(Exam 1 tomorrow)

Today: Sorting Day 3

~~Must~~ Prove $O(n \log n)$ ~~methods~~ is best

Then a $O(n)$ method

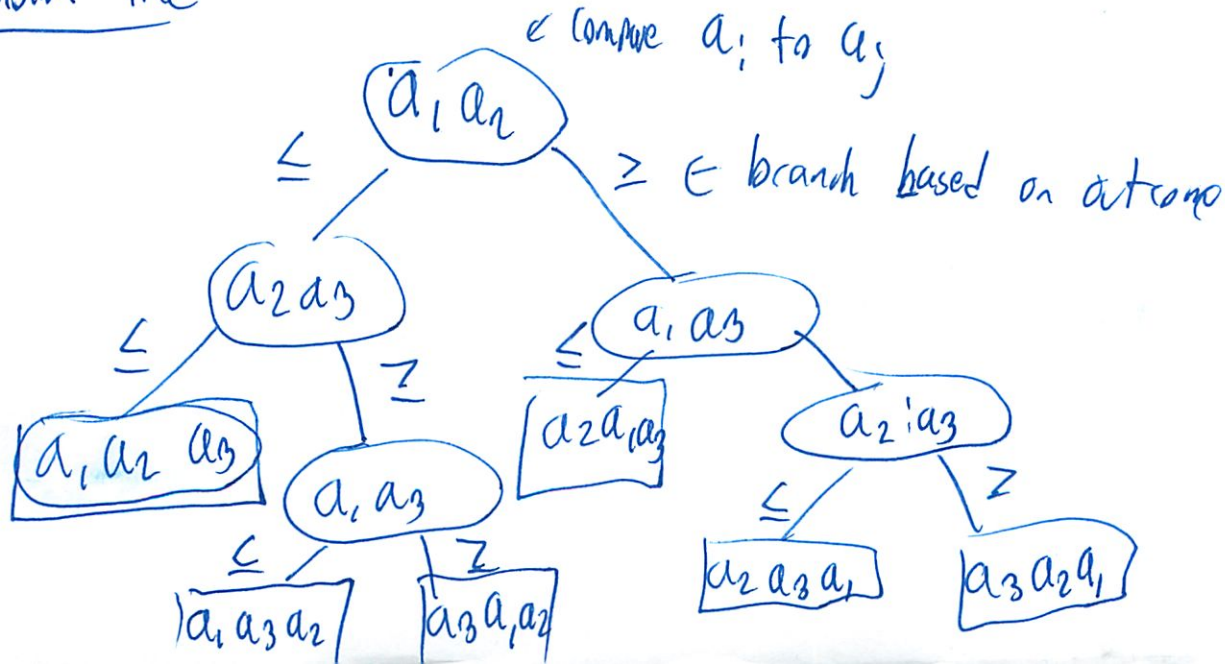
Comparison Sort

all the algorithms we saw so far

Use comparisons to test

$O(n \log n)$ is the best we can do

Decision Tree



(2)

Every node does a comparison
Go left and right based on a comparison

Each leaf has a permutation showing the order

Every comparison tree has a decision tree behind it
└ the running time = lenght of path taken
Worst possible = height of tree
and best height is $n \log n$ height
$\Omega(n \log n) \leftarrow$ height

Tree must contain $\geq n!$ leaves
since $n!$ permutations

Height h binary tree has $\leq 2^h$ leaves

So proof
$$2^h \geq n!$$
$$h \geq \log(n!)$$
$$\geq \log\left(\left(\frac{n}{e}\right)^n\right)$$
$$= n \log n - n \log e$$
$$= \Omega(n \log n)$$

(3)

Now we will beat that! $O(n)$

- but does not work in general case

does not work on sets of stuff → Only #s

Counting Sort

Could use auxiliry storage
- store stats

(went over in recitation I believe)

(counts freq that stuff shows up

$$[4\ 1\ 3\ 4\ 3] \rightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ [1 & 0 & 2 & 2] \end{array}$$

$$[1\ 2\ 3\ 3\ 4\ 4]$$ walk through freq array

But sort is not <u>stable</u>

Could build cumulative distribution

$$C[i] = \{keys \leq i\}$$

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ [1 & 1 & 3 & 5] \end{array}$$

sum of keys $\leq i$

(4)

So know what position to put each thing
- go from right end

Decrease count by 1

Repeat

(Example in slides)

Complicated way to do easy thing

<u>Running time</u> O(n + k)

<u>Stable sorting property</u>

- rel position in input preserved
- ie the first 3 is still just a 3
- does not make sense here
- but if 3s come w/ add'l order to preserve

Is <u>NOT</u> a comparison sort

Better in limited set of #s

Must be a finite set of integers

Complexity scales w/ the <u>range</u> input

(5)

Radix Sort

Goes back to 1890 Census

digit by digit

~~Sort~~ Sort on least significant w/ stability

└ kinda unintuitive └ must have

(Proof in slides)

Running time . n words w/ b bits each

Each word has $\frac{b}{r}$ base $\rightarrow 2^r$ digits

Example $b=32$ -bit word

If each b-bit word is broken into r-bit pieces

each pass takes $\Theta(n+2^r)$

So $r=\log n$ gives $\Theta(n)$ per pass or

$\Theta\left(\frac{nb}{\log n}\right)$ total

(6)

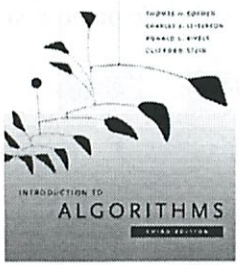
Midterm

Anything except radix + counting sort

Go to the right room!

I tried to remove

6.006- *Introduction to Algorithms*



Lecture 10

Prof. Constantinos Daskalakis

CLRS 8.1-8.4

Menu

- Show that $\Theta(n \lg n)$ is the best possible running time for a sorting algorithm.
- Design an algorithm that sorts in $\Theta(n)$ time.
- Hint: maybe the models are different ?

Comparison sort

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

So the elements could be numbers, water-samples compared on the basis of their concentration in chloride, etc.

The best running time that we've seen for comparison sorting is $O(n \log n)$.

Is* $\mathbf{O(n \log n)}$ *the best we can do?

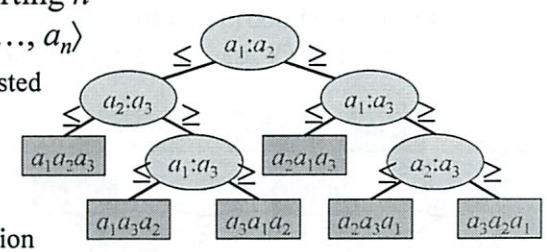
Decision trees can help us answer this question.

Decision-tree

A recipe for sorting n things $\langle a_1, a_2, \ldots, a_n \rangle$

- Nodes are suggested comparisons: $a_i{:}a_j$ means compare a_i to a_j.

- Branching direction depends on outcome of comparisons.

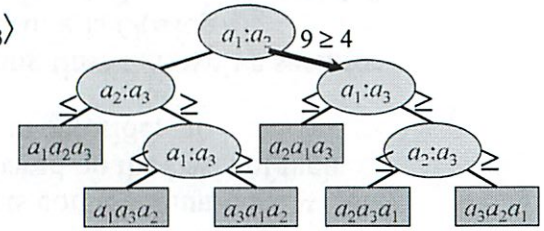


- Leaves are labeled with permutations corresponding to the outcome of the sorting.

3/13

Decision-tree example

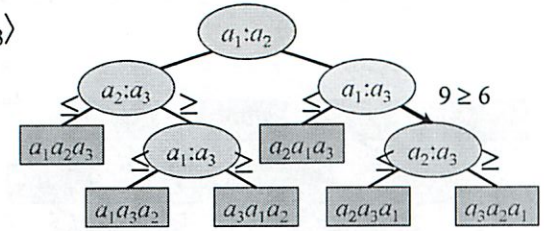
Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i{:}a_j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.
- Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ was found.

Decision-tree example

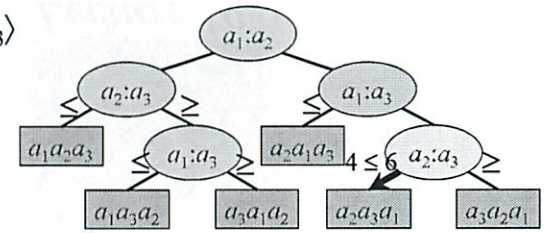
Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i{:}a_j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.
- Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ was found.

Decision-tree example

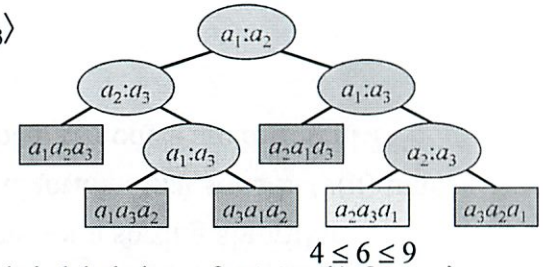
Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i{:}a_j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.
- Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ was found.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $a_i{:}a_j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.
- Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ was found.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- A path from the root to the leaves of the tree represents a trace of comparisons that the algorithm may perform.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision-tree sorting

Theorem. Any decision tree for n elements must have height $\Omega(n \log n)$.

Proof. (Hint: how many leaves are there?)

- The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations.
- A height-h binary tree has $\leq 2^h$ leaves.
- For it to be able to sort it must be that:

$$
\begin{aligned}
2^h &\geq n! \\
h &\geq \log(n!) && \text{(log is mono. increasing)} \\
&\geq \log((n/e)^n) && \text{(Stirling's formula)} \\
&= n \log n - n \log e \\
&= \Omega(n \log n).
\end{aligned}
$$

Sorting in linear time

Counting sort: No comparisons between elements.

- ***Input***: $A[1 \,..\, n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1 \,..\, n]$, a sorted permutation of A
- ***Auxiliary storage***: $C[1 \,..\, k]$.

Counting-sort example

n=5, k=4

one index for each possible key stored in A

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:				

Loop 1: initialization

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	0	0	0	0

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

Loop 2: count frequencies

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	0	0	0	1

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Loop 2: count frequencies

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	0	1

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Loop 2: count frequencies

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	1	1

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ $\triangleright$ $C[i] = |\{\text{key} = i\}|$

Loop 2: count frequencies

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	1	2

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2: count frequencies

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2: count frequencies

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2

for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

[A parenthesis: a quick finish

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2

Walk through frequency array and place the appropriate number of each key in output array…

A parenthesis: a quick finish

	1	2	3	4	5
A:	4	1	3	4	3
B:	1				

	1	2	3	4
C:	1	0	2	2

A parenthesis: a quick finish

	1	2	3	4	5
A:	4	1	3	4	3
B:	1				

	1	2	3	4
C:	1	0	2	2

A parenthesis: a quick finish

	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		

	1	2	3	4
C:	1	0	2	2

A parenthesis: a quick finish

	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3	4	4

	1	2	3	4
C:	1	0	2	2

B is sorted!

but it is not "stably sorted"…]

Loop 3: from frequencies to cumulative frequencies…

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Loop 3: from frequencies to cumulative frequencies…

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Loop 3: from frequencies to cumulative frequencies…

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Loop 3: from frequencies to cumulative frequencies…

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	1	3	5

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	1	3	5

There are exactly 3 elements ≤A[5]. So where should I place A[5]?

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	3	5

Used-up one 3; update counter in C for the next 3 that shows up...

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	2	5

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	2	5

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	2	5

There are exactly 5 elements ≤A[4]. So where should I place A[4]?

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

	1	2	3	4
C:	1	1	2	4

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

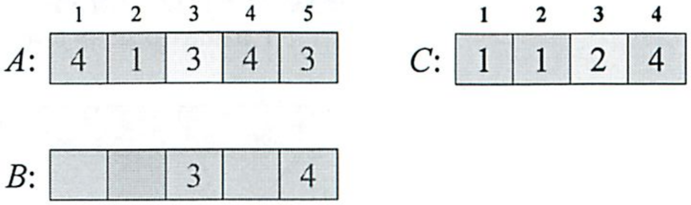
Loop 4: permute elements of A

	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

	1	2	3	4
C:	1	1	2	4

for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

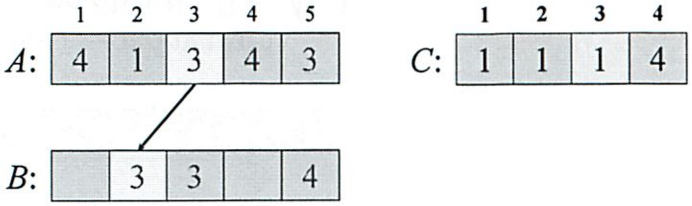


	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

	1	2	3	4
C:	1	1	2	4

```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A

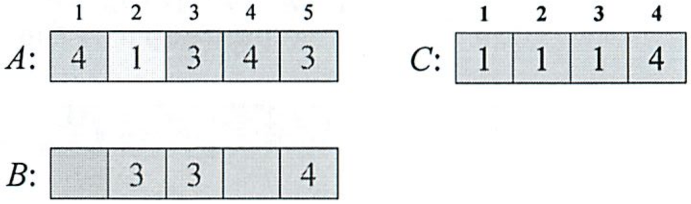


	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	1	4

