CS 473: Algorithms

Ruta Mehta

University of Illinois, Urbana-Champaign

Spring 2018

Review session

Lecture 99

Feb 22, 2018

Some of the slides are courtesy Prof. Chekuri

What we saw so far...

Fast Fourier Transform (FFT).

Dynamic Programming

- String algorithms.
- Graph algorithms: shortest path, independent set, dominating set, etc.

Randomozed Algorithms

- Quick sort,
- High probability analysis: Markov, Chebyshev, and Chernoff inequalities

What we saw so far...

Fast Fourier Transform (FFT).

Dynamic Programming

- String algorithms.
- Graph algorithms: shortest path, independent set, dominating set, etc.

Randomozed Algorithms

- Quick sort,
- High probability analysis: Markov, Chebyshev, and Chernoff inequalities
- Hashing, Fingerprinting

Part I

FFT

What is Fast Fourier Transform

Definition

Given a polynomial $a = (a_0, a_1, \ldots, a_{n-1})$ in coefficient representation the *Discrete Fourier Transform* (DFT) of a is the vector $a' = (a'_0, a'_1, \ldots, a'_{n-1})$ where $a'_j = a(\omega_n^j)$ for $0 \leq j < n$.

a' is a sample representation of polynomial with coefficient reprentation a at n'th roots of unity.

We have shown that a' can be computed from a in $O(n \log n)$ time. This divide and conquer *algorithm* is called the *Fast Fourier Transform* (FFT).

Why FFT? Convolution and Polynomial Multiplication

Convolution

Convolution of vectors $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$ is a vector $c = (c_0, c_1, \ldots, c_{2n-2})$, where

$$c_k = \sum_{i,j:\, i+j=k} a_i \cdot b_j$$

Why FFT? Convolution and Polynomial Multiplication

Convolution

Convolution of vectors $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$ is a vector $c = (c_0, c_1, \ldots, c_{2n-2})$, where

$$c_k = \sum_{i,j:\, i+j=k} a_i \cdot b_j$$

Polynomial Multiplication

If vectors a and b are coefficients of two $n-1$ degree polynomials, (abusing notation) $a(x) = \sum_{i=0}^{n-1} a_i x^i, \quad b(x) = \sum_{i=0}^{n-1} b_i x^i$ then c is the coefficient vector of the product polynomial $a(x) * b(x)$.

Why FFT? Convolution and Polynomial Multiplication

Convolution

Given vectors $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$ find its convolution vector $c = (c_0, c_1, \ldots, c_{2n-2})$.

1. Evaluate polynomials a and b at the $2n$th roots of unity, to get their sample representation a' and b'.
2. Compute sample representation $c' = (a'_0 b'_0, \ldots, a'_{2n-2} b'_{2n-2})$ of product $c = a \cdot b$
3. Compute c from c' using inverse Fourier transform.

- Step 1 takes $O(n \log n)$ using two FFTs
- Step 2 takes $O(n)$ time
- Step 3 takes $O(n \log n)$ using one FFT

Problem

Let $\bar{a} = a_0, a_1, \ldots, a_{n-1}$ be a sequence of n numbers representing value of a function at different points, we would like to "smooth" it using vector $\bar{b} = (b_0, b_1, \ldots, b_{k-1})$ for $k \leq n$ as follows: $\bar{a}' = a'_0, a'_1, \ldots, a'_{n-1}$ where $a'_i = a_i b_0 + (a_{i+1} b_1 + \ldots + a_{i+k-1} b_{k-1}) + (a_{i-1} b_1 + a_{i-2} b_2 + \ldots + a_{i-k+1} b_{k-1})$. If an index goes out of bounds we assume that the corresponding value is $\mathbf{0}$. Given $\bar{a}$ and $\bar{b}$ describe how $\bar{a}'$ can be computed in $O(n^2)$ time.

Application of FFT

Let $\bar{a} = a_0, a_1, \ldots, a_{n-1}$ be a sequence of n numbers representing value of a function at different points, we would like to "smooth" it using vector $\bar{b} = (b_0, b_1, \ldots, b_{k-1})$ for $k \leq n$ as follows: $\bar{a}' = a'_0, a'_1, \ldots, a'_{n-1}$ where $a'_i = a_i b_0 + (a_{i+1} b_1 + \ldots + a_{i+k-1} b_{k-1}) + (a_{i-1} b_1 + a_{i-2} b_2 + \ldots + a_{i-k+1} b_{k-1})$. If an index goes out of bounds we assume that the corresponding value is $\mathbf{0}$. Given $\bar{a}$ and $\bar{b}$ describe how $\bar{a}'$ can be computed in $O(n \log n)$ time.

