wedge A[k, j] = 0 \quad \forall\, k \in V$.

Consider the matrix $B = A \times A = A^2$. By definition, $B[i,j] = \sum_{k=1}^{n} A[i,k]A[k,j]$. In particular, $B[i,j] = 0$ if and only if $A[i,k]A[k,j] = 0$ for all $k \in V$. (This is due to the fact that $A[i,j] \in \{0,1\}$ for all $i,j \in V$.)

$B[i,j]$ is non zero otherwise. i.e., $B[i,j]$ is non zero if there exists k such that $A[i,k]A[k,j] > 0$. Note that this is the same as condition (ii) above.

Thus, we can construct A^* from B in this way:
- $A^*[i,j] = 1$ if $B[i,j] > 0$ or $A[i,j] = 1$
- $A^*[i,j] = 0$ otherwise

To find B, we take the matrix multiplication of A with itself.

(i) Compute the matrix A^2
(ii) Compute the matrix A^* as per:

$$A^*_{ij} = \begin{cases} 1 & \text{if } A^2_{ij} > 0 \text{ or } A_{ij} = 1 \\ 0 & \text{otherwise} \end{cases}$$

Time Complexity: $O(|V|^3 + |V|^2) = O(|V|^3)$

Comparison
- For **sparse graphs**, $|E| \ll |V|^2$, (a) is a much better choice. Since (a) operates on the adjacency list, it exploits the sparsity of the graph by only iterating over the valid edges in the graph.
- For **dense graphs**, $|E| \approx |V|^2$, and we have $\Delta = O(|V|)$, so both (a) and (b) are asymptotically equivalent in their time complexity, $O(|V|^3)$.

4. Show how to determine whether a directed graph G contains a **universal sink** — a vertex with in-degree $|V| - 1$ and out-degree 0 — in time $O(|V|)$, given the adjacency matrix $A[1...n][1...n]$ of G.

Sol. Note that if a graph G has a universal sink s, then its adjacency matrix will have the property: $A[s, j] = 0$ and $A[j, s] = 1$ for all $j \neq s$.

On the adjacency matrix, for any $i \neq j$, if $A[i, j] = 0$, then j cannot be a universal sink (as there is no edge from i to j). If $A[i, j] = 1$, then i cannot be a universal sink (since there is an outgoing edge from i). Thus we can eliminate one vertex from being a universal sink by using a single access from the adjacency matrix.

By using this observation we can arrive at the following algorithm:

Algorithm 2: Find Universal Sink

```
1  procedure FIND-SINK(A)
2      i, j ← 1, n
3      while i < j do
4          if A[i, j] = 0 then
5              j ← j − 1
6          else
7              i ← i + 1
8
9
10     ▷ Check if i is a universal sink
11     for k ← 1 to n do
12         if k ≠ i ∧ (A[i, k] = 1 ∨ A[k, i] = 0) then
13             return NIL
14
15
16     return i
17
```

$j =$	1	2	3	4	5	6	7
$i = 1$	0	1	0	1	1	0	1
2	0	0	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0
7	0	0	0	1	1	0	0

$i = 1, j = 7$	$A[i, j] = A[1, 7] = 1$	$i \leftarrow i + 1$
$i = 2, j = 7$	$A[i, j] = A[2, 7] = 0$	$j \leftarrow j - 1$
$i = 2, j = 6$	$A[i, j] = A[2, 6] = 0$	$j \leftarrow j - 1$
$i = 2, j = 5$	$A[i, j] = A[2, 5] = 1$	$i \leftarrow i + 1$
$i = 3, j = 5$	$A[i, j] = A[3, 5] = 1$	$i \leftarrow i + 1$
$i = 4, j = 5$	$A[i, j] = A[4, 5] = 1$	$i \leftarrow i + 1$
$i = 5, j = 5$	-	

Figure 1: Demonstration of working of Algorithm 2 on a directed graph with universal sink $s = 5$.

Correctness:

We claim that the following invariance holds before and after each iteration of the loop:
- If s is a universal sink in G, then $s \in S = \{i, i + 1, ..., j\}$.

We inductively prove this invariance.

[Basis] Before the loop, $S = \{1, 2, ..., n\} = V$, clearly if s is a universal sink, $s \in V$.

[Step] Assume that $s \in S = \{i, i+1, ..., j\}$. We check the value of $A[i, j]$.

- If $A[i, j] = 1$, then vertex i has an outgoing edge. This means i cannot be a universal sink. Thus, if s is a universal sink, then $s \in S' = \{i+1, ..., j\}$
- If $A[i, j] = 0$, then vertex j has no incoming edge from i. This means j cannot be a universal sink. Thus, if s is a universal sink, then $s \in S'' = \{i, ..., j-1\}$

In either case, the property holds at the end of the body of the loop. This proves that the condition is indeed an invariance.