```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A

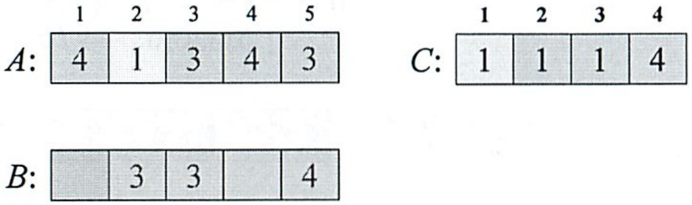


	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	1	4

```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A

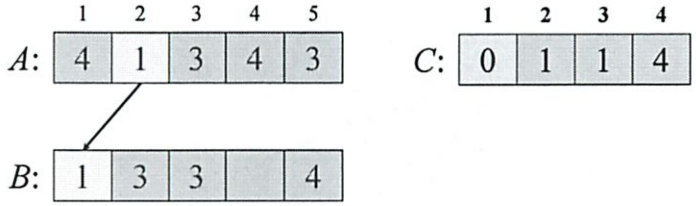


	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	1	4

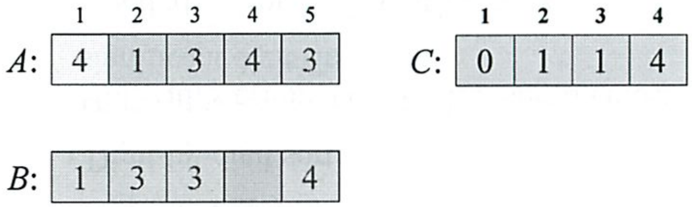
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A



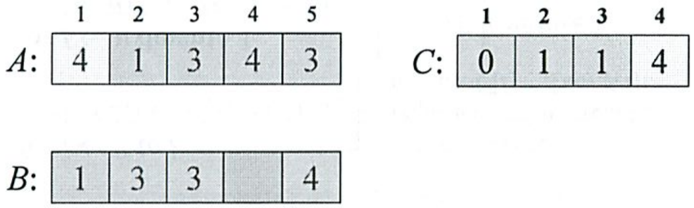
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A



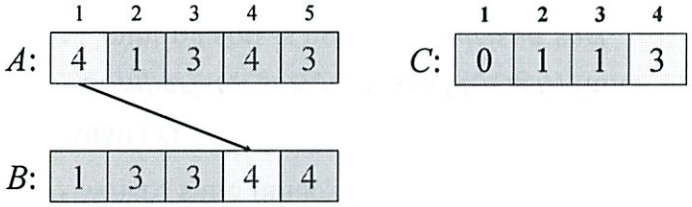
```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A



```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4: permute elements of A



```
for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort

```
for i ← 1 to k                                          Θ(k)
    do C[i] ← 0
for j ← 1 to n                  store in C the frequencies
    do C[A[j]] ← C[A[j]] + 1    of the different keys in A   Θ(n)
                                i.e. C[i] = |{key = i}|

for i ← 2 to k                  store in C the cumulative
    do C[i] ← C[i] + C[i–1]     frequencies of different keys Θ(k)
                                in A, i.e. C[i] = |{key ≤ i}|

for j ← n downto 1              using cumulative
    do B[C[A[j]]] ← A[j]        frequencies build            Θ(n)
       C[A[j]] ← C[A[j]] – 1    sorted permutation
                                                        Θ(n + k)
```

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

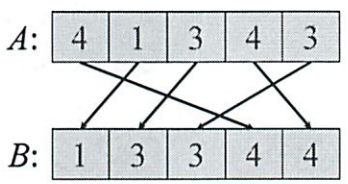
- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.



This does not seem useful for this example, but imagine a situation where each element stored in A comes with some "personalized information" (wait 2 slides…).

Radix sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix ◉.)
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

Operation of radix sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t

720	329
329	355
436	436
839	457
355	657
457	720
657	839

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.

720	329
329	355
436	436
839	457
355	657
457	720
657	839

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.

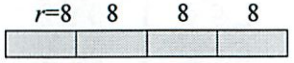
(just used stability property!)

720	329
329	355
436	436
839	457
355	657
457	720
657	839

Runtime Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: b=32-bit word



- If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time.
- So overall $\Theta(b/r\ (n + 2^r))$ time.
- Setting $r=\log n$ gives $\Theta(n)$ time per pass, or $\Theta(n\ b/\log n)$ total

Appendix: Punched-card technology

- Herman Hollerith (1860-1929)
- Punched cards
- Hollerith's tabulating system
- Operation of the sorter
- Origin of radix sort
- "Modern" IBM card
- Web resources on punched-card technology

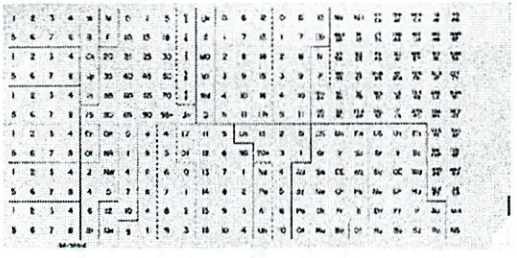
Return to last slide viewed.

Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

Punched cards

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.



Replica of punch card from the 1900 U.S. census. [Howells 2000]

Hollerith's tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

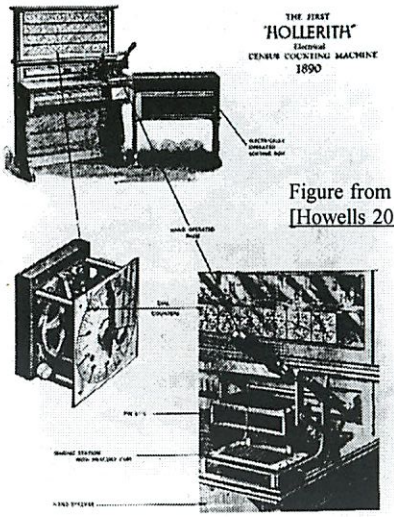
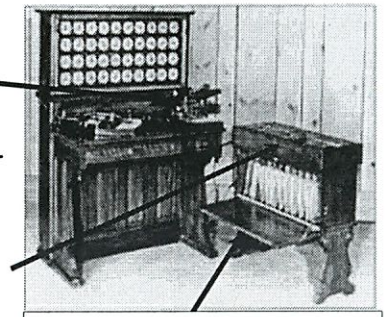


Figure from [Howells 2000].

Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.



Hollerith Tabulator, Pantograph, Press, and Sorter

Origin of radix sort

Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

> *"The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards."*

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

"Modern" IBM card

- One character per column.

Produced by the WWW Virtual Punch-Card Server.

So, that's why text windows have 80 columns!

Web resources on punched-card technology

- Doug Jones's punched card index
- Biography of Herman Hollerith
- The 1890 U.S. Census
- Early history of IBM
- Pictures of Hollerith's inventions
- Hollerith's patent application (borrowed from Gordon Bell's CyberMuseum)
- Impact of punched cards on U.S. history

Lecture 11

3/15

Today: Graphs, Representations, Search

Useful in Combinatorics (6.042)

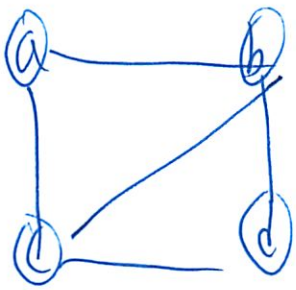
V = set of vertices

$E \subseteq V \times V$ (pairs of vertices)

└ # of $= m$

$m < n(n-1) = O(n^2)$

Some examples



Lots of applications

- Web DAG = Google
- Routing
- Games

(2)

<u>Example</u> 2x2x2 Rubix cube

Graph: One vertex for each state of the cube

One edge for each move

- 6 faces
- 3 ways to twist 1/4 3/4 turns 1/2
- = 18 edges for each state

Solve by finding a path from given to solved

So how big is this?

8 cublets in 8 positions = 8! arrangements

each cublet 3 orientations = 3^8 possibilities

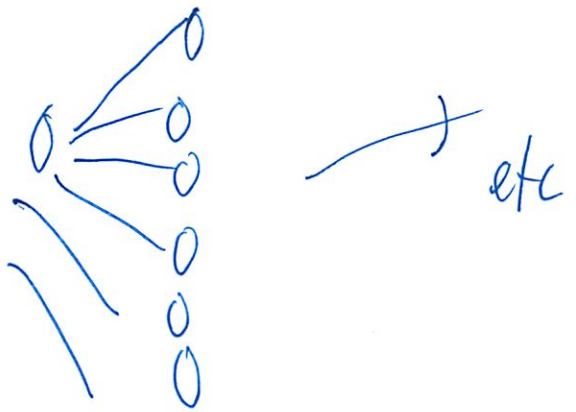
$8! \cdot 3^8 = 264{,}539{,}320$

But divide at 24 orientations of whole cube

= 3.[illegible] million states

(3)

Starting vertex



etc (see slides)

6 vertices 27 others
by another
(after removing duplicates)

Get table → God's Number

How do you represent?

4 possibilities

- Adj list
- Incidence list
- Adj matrix
- Implicit representation

(4)

Adjacency List

For every vertex v, list its neighbors

Array A of |V| linked list

For every $v \in V$, linked list $A[v]$ stores neighbors

Directed → outgoing only

Bi → both ways

Incidence List

For every vertex v, list its edges

Array A of |V| linked lists

For $v \in V$ store edges

(How is this different?)

OO Way

- adj list - object for each vertex u
- u.neighbors

(5)

incidence list

V, edges

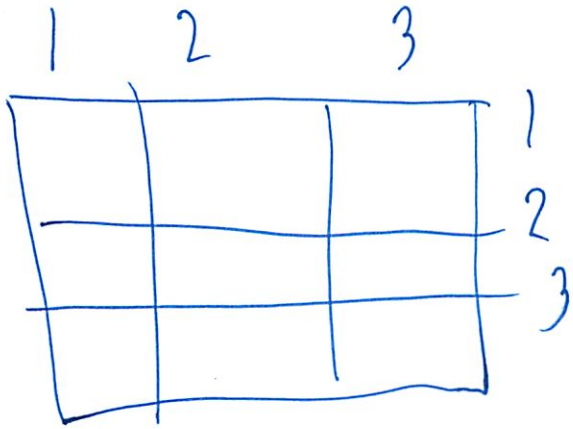
Adjacency Matrix

assume $V = \{1, \ldots, n\}$

$n \times n$ matrix $A = (a_{ij})$

$a_{ij} = 1$ if $(i,j) \in E$

$= 0$ otherwise



(6)

Graph Algebra

- adj matrix is a matrix → Linear Algebra
- eg A^2 = # lenght 2 paths b/w vertices

  A^∞ (after normalizing) is the PageRank
- Undirected graph → symmetric matrix
- Could use eigenvalues

Tradeoff: Space

Assume vertices $\{1, \ldots, n\}$

Adj list use one list node per edge

Space $\Theta(n + m \log n)$

Adj matrix uses n^2 entries

But each entry can be just 1 bit

So $\Theta(n^2)$

Matrix better only for very dense graphs

m near n^2

ie. Google can't use

(7)

Trade off : Time

~~These notes are wrong~~

	Matrix	Adj List
Add edge	O(1) flip bit	O(1) add at start
Check "is there edge from v to v"	O(1)	scan
Visit all neighbors of v (verse common)	O(n)	O(neighbors)
Remove edge		

- ~~Matrix is O(1)~~
- ~~Adj list of v must be scanned~~
- like find + add

Implicit Representation

- Don't store graph at all
- Implicit fn Adj(v) that returns list of neighbors / edges
- Requires no space - use as you need it

(8)

How do we explore/search graph?

Breath 1st

- Start w/ vertex v
- list all its neighbors (distance 1)
- then all their neighbors (distance 2)
- etc

- algorithm starting at s:
  - define frontier F
  - initally $F = \{s\}$
  - repeat till F = all neighbors of vertices in F
  - Until found

Greedy algo - find as much as possible in each reach

DFS is optimistic

↓

DFS

- like exploring a maze
- when get stuck - back track
- eg left hand rule

Cycles

What if we revisit a vertex?

BFS - get wrong ~~[illegible]~~ notion of distance

DFS - go in circles

So mark vertices

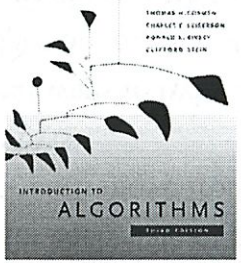
~~So today~~?