$$\bar{a}(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

$$c = b(x) = \overset{c_0}{b_{k-1}} + \overset{c_1}{b_{k-2}} x + \cdots + \overset{c_{k-1}}{b_0} x^{k-1} + b_1 x^{k} + \cdots + b_{k-1} x^{2k-2}$$

c_{k+1} $\quad c_{2k-2}$

$$\bar{a} * b \;\operatorname{coeff}(x^{\ell}) = \sum_{i,j,\, i+j=\ell} a_i \cdot x^i \cdot c_j x^j = \sum_{i,j:\, i+j=\ell} a_i c_j$$

$$a'_0 = \text{coeff}(x^{k-1}) = a_0 \cdot b_0 + a_1 b_1 + \cdots + a_{k-1} b_{k-1} + 0 + 0 + \cdots + 0$$

$$b_1(x) = b_0 + b_1 x + \cdots + b_{k-1} x^{k-1}$$

$$a * b_1 : \quad \text{coeff}(x^i) = \underset{x^0 \cdot x^i}{b_0 \cdot a_i} + \underset{x^1 \;\; x^{i-1}}{b_1 \cdot a_{i-1}} + \cdots$$

$$\cdots + \underset{x^{k-1} \;\; x^{i-k+1}}{b_{k-1} \, a_{i-k+1}}$$

Part II

Dynamic Programming

Recursion

Reduction:

Reduce one problem to another

Recursion

A special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size n is reduced to one or more instances of size $n - 1$ or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.

What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

Dynamic Programming:

A recursion that when memoized leads to an *efficient* algorithm.

Edit Distance

Definition

Edit distance between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X.

Example

The edit distance between FOOD and MONEY is at most **4**:

$$\mathbf{\underline{F}OOD} \rightarrow \mathbf{MO\underline{O}D} \rightarrow \mathbf{MON\underline{O}D} \rightarrow \mathbf{MONE\underline{D}} \rightarrow \mathbf{MONEY}$$

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F	**O**	**O**		**D**
M	**O**	**N**	**E**	**Y**

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears exactly once, and there is no "crossing": if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of i or j could be empty, in which case no comparision. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (\ , 4), (4, 5)\}$.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set M of pairs (i, j) such that each index appears exactly once, and there is no "crossing": if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of i or j could be empty, in which case no comparision. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (\ , 4), (4, 5)\}$.

Cost of an alignment: the number of mismatched columns.

Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

Edit Distance

Basic observation

Let $A = \alpha x$ and $B = \beta y$
α, β: strings. x and y single characters.
Possible alignments between A and B

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

Edit Distance

Basic observation

Let $A = \alpha x$ and $B = \beta y$
α, β: strings. x and y single characters.
Possible alignments between A and B

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

$$EDIST(A, B) = \min \begin{cases} EDIST(\alpha, \beta) + [x \neq y] \\ 1 + EDIST(\alpha, B) \\ 1 + EDIST(A, \beta) \end{cases}$$

Recursive Algorithm

Assume strings are given as arrays $A[1..m]$ and $B[1..n]$

```
EDIST(A[1..i], B[1..j])
    If (i = 0) return j
    If (j = 0) return i
    m1 = 1 + EDIST(A[1..(i − 1)], B[1..j])
    m2 = 1 + EDIST(A[1..i], B[1..(j − 1)]))
    If (A[m] = B[n]) then
        m3 = EDIST(A[1..(i − 1)], B[1..(j − 1)])
    Else
        m3 = 1 + EDIST(A[1..(i − 1)], B[1..(j − 1)])
    return min(m1, m2, m3)
```

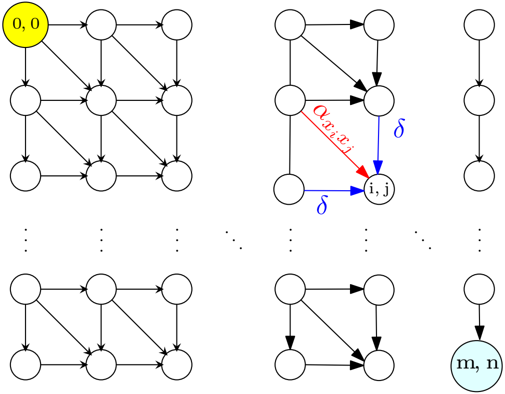
Call $EDIST(A[1..m], B[1..n])$

Memoizing the Recursive Algorithm

```
int M[0..m][0..n]
Initialize all entries of M[i][j] to ∞
return EDIST(A[1..m], B[1..n])
```

```
EDIST(A[1..i], B[1..j])
    If (M[i][j] < ∞) return M[i][j]   (* return stored value *)
    If (i = 0)
        M[i][j] = j
    ElseIf (j = 0)
        M[i][j] = i
    Else
        m1 = 1 + EDIST(A[1..(i − 1)], B[1..j])
        m2 = 1 + EDIST(A[1..i], B[1..(j − 1)]))
        If (A[i] = B[j]) m3 = EDIST(A[1..(i − 1)], B[1..(j − 1)])
        Else m3 = 1 + EDIST(A[1..(i − 1)], B[1..(j − 1)])
        M[i][j] = min(m1, m2, m3)
    return M[i][j]
```