Thus, it must hold even after the loop. After the loop exits, $i = j$, so $S = \{i\}$. By the invariance, it follows that if s is a universal sink in G, then $s = i$. Hence the given algorithm correctly finds the universal sink if it exists.

Time Complexity: At each iteration the value of $j - i$ decreases by 1. At the beginning of the program, $j - i = n - 1$ and the loop exits when $i = j$. Thus the main loop runs in $O(n)$. The verification loop also runs in $O(n)$, so the overall time complexity is $O(n) = O(|V|)$.

---

5. An undirected weighted graph is given as input in terms of adjacency matrix. Provide an implementation of Prim's algorithm that runs in $O(V^2)$ time.

Sol. By convention, for $u \neq v$, $A[u,v] = w(u,v)$ if $(u,v) \in E$, and $A[u,v] = \infty$ otherwise.

Algorithm 3: Minimum Spanning Tree using Prim's Algorithm using Adjacency Matrix

```
1  procedure MST-Prim(A)
2      for u = 1 to n do
3          visited[u] ← 0                    ▷ whether a vertex has been added to the MST
4          key[u] ← ∞                        ▷ weight of minimal crossing edge to u
5          π[u] ← NIL
6
7      key[1] ← 0
8      for i = 1 to n do
9          u ← Get-Min-Key-Vertex()          ▷ add u to the MST
10         visited[u] ← 1
11         for v ← 1 to n do                 ▷ update info for adjacent nodes
12             if (¬ visited[v]) ∧ (A[u,v] < ∞) ∧ (key[v] > A[u,v]) then
13                 key[v] ← A[u,v]
14                 π[v] ← u
15
16
17
18
19
20 function Get-Min-Key-Vertex()
21     min-v ← NIL
22     best ← ∞
23     for v = 1 to n do
24         if ¬ visited[v] ∧ best > key[v] then
25             best ← key[v]
26             min-v ← v
27
28
29     return min-v
30
```

The MST can be constructed by using the π values.

Time Complexity:

- Initialisation: $O(n)$
- Inside the loop (n iterations)
  - Get-Min-Key-Vertex: $O(n)$
  - Update key and π for adjacent nodes: $O(n)$

Thus, the overall complexity of the above implementation is $O(n^2) = O(V^2)$

6. Let $G = (V, E)$ be a weighted, directed graph with a nonnegative weight function $w : E \to \{0, 1, ..., k\}$ for some integer $k > 0$. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(kV + E)$ time.

Sol. Consider the simple version of Dijkstra's algorithm which uses arrays to store the vertices $Q = V - S$ (S denotes the set of vertices whose shortest path weights have been determined). The complexity of this version is $O(V^2 + E)$, owing to the total $O(V^2)$ computation in all the EXTRACT-MIN operations, which finds the vertex with the minimum $d[v]$ value out of all the vertices in Q.

In our modified version of the problem, we have $0 \leq w(e) \leq k \quad \forall\ e \in E$. This gives a bound on the shortest path weights themselves. The weight of any shortest path P is bounded by

$$0 \leq w(P) \leq k(n-1) < kn$$

This is because any shortest path can have at most $n - 1$ edges, and each edge can have weight at most k.

By using this observation, we have to make our EXTRACT-MIN operations more efficient, so that all of them can be done in $O(kV)$ time. We can do so by storing the vertices in $Q = V - S$ in a bucket-list: Let B be an array of linked lists, where $B[d]$ stores the vertices v with tentative distance $d[v] = d$. In other words, for all $v \in Q, v \in B[d[v]]$. B will need to store at most $k(n - 1)$ lists.

Algorithm 4: Dijkstra's algorithm optimised for small edge weights

```
1  procedure MODIFIED-DIJKSTRA(G, s)
2      for v = 1 to n do
3          d[v] ← ∞
4          π[v] ← NIL
5
6      d[s] ← 0
7      B[0] ← insert s
8      curr ← 0
9      count ← 0
10     while count < n do
11         while B[curr] is empty do
12             curr ← curr + 1
13
14         u ← B[curr].pop()
15         if curr ≠ d[u] then
16             CONTINUE
17
18         count ← count + 1
19         for v in adj[u] do
20             if d[v] > d[u] + w(u, v) then
21                 d[v] ← d[u] + w(u, v)
22                 π[v] ← u
23                 B[d[v]] ← insert v
24
25
26
27
```

Correctness:

We need to show that this is equivalent to the original Dijkstra's algorithm. In particular, we need to show that (i) in each iteration of the loop of line 10, we find the vertex with minimum tentative distance from $V - S$ in the variable u. Also, we need to show that (ii) in the relaxation step, we update our data structures properly.