Next time formalize

Ariande / Minitor fable

- invented DFS
- string to backtrack

6.006- *Introduction to Algorithms*



Lecture 11

Prof. Costis Daskalakis

CLRS 22.1-22.3, B.4

Lecture Overview

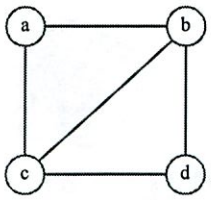
Graphs, Graph Representation, and Graph Search

Graphs

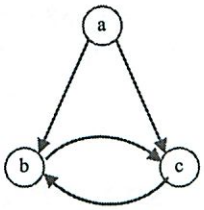
- Useful object in Combinatorics
- G=(V,E)
- V a set of vertices
  - usually number denoted by n
- $E \subseteq V \times V$ a set of edges (pairs of vertices)
  - usually number denoted by m
  - note $m \leq n(n-1) = O(n^2)$
- Two Flavors:
  - order of vertices on the edges matters: ***directed graphs***
  - ignore order: ***undirected graphs***
    - Then at most n(n-1)/2 possible edges

Examples

- Undirected
- V={a,b,c,d}
- E={{a,b}, {a,c}, {b,c}, {b,d}, {c,d}}



- Directed
- V = {a,b,c}
- E = {(a,c), (a,b) (b,c), (c,b)}

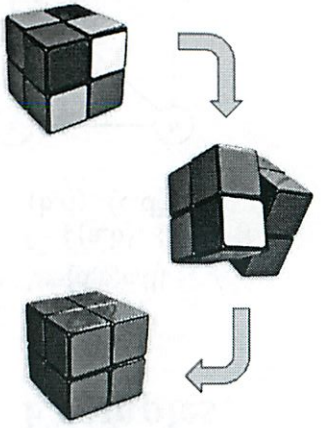


3/5

Instances/Applications

- Web
  - crawling
  - ranking
- Social Network
  - degrees of separation
- Computer Networks
  - routing
  - connectivity
- Game states
  - solving Rubik's cube, chess

Pocket Cube (aka Harvard Sophomores' cube)



- $2 \times 2 \times 2$ Rubik's cube
- Start with a given configuration
- Moves are quarter turns of any face
- "Solve" by making each side one color

Configuration Graph

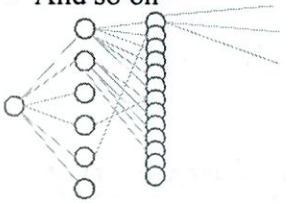
- Imagine a graph that has:
  - One vertex for each state of cube
  - One edge for each move from a vertex
    - 6 faces to twist
    - 3 nontrivial ways to twist (1/4, 2/4, 3/4)
    - So, 18 edges out of each state
- Solve cube by finding a path (of moves) from initial state (vertex) to "solved" state

Number of States

- One state per arrangement of cubelets and orientation of the cubelets:
  - 8 cubelets in 8 positions: so 8! arrangements
  - each cubelet has 3 orientations: 3^8 Possibilities
  - Total: 8! ´ 3^8= 264,539,320 vertices
- But divide out 24 orientations of whole cube
- And there are three separate connected components (twist one cube out of place 3 ways)
- Result: **3,674,160** states to search

GeoGRAPHy

- Starting vertex
- 6 vertices reachable by one 90° turn
- From those, 27 others by another
- And so on



distance	90°	90° and 180°
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,526
8	114,149	870,072
9	360,508	1,887,748
10	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

diameter

(aka God's number)

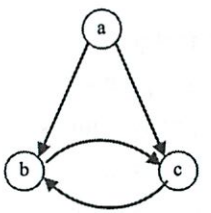
Representation

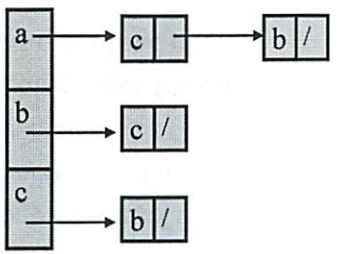
- To solve graph problems, must examine graph
- So need to represent in computer
- Four representations with pros/cons
  - Adjacency lists (of neighbors of each vertex)
  - Incidence lists (of edges from each vertex)
  - Adjacency matrix (of which pairs are adjacent)
  - Implicit representation (as neighbor function)

Adjacency List

- For each vertex v, list its neighbors (vertices to which it is connected by an edge)
  - Array A of $|V|$ linked lists
  - For $v \in V$, list A[v] stores neighbors $\{u \mid (v,u) \in E\}$
  - Directed graph only stores outgoing neighbors
  - Undirected graph stores edge in two places
- In python, A[v] can be hash table
  - v any hashable object

Example





Incidence List

- For each vertex v, list its edges
  - Array A of |V| linked lists
  - For $v \in V$, list A[v] stores edges $\{e \mid e=(v,u) \in E\}$
  - Directed graph only stores outgoing edges
  - Undirected graph stores edge in two places
- In python, A[v] can be hash table

(Object Oriented Variants)

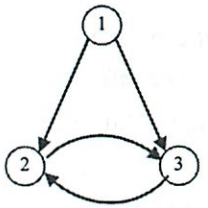
- adjacency list: object for each vertex u
  - u.neighbors is list of neighbors for u
- incidence list: object for each edge e
  - u.edges = list of outgoing edges from u
  - e object has endpoints e.head and e.tail

- can store additional info per vertex or edge without hashing

Adjacency Matrix

- assume $V=\{1, \ldots, n\}$
- $n \times n$ matrix $A=(a_{ij})$
  - $a_{ij} = 1$ if $(i,j) \in E$
  - $a_{ij} = 0$ otherwise
- (store as, e.g., array of arrays)

Example



1	2	3	
0	1	1	1
0	0	1	2
0	1	0	3

Graph Algebra

- can treat adjacency matrix as matrix
- e.g., A^2 = #length-2 paths between vertices ..
- A^∞ gives pagerank of vertices (after appropriately normalizing of A)
- undirected graph → symmetric matrix
- [eigenvalues carry information about the graph]

Tradeoff: Space

- Assume vertices {1,…,n}
- Adjacency lists use one list node per edge
  - So space is $\Theta(n+ m \log n)$
- Adjacency matrix uses n^2 entries
  - But each entry can be just one bit
  - So $\Theta(n^2)$ bits
- Matrix better only for very dense graphs
  - m near n^2
  - (Google can't use matrix)

Tradeoff: Time

- Add edge
  - both data structures are O(1)
- Check "is there an edge from u to v"?
  - matrix is O(1)
  - adjacency list of u must be scanned
- Visit all neighbors of u (very common)
  - adjacency list is O(neighbors)
  - matrix is $\Theta(n)$
- Remove edge
  - like find + add

Implicit representation

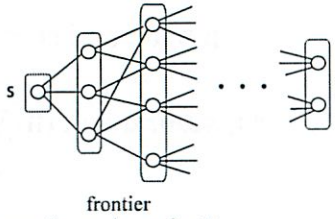
- Don't store graph at all
- Implement function Adj(u) that returns list of neighbors or edges of u
- Requires no space, use it as you need it
- And may be very efficient
- e.g., Rubik's cube

Searching Graph

- We want to get from current Rubik state to "solved" state
- How do we explore?

Breadth First Search

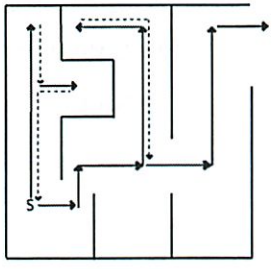
- start with vertex v
- list all its neighbors (distance 1)
- then all their neighbors (distance 2)
- etc.



- algorithm starting at s:
  - define frontier F
  - initially F={s}
  - repeat F=all neighbors of vertices in F
  - until all vertices found

Depth First Search

- Like exploring a maze
- From current vertex, move to another
- Until you get stuck
- Then backtrack till you find a new place to explore
- e.g "left-hand" rule



Problem: Cycles

- What happens if unknowingly revisit a vertex?
- BFS: get wrong notion of distance
- DFS: go in circles
- Solution: mark vertices
  - BFS: if you've seen it before, ignore
  - DFS: if you've seen it before, back up

Conclusion

- Graphs: fundamental data structure
  - Directed and undirected
- 4 possible representations
- Basic methods of graph search

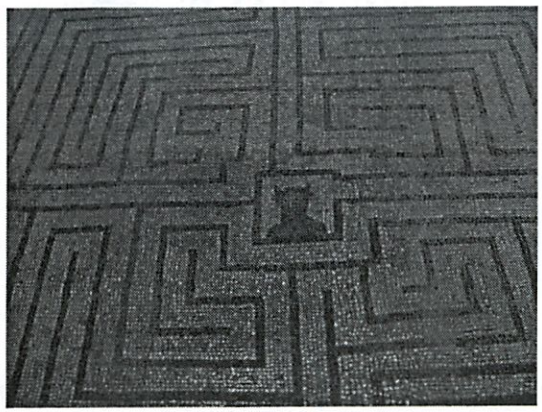
- Next time:
  - Formalize BFS and DFS
  - Runtime analysis
  - Applications

Inventor of DFS?

Daughter of Minos king of Crete
And sister of…

The Minotaur

The Minotaur resided in a maze next to Minos's palace. The best of the youth from around Greece was brought to the maze, and unable to navigate inside it got lost and tired, and eventually eaten by the Minotaur…



Inventor of DFS fell in love with Theseus and explained the algorithm to him before he was thrown to the maze..

and...

Theseus follows algorithm, finds the Minotaur...

Theseus and Ariadne then sail happily to Athens..

The rest of the story is not uneventful though..

6.006 Recitation

3/16

Prelim avg 50 $\sigma = 15$
(not done grading

He wrote 5+6 - thought they turned out harder
than he though

~~they~~ solutions were lots more complex than had to be

5+6a no BSTs
6a - made to think about program

Have (a_i, b_i)
Throw out all players i
for which there exits j s. t.
$a_i < a_j$
$b_i < b_j$

Looks like Pset
but universe operating in was diff
a_i, b_i were comparable to each other $a_i < b_i$

(2)

best can $O(n \log n)$

Twitter

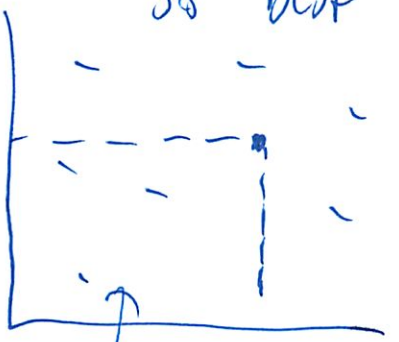
Takes at least n^2 for list
- just asked for count

Here list is at most n

These are not intervals

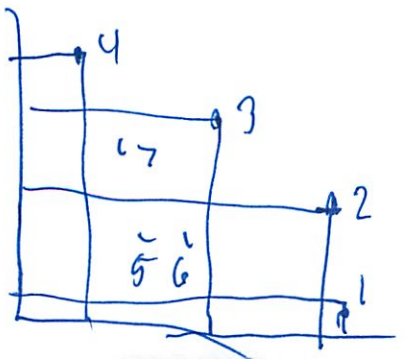
These guys are 2 points that don't have any thing to do w/ each other

So best way to represent



points ~~[illegible]~~ majorized by rectange this pt are in

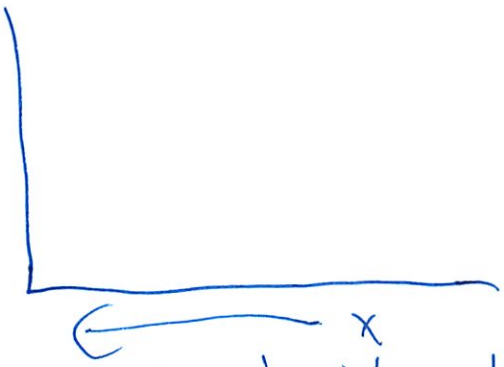
So looks like



← pardo ~~[illegible]~~ efficent boundry

③

Sort on 1 dimension of the problem



Sort in this order

(so simple in retrospect)

Pick a point at start of x list

go left (through list)

~~when see some~~ decreasing x ~~point (some x) as where are now~~

when point > y max

make that new y max

ans = ans ++

O(n log n) ← for sorting

He realizes you might not have thought of this

(I never saw such a method)

(4)

Other possible (more complex) solutions (still correct)

- Sort on x's and iterate into ~~the~~ decreasing x order

- Insert item (x_i, y_i) into BST based on y
- Immedity delete it if (x_i, y_i) <u>does not</u> have the largest y.

Will tree just be a line

- Not really a tree - more like a linked list

Try not have complexity in sol

- will cause you to make mistakes

Bring smallest a guy to middle of b (:)

- Same frontier thing

1/3 got it correct

He said the plot looks like natural representation of data

- will maybe do more algos like this

(5)

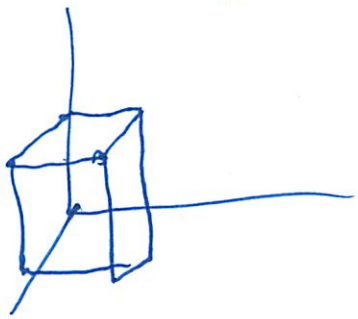
Nieve is n^2 ← give 1 pt
can put down on test

But search w/ binary tree is $n^2 \log n$

- Complicated
- <u>Worse</u> than nieve sol

---

b) Adds 3rd dimension (z axis)



Majorized is now a rectange
Can do problem w/o augmentation

Sort w/ z order
Go in decreasing order of z
Take a point. Check againts x, y
will know its < z

(6)

Only care about points w/ a high x, y

The x, y frontier are points that are high

If BST keyed by x, would be keyed by y backwards

When get new pt search for
Sucessor in x tree
predecessor in y tree

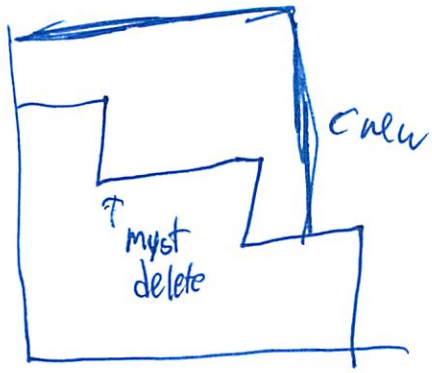
So outside -

Also delete

- if y is greater - must delete others (not that majorized - but not part of BST

So more than $O(\log n)$ since delete

(7)



So not every step is log n for n times

But the total is $O(n \log n)$

Will do more algo design

kinda the same as last problem

$\uparrow z$ or $\downarrow z$

At each pt - how do we know if to include it

Counter example

(1, 1, 1)

(2, 2, 0)

(2, 0, 2)

(0, 2, 2)

(8)

How do know no better $O(n \log n)$

↓

Involves order

↓

So likely will involve sorting

They thought telling you runtime was not needed

Q) Part B did not say efficently

↳ efficency + ~~sim~~ simplicity are a standing order

---

5) They didn't expect it would be that hard either

Find the # of indices (i, j) in the list such that

$$i < j$$

$$|A[i] - A[j]| < d$$

(9)

Niave n^2

So do better. Sort.

