Soham Chatterjee
Email: soham.chatterjee@tifr.res.in
Course: Algorithms

Assignment - 4
Dept: STCS
Date: January 10, 2025

Problem 2 (20 marks)

Given a directed graph with positive edge weights, give an algorithm to find the second min-weight path between vertices s and t. Note that multiple $s-t$ paths may have the same weight, so the path returned must have weight strictly greater than the minimum weight path.

Solution: For some graphs second min-weight path doesn't exists. For example the graph with 3 vertices with edges $(1,2),(2,3)$ and $(3,1)$ with each edge weight is 1 and $s=1$ Now there is no second min-weight path from 1 to 2 unless we take cycle to return to 1 then take the edge $(1,2)$. So here we assume that we are allowed to visit same vertex multiple times

Algorithm 1: FIND-MAX-PATH(G,s,t,W)

Input: A directed graph $G=(V,E)$ with 2 vertices s,t and non-negative weights on edges w_e for all $e \in E$.
Output: Second minimum weight path from $s \rightsquigarrow t$

```
begin
    dist_1(s) ⟵ 0
    MinPath ⟵ Dijkstra(G, s, {w_e})
    dist_2(s) ⟵ min_{u∈V} {MinPath[u][1] + w_{u,s} : (u,s) ∈ E}        // Suppose minimum at u
    s.parent_1 ⟵ Null, s.parent_2 ⟵ u
    for v ∈ V − {s} do
        dist_1(v) ⟵ ∞, dist_2(v) ⟵ ∞
        v.parent_1 ⟵ Null, v.parent_2 ⟵ Null
    U ⟵ {s}
    while U ≠ ∅ do
        u ⟵ Extract first element of U
        for (u,v) ∈ E do
            if dist_1(v) > dist_1(u) + w(u,v) then
                dist_2(v) ⟵ dist_1(v), dist_1(v) ⟵ dist_1(u) + w_{u,v}
                v.parent_2 ⟵ v.parent_1, v.parent_1 ⟵ u
                U ⟵ U ∪ {v}
            else if dist_2(v) > dist_1(u) + w_{u,v} then
                dist_2(v) ⟵ dist_1(u) + w_{u,v}
                v.parent_2 ⟵ u
                U ⟵ U ∪ {v}
    P ⟵ ∅, p ⟵ t
    while True do
        P ⟵ P ∪ {p}
        if p.parent_2 == Null then
            Break
        p ⟵ p.parent_2
    return Reverse(P)
```

Let $dist_1(v)$ is the minimum weight distance from s to v and $dist_2(v)$ is the second minimum weight distance from s to v for any $v \in V$. Now we have the following relation for the minimum distance $dist_1(v) = \min_{u:(u,v)\in E} \{dist(u) + w_{u,v}\}$. Therefore if we have $dist_1(u)$ and $dist_1(v)$ at any point then we can update $dist_1(v)$ by $\min\{dist_1(v), dist_1(u) + w_{u,v}\}$. Now if $dist_1(u) + w_{u,v} < dist_1(v)$ then we know second minimum distance

of v form s is $dist_1(v)$. So we update $dist_2(v)$ by $dist_1(v)$. Now suppose $dist_1(u) + w_{u,v} \geq dist_1(v)$. Now if $dist_1(v) \leq dist_1(u) + w_{u,v} < dist_2(v)$ then we know second minimum distance of v from s is $dist_1(u) + w_{u,v}$. So for s we have the $dist_1(s) = 0$ but the $dist_2(s)$ to be the weight of the minimum weight cycle which contains s. So in this case we update $dist_2(v)$ by $dist_1(u) + w_{u,v}$.

Here we assume that $\textsc{Dijkstra}(G, s, \{w_e\})$ algorithm basically returns the list minimum distance paths from s to all the other vertices in the array $\textsc{MinPath}$. Let $\textsc{MinPath}[u]$ is the pair $(p_u, dist(u))$ where p_u is the minimum weight path from s to u and $dist(u)$ is the distance from s to u. We also assume this also updates the $parent_2$ pointers to get the minimum weight paths. Since it returns the minimum paths to all the other vertices we can assume at most we run the original $\textsc{Dijkstra}$ algorithm on all the vertices as the end points. Hence it takes $O(n^3)$ time.