Matrix and DAG of Computation



Removing Recursion to obtain Iterative Algorithm

EDIST$(A[1..m], B[1..n])$
    int $M[0..m][0..n]$
    for $i = 0$ to m **do** $M[i, 0] = i$
    for $j = 0$ to n **do** $M[0, j] = j$

    for $i = 1$ to m **do**
        for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} [x_i \neq y_j] + M[i-1][j-1], \\ 1 + M[i-1][j], \\ 1 + M[i][j-1] \end{cases}$$

Removing Recursion to obtain Iterative Algorithm

$EDIST(A[1..m], B[1..n])$
    $int \quad M[0..m][0..n]$
    for $i = 0$ `to` m **do** $M[i,0] = i$
    for $j = 0$ `to` n **do** $M[0,j] = j$

    for $i = 1$ `to` m **do**
        for $j = 1$ `to` n **do**

$$M[i][j] = \min \begin{cases} [x_i \neq y_j] + M[i-1][j-1], \\ 1 + M[i-1][j], \\ 1 + M[i][j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.

Removing Recursion to obtain Iterative Algorithm

```
EDIST(A[1..m], B[1..n])
    int  M[0..m][0..n]
    for i = 0 to m do M[i, 0] = i
    for j = 0 to n do M[0, j] = j

    for i = 1 to m do
        for j = 1 to n do
```

$$M[i][j] = \min \begin{cases} [x_i \neq y_j] + M[i-1][j-1], \\ 1 + M[i-1][j], \\ 1 + M[i][j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.
2. Space used is $O(mn)$.

Matrix and DAG of Computation

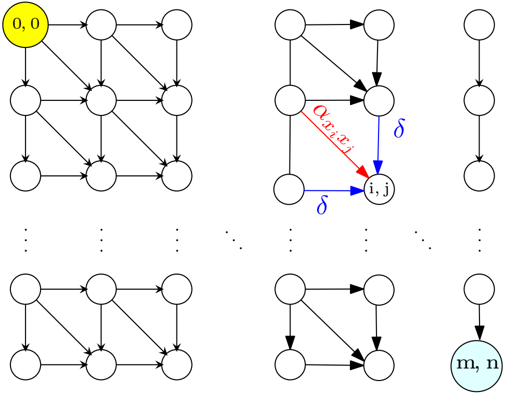
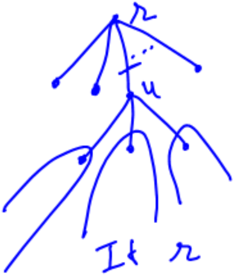


Figure: **Iterative algorithm in previous slide computes values in row order.**

Problem

Given a graph $G = (V, E)$ a matching is a set of edges $M \subset E$ such that no two edges in M share an end point. Describe an efficient algorithm that given a tree $T = (V, E)$ and non-negative weights $w : E \to R^+$ finds a maximum weight matching in T.



If r is not matched then: $M(r) = \sum_{u \in child(r)} M(u) \to ①$

If r is indeed matched then

$$M(r) = \max_{u \in child(r)} w(r,u) + \sum_{\substack{v \neq u \\ v \in child(r)}} M(v) + \sum_{v \in child(u)} M(v)$$

$\to ②$

$$M(r) = \max \begin{cases} ① \\ ② \end{cases}$$

Iterative Algorithm:

Initialization: $M(u) = 0$ $\forall u$ leaf

Evaluate in post order traversal of tree rooted at r.