Since order matters

$A[i] - A[j] < d$

$A[i] - A[j] > -d$

Sort A

Iterate in increasing order

$a_0 \quad a_1 \quad a_2 \quad a_i \ldots \quad a_n$

Go to a_i

Then go to $a_i + d$

Find people in the middle

Can add the # in b/w the 2

Lots of ways to make this more complex - BST

(10)

Sort $n \log n$

Search $\log n$ | $O(n \log n)$

Could you do 2nd phase in $O(n)$

After sorted list → linear time for 2 close pairs

The +d part is constant

- going $a_1 \to a_2$
- moves 2nd pointer same amount ?

a_0 a_1 a_j a_n

↑

a_i right now

a_{i+d} right now

close pairs

move 1 →

verify if works → if not keep moving forwards

L12 Graphs 2

(no PPT)

3/20

$G = (V, E)$

V = set of vertices

$E \subset V \times V$

directed (i, j)

undirected $\{i, j\}$

Model

friendship

powergrids

Maps

....

Since functions are basic

$F: X \to Y$

Relations even more basic

$R \subset X \times Y$

②

Last time 4 representations

- Adj list
  - graphs can have few of edges
  - like friendship graph w/ n = words population

Searching graphs

Find all vertices connected to a given vertex s

Class: Undirected
Recitation: Directed

s connected to t if a Path P from s to t

$P = s = s_0, s_1, \ldots, s_k = t$

Lenghth of $P = k$

Distance b/w $s \to t$ = lenght of shortest path

Plan

First Use Pseudo² Code — correctedness
2nd Pseudo Code — complexity

③

BFS (wrong)

~~Find~~ Find all the vertices connected to me

But might have ones w/ no connection

(Didn't get the fix)

$$s \to \begin{matrix} a \\ x \end{matrix} \to \begin{matrix} z \\ d \\ c \end{matrix} \to \begin{matrix} f \\ v \end{matrix}$$

Not really exploring - since every direction

It marks all vertices connected to s

- must be some path $s \to t$

$$(s) - o - o - o - o - (t)$$

If last one - will be no more

but still more if neighbors

Complexity?

need an array of items you marked

(4)

log (n) ?

But can have

(s) - o - o - o - o - o - o — (t)

So must compare each one each time

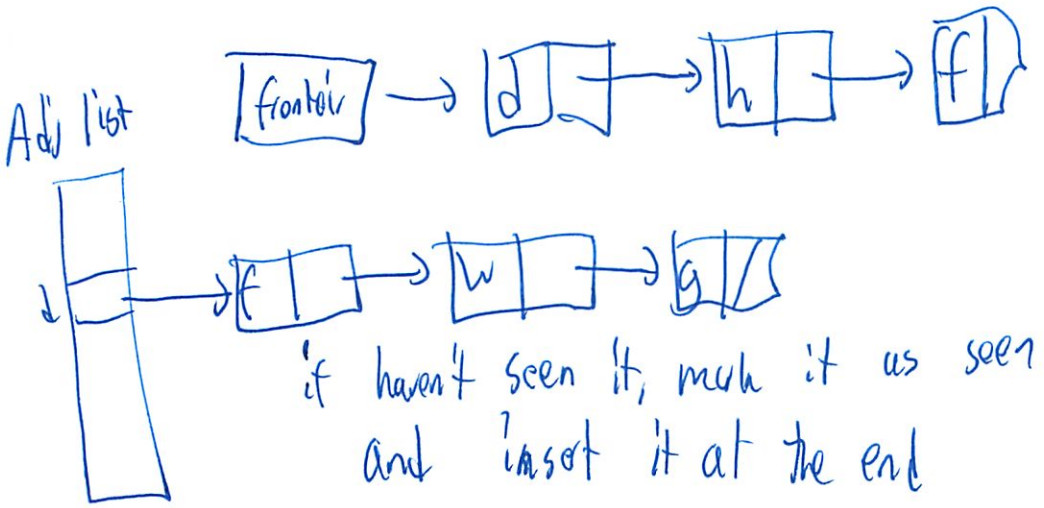
$O(n^2)$

Can you get worse?

- final / PS qu

Better Pseudo Code

track frontier of items you marked



if haven't seen it, mark it as seen
and insert it at the end

$\Theta(n+m)$ for all graphs

vertex
mark once
process once

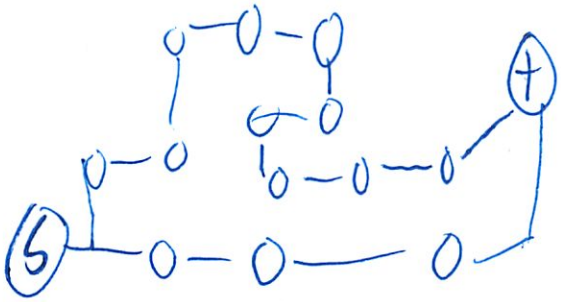
edge
process once

(5) Can you do better?

Augmented BFS (shortest path) Pseudo?

Initially mark $s \to 0$, others ∞

Then mark vertices $\to i+1$ (its distance from s)

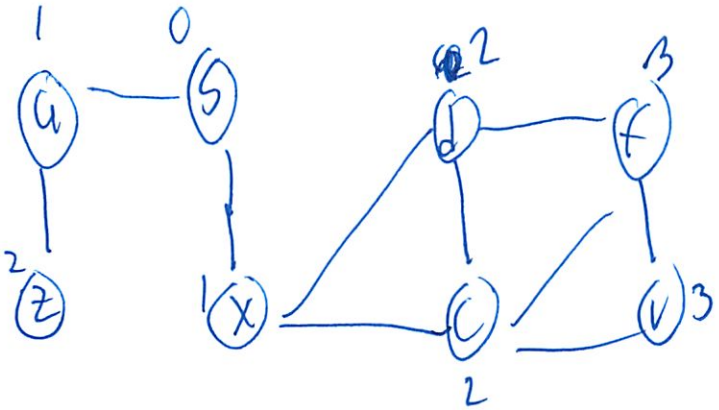


2 possible paths

↑ labled 2

can't be 1 since no path of lenght 1

Complexity: Same! - nothing has changed



(4)

Now pseudo code

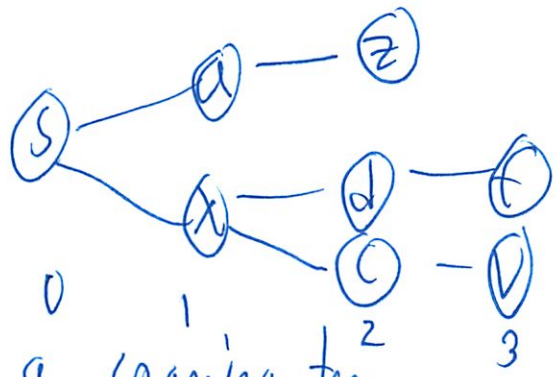
~~Will mark~~

uses the linked list

will mark them in order, 1 at a time

└specific order you mark stuff in

└can build a BFS tree

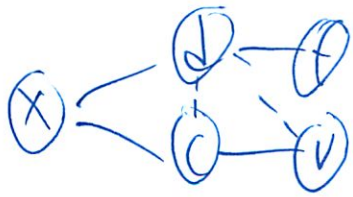


- is a spanning tree

$V' = V$

(V', E') is a ~~[illegible]~~ subgraph of (V, E)

Can't have $d \to v$



since c responsible to mark v

possible

Nor $d \to c$ would be a longer path

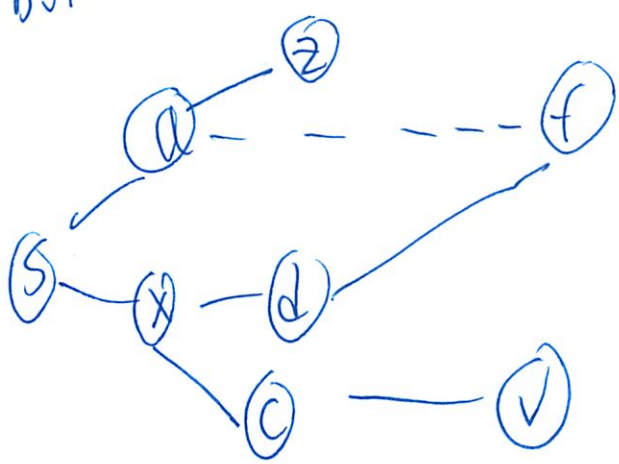
possible too

(2)

There will always be a longer path

Can go ~~the~~ Boston → Phila via Nx

But



No!

would be a shorter path

Mazes

graphs model mazes

but searching graphs ≠ ~~gra~~ searching mazes

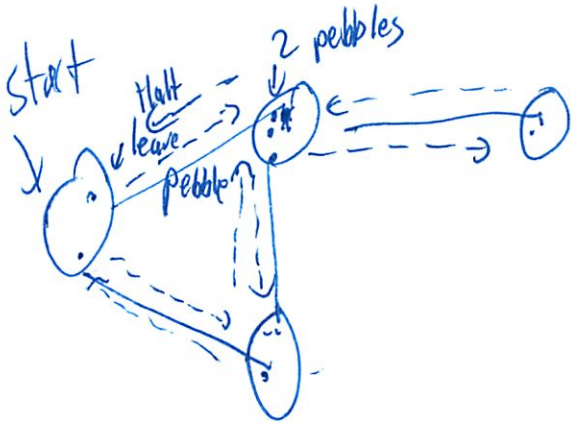
~~Tal~~ 1800s Pseudo 3 DFS

How to visit the ~~[illegible]~~ Louvre in an hr?

No strings

(8)

See all possible rooms + come back



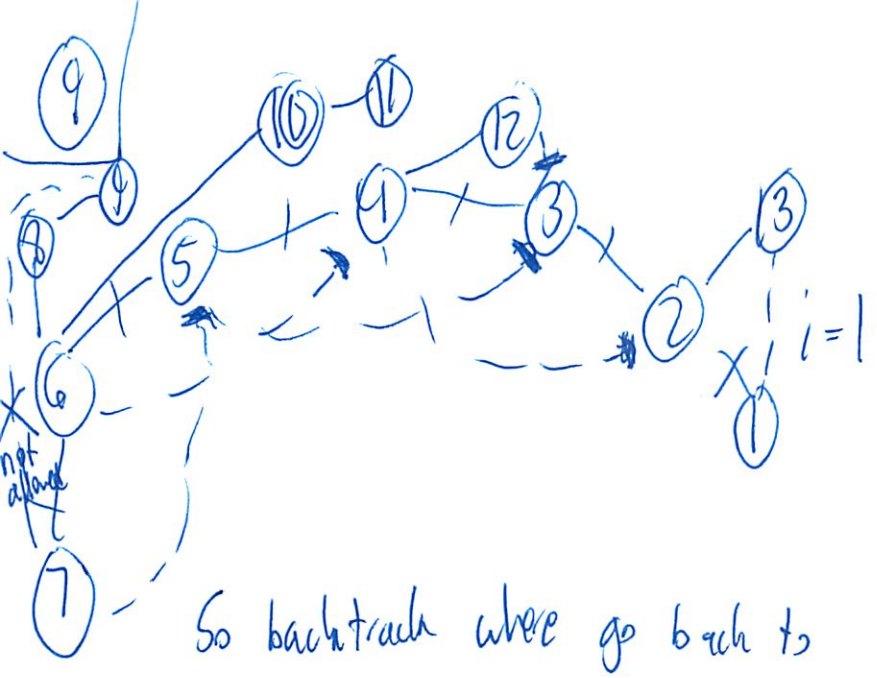
Claim 1 No edge is traversed twice in same dir

Claim 2 (missed)

Linear time

Hopcroft's + Tarjan's DFS

- Mark edges - not entrances + exits
- Number vertices (avg. for future use)
- Number your father node

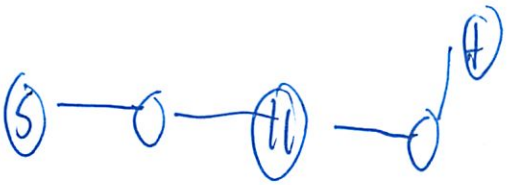


WTF?

So backtrack where go back to

Claim: DFS visits all vertices connected to s

is a path from s to t



Only backtrack if new edge we have not explored

DFS Tree

- Tree edges
- Back edges

(10)

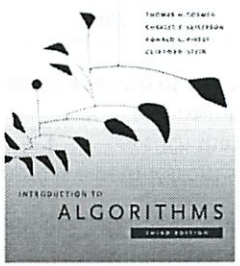
Odds + Ends

- Queses vs Stacks
- Strings ?!
  - Minatour maze

Assume part of maze where Minatar lives

but string can get sucked it
must nail string to the floor

6.006- *Introduction to Algorithms*



Lecture 12

Prof. Silvio Micali

CLRS 22.2-22.3

Graphs

$G = (V, E)$

- V a set of vertices
  $|V|$ denoted by n. Often: $V = \{1, \dots, n\}$
- $E \subset V \times V$ a set of *edges* (pairs of vertices)
  $|E|$ denoted by m $\leq n(n-1) = O(n^2)$

Graph Flavors

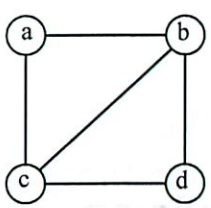
- *Directed*: "edges have a direction" I.e., $(i,j) \equiv i \to j$
- *Undirected:* "$\{i,j\}$ not (i,j)": $(i,j) \equiv (j,i) \equiv i - j$
  $\leq n(n-1)/2$ possible edges

PPT has not shown in class

Examples

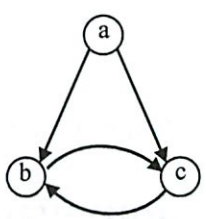
Undirected

- V={a,b,c,d}
- E={{a,b}, {a,c}, {b,c}, {b,d}, {c,d}}



Directed

- V = {a,b,c}
- E = {(a,c), (a,b) (b,c), (c,b)}



Graphs model lots of stuff:

- Friendship
- Powergrids
- Maps
- ...

Why?