Time Complexity: The lines 2,5 takes constant time. The Dijkstra algorithm in line 3 takes $O(n^3)$ time. The calculating $dist_2$ at line 4 takes $O(n)$ time. the for loop at line 6 takes $O(n)$ time since each iteration it takes constant time. The while loop at line 10 picks a vertex and then goes through all its neighbors and adds them to U. So the while loop visits at most every vertex and and every edges n times so that it have to update the distances every time for each vertex and its neighbors. In each iteration it takes constant time. Therefore the while loop takes $O(n^3)$ time. Now for the while loop at line 21 it can be at most n many iterations and in each iteration it takes constant time. The $\textsc{Reverse}$ function also take linear time. Therefore the algorithm runs in $O(n^3)$ time. ■

[Me and Soumyadeep discussed the problem together]

Problem 3 (15 marks)

Let G be a graph with $2n$ vertices. A bisection of G is a cut (S, T) with $|S| = |T|$. Since G has an even number of vertices, it clearly has a bisection. Find a probabilistic polynomial-time algorithm that produces a bisection with an expected cut (i.e., number of edges across cut) at least half that of the maximum. Then convert your algorithm to a deterministic polynomial-time algorithm.

Solution:

Algorithm 2: $\textsc{Bisection-Cut}$

Input: Graph $G = (V, E)$ where $|V| = 2n$
Output: Bisection of G, (S, T)

```
begin
    U ⟵ V;
    S ⟵ ∅, T ⟵ ∅;
    for i ∈ [2n] do
        v ⟵ uniformly random vertex picked from U.;
        U ⟵ U \ {v};
        if i ≡ 0 mod 2 then
            S ⟵ S ∪ {v}
        else
            T ⟵ T ∪ {v}
    return (S, T)
```

Let r be any permutation of the $2n$ vertices. Then probability of at i^{th} iteration the sample vertex is $r(i)$ for all $i \in [2n]$ is $\prod_{i=1}^{2n} \frac{1}{i} = \frac{1}{(2n)!}$. Now for any bisection cut (S, T) probability of obtaining that cut is $\frac{n!n!}{(2n)!}$ since all the permutations where the odd indexed elements are same as in S and even indexed elements are same as T gives output to the bisection (S, T) and number such permutations are $n!n!$ since we just have to permute the elements of S within themselves and permute the elements of T within themselves. Now observe there are $\binom{2n}{n}$ many bisections of the $2n$ vertices are possible. Hence we sample any bisection of the $2n$ vertices uniformly at random.

Let for any edge $e \in E$, $e = (u, v)$, I_e be the indicator random variable that it becomes a cut edge. So

$\mathbb{P}[I_e = 1] = \mathbb{P}[e \text{ becomes a cut edge}]$. Now there are $\binom{2n-2}{n-1}$ bisections for which $u \in S$ and $v \in T$ and similarly $\binom{2n-2}{n-1}$ bisections for which $u \in T$ and $v \in E$. Hence

$$\mathbb{P}[I_e = 1] = \frac{2\binom{2n-2}{n-1}}{\binom{2n}{n}} = \frac{2n^2}{(2n)(2n-1)} = \frac{n}{2n-1} \geq \frac{n}{2n} = \frac{1}{2}$$

Now if we consider S_e as the random variable for the number of edges across bisection cut given by the algorithm then

$$\mathbb{E}[S_e] = \mathbb{E}\left[\sum_{e \in E} I_e\right] = \sum_{e \in E} \mathbb{E}[I_e] \geq \frac{|E|}{2}$$

Hence the given probabilistic algorithm gives a bisection cut with an expected cut at least half that of the maximum.