Dijkstra's Algorithm

```
Initialize for each node v, dist(s, v) = ∞
Initialize S = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−S} dist(s, u)
    S = S ∪ {v}
    for each u in Adj(v) \ S do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

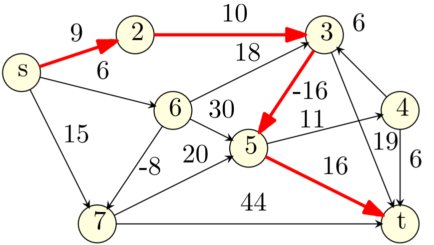
1. Using Fibonacci heaps. Running time: $O(m + n \log n)$.
2. Can compute shortest path tree.

Single-Source Shortest Paths with Negative Edge Lengths

Single-Source Shortest Path Problems

Input: A *directed* graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

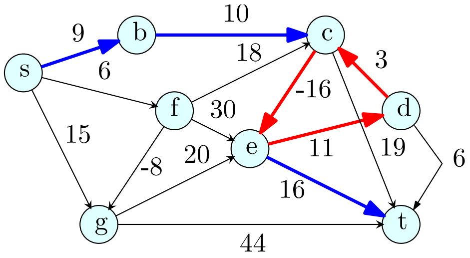
- Given nodes s, t find shortest path from s to t.
- Given node s find shortest path from s to all other nodes.



Negative Length Cycles

Definition

A cycle C is a negative length cycle if the sum of the edge lengths of C is negative.



Dijkstra's algorithm does not work with negative edges.

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_i$ *is a shortest path from s to v_i*

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_i$ *is a shortest path from s to v_i*

Sub-problem idea: paths of fewer hops/edges

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
Assume that all nodes can be reached by s in G. (Remove nodes unreachable from s).

$d(v, k)$: shortest walk length from s to v using at most k edges.

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
Assume that all nodes can be reached by s in G. (Remove nodes unreachable from s).

$d(v, k)$: shortest walk length from s to v using at most k edges.
Recursion for $d(v, k)$:

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
Assume that all nodes can be reached by s in G. (Remove nodes unreachable from s).

$d(v, k)$: shortest walk length from s to v using at most k edges.
Recursion for $d(v, k)$:

$$d(v,k) = \min \begin{cases} \min_{u \in V}(d(u, k-1) + \ell(u, v)). \\ d(v, k-1) \end{cases}$$

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

A Basic Lemma

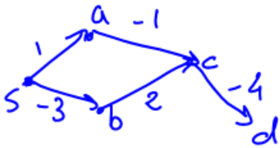
Lemma

Assume s can reach all nodes in $G = (V, E)$. Then,

1. *There is a negative length cycle in G iff $d(v, n) < d(v, n-1)$ for some node $v \in V$.*
2. *If there is no negative length cycle in G then $dist(s, v) = d(v, n-1)$ for all $v \in V$.*

Bellman-Ford Algorithm

for each $u \in V$ **do**
    $d(u, 0) \leftarrow \infty$
$d(s, 0) \leftarrow 0$



K	s	a	b	c	d
0	0	∞	∞	∞	∞
1	0	1	-3	∞	∞
2	0	1	-3	-1	∞
3	0	1	-3	-1	-5

Bellman-Ford Algorithm

for each $u \in V$ **do**
  $d(u, 0) \leftarrow \infty$
$d(s, 0) \leftarrow 0$

for $k = 1$ to n **do**
    for each $v \in V$ **do**
      $d(v, k) \leftarrow d(v, k-1)$
      for each edge $(u, v) \in In(v)$ **do**
        $d(v, k) = \min\{d(v, k), d(u, k-1) + \ell(u, v)\}$

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}


for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
        If d(v, n) < d(v, n − 1)
            Return ``Negative Cycle in G''
```

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u,0) ← ∞
d(s,0) ← 0

for k = 1 to n do
        for each v ∈ V do
            d(v,k) ← d(v,k−1)
            for each edge (u,v) ∈ In(v) do
                d(v,k) = min{d(v,k), d(u,k−1) + ℓ(u,v)}


for each v ∈ V do
        dist(s,v) ← d(v,n−1)
        If d(v,n) < d(v,n−1)
            Return ``Negative Cycle in G''
```

Running time:

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}


for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
        If d(v, n) < d(v, n − 1)
            Return ``Negative Cycle in G''
```

Running time: $O(mn)$

Bellman-Format Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}


for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
        If d(v, n) < d(v, n − 1)
            Return ``Negative Cycle in G''
```

Running time: $O(mn)$ Space:

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u,0) ← ∞
d(s,0) ← 0

for k = 1 to n do
        for each v ∈ V do
            d(v,k) ← d(v,k−1)
            for each edge (u,v) ∈ In(v) do
                d(v,k) = min{d(v,k), d(u,k−1) + ℓ(u,v)}


for each v ∈ V do
        dist(s,v) ← d(v,n−1)
        If d(v,n) < d(v,n−1)
            Return ``Negative Cycle in G''
```