Functions are basic
$F: X \to Y$
Relations are more basic!
$R \subset X \times Y$

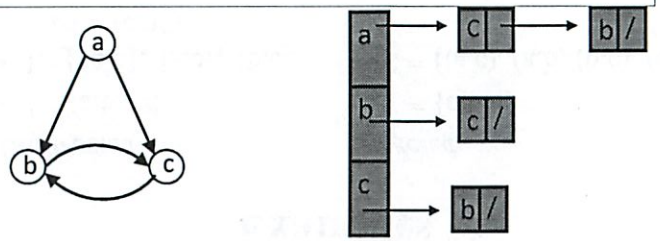
Graph Power!

3/20

Computer Representation

Four representations with pros/cons

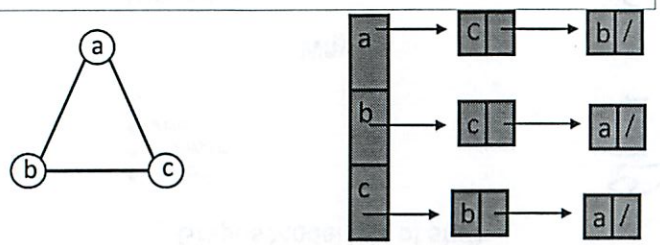
- *Adjacency lists* (of neighbors of each vertex)



Computer Representation

Four representations with pros/cons

- *Adjacency lists* (of neighbors of each vertex)



Searching Graphs

Finding all vertices connected to a given vertex s WLOG $s = 1$

Class: Undirected Graphs
Recitation: Directed Ones

- s connected to t if there is a path P from s to t
- $P \equiv s = s_0, s_1, \ldots, s_k = t$ such that $\{s_i, s_{i+1}\} \in E$
- Length of $P = k$ (we count edges!)
- Distance between s and t = length of shortest path from s to t

Plan		
Pseudo²Code	first	for "mathematical" correctness
Pseudo Code	next	(implementation ideas) for complexity
Real Code	home	for getting an output!

Breadth First Search (Wrong)

All vertices initially unmarked, but s

1. Until all vertices are marked, mark all neighbors of currently marked vertices

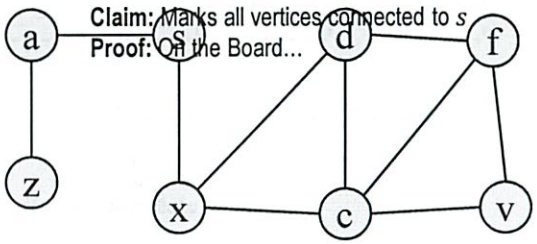
Breadth First Search ($Pseudo^2$)

All vertices initially unmarked, but s

1. Until no new vertices are marked, mark all neighbors of currently marked vertices

Example

Claim: Marks all vertices connected to s
Proof: On the Board...



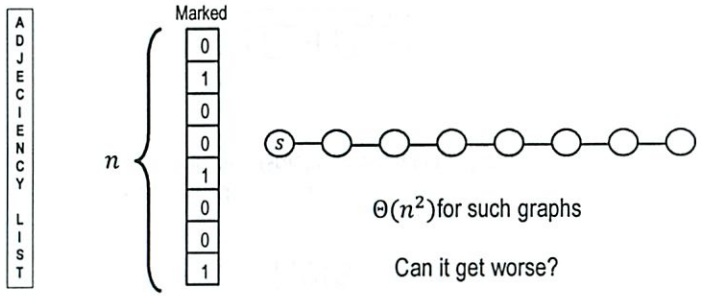
Breadth First Search ($Pseudo^2$)

All vertices initially unmarked, but s

1. Until no new vertices are marked, mark all neighbors of currently marked vertices

Complexity?

At least: $Pseudo\ Code$ or Implementation Details!



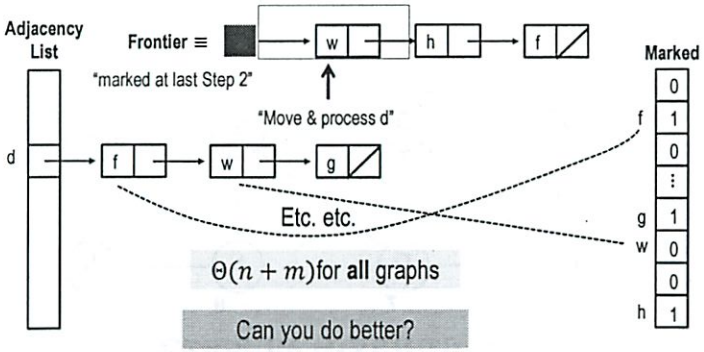
$\Theta(n^2)$for such graphs

Can it get worse?

Breadth First Search (Better $Pseudo$ Code)

All vertices initially unmarked, but s

1. Until no new vertices are marked, mark all neighbors of currently marked vertices



$\Theta(n+m)$for **all** graphs

Can you do better?

Augmented Breadth First Search =Shortest Path Alg ($Pseudo^2$)

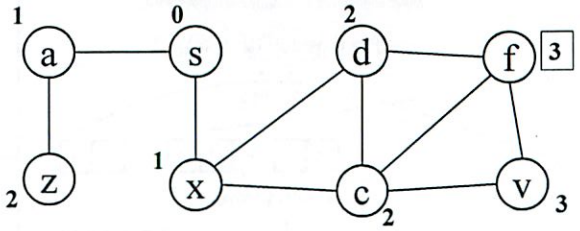
Initially, s is marked 0, all other vertices are marked ∞

1. $i \leftarrow 0$
2. Find all neighbors of at least one vertex marked i. If none, STOP.
3. Mark all vertices found in (3) with $i+1$.
4. $i \leftarrow i+1$

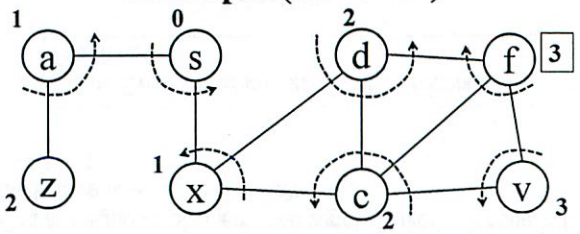
Claim: Every vertex is marked with its distance form s
Proof: ...

Complexity: ...

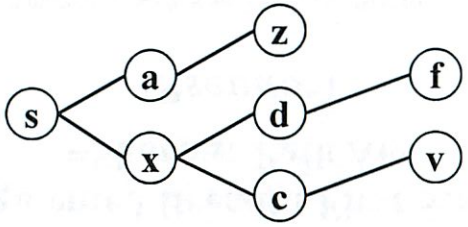
Example ($Pseudo^2$)



Example ($Pseudo$)



If you keep track of the edge through which you got your mark: **BFS Tree!**

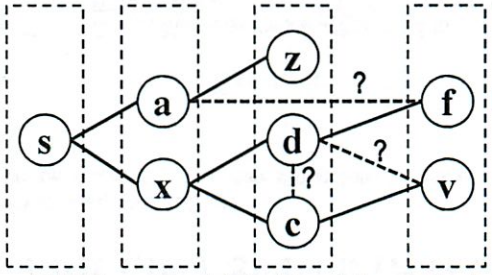


BFS Tree Structure

- Spanning Tree: subgraph (V', E') that
  1. is a tree
  2. $V' = V$

  (V', E') subgraph of (V, E) iff $V' \subset V$ and $E' \subset E$

- Extra Structure:



Noooooooo!!!!

Possible!

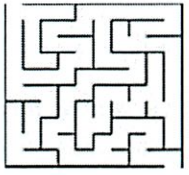
Possible Too

BFS Tree: Few Data, Very Informative!

Note!

Graphs model mazes.

But: Searching Graphs $\neq$ Searching Mazes



1800s Depth First Search ($Pseudo^3$)

In spocken Englsh (sort of...)

How to visit the Louvre in an hour and come out ALIVE!

No strings allowed!

(No questions either!)

Claim 1: No edge is traversed twice in the same direction

Claim 2: Upon Termination each edge has been traversed once in each direction

Hopcroft's & Tarjan's DFS

- Mark edges rather than their ``entrances" and "exits"
- Number vertices (augmentation for future use)
- Remember your father node rather than the edge who discovered you

0. Mark all edges "unused". For all $v \in V, \#(v) := 0$. Let $i := 0$ and $CoA := s$.
1. $i \leftarrow i + 1$ $\#(CoA) \leftarrow i$
2. If CoA has no unused edges, go to (4)
3. Choose an unused edge $CoA \overset{e}{\leftrightarrow} u$. Mark e used. If $\#(u) \neq 0$ go to (2). Else $F(u) \leftarrow CoA$ $CoA \leftarrow u$ and go to (1)
4. If $\#(CoA) = 1$ HALT
5. $CoA \leftarrow F(CoA)$ and go to (2)

Thm: DFS Visits all vertices connected to s

Proof: ...

DFS Tree

- Tree edges
- Back edges

Odds & Ends

- Queues vs. Stacks
- Strings??!

Good News

More Board Explanations!

Good Implications

More Reasons to come to class!

Enjoy it!

Lecture is over!

Please walk **calmly** to the nearest EXIT

Original

Problem Set 3

This problem set is due **Wednesday, March 21** at **11:59PM.**

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **Hash Collisions (20 points)**

   Consider hashing integers which are selected independently at random from the universe $U = [1, 2, \ldots, 84]$. Recall that a hash family h_i from U to $\{0, 1, \ldots, m-1\}$ is universal if, for any distinct x and y:

   $$\Pr_i\big[h_i(x) = h_i(y)\big] \leq \frac{1}{m}.$$

   a. Suppose we would like $m = 4$. Consider the hash family $h_i(x) = x^2 + x + i \pmod 4$, for $i \in \{0, 1, 2, 3\}$.

      1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_2(x) = x^2 + x + 2 \pmod 4$?
      2. The family $h_i(x)$ a universal hash family. `True` or `False`?

   b. Suppose we would like $m = 3$. Consider the hash family $h_i(x) = x^2 + i \pmod 3$, for $i \in \{0, 1, 2\}$.

      1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_1(x) = x^2 + 1 \pmod 3$?
      2. The family $h_i(x)$ a universal hash family. `True` or `False`?

   c. Suppose we would like $m = 12$. Consider the hash family $h_i(x) = ix + 2 \pmod{12}$, for $i \in \{0, \ldots, 11\}$.

1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_7(x) = 7x + 2 \pmod{12}$?
2. The family $h_i(x)$ a universal hash family. `True` or `False`?

d. Suppose we would like $m = 7$. Consider the hash family $h_i(x) = ix + 2 \pmod 7$, for $i \in \{0, \ldots, 6\}$.

1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_5(x) = 5x + 2 \pmod 7$?
2. The family $h_i(x)$ a universal hash family. `True` or `False`?

Solution Format:

Your answer for the first part of each question should consist of a float probability in the range $[0, 1]$, accurate to within 0.001 of the correct answer. Your answer to the second part of each question should be a boolean.

2. **Open Addressing (30 points)**

Suppose you are hashing items into the hash table of size 10 below, using the hash function $h(k) = k \bmod 10$ to find the location of key k and using linear probing to resolve collisions.

After inserting 6 values into the empty hash table, the table is in the state below:

0	
1	
2	22
3	13
4	54
5	32
6	46
7	43
8	
9	

(a) Which one of the following insertion orders would result in this state?

1) 46, 22, 54, 32, 13, 43
2) 54, 22, 13, 32, 43, 46
3) 46, 54, 22, 13, 32, 43
4) 22, 46, 43, 13, 54, 32

(b) Suppose that 46 was deleted from the table. How many cells would be inspected if you then searched the table for 65?

(c) Is there some sequence of insertions and deletions, starting from an empty table, after which each cell i contains the value $i+1$? Give an example of such a sequence, or prove that no such sequence exists.

(d) Is there some sequence of insertions and deletions, starting from an empty table, after which each cell i contains the value $9 - i$? Give an example of such a sequence, or prove that no such sequence exists.

Solution Format:

For part a), your answer should be an integer choice, and for part b) it should be a integer answer. For parts c) and d), if you believe that no such sequence exists, your answer should be a string containing a proof of this fact. If you have a counterexample, you should enter it as a list of tuples; the first element of each tuples should either be an 'i' for insertion or a 'd' for deletion, and the second should be the key being inserted or deleted.

3. **Price changes (20 points)**

The local supermarket sells n products whose prices are stored in a sorted array $[p_1, p_2, \ldots p_n]$, where $p_i \leq p_{i+1}$ for all $i \leq n-1$.

After some seasonal price cuts, k of these prices are updated. Suppose you are given the original price array and an array $[d_1, d_2, \ldots d_n]$ of the price changes, all but k of which are 0.

Give a fast algorithm (in terms of n and k for computing the resorted array of new prices after the changes, and analyze its running time.

Solution Format:

You answer to this problem should be a string containing a concise description of your algorithm and its runtime.

4. **One-Bit Error Correction (60 points)**

Suppose that you want to recover messages sent over a noisy channel. You are given a list of k valid messages m_1, m_2, ... m_k, each of which is an n-bit binary string. The messages r received from the channel are all corruptions of one of these k strings - each one differs from exactly one of the m_i in exactly one position. Your goal is to find the index i of the valid message that r is derived from.

Write a function `recover_original_messages` that takes two parameters, the lists `valid_messages` and `corrupted_messages`, and returns a list containing the indices of the valid messages corresponding to each corrupted message. Each of the valid and corrupted messages will be a string containing only the characters `'0'` and `'1'`. All of these strings will be the same length.

Note that the number of valid messages, the number of corrupted messages, and the length of each message will be quite large. Your algorithm should scale well with all of these parameters.