Let at i^{th} iteration of the algorithm R_i denote the random variable which takes value from any of the vertex from U. So we write $R_i = r_i$ when the uniformly random sampled vertex at i^{th} iteration is r_i where $r_i \in U$. Let $S_i = \{r_j : j \in [i], j \equiv 1 \bmod 2\}$ and $T_i = \{r_j : j \in [i], j \equiv 0 \bmod 2\}$ i.e. S_i and T_i are the state of the sets S, T in the algorithm after i^{th} iteration. Also denote U_i be the set of rest of the vertices i.e. $U_i = V \setminus \{r_1, \ldots, r_i\}$. So denote

$$C(r_1, \ldots, r_i) = \mathbb{E}[S_e \mid R_1 = r_1, \ldots, R_i = r_i]$$

We already found that $C(\emptyset) = \mathbb{E}[S_e] \geq \frac{|E|}{2}$. First of all we can see that

$$C(r_1, \ldots, r_i) = Cut(S_i, T_i) + \sum_{\text{e incident on } U_i} \mathbb{E}[I_e]$$

We will now analyze this case wise.

Suppose i is even. Then there are even number of vertices are remaining. Now consider all the edges which are not incident on $S_i \cup T_i$. Then for all such edges it is as if the problem is reduced to a graph with $2n - i = 2(n-k)$ vertices and we are starting with that. So the for such edge, e the probability of such edge becoming a cut edge is $\mathbb{P}[I_e = 1] = \frac{n-\frac{i}{2}}{2n-i-1} = \frac{2n-i}{2(2n-i-1)}$. Now take an edge $e = (u, v)$ where $u \in S_i$ but $v \notin S_i \cup T_i$. Then probability of such edge becomes a cut edge i.e. $\mathbb{P}[I_e = 1] = \frac{\binom{2n-i-1}{n-\frac{i}{2}}}{\binom{2n-i}{n-\frac{i}{2}}} = \frac{1}{2}$. Because of similarity for any edge e whose one vertex is in T_i and other is not in $S_i \cup T_i$ the probability of such edge becomes a cut edge is $\frac{1}{2}$. Therefore we finally get that

$$\begin{aligned}
&\sum_{\text{e incident on } U_i} \mathbb{E}[I_e] \\
&= \sum_{e \text{ not incident on } S_i \cup T_i} \mathbb{E}[I_e] + \sum_{e \in S_i \times \overline{S_i \cup T_i} \cap E} \mathbb{E}[I_e] + \sum_{e \in T_i \times \overline{S_i \cup T_i} \cap E} \mathbb{E}[I_e] \\
&= \frac{2n-i}{2(2n-i-1)} |\#\text{edges not incident on } S_i \cup T_i| + \frac{1}{2} |\#\text{edges whose only one vertex is incident on } S_i \cup T_i|
\end{aligned}$$

Therefore

$$\begin{aligned}
C(r_1, \ldots, r_i) = Cut(S_i, T_i) + \frac{(2n-i)|\#\text{edges not incident on } S_i \cup T_i|}{2(2n-i-1)} \\
+ \frac{|\#\text{edges whose only one vertex is incident on } S_i \cup T_i|}{2}
\end{aligned}$$

Now suppose i is odd. Let $i = 2k - 1$ form for some k. Consider the edges which are not incident on $S_i \cup T_i$. Now let $e = (u, v)$ be such an edge. There are $\binom{2n-(2k+1)}{n-k-1}$ bisection cuts for which $u \in S$ and $v \in T$. Similarly there are $\binom{2n-(2k+1)}{n-k-1}$ bisection cuts for which $u \in T$ and $v \in S$. Then probability of e becomes a cut edge is $\frac{2\binom{2n-(2k+1)}{n-k-1}}{\binom{2n-(2k-1)}{n-i}} = \frac{n-k+1}{2n-2k+1}$. Now let $e = (u, v)$ is an edge such that $u \in S_i$ but $v \notin S_i \cup T_i$. The number of bisection for which $v \in T$ is $\binom{2n-2k}{n-k}$. Then probability e becomes a cut edge is $\frac{\binom{2n-2k}{n-k}}{\binom{2n-(2k-1)}{n-i}} = \frac{n-k+1}{2n-2k+1}$. Now suppose $e = (u, v)$ is an

