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

**Assignment - 4**
Dept: STCS
Date: November 28, 2024

**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:*** 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 in this case we update $dist_2(v)$ by $dist_1(u) + w_{u,v}$. Otherwise we still have the minimum and second minimum distance so we don't need to update the distances. So we have the following algorithm:

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

**Input:** A directed acyclic 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 $s \rightsquigarrow t$

```
1  begin
2      dist_1(s) ⟵ 0, dist_2(s) ⟵ 0
3      s.parent_1 ⟵ Null, s.parent_2 ⟵ Null
4      for v ∈ V − {s} do
5          dist_1(v) ⟵ ∞                  // dist_1 stores the minimum distance from s
6          dist_2(v) ⟵ ∞          // dist_2 stores the second minimum distance from s
7          v.parent_1 ⟵ Null, v.parent_2 ⟵ Null
8      U ⟵ {s}
9      while U ≠ ∅ do
10         u ⟵ Extract first element of U
11         for (u, v) ∈ E do
12             if dist_1(v) > dist_1(u) + w(u, v) then
13                 dist_2(v) ⟵ dist_1(v), dist_1(v) ⟵ dist_1(u) + w(u, v)
14                 v.parent_2 ⟵ v.parent_1, v.parent_1 ⟵ u
15             else if dist_2(v) > dist_1(u) + w(u, v) then
16                 dist_2(v) ⟵ dist_1(u) + w(u, v)
17                 v.parent_2 ⟵ u
18             U ⟵ U ∪ {v}
19     P ⟵ ∅, p ⟵ t
20     while True do
21         P ⟵ P ∪ {p}
22         if p.parent_2 == Null then
23             Break
24         p ⟵ p.parent_2
25     return Reverse(P)
```

**Time Complexity:** The lines 2-3 takes constant time and the for loop at line 4 takes $O(n)$ time since each iteration it takes constant time. The while loop at line 9 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. In each iteration it takes constant time. Therefore the while loop takes $O(|V|+|E|) = O(n^2)$ time. Now for the while loop at line 20 it can be at most $n$ many iterations and in each iteration it takes constant time. The REVERSE function also take linear time. Therefore the algorithm runs in $O(n^2)$ time or $O(|V|+|E|)$ time. ■

**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:*** ■

**Problem 4** (10 marks)

Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ 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

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

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 $k \in [n-1]$. Therefore we have

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

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

■

**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 out-

put 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 2:** VERIFY-MATRIX-MULT

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

```
1 begin
2     Pick r ∈ {0,1}^n uniformly at random
3     if A(Br) − (Cr) == 0^n then
4         return True
5     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 $\begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$. Let $A = (a_{i,j})_{1 \leq i,j \leq n}$, $B = (b_{i,j})_{1 \leq i,j \leq n}$, $B = (b_{i,j})_{1 \leq i,j \leq n}$. 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. Therefore $A(Bx) - Cx$ is a linear $n-$variate polynomial. 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. ■

**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 $1 + \min\{l(v) : (u,v) \in E_f\}$. To calculate the minimum it takes $O(n)$ time. Now we first make $u.push - vertices$ empty. Then for each vertex $v \in N(u)$ 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. ■

**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:***

■