Here are some tests which your function should pass:

```
recover_original_messages(['000', '111'], ['110', '010']) == [1, 0]
recover_original_messages(['000000', '110001', '001110', '111111'],
                          ['001010', '110000', '110111']) == [2, 1, 3]
```

Solution Format:

You should answer this problem by filling in the body of the `recover_original_messages` function in the solution template.

Problem Set 3

This problem set is due **Wednesday, March 21** at **11:59PM.**

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **Hash Collisions (20 points)**

   Consider hashing integers which are selected independently at random from the universe $U = [1, 2, \ldots, 84]$.[1] Recall that a hash family h_i from U to $\{0, 1, \ldots, m-1\}$ is universal if, for any distinct x and y:

   $$\Pr_i\big[h_i(x) = h_i(y)\big] \leq \frac{1}{m}.$$

   a. Suppose we would like $m = 4$. Consider the hash family $h_i(x) = x^2 + x + i \pmod 4$, for $i \in \{0, 1, 2, 3\}$.

      1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_2(x) = x^2 + x + 2 \pmod 4$?
      2. The family $h_i(x)$ a universal hash family. `True` or `False`?

   b. Suppose we would like $m = 3$. Consider the hash family $h_i(x) = x^2 + i \pmod 3$, for $i \in \{0, 1, 2\}$.

      1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_1(x) = x^2 + 1 \pmod 3$?
      2. The family $h_i(x)$ a universal hash family. `True` or `False`?

[1] The integer keys are chosen independently. It is possible that we attempt to hash two things with the same key, in which case we will consider this a hash collision.

c. Suppose we would like $m = 12$. Consider the hash family $h_i(x) = ix + 2 \pmod{12}$, for $i \in \{0, \ldots, 11\}$.

   1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_7(x) = 7x + 2 \pmod{12}$?
   2. The family $h_i(x)$ a universal hash family. `True` or `False`?

d. Suppose we would like $m = 7$. Consider the hash family $h_i(x) = ix + 2 \pmod{7}$, for $i \in \{0, \ldots, 6\}$.

   1. What is the probability of having NO collisions when TWO random elements are hashed using the function $h_5(x) = 5x + 2 \pmod{7}$?
   2. The family $h_i(x)$ a universal hash family. `True` or `False`?

Solution Format:

Your answer for the first part of each question should consist of a float probability in the range $[0, 1]$, accurate to within 0.001 of the correct answer. Your answer to the second part of each question should be a boolean.

2. **Open Addressing (30 points)**

Suppose you are hashing integers into the hash table of size 10 below, using the hash function $h(k) = k \bmod 10$ to find the location of key k and using linear probing to resolve collisions.

After inserting 6 values into the empty hash table, the table is in the state below:

0	
1	
2	22
3	13
4	54
5	32
6	46
7	43
8	
9	

(a) Which one of the following insertion orders would result in this state?

1) 46, 22, 54, 32, 13, 43
2) 54, 22, 13, 32, 43, 46
3) 46, 54, 22, 13, 32, 43
4) 22, 46, 43, 13, 54, 32

(b) Suppose that 46 was deleted from the table. How many cells would be inspected if you then searched the table for 65?

(c) Is there some sequence of insertions and deletions, starting from an empty table, after which each cell i contains the value $i+1$? Give an example of such a sequence, or prove that no such sequence exists.

(d) Is there some sequence of insertions and deletions, starting from an empty table, after which each cell i contains the value $9 - i$? Give an example of such a sequence, or prove that no such sequence exists.

Solution Format:

For part a), your answer should be an integer choice, and for part b) it should be a integer answer. For parts c) and d), if you believe that no such sequence exists, your answer should be a string containing a proof of this fact. If you have a counterexample, you should enter it as a list of tuples; the first element of each tuples should either be an 'i' for insertion or a 'd' for deletion, and the second should be the key being inserted or deleted.

3. **Price changes (20 points)**

The local supermarket sells n products whose prices are stored in a sorted array $[p_1, p_2, \ldots p_n]$, where $p_i \leq p_{i+1}$ for all $i \leq n-1$.

After some seasonal price cuts, k of these prices are updated. You are informed of an array of price changes $[(i_1, d_1), \ldots, (i_k, d_k)]$. Here, a tuple (i, d) means that the ith price should be changed by d (which may be negative), so p_i is changed to $p_i + d$.

Give a fast algorithm which takes the original price array and the array of price changes, and computes the resorted array of new prices after the changes. Analyze its running time in terms of n and k.

You can assume that comparison of prices can be done in constant time (and you may not assume anything else about the prices).

Solution Format:

You answer to this problem should be a string containing a concise description of your algorithm and a brief analysis of its runtime.

4. **One-Bit Error Correction (60 points)**

Suppose that you want to recover messages sent over a noisy channel. You are given a list of k valid messages m_1, m_2, ... m_k, each of which is an n-bit binary string. The messages r received from the channel are all corruptions of one of these k strings - each one differs from exactly one of the m_i in exactly one position. Your goal is to find the index i of the valid message that r is derived from.

Write a function `recover_original_messages` that takes two parameters, the lists `valid_messages` and `corrupted_messages`, and returns a list containing the indices of the valid messages corresponding to each corrupted message. Each of the valid and corrupted messages will be a string containing only the characters `'0'` and `'1'`. All of these strings will be the same length.

Note that the number of valid messages, the number of corrupted messages, and the length of each message will be quite large. Your algorithm should scale well with all of these parameters.

Here are some tests which your function should pass:

```
recover_original_messages(['000', '111'], ['110', '010']) == [1, 0]
recover_original_messages(['000000', '110001', '001110', '111111'],
                          ['001010', '110000', '110111']) == [2, 1, 3]
```

Solution Format:

You should answer this problem by filling in the body of the `recover_original_messages` function in the solution template.

6.006 PS3

3/19

1. Hash Collisions

(I thought we were done w/ hashes...)

Hash integers $1 \to 84$

- pick ind.

Hash

$U \to 0 \cdots m-1$

distinct

$P_i \left[h_i(x) = h_i(y) \right] \leq \frac{1}{m}$

hash collision

a) Suppose $m = 4$

$h_i(x) = x^2 + x + i \pmod 4$

for $i \in \{0, 1, 2, 3\}$

1. P of no collisions when 2 random els are hashed using fn $h(x) = x^2 + x + 2 \pmod 4$

(2)

6.041 problem

So can have any input

i is fixed at 2 - does not seem to be important here

So is it = prob of 0 to 4

Yeah I'm pretty sure

So

	1st one	2nd one	
		0	
0		1, 2, 3	1/4
1		0, 1, 2, 3	1/4
2		0, 1, 2, 3	1/4
3		0, 1, 2, 3	1/4
w/ prob	1/4	0, 1, 2, 3	1/4

$$4\left(\frac{1}{4}\right) \cdot \frac{1}{4} = 1/4$$

i too simple

(3)

b) $h_i(x)$ are a universal hash family?

Check def

Collection universal if for each pair of distinct keys $k, l \in U$ the # of hash fns $h \in H$ for which $h(k) = h(l)$ is at most $\frac{|H|}{m}$

Basically w/ a hash function randomly chosen from H, the chanch of collision is no more that $\frac{1}{m}$ if keys randomly chosen

At first I was thinking no

But the key + 0, 1, 2, 3 means that it can ~~be~~ really be any value

like if hash 5

key 0

$5^2+5+0 = 2$
mod 4

key 1

$5^2+5+1 = 3$

(1)

Actually no chance of collison ??

and chance can be no more than $\frac{1}{m}$

True

b) Suppose $m = 3$

$h(x) = x^2 + i \pmod 3$

1. Then $i = 1$

i Same problem

~~say x = 5~~ values

~~can~~ 1st try

0
1
2

0
2
0
2
0
2

$3\left(\frac{1}{3}\right) : \frac{1}{3} = \frac{1}{3}$

i or is each option not equiv

2. Universal hash family

i is one as before → so true

(5)

c) $m = 12$

$h_i(x) = ix + 2 \pmod{12}$

1. ~~$h_7(x) + 7 \pmod{12}$~~

$h_7(x) = 7x + 2 \pmod{12}$

So if $x = 5$ $\quad x = 6$

$37 \bmod 12$

W Shri

$U = 1$ to 84

So can we use that extra info

Shri $\frac{41}{83}$ Take every # mod 4

0 mod 4 → 2

1 → 0

2 → 0

3 → 2

(6)

Math

$7 \cdot 6 \mod 4$

One way

$42 \mod 4 = 2$

Or

$(7 \bmod 4 \cdot 6 \bmod 4) \bmod 4 =$

Bad notation

$3 \qquad 2$

$\downarrow$

$6 \bmod 4 = 2$ ✓

So I didn't pay attention to mapping

we can have $01 \to 84$

Lets say 15

$15 \mod 4 = 3$

So

$3 \cdot 3 \bmod 4 + 3 + 2 \pmod 4$

$1 + 3 + 2 \equiv$

$= 2$

(7)

Ahh so its fixed input leads to fixed output

(So helpful!!)

So from 1 to 84

there are ? how many #s

mod 4 = 0

4, 8, ... 84 = 21 0s

21 1s

21 2s

21 3s

1st choice

$\frac{42}{84} = \frac{1}{2}$ 0

2nd

$\frac{42}{83}$ 0

since 1 away

$\frac{42}{84} = \frac{1}{2}$ 2

(8)

So $\frac{42}{84} \cdot \frac{\cancel{42}\,41}{83} + \frac{42}{84} \cdot \frac{\cancel{42}\,41}{83} + \cancel{\frac{42}{84} \cdot \frac{1}{83} = \frac{63}{166}}$

$= \frac{42}{83}$

~~will collide~~

~~want $1 - .3795 = .6205$~~

$1 - \frac{42}{83} = \cancel{.4939}$ .5060 $= \frac{41}{83}$

2) False since that mapping

√ I agree

---

b) $x^2 + 1 \pmod 3$

So $0 \to 1$

$1 \to 2$

$2 \to 2$

So then 16 is $16 \bmod 3 = 0$

1.1 mod1

(7)

(9)

So then

$$\frac{28}{84} \cdot \frac{27}{83} + \frac{56}{84} \cdot \frac{55}{83} < \frac{137}{249} \quad .5502$$

$$1 - \text{that} = \frac{112}{249} = .4497$$

2. still no good

---

c) $m = 12$ $\quad ix + 2 \pmod{12}$

$7x + 2 \pmod{12}$

input mod 12		
0	→	2
1		9
2		4
3		11
4		6
5		1
6	→	8
7		3
8		10
9		5
10		0
11	→	7

(10)

$$12\left(\frac{7}{84}\right) = \frac{6}{83} = \frac{6}{83}$$

$$1 - \frac{6}{83} = \frac{77}{83}$$

19277

2. True?

False since all i

Only works for rel prime

With a linear modulus get whole range of congruences

Otherwise cycles congruences → so not evenly distributed

---

d) $h_i(x) = ix + 2 \pmod 7$

a) Same method

$i = 5$

(11)

0	2
1	0
2	5
3	3
4	1
5	6
6	4

$$7 \cdot \left(\frac{1}{7}\right) \cdot \frac{11}{83} = \frac{11}{83}$$

$$1 - \frac{11}{83} = 86 - 14 = \frac{72}{83}$$ (1)

2. True

Since every < 7 is rel prime since 7 is prime

(11) b

2. Open addressing

$h(k) = k \bmod 10$

linear probing

table size 10

0	
1	
2	22
3	13
4	54
5	32
6	46
7	43
8	
9	

a) ~~[illegible]~~ What insertion order leads to this

I guess test each

$46 \bmod 10 = 6$

↑ just that

1)

0	
1	
2	22
3	32
4	54
5	13
6	43
7	46
8	
9	

(X)

(12)

2)

0	
1	
2	22
3	13
4	54
5	32
6	43
7	46
8	
9	

(X)

3)

0	
1	
2	22
3	13
4	54
5	32
6	46
7	43
8	
9	

(✓)

b) Suppose 46 deleted

How many cells inspected for 65

0		
1		
2	22	
3	13	
4	54	
5	32	←
6	deleted	←
7	43	←
8		←
9		

(4)

(3)

c) Is there some seq of insertions so that each sell cell contains $i+1$

No - one cell must be i

Then the rest

0	9
1	0
2	1
3	2
4	3
5	5
6	5
7	6
8	7
9	8

Say 5, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4

table full

Book stops at $h[h(k)-1]$

Shi: can delete later ?

so 5, 5, 6, 7 ..., 4

Then delete 5

~~get~~ insert 4

(4)

Put 15 ?

~~can~~ can delete later

5, 15, 6, 7, 8, 9, 0, 1, 2, 3, 4

delete 5

~~delete 15~~

insert 6

delete 15

insert 7

Oh want i + 1

~~5, 15,~~ how do you do this

0 → 1

Could do w/ ~20 yeah

So 5, 15, 6, . . . , 4

delete 5

insert 6

delete 15

insert 7

(15)

delete 6 ← which one?

So 1st time do mod values

5 6 7

5, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

delete 5
insert 6
d 15
i 7
d 16
i 8
etc

Nice + tricky

d) Now 9 − i

So 5 is 9−5 = 4

Shri Works

Want	
0	4
1	8
2	7
3	6
4	5
5	4
6	3
7	2
8	1
9	0

← (good to draw want table)

(16)

Do the same start

5
15
16
etc

Then replace

d 5
i 4
d 15
i 3

Not cleanest, but should work

Does not have to be <u>Optimal</u> counter example

(17)

3. Price Changes

n products

Sorted array $[p_1, p_2, \ldots, p_n]$

$p_i \leq p_{i+1}$ for $i < n-1$

Seasonal price cuts

k prices updated

$[(i_1, d_1), \ldots, (i_k, d_k)]$

A tuple (i, d) means ith price should be changed by d

$p_i = p_i + d$

Give a fast algo which computes array and resorts

Running time?