edge such that $u \notin S_i \cup T_i$ but $v \in T_i$. Then number of bisections for which $u \in S$ is $\binom{2n-2k}{n-k-1}$. Therefore probability that e becomes a cut edge is $\frac{\binom{2n-2k}{n-k-1}}{\binom{2n-(2k-1)}{n-i}} = \frac{n-k}{2n-2k+1}$. Therefore we get

$$\begin{aligned} C(r_1,\dots,r_i) = Cut(S_i,T_i) &+ \frac{n-k+1}{2n-2k+1}|\#\text{edges not incident on } S_i \cup T_i| \\ &+ \frac{n-k+1}{2n-2k+1}|\#\text{edges whose only one vertex incident on } S_i| \\ &+ \frac{n-k}{2n-2k+1}|\#\text{edges whose only one vertex incident on } T_i| \end{aligned}$$

Now from all the edges we can calculate $C(r_1,\dots,r_i)$ efficiently. At i^{th} step of the algorithm instead of uniformly picking a random vertex from U_i we calculate $C(r_1,\dots,r_{i-1},v)$ for all $v \in U_i$ and pick the v for which the value of the function is largest. This way we have derandomized the problem to get a polynomial time problem. To calculate $C(r_1,\dots,r_{i-1},v)$ it takes $O(|E|)$ time. Now we calculate $C(r_1,\dots,r_{i-1},v)$ for all $v \in U_i$. Hence at each iteration it takes $O(|V|\cdot|E|)$ time. Now there are $|V|$ iterations. Hence the derandomized algorithm takes $O(|V|^2|E|)$ time. ■

[Me and Soumyadeep discussed the problem together]

Problem 4 (10 marks)

Consider a random walk on a path with vertices numbered $1,2,\dots,n$ from left to right. At each step, we flip an unbiased coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path either by moving left from vertex 1 or right from vertex n. If we start from vertex 1 , what is the probability that the walk ends by falling off the right end of the path?

Solution: We are in the new line starting from 1 we want to go to the state n without hitting 0. Let

$$f(k) = \mathbb{P}[\text{Falling off the right end of the path starting at } k]$$

Then we have

$$f(1) = \frac{1}{2}f(2),\ f(2) = \frac{1}{2}f(1) + \frac{1}{2}f(3),\ \dots,\ f(k) = \frac{1}{2}f(k-1) + \frac{1}{2}f(k+1),\ \dots,\ f(n) = \frac{1}{2} + \frac{1}{2}f(n-1)$$

Now $f(1) = \frac{1}{2}f(2)$.

$$f(2) = \frac{1}{2}f(1) + \frac{1}{2}f(3) \implies \frac{1}{4}f(2) + \frac{1}{2}f(3) \implies f(2) = \frac{2}{3}f(3)$$

We will show using induction that $f(k) = \frac{k}{k+1}f(k+1)$ for all $k \in [n-1]$. For base cases we just showed this is true. Let this is true for $k-1$. So $f(k-1) = \frac{k-1}{k}f(k)$. Now we have

$$f(k) = \frac{1}{2}f(k-1) + \frac{1}{2}f(k+1) = \frac{1}{2}\frac{k-1}{k}f(k) + \frac{1}{2}f(k+1) \implies \frac{k+1}{2k}f(k) = \frac{1}{2}f(k+1) \implies f(k) = \frac{k}{k+1}f(k+1)$$

Hence by induction this is true for all $k \in [n-1]$. Hence

$$f(n) = \frac{1}{2} + \frac{1}{2}\frac{n-1}{n}f(n) \implies \frac{n+1}{2n}f(n) = \frac{1}{2} \implies f(n) = \frac{n}{n+1}$$

Therefore we have

$$\prod_{i=1}^{n-1} f(i) = \prod_{i=1}^{n-1} \frac{i}{i+1}f(i+1) \implies f(1) = \frac{1}{n}f(n) = \frac{1}{n}\frac{n}{n+1} = \frac{1}{n+1}$$

Therefore starting from vertex 1 the probability that walk ends by falling off the right end of the path is $\frac{1}{n+1}$. ■