Running time: $O(mn)$ Space: $O(n^2)$

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u,0) ← ∞
d(s,0) ← 0

for k = 1 to n do
        for each v ∈ V do
            d(v,k) ← d(v,k−1)
            for each edge (u,v) ∈ In(v) do
                d(v,k) = min{d(v,k), d(u,k−1) + ℓ(u,v)}


for each v ∈ V do
        dist(s,v) ← d(v,n−1)
        If d(v,n) < d(v,n−1)
            Return ‘‘Negative Cycle in G’’
```

Running time: $O(mn)$ Space: $O(n^2)$

Space can be reduced to $O(m+n)$.

Bellman-Ford with Space Saving

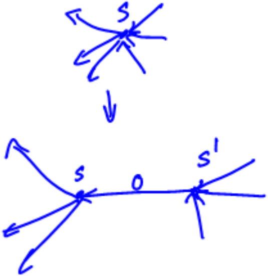
```
for each u ∈ V do
    d(u) ← ∞
d(s) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            for each edge (u, v) ∈ In(v) do
                d(v) = min{d(v), d(u) + ℓ(u, v)}
(* One more iteration to check if distances change *)
for each v ∈ V do
    for each edge (u, v) ∈ In(v) do
        if (d(v) > d(u) + ℓ(u, v))
            Output ``Negative Cycle''