(18)

Selection → constant time

go that far in

But moment you rearrange

Make new array

1. Copy entire array n w/ old array as a value

n times

2. Find i 1
3. Compute $p_i + d$ 1
4. In new array find the old value i
5. Increment it up or down

- Comparison as much as n

So n^2 - sucks

AVL:

1. Build AVL of ps $n \log n$
2. For each price change k
3. Find item + delete $\log n$

(19)

4. Insert $p_i + d_i$ $\log n$) $k \log n$

Since $n \geq k$

$= n \log n$

Anything better?

Counting sort?

In class decision tree proved $n \log n$ as the lower bound

~~Still~~ So have bins for \$0 → 20.00

2000 bins

build n

Then each bin linked list of items w/ that price ~~the~~ (that could get big)

Worst case	n
Delete item	1
insert item	1
Report out	n

(20)

If you can have max price $O(n)$

So write 2 solutions

Note this i does supermarket imply max price i

Or max price m

so $O(m)$ in space

but still $O(n)$ in time

* Can insert at top of linked list

~~O(m)~~ $O(n)$ to find + delete.

Finding item #:

$4.98 $4.99 $5.00

item 1

item 5

item 7

item 9

Scanning whole thing O(n)

On can look up orginal price in <u>original</u> array

(22)

So no linked list just count

(I was ~~am~~ trying to think of pratical application
item id → price)

Rolling Hash (Rabin-Karp Algorithm)

Objective

If we have text string S and pattern string P, we want to determine whether or not P is found in S, i.e. P is a substring of S.

Notes on Strings

Strings are arrays of characters. Characters however can be interpreted as integers, with their exact values depending on what type of encoding is being used (e.g. ASCII, Unicode). This means we can treat strings as arrays of integers. Finding a way to convert an array of integers into a single integer allows us to hash strings with hash functions that expect numbers as input.

Since strings are arrays and not single elements, comparing two strings for equality is not as straightforward as comparing two integers for equality. To check to see if string A and string B are equal, we would have to iterate through all of A's elements and all of B's elements, making sure that $A[i] = B[i]$ for all i. This means that string comparison depends on the length of the strings. Comparing two n-length strings takes $O(n)$ time. Also, since hashing a string usually involves iterating through the string's elements, hashing a string of length n also takes $O(n)$ time.

Method

Say P has length L and S has length n. One way to search for P in S:

1. Hash P to get $h(P)$ **O(L)**

   L = window size

2. Iterate through all length L substrings of S, hashing those substrings and comparing to $h(P)$ **O(nL)**

3. If a substring hash value does match $h(P)$, do a string comparison on that substring and P, stopping if they do match and continuing if they do not. **O(L)**

This method takes $O(nL)$ time. We can improve on this runtime by using a **rolling hash**. In step 2. we looked at $O(n)$ substrings independently and took $O(L)$ to hash them all. These substrings however have a lot of overlap. For example, looking at length 5 substrings of "algorithms", the first two substrings are "algor" and "lgori". Wouldn't it be nice if we could take advantage of the fact that the two substrings share "lgor", which takes up most of each substring, to save some computation? It turns out we can with rolling hashes.

"Numerical" Example

Let's step back from strings for a second. Say we have P and S be two integer arrays:

$$P = [9, 0, 2, 1, 0] \tag{1}$$
$$S = [4, 8, 9, 0, 2, 1, 0, 7] \tag{2}$$

The length 5 substrings of S will be denoted as such:

$$S_0 = [4, 8, 9, 0, 2] \tag{3}$$
$$S_1 = [8, 9, 0, 2, 1] \tag{4}$$
$$S_2 = [9, 0, 2, 1, 0] \tag{5}$$
$$\ldots \tag{6}$$

Window size = 5

We want to see if P ever appears in S using the three steps in the method above. Our hash function will be:

Our window size $\frac{1}{2}n$

$$h(k) = (k[0]10^4 + k[1]10^3 + k[2]10^2 + k[3]10^1 + k[4]10^0) \bmod m \tag{7}$$

Or in other words, we will take the length 5 array of integers and concatenate the integers into a 5 digit number, then take the number mod m. $h(P) = 90210 \bmod m$, $h(S_0) = 48902 \bmod m$, and $h(S_1) = 89021 \bmod m$. Note that with this hash function, we can use $h(S_0)$ to help calculate $h(S_1)$. We start with 48902, chop off the first digit to get 8902, multiply by 10 to get 89020, and then add the next digit to get 89021. More formally:

$$h(S_{i+1}) = [(h(S_i) - (10^5 * \text{ first digit of } S_i)) * 10 + \text{ next digit after } S_i] \bmod m \tag{8}$$

We can imagine a window sliding over all the substrings in S. Calculating the hash value of the next substring only inspects two elements: the element leaving the window and the element entering the window. This is a dramatic difference from before, where we calculated each substring's hash values independently and would have to look at L elements for each hash calculation. Finding the hash value of the next substring is now a $O(1)$ operation.

In this numerical example, we looked at single digit integers and set our base $b = 10$ so that we can interpret the arithmetic easier. To generalize for other base b and other substring length L, our hash function is

$$h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + k[2]b^{L-3} \ldots k[L-1]b^0) \bmod m \tag{9}$$

And calculating the next hash value is:

$$h(S_{i+1}) = b(h(S_i) - b^{L-1}S[i]) + S[i+L] \bmod m \tag{10}$$

Back to Strings

Since strings can be interpreted as an array of integers, we can apply the same method we used on numbers to the initial problem, improving the runtime. The algorithm steps are now:

1. Hash P to get $h(P)$ **O(L)**
2. Hash the first length L substring of S **O(L)**
3. Use the rolling hash method to calculate the subsequent $O(n)$ substrings in S, comparing the hash values to $h(P)$ **O(n)**
4. If a substring hash value does match $h(P)$, do a string comparison on that substring and P, stopping if they do match and continuing if they do not. **O(L)**

This speeds up the algorithm and as long as the total time spent doing string comparison is $O(n)$, then the whole algorithm is also $O(n)$. We can run into problems if we expect $O(n)$ collisions in our hash table, since then we spend $O(nL)$ in step 4. Thus we have to ensure that our table size is $O(n)$ so that we expect $O(1)$ total collisions and only have to go to step 4 $O(1)$ times. In this case, we will spend $O(L)$ time in step 4, which still keeps the whole running time at $O(n)$.

Common Substring Problem

The algorithm described above takes in a specific pattern P and looks for it in S. However, the problem we've dealt with in lecture is seeing if two long strings of length n, S and T, share a common substring of length L. This may seem like a harder problem but we can show that it too has a runtime of $O(n)$ using rolling hashes. We will have a similar strategy:

1. Hash the first length L substring of S **O(L)**
2. Use the rolling hash method to calculate the subsequent $O(n)$ substrings in S, adding each substring into a hash table **O(n)**
3. Hash the first length L substring of T **O(L)**
4. Use the rolling hash method to calculate the hash values subsequent $O(n)$ substrings in T. For each substring, check the hash table to see if there are any collisions with substrings from S. **O(n)**
5. If a substring of T does collide with a substring of S, do a string comparison on those substrings, stopping if they do match and continuing if they do not. **O(L)**

However, to keep the running time at $O(n)$, again we have to be careful with limiting the number of collisions we have in step 5 so that we don't have to call too many string comparisons. This time, if our table size if $O(n)$, we expect $O(1)$ substrings in each slot of the hash table so we expect $O(1)$ collisions for each substring of T. This results in a total of $O(n)$ string comparisons

which takes $O(nL)$ time, making string comparison the performance bottleneck now. We can increase table size and modify our hash function so that the hash table has $O(n^2)$ slots, leading to an expectation of $O(\frac{1}{n})$ collisions for each substring of T. This solves our problem and returns the total runtime to $O(n)$ but we may not necessarily have the resources to create a large table like that.

Instead, we will take advantage of string **signatures**. In addition to inserting the actual substring into the hash table, we will also associate each substring with another hash value, $h_s(k)$. Note that this hash value is different from the one we used to insert the substring into the hash table. The $h_s k$ hash function actually maps strings to a range 0 to n^2 as opposed to 0 to n like $h(k)$. Now, when we have collisions inside the hash table, before we actually do the expensive string comparison operation, we first compare the signatures of the two strings. If the signatures of the two strings do not match, then we can skip the string comparison. For two substrings k_1 and k_2, only if $h(k_1) = h(k_2)$ and $h_s(k_1) = h_s(k_2)$ do we actually make the string comparison. For a well chosen $h_s(k)$ function, this will reduce the expected time spent doing string comparisons back to $O(n)$, keeping the common substring problem's runtime at $O(n)$.

(23)

#4 One Bit Error Correction

k valid messages $m_1, m_2, \ldots m_k$
each of which is a n bit string

The messages r are all corruptions k
each 1 differs in exactly 1 position

Find the index i of the valid message
r is derived from

Which message # it was originally

∴ So every hash 1 bit permutation
$k \cdot n$

All strings are same strenght

Should scale w/ k, n

$k \neq r$

$k \leq r$

(24)

Counting sort or radix sort
↑ silly ↑ good

Find hamming distance
↳ must be n-1
but how do efficently

Tree → decision tree
Similar to sorting - if 1 is wrong
(Try to come up w/ good sol before coding!)
This was the search unit

Compare each → k·r (or k^2)

Calc #of 1s
↳ "parity bit" must be ± 1
easy to compare

Matrix - multiply each

25

Parity in worst case - no help

but this is actual time

they could have algorithms that suck at this

Must hash or sort or heap

Decision tree w/ the 1 bit flip

---

like function in SQL?

Should we write nieve lst + fix?

Hint Piazza #252

"Think of a nieve solution that re does a lot of computation. Have you seen this in lecture before"

rolling hash??

(26)

Hash whole thing
Tweak 1st bit - rehash that bin
2nd bit rehash that bit

¿Is this the best you can go

~~Neve~~

¿Modify original + more
but $k \geq c$?

So could do diff for $k > c$ or $c > k$?

Mei Rec 6 notes Robin-Karp
Must have common substring ≥ 500
If strings randomly distributed good

Convert binary to decimal
- ¿faster but more hash collissions
- but can't flip bits
- ahh but common substring

E7

but then "16" vs "6"

is 1 binary bit off

but multiple decimals

Lets just rolling hash to find 2 500 common chars. Then think later what to do w/ results

---

Shri did XOR test case

On my own

¿How to do a rolling hash?

¿Build on my own...

Rec 06 notes from spring 11

$$h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + k[2]b^{L-3} + \cdots k[L-1]b^{0}) \bmod m$$

L = lenght of window

(28)

Hash what

- the ones that less

$k < r$

So hash r

Check for P in S

? hash both

- think about it later

M ?

? some prime?

$2^{19} - 1$

$p = 2 * * 31 - 1$

3/20

Kishore: Rolling hash where can easily change value
Rabin-Karp could be made to work - but
do efficently

So need to invent a hash

Or simpler

$1 \cdot (L)^{position}$

base

Then subtract $(L)^{position}$ mod m

? What base, what mod?

What did we talk about in class
base is a prime

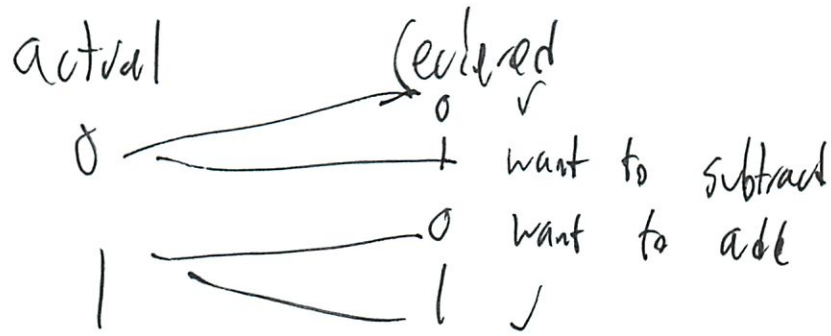
Slides mention b^L mod m

6.006 More Work

3/20

Ok now try to implement

So can be



if low check for existance
- if try inserting

No - Use hash value as key

Got finished version - now figure how to test...
why is it blanking ??
Ok pasted into world

Now fixing little bugs...

Ok all are failing
print + see

(2)

Answers I am getting are pretty similar

Ahh n not in tree

Stuff that is 0 is not going right!

Parenthesis issue!!

└ Yes - grrrr

~~Ok~~

i Hash not unique enough

- Verify running right

Lots of hash collissions!

└ its the flow

Ohh I never changed to string for function!!!

(really stupid mistake!

So now the check

either

(3)

I think hash collision - matched too early!

; better hash

I am doing a full one!

Oh 100 matches 000

Was exponentiating wrong ** not ^

(✓) 2 simple cases

Test

(✗) earlier problem crashed

Tried to insert in full table

(✓) Fixed WA those

~~Now Test offline~~

(✗) Timeout

Test things off line

(✓) seems to work on medium!

(4)

Test medium 2 seems to time out

Something never finishes

flashing very slowly!

emailed in - but he was slow at getting back

both my partners turning into int

Due Wed at ~~[illegible]~~ 11:45PM

I need to work on other stuff!

Crystal int(string, 2)

then

+ bitflip 2^x

$\pm$

Works much ~~better~~ faster

- but wrong!

(✓) Fixed. Passed simple test cases

(✓) Passes medium - faster!

(5)

Instead keep temp variable - multiply by 2 each time
; or bit shift

Passed online ~~all~~ easy 1-5
medium 1-3
fail 4 in time

Bitshift is squared?
- last p-set I forgot

Bitshift is 2^n

$\# < n$

#	is multiply by
1	2
2	4
3	8

So the temp ~~[illegible]~~ divide by 2 wrong

(6)

(X) Wrong now

Oh put temp in wrong place!