[Me and Soumyadeep discussed the problem together]

Problem 5 (20 marks)

This problem deals with an efficient technique for verifying matrix multiplication. The fastest known algorithm for multiplying two $n \times n$ matrices runs in $O(n^{\omega})$ time, where $\omega \approx 2.37$. This is significantly faster than the obvious $O\left(n^3\right)$ algorithm but this $O(n^{\omega})$ algorithm has the disadvantage of being extremely complicated. Suppose we are given an implementation of this algorithm and would like to verify its correctness. Since program verification is a difficult task, a reasonable goal might be to verify the correctness of the output produced on specific executions of the algorithm. In other words, given $n \times n$ matrices A, B, and C with entries from rational numbers, we would like to verify that $AB = C$. Note that here we want to use the fact that we do not have to compute C; rather, our task is to verify that the product is indeed C. Give an $O\left(n^2\right)$ time randomized algorithm for this problem with error probability at most $1/2$.

Solution: We will use the following lemma:

Lemma 1 (Schwartz–Zippel lemma)**.** *Let $\mathbb{F}$ be a field and $P \in \mathbb{F}[X_1, \ldots, X_n]$ be a n-variate degree d nonzero polynomial over $\mathbb{F}$. Then for any finite set $S \subseteq \mathbb{F}$, $\mathop{\mathbb{P}}_{a \in S^n}[P(a) = 0] \leq \frac{d}{|S|}$*

Now we will explain the algorithm:

Algorithm 3: VERIFY-MATRIX-MULT

Input: $n \times n$ matrices A, B, C
Output: Verify if $AB = C$

```
begin
    Pick r ∈ {0,1}^n uniformly at random
    if A(Br) − (Cr) == 0^n then
        return True
    return False
```

Probability Analysis: If $AB = C$ then this algorithm always returns *True*. Suppose $AB \neq C$. Think of the vector x as the variable vector $\begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$. Then Bx is a vector where each entry is a n-variate linear polynomial. Therefore each entry of $A(Bx)$ linear combination of entries in Bx. Therefore each entry of $A(Bx)$ is also n–variate linear polynomial. Similarly Cx is also a linear n–variate polynomial vector. Therefore $A(Bx) - Cx$ is a linear n–variate polynomial vector. So we have $AB = C \iff A(Bx) - Cx \equiv 0$. Since we assumed $AB \neq C$, $A(Bx) - Cx \not\equiv 0$. Let F be the polynomial be the linear polynomial vector $A(Bx) - Cx$. Let $F = \begin{bmatrix} f_1 & \cdots & f_n \end{bmatrix}^T$. Then there exists at least one $i \in [n]$ such that $f_i(X_1, \ldots, X_n) \neq 0$. Now for a random $r \in \{0.1\}^n$ for any $j \in [n]$, $f_j(r) = 0$ with probability at most $\frac{1}{2}$ by the Lemma 1. Therefore for a random $r \in \{0,1\}^n$, $F(r) = 0 \implies f_i(r) = 0$ which can happen with probability at most $\frac{1}{2}$. Therefore $\mathop{\mathbb{P}}_{a \in S^n}[F(r) = 0 \mid AB \neq C] \leq \frac{1}{2}$. Hence the error probability of this algorithm is at most $\frac{1}{2}$.

Time Complexity: Let M be any $n \times n$ matrix. For any row m_i of M to compute $m_i r$ it takes $O(n)$ time. There are n rows. Hence to compute Mr it takes $O(n^2)$ time. Here we are first computing Br and Cr and the $A(Br)$ which takes in total $O(n^2)$ time. Therefore the algorithm runs in $O(n^2)$ time. ■

[Me and Soumyadeep discussed the problem together]

Problem 6 (20 marks)

Show how to implement the push-relabel algorithm using $O(n)$ time per relabel operation, $O(1)$ time per push, and $O(1)$ time to select an applicable operation, for a total running time of $O\left(mn^2\right)$.