for each v ∈ V do
        dist(s, v) ← d(v)
```

Problem

Given a directed graph $G = (V, E)$ with non-negative edge lengths $\ell : E \to R^+$, describe an algorithm that finds the shortest cycle in G that contains a specific node s.



Shortest path from s to s' is the sol^n.

Problem

Given a directed graph $G = (V, E)$ with non-negative edge lengths $\ell : E \to R^+$. Describe an algorithm to find the shortest cycle containing s with at most k edges.

$d(s', k)$

Run Bellman-Ford from s.

for k iterations

$d(v,k)$: length of the shortest walk from s to v that has at most k edges.

$\Rightarrow d(s', k)$: length of the shortest walk from s to s' with at most k edges.

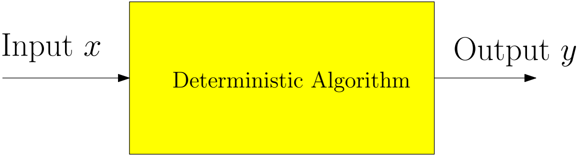
Since edge weights are positive walk is essentially a path.

$\therefore$ $d(s', K)$ is the sol^n.

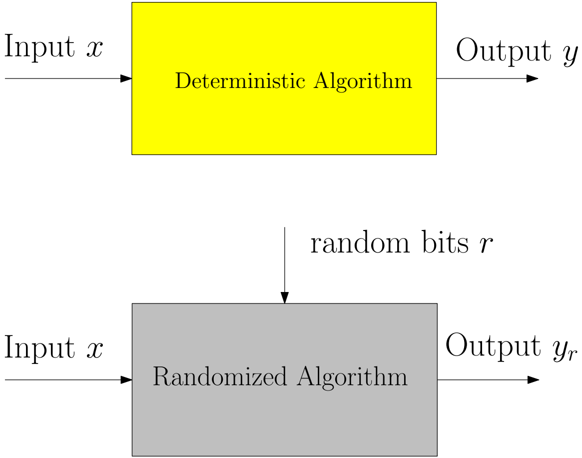
Part III

Randomization

Randomized Algorithms



Randomized Algorithms



Types of Randomized Algorithms

Typically one encounters the following types:

1. **Las Vegas randomized algorithms:** for a given input x output of *algorithm is always correct* but the *running time is a random variable*. Analyze **expected running time**.

Types of Randomized Algorithms

Typically one encounters the following types:

1. **Las Vegas randomized algorithms:** for a given input x output of *algorithm is always correct* but the *running time is a random variable*. Analyze **expected running time**.
2. **Monte Carlo randomized algorithms:** for a given input x the *running time is deterministic* but the *output is random*; correct with some probability. Analyze the **probability of the correct output** (and also the running time).
3. Algorithms whose running time and output may both be random.

Ping and find.

Consider a deterministic algorithm A that is trying to find an element in an array X of size n. At every step it is allowed to ask the value of one cell in the array, and the adversary is allowed after each such ping, to shuffle elements around in the array in any way it seems fit. For the best possible deterministic algorithm the number of rounds it has to play this game till it finds the required element is

(A) $O(1)$

(B) $O(n)$

(C) $O(n \log n)$

(D) $O(n^2)$

(E) ∞.

Ping and find randomized.

Consider an algorithm **randFind** that is trying to find an element in an array X of size n. At every step it asks the value of one random cell in the array, and the adversary is allowed after each such ping, to shuffle elements around in the array in any way it seems fit. This algorithm would stop in expectation after

- **(A)** $O(1)$
- **(B)** $O(\log n)$
- **(C)** $O(n)$
- **(D)** $O(n^2)$
- **(E)** ∞.

steps.

Median

Consider the problem of finding an "approximate median" of an unsorted array $A[1..n]$: an element of A with rank between $n/4$ and $3n/4$.

- Finding an approximate median is not any easier than a proper median.
- $n/2$ elements of A qualify as approximate medians and hence a random element is good with probability $1/2$!

Part IV

Basics of Randomization

Discrete Probability Space

Definition

A discrete probability space is a pair $(\mathbf{\Omega}, \mathbf{Pr})$ consists of finite set $\mathbf{\Omega}$ of **elementary events** and function $p : \mathbf{\Omega} \to [\mathbf{0}, \mathbf{1}]$ which assigns a probability $\mathbf{Pr}[\omega]$ for each $\omega \in \mathbf{\Omega}$ such that $\sum_{\omega \in \Omega} \mathbf{Pr}[\omega] = \mathbf{1}$.

Example

An unbiased coin. $\mathbf{\Omega} = \{H, T\}$ and $\mathbf{Pr}[H] = \mathbf{Pr}[T] = \mathbf{1}/\mathbf{2}$.

Events

Definition

Event is a collection of elementary events. The probability of an event $A \subset \Omega$, denoted by $\mathbf{Pr}[A]$, is $\sum_{\omega \in A} \mathbf{Pr}[\omega]$.

Events

Definition

Event is a collection of elementary events. The probability of an event $A \subset \Omega$, denoted by $\mathbf{Pr}[A]$, is $\sum_{\omega \in A} \mathbf{Pr}[\omega]$.

Union Bound

For any two events $\mathcal{E}$ and $\mathcal{F}$, we have that $\mathbf{Pr}\left[\mathcal{E} \cup \mathcal{F}\right] \leq \mathbf{Pr}\left[\mathcal{E}\right] + \mathbf{Pr}\left[\mathcal{F}\right]$.

Events

Definition

Event is a collection of elementary events. The probability of an event $A \subset \Omega$, denoted by $\mathbf{Pr}[A]$, is $\sum_{\omega \in A} \mathbf{Pr}[\omega]$.

Union Bound

For any two events $\mathcal{E}$ and $\mathcal{F}$, we have that $\mathbf{Pr}\left[\mathcal{E} \cup \mathcal{F}\right] \leq \mathbf{Pr}\left[\mathcal{E}\right] + \mathbf{Pr}\left[\mathcal{F}\right]$.

Independence

Events A and B are called independent if $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$.

Random Variables

Definition

Given a probability space $(\Omega, \Pr)$ a (real-valued) random variable X over Ω is a function $X : \Omega \rightarrow \mathbb{R}$.

Random Variables

Definition