(i) As described earlier, the vertices in $V - S$ are stored in the array of lists B, such that $v \in B[d[v]]$ for all $v \in V - S$. Also, the vertex u that was moved to S (removed from B) in the last iteration had $d[u] = \text{curr}$. Let v^* be a vertex in $V - S$ with minimum tentative distance. We claim that for vertex v^*, $d[v^*] \geq d[u] = \text{curr}$ (Claim 1). Thus, to find v^*, we increment curr until $B[\text{curr}]$ is non empty, and take an element from $B[\text{curr}]$. This ensures we correctly find the minimum-distance unexplored vertex.

Proof of **Claim 1**: We always remove vertices from $V - S$ in non increasing order of $d[]$ values. This is because while removing, we always remove the minimal element. Also, after a vertex u is removed, some edges will be relaxed, which will update $d[]$ values for some vertices in $V - S$, $(d[v] = d[u] + w(u, v))$ but even after the updates, all the $d[]$ values of vertices in $V - S$ will have $d[v] \geq d[u]$, since all edges have non negative weights.

(ii) Let's look at the relaxation step. Let the current vertex being processed be u. For all $(u, v) \in E$, we relax the edge (u, v), and if it can be relaxed, update $d[v]$ to $d[u] + w(u, v)$ and set $\pi[v] = u$. Let the value of $d[v]$ before relaxing be d_1 and after relaxing be d_2. $(d_2 < d_1)$

Note that in B, we store the vertices $v \in V - S$ according to their $d[v]$ values. Now since we changed the $d[v]$ value for vertex v from d_1 to d_2, we would need to remove v from $B[d_1]$ and insert v to $B[d_2]$. However, removing an arbitrary element from a linked list will increase the complexity. Instead, we retain the outdated copy of v in $B[d_1]$, just insert v in $B[d_2]$. Whenever we pop an item from B, we can check if it is outdated, i.e. if the $d[]$ value of the vertex does not match the index of B from where we popped the vertex. If so, we ignore it.

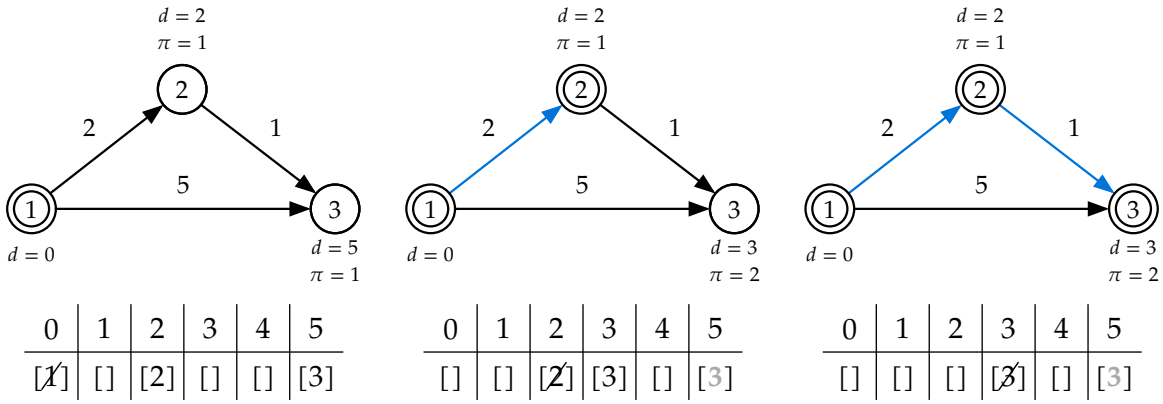

Figure 2: Working out of Modified-Dijkstra on a small example graph. The table shows values stored by $B[d]$ for different values of d. The vertex that was removed in this iteration is striked out. An outdated copy of a vertex is shown in grey. (The initial stage is not shown, when $d[v] = \infty \forall\ v \neq s, d[s] = 0$)

Time Complexity:

1. Lines 11-12: We increment curr; since shortest path distances can only go up to $k\,|V|$, curr $\leq k\,|V|$, so all the increments run in $O(kV)$.

2. Lines 14-17: We pop from B. There will be at most $|E|$ elements in B (Claim 2), so this runs in $O(E)$.
3. Lines 19-23: Since this loops over the adjacency list of all vertices, the loop will run $|E|$ times. This runs in $O(E)$.

Proof of **Claim 2:** Since the loop in lines 19-23 runs $|E|$ times, and we only ever insert into B once in this loop every iterations, there will be at most $|E|$ elements in B.

Thus, the overall time complexity of this algorithm is $O(kV + E)$.