Solution: Let $l(v)$ denote the label of v for any $v \in V$. For each vertex $v \in V$ we keep a list of vertices $v.push-vertex$ which contains all the vertices $u \in V$ such that $l(v) = l(u) + 1$. Each time we relabel we update the $v.push-vertices$ list.

In the Relabel operation we update $l(u)$ by $1 + \min\{l(v) \colon (u,v) \in E_f\}$. To calculate the minimum it takes $O(n)$ time. Now we first make $u.push-vertices$ empty. Then for each vertex $v \in N(u)$ in E_f if $l(u) = l(v) + 1$ then we add v to the list $u.push-vertices$. This also takes at most $O(n)$ time. Hence the Relabel operation takes $O(n)$ time.

Now in each Push operation it is executed in constant time if we can obtain $excess_f(u)$ and $c_f(u,v)$. We keep a list *overflow* which contains the list of vertices which are overflowing. So everytime we Push along (u,v) we update $excess_f(u), excess_f(v)$ and after updating them if $excess_f(u) > 0$ keep u in *overflow* list and similarly if $excess_f(v) > 0$ then keep v in *overflow* list. All of these can be done in constant time. Therefore the Push operation takes constant time.

So every iteration of the while loop we check if the *overflow* list is empty. If it is nonempty we take the first vertex, say u. Then we check if $u.push-vertices$ empty or not. If it is empty we do the Relabel operation. If $u.push-vertices$ is nonempty we take a $v \in u.push-vertices$. Then do the Push operation along (u,v). Hence to select an applicable operation it also takes constant time. ■

[Me and Soumyadeep discussed the problem together]

Problem 7 (15 marks)

Given an undirected graph G, an edge coloring is an assignment of colors to the edges of G such that no two edges incident to the same vertex get the same color. The edge coloring problem asks for an edge coloring using the minimum number of colors. This is a hard problem to solve. Instead, design a 2-approximation algorithm for the edge coloring problem that runs in polynomial time (i.e., if the optimal solution in an instance uses k colors, your algorithm should use at most $2k$ colors).

Solution: Let the maximum degree of the graph D. Since for any vertex v all the edges incident of v has different color the optimal edge coloring of the graph uses. Therefore $k \geq D$.

The algorithm starts from any vertex and colors the edges incident on it then moves to its neighbors. In this the algorithm tries the color the edges with the lowest possible number in $\mathbb{Z}_+$.

In this algorithm suppose at any step the algorithm colors l to an edge incident on a vertex v on which the incident edges we are coloring currently. Let the edge is (v,u). Since the algorithm colored the edge l then all the colors $[l-1]$ appeared on the edges incident on either v or u. Now degree of u, v are at most D. Therefore there are at most $2D-2$ except the edge (u,v) which are incident on either u or v. Hence the color of the edge (u,v) has to be different than all these other $2D-2$ edges. Therefore l can be at most $2D-1$. Since if $l \geq 2D$ then there are total $2D-1$ edges except (u,v) incident on either u or v but that is not possible.

Therefore this algorithm gives an edge coloring with at most $2D-1$ colors. Therefore the number of colors is less than $2D \leq 2k$. Hence this algorithm gives a 2–approximation algorithm for edge coloring. ■

[Me and Soumyadeep discussed the problem together]

Problem 8 (10 marks)

Show that the number of distinct minimum cuts in an undirected graph (assume all edge weights are one) is at most $\binom{n}{2}$. That is, show that the number of distinct cuts whose value is equal to the value of the min-cut in the graph is at most $\frac{n(n-1)}{2}$.

Solution: Suppose there are k min cuts in an undirected graph. By Karger's algorithm we know for any $I \in [k]$, probability of getting the ith mincut as output is at least $\frac{1}{\binom{n}{2}}$. Hence probability of getting any mincut as output by the the Karger's algorithm is at least $\frac{k}{\binom{n}{2}}$. Now this probability have to be less than or equal to 1. Therefore we get $\frac{k}{\binom{n}{2}} \leq 1 \implies k \leq \binom{n}{2}$. Therefore in the graph there can be at most $\binom{n}{2}$ many mincuts. ■

[I discussed with Vivek and Soumyadeep.]