Given a probability space $(\Omega, \Pr)$ a (real-valued) random variable X over Ω is a function $X : \Omega \to \mathbb{R}$.

Definition (Expectation: Average of X as per Pr)

Expectation of X, $\mathbf{E}[X]$, is defined as $\sum_{\omega \in \Omega} \Pr[\omega]\, X(\omega)$.
If S is the set of all values that X takes, then expectation can also be written as $\sum_{x \in S} x \Pr[X = x]$.

Random Variables

Definition

Given a probability space $(\Omega, \Pr)$ a (real-valued) random variable X over Ω is a function $X : \Omega \rightarrow \mathbb{R}$.

Definition (Expectation: Average of X as per Pr)

Expectation of X, $\mathbf{E}[X]$, is defined as $\sum_{\omega \in \Omega} \Pr[\omega]\, X(\omega)$.
If S is the set of all values that X takes, then expectation can also be written as $\sum_{x \in S} x \Pr[X = x]$.

Linearity of Expectation

Given two random variables X_1 and X_2,
$\mathbf{E}[X_1 + X_2] = \mathbf{E}[X_1] + \mathbf{E}[X_2]$.

Independence of Random Variables

Random variables X and Y are said to be independent if

$$\forall x, y, \quad \Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$$

Multiplication

If X and Y are independent then $\mathbf{E}[XY] = \mathbf{E}[X]\,\mathbf{E}[Y]$.

Part V

Randomized Quick Sort

Randomized **QuickSort**

Randomized **QuickSort**

1. Pick a pivot element *uniformly at random* from the array.
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Analysis via Recurrence

1. Given array A of size n, let $Q(A)$ be number of comparisons of randomized **QuickSort** on A.
2. Note that $Q(A)$ is a random variable.
3. Let A^i_{left} and A^i_{right} be the left and right arrays obtained if:

Let X_i be indicator random variable, which is set to 1 if the pivot is of rank i in A, else zero.

$$Q(A) = n + \sum_{i=1}^{n} X_i \cdot \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right).$$

Analysis via Recurrence

1. Given array A of size n, let $Q(A)$ be number of comparisons of randomized **QuickSort** on A.
2. Note that $Q(A)$ is a random variable.
3. Let A^i_{left} and A^i_{right} be the left and right arrays obtained if:

Let X_i be indicator random variable, which is set to 1 if the pivot is of rank i in A, else zero.

$$Q(A) = n + \sum_{i=1}^{n} X_i \cdot \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right).$$

Since each element of A has probability exactly of $1/n$ of being chosen:

$$\mathbf{E}[X_i] = \mathbf{Pr}[\text{pivot is the element with rank } i] = 1/n.$$

Independence of Random Variables

Lemma

Random variables $\boldsymbol{X_i}$ *is independent of random variables* $\boldsymbol{Q(A^i_{left})}$ *as well as* $\boldsymbol{Q(A^i_{right})}$, i.e.

$$\mathbf{E}\left[X_i \cdot Q(A^i_{left})\right] = \mathbf{E}[X_i]\,\mathbf{E}\left[Q(A^i_{left})\right]$$
$$\mathbf{E}\left[X_i \cdot Q(A^i_{right})\right] = \mathbf{E}[X_i]\,\mathbf{E}\left[Q(A^i_{right})\right]$$

Proof.

This is because the algorithm, while recursing on $\boldsymbol{Q(A^i_{\text{left}})}$ and $\boldsymbol{Q(A^i_{\text{right}})}$ uses new random coin tosses that are independent of the coin tosses used to decide the first pivot. Only the latter decides value of $\boldsymbol{X_i}$. □

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We have, for any A:

$$Q(A) = n + \sum_{i=1}^{n} X_i \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right)$$

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We have, for any A:

$$Q(A) = n + \sum_{i=1}^{n} X_i \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right)$$

By linearity of expectation, and independence random variables:

$$\mathbf{E}\Big[Q(A)\Big] \;\; = \;\; n + \textstyle\sum_{i=1}^{n} \mathbf{E}[X_i] \Big(\mathbf{E}\Big[Q(A^i_{\text{left}})\Big] + \mathbf{E}\Big[Q(A^i_{\text{right}})\Big] \Big)$$

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We have, for any A:

$$Q(A) = n + \sum_{i=1}^{n} X_i \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}}) \right)$$

By linearity of expectation, and independence random variables:

$$\begin{aligned}
\mathbf{E}\Big[Q(A)\Big] &= n + \sum_{i=1}^{n} \mathbf{E}[X_i] \Big(\mathbf{E}\Big[Q(A^i_{\text{left}})\Big] + \mathbf{E}\Big[Q(A^i_{\text{right}})\Big] \Big) \\
&\leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i) \right).
\end{aligned}$$

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.
We derived:

$$\mathbf{E}\Big[Q(A)\Big] \leq n + \sum_{i=1}^{n} \frac{1}{n}\left(T(i-1) + T(n-i)\right).$$

Note that above holds for any A of size n. Therefore

Here note the difference between $T(n)$ and $E[Q(A)]$. $T(n)$ is the "worst" expected running time among all n-sized arrays, while $E[Q(A)]$ is the expect running time for input array A.

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.
We derived:

$$\mathbf{E}\Big[Q(A)\Big] \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i) \right).$$

Note that above holds for any A of size n. Therefore

$$\max_{A:|A|=n} \mathbf{E}[Q(A)] = T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i) \right).$$

Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n}\left(T(i-1) + T(n-i)\right)$$

with base case $T(1) = 0$.

Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n}\left(T(i-1) + T(n-i)\right)$$

with base case $T(1) = 0$.

Lemma

$T(n) = O(n \log n)$.

Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n}(T(i-1) + T(n-i))$$

with base case $T(1) = 0$.

Lemma

$T(n) = O(n \log n)$.

Proof.

(Guess and) Verify by induction. □

Part VI

Inequalities

Markov's Inequality

Markov's inequality

Let X be a **non-negative** random variable over a probability space $(\Omega, \Pr)$. For any $a > 0$,

$$\Pr[X \geq a] \leq \frac{\mathbf{E}[X]}{a}$$

Chebyshev's Inequality

Variance

Variance of X is the measure of how much does it deviate from its mean value. Formally,
$Var(X) = \mathbf{E}\left[(X - \mathbf{E}[X])^2\right] = \mathbf{E}\left[X^2\right] - \mathbf{E}[X]^2$

Chebyshev's Inequality

Given $a \geq 0$, $\mathbf{Pr}[|X - \mathbf{E}[X]| \geq a] \leq \frac{Var(X)}{a^2}$

Chebyshev's Inequality

Variance

Variance of X is the measure of how much does it deviate from its mean value. Formally,
$Var(X) = \mathbf{E}\left[(X - \mathbf{E}[X])^2\right] = \mathbf{E}\left[X^2\right] - \mathbf{E}[X]^2$

Chebyshev's Inequality

Given $a \geq 0$, $\Pr[|X - \mathbf{E}[X]| \geq a] \leq \frac{Var(X)}{a^2}$

If X and Y are independent then
$Var(X + Y) = Var(X) + Var(Y)$.

Chebyshev's Inequality: Under Mutual Independence

Let $X_1, \ldots, X_k$ be k independent random variables such that, for each $i \in [1, k]$, X_i equals 1 with probability p_i, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbf{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$$Var(X) \leq \mu \Rightarrow \mathbf{Pr}[|X - \mu| \geq a] \leq \frac{Var(X)}{a^2} < \frac{\mu}{a^2}$$

Chebyshev's Inequality: Under Mutual Independence

Let $X_1, \ldots, X_k$ be k independent random variables such that, for each $i \in [1, k]$, X_i equals 1 with probability p_i, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbf{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$$Var(X) \leq \mu \Rightarrow \Pr[|X - \mu| \geq a] \leq \frac{Var(X)}{a^2} < \frac{\mu}{a^2}$$

For $\delta > 0, \Pr[X \geq (1 + \delta)\mu] \leq \frac{1}{\delta^2 \mu}$

For $0 < \delta < 1, \Pr[X \leq (1 - \delta)\mu] \leq \frac{1}{\delta^2 \mu}$

Chernoff Bound

Let $X_1, \ldots, X_k$ be k independent random variables such that, for each $i \in [1, k]$, X_i equals 1 with probability p_i, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbf{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

Chernoff Bound

Let $X_1, \ldots, X_k$ be k independent random variables such that, for each $i \in [1, k]$, X_i equals 1 with probability p_i, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbf{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$\Pr[X \geq (1+\delta)\mu] \leq e^{\frac{-\delta^2\mu}{3}}$ and $\Pr[X \leq (1-\delta)\mu] \leq e^{\frac{-\delta^2\mu}{2}}$

Chernoff Bound

Let $X_1, \ldots, X_k$ be k independent random variables such that, for each $i \in [1, k]$, X_i equals 1 with probability p_i, and 0 with probability $(1 - p_i)$. Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathbf{E}[X] = \sum_i p_i$. For any $0 < \delta < 1$, it holds that:

$$\Pr[X \geq (1+\delta)\mu] \leq e^{\frac{-\delta^2\mu}{3}} \text{ and } \Pr[X \leq (1-\delta)\mu] \leq e^{\frac{-\delta^2\mu}{2}}$$

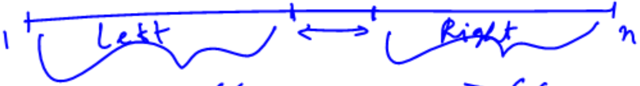
Tighter bound

For any $\delta > 0$, $\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$

$\Pr[X \geq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu}$

Problem: Approximate Median

Suppose you are presented with a very large set S of real numbers, and you would like to approximate the median of these numbers by sampling. Let $|S| = n$. We say x is an ϵ-approximate median of S if at least $(1/2 - \epsilon)n$ are less than x and at least $(1/2 - \epsilon)n$ are greater than x. Consider an algorithm that samples a number c times u.a.r. from S, forms set S' of sampled numbers, and outputs a median of S'. Show that for the algorithm to return ϵ-approximate median w.p. at least $(1 - \delta)$, it suffices to have sample size c that is an absolute constant, independent of n.



Ans is wrong if $> c/2$ OK $> c/2$

So, bound probability for each by $\frac{\delta}{2}$

$c = |S'|$

$|Left| = \left(\frac{1}{2} - \varepsilon\right)n$

$Pr(\text{sample } i \text{ is from left}) = \frac{\left(\frac{1}{2} - \varepsilon\right) \cdot \cancel{n}}{\cancel{n}}$

$X_i = 1$ if i^{th} sample from left
$\quad\; = 0$ o.w.

$E[X_i] = \frac{1}{2} - \varepsilon$.

$Z = \#$ ele. from Left $= \sum_{i=1}^{c} X_i$

$E[Z] = \left(\frac{1}{2} - \varepsilon\right) \cdot c$

Want $Pr\left[Z > c/2\right] \leq \frac{\delta}{2}$. Then what should be '$c$'?
(using chernoff)