(✓) 56 points !!!

Recs: Start at end $0 \rightarrow 2^{n-1}$ ← add not multiply

not $2^{n-1} \rightarrow 0$

Also /2 vs >>1

no Seperate function

Divide by 2 (X) 36 pts - sucks

No sep fn (−) 56 pts - same

flip (X) 55 pts

Other optimizations (✓) 57 pts / 60 pts

Give up

```
collaborators = 'Arianna, Crystal, and Shri'

# Enter a float in the interval [0.0, 1.0] for each part 1 of problem 1.
# Enter True or False for each part 2.
answer_for_problem_1_part_a_1 = .500
answer_for_problem_1_part_a_2 = False
answer_for_problem_1_part_b_1 = .444
answer_for_problem_1_part_b_2 = False
answer_for_problem_1_part_c_1 = .916
answer_for_problem_1_part_c_2 = False
answer_for_problem_1_part_d_1 = .857
answer_for_problem_1_part_d_2 = True


# On problem 2, enter an integer for parts a and b.
# For parts c and d, enter a string proving that no such sequence exists,
# or an insertion sequence providing a counter-example, not both.
answer_for_problem_2_part_a = 3
answer_for_problem_2_part_b = 4
# Uncomment one of the following lines for part c, and enter your answer
#answer_for_problem_2_part_c = 'Type your proof here.'
answer_for_problem_2_part_c = [('i', 5), ('i', 15), ('i', 16), ('i', 17), ('i', 18), ('i',
19), ('i', 20), ('i', 21), ('i', 22), ('i', 23), ('d', 5), ('i', 6), ('d', 15), ('i', 7), (
'd', 16), ('i', 8), ('d', 17), ('i', 9), ('d', 18), ('i', 10), ('d', 19), ('i', 1), ('d', 20
), ('i', 2),('d', 21), ('i', 3),('d', 22), ('i', 4),('d', 23), ('i', 5)]
# Uncomment one of the following lines for part d, and enter your answer
#answer_for_problem_2_part_d = 'Type your proof here.'
answer_for_problem_2_part_d = [('i', 5), ('i', 15), ('i', 16), ('i', 17), ('i', 18), ('i',
19), ('i', 20), ('i', 21), ('i', 22), ('i', 23), ('d', 5), ('i', 4), ('d', 15), ('i', 3), (
'd', 16), ('i', 2), ('d', 17), ('i', 1), ('d', 18), ('i', 0), ('d', 19), ('i', 9), ('d', 20
), ('i', 8),('d', 21), ('i', 7),('d', 22), ('i', 6),('d', 23), ('i', 5)]

# Enter your answer to problem 3 here.
answer_for_problem_3 = '''
So this has 2 possible answers depending on this question: is there a max price in our
supermarket?  Generally the prices of items are in the $1 to $7 range.  There are very vew
to no items above $20.  Perhaps there are no items over $100.  (If so those items could be
done under a nieve, parallel process that is optimized for very few items)

If yes: basically counting sort in O(n).
1.  Have m bins for each price from $0.00 to $(m/100) max price.
2. Sort each item into a bin.  We don't need to maintain which item the price is for, only
the count of the items which are that price.  This takes O(1) for n items = O(n)
3. Iterate through the k price changes.  Look up the i-th item's original price in the
original array.  This takes O(1).
4. Find this price's bin and decrement the count by 1.  O(1)
5. Calculate p+d.  Find this new price's bin and increment by 1.  O(1) as before. O(k) for
all
6. When done, build the output array. O(n)
So because it is O(n)+O(k)+O(n) and n >= k, it's O(n).

If no, no max price: AVL tree in O(nlogn)
```

```
1. Build an AVL of Ps O(nlogn)
2. For each of the k price changes, find the item and delete it.  This takes O(log n)
3. Insert p+d.  O(logn)  For k price changes O(klogn)
4. Output the final array O(n) to iterate through in order.
So since n >=k, it's O(nlogn)
'''

# Fill in the body of the code for problem 4.
def recover_original_messages(valid_messages, corrupted_messages):
    n = len(valid_messages[0]) #they are all the same

    #hash all of the valid messages
    valid_hashes = {}
    i = 0
    for valid_message in valid_messages:
        valid_hashes[int(valid_message, 2)] = i
        i = i + 1

    #for each received message, flip bits tell we find
    answer = []
    for corrupted_message in corrupted_messages:
        hashsum = int(corrupted_message, 2)
        #flip each bit and check
        i = 0
        temp = 1
        while i < n:
            hashsumtemp=hashsum^temp
            if hashsumtemp in valid_hashes:
                answer.append(valid_hashes[hashsumtemp])
                break;
            i = i + 1
            temp = temp << 1

    return answer
```

```
import random

collaborators = ''

# Enter a float in the interval [0.0, 1.0] for each part 1 of problem 1.
# Enter True or False for each part 2.

"""
Notice that none of these are universal hash families, since if m is what we are modding by,
then we have that for all i: h_i(x) = h_i(x + m)
Universal hash constructions are typically more complicated.
"""

answer_for_problem_1_part_a_1 = 1/2.0
answer_for_problem_1_part_a_2 = False
answer_for_problem_1_part_b_1 = 4/9.0
answer_for_problem_1_part_b_2 = False
answer_for_problem_1_part_c_1 = 11/12.0
answer_for_problem_1_part_c_2 = False
answer_for_problem_1_part_d_1 = 6/7.0
answer_for_problem_1_part_d_2 = False

# On problem 2, enter an integer for parts a and b.
# For parts c and d, enter a string proving that no such sequence exists,
# or an insertion sequence providing a counter-example, not both.
answer_for_problem_2_part_a = 3
answer_for_problem_2_part_b = 4

"""
Here is a general solution:
desired is an array, with what you want in slot i of the open addressing table in desired[i]
"""

def get_sequence(desired):
  sequence = [('i', 100 + x) for x in xrange(10)]
  for i in range(10):
    sequence.append(('d', 100 + i))
    sequence.append(('i', desired[i]))
  return sequence

answer_for_problem_2_part_c = get_sequence([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
answer_for_problem_2_part_d = get_sequence([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

"""
Here are some shorter sequences that also work
"""
answer_for_problem_2_part_c = [('i', 0), ('i', 11), ('i',12), ('i', 13), ('i', 14), ('i', 15
),\
                               ('i', 16), ('i', 17), ('i', 18), ('i', 10), ('d', 18), ('i',
9),\
                               ('d', 17), ('i', 8), ('d', 16), ('i', 7), ('d', 15), ('i', 6
), \
```

```
                                ('d', 14), ('i', 5), ('d', 13), ('i', 4), ('d', 12), ('i', 3
), \
                                ('d', 11), ('i', 2), ('d', 0), ('i', 1)]
answer_for_problem_2_part_d = [('i', 19), ('i', 9), ('i',18), ('i', 8), ('i', 17), ('i', 7),\
                                ('i', 16), ('i', 6), ('i', 15), ('i', 5), ('d', 19), ('d', 18
),\
                                ('d', 17), ('d', 16),('d', 15), ('i', 4),('i', 3),('i', 2),(
'i', 1),('i',0)]


# Enter your answer to problem 3 here.
answer_for_problem_3 = '''
Algorithm:
  Make a new array of size k, which contains the updated entries p_(i_j) + d_j for j in
range(1,k+1).
  While doing so, mark the changed prices in the original array with a special 'Changed'
marker.  This all takes O(k).
  Sort the new array, in O(k log k)
  Merge the new array and the original array, using the same algorithm as that from
merge-sort,
     but skip the finger in the original array over entries with the 'Changed' marker.
This takes O(n).
  Return this merged array

The algorithm is clearly correct, and runs in O(k log k + n).

An informal, non-rigorous argument that this is optimal:
We can't do better than O(k log k), since we need to sort k new prices, which could be
arbitrary.
We can't do better than O(n), since if the smallest price was updated to the middle, and
nothing else changed,
we would still need to shift at least n/2 elements in the array
Thus we can't do better than O(max(k log k, n)) = O(k log k + n)
'''

"""
Our original solution to problem 4, which was actually designed for
the problem of correcting one-off polyominoes, instead of bitstrings
"""

# First, preprocess a bunch of random 64 bit integers r_i, each of which corresponds to one
position in a messgae
r = [random.randint(1, (2**63) - 1) for i in xrange(50000)]
roll = [r[0]] + [r[i-1] ^ r[i] for i in xrange(1, 50000)]

def recover_original_messages(valid_messages, corrupted_messages):
  (n, k) = (len(valid_messages[0]), len(valid_messages))

  # Our hash function is the xor of the r_i's where message[i] is 1
  # This hash is easy to roll
  def myhash(message):
    return reduce(lambda x, y : x ^ y, (r[i] for i in xrange(n) if message[i] == '1'), 0)
```

```
  valid_hash = {}
  for i in xrange(k):
    valid_hash[myhash(valid_messages[i])] = i

  answer = []
  for cor in corrupted_messages:
    hash = myhash(cor)
    for j in xrange(n):
      hash ^= roll[j]
      if hash in valid_hash:
        answer.append(valid_hash[hash])
        break

  return answer

"""
Here's a time-optimized solution which gets ~0.20 seconds total on the large test cases
Code stolen (with permission) from Joshua Blum, Tal Tchwella, and Rishikesh Tirumala and
modified.
I'm sure many others had similar code, and it's possible there were faster things out there
"""

def recover_original_messages(valid_messages, corrupted_messages):
  (n, k, c) = (len(valid_messages[0]), len(valid_messages), len(corrupted_messages))

  answer = [0] * c

  valid_hash = {}
  for i in xrange(k):
    valid_hash[int(valid_messages[i], 2)] = i

  corrupt_hash = {}
  for i in xrange(c):
    corrupt_hash[int(corrupted_messages[i],2)] = i

  if (n < 5000): # This part due to Rishikesh Tirumala

    pows = [(1 << i) for i in xrange(n)]
    for c_hash in corrupt_hash:
      for i in xrange(n):
        hash = c_hash ^ pows[i] # Flip bit i
        if hash in valid_hash:
          answer[corrupt_hash[c_hash]] = valid_hash[hash]
          break

  else: # This part due to Joshua Blum, Tal Tchwella

    for c_hash in corrupt_hash:
      for v_hash in valid_hash:
        xor = c_hash ^ v_hash # bitwise xor should have exactly one 1
        if not (xor & (xor-1)): # check if xor is a power of two
```

```
        answer[corrupt_hash[c_hash]] = valid_hash[v_hash]
        break

    return answer
```

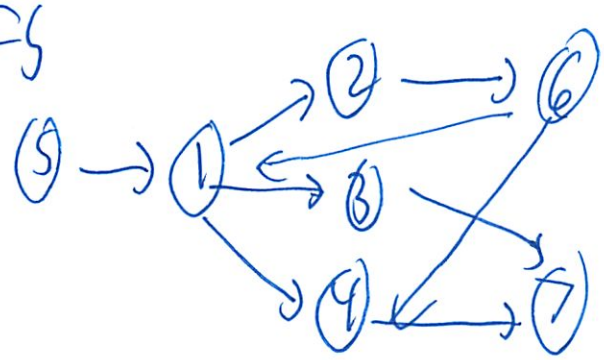
6.006 Recitation

3/21

(substitute TA)

DFS + BFS on directed graphs

DFS



(he mumbles)

1, 2, 6, 4, 7

3

4

Not possible to reach 5 from 1

DFS(v) ← recursive code

```
def dfs(w):
    visited[w] = True
    visit(w)   // could be anything
    for u in w neighbors:
        if not visited[u]:
            dfs[u]
```

②

BFS

Same chart

1

2, 3, 4

6, 7

Could each list individually

Pretty straightforward

Or put everything on 1 list

$\underbrace{1}_{\text{source}}, \underbrace{2, 3, 4}_{\text{distance } 1}, \underbrace{6, 7}_{\text{distance } 2}$

BFS (v)

initialize hasSeen array to False

queue[0] = v

hasSeen[v] = True

(A) (3)

```
curPosition = 0

while curPosition < queue.length
    w = queue[curPosition]
    visit(w)
    curPosition = curPosition + 1
    for v in w.neighbors:
        if not hasSeen[v]:
            queue.append(v)
            hasSeen[v] = True
```

(Handed exam back)
(Recitation ended 30 min early)

6.006 Lecture

3/22

(3 min late)

(no PPTs) again

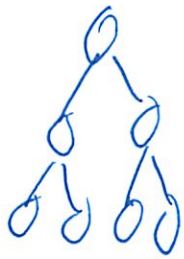
(he's calling on people w/ laptops...) haha

BFS $O(n+m)$

DFS $O(n+m)$

Goal: find who is connected to us

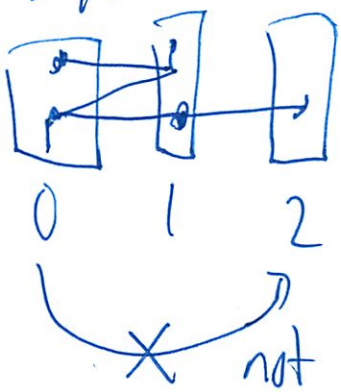
Structure → spanning tree



Connects all nodes

can see distances from height

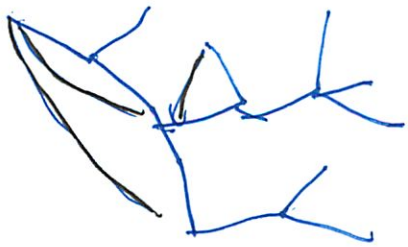
Build layers



not skipping layers

Free and back edges

- store a lot of info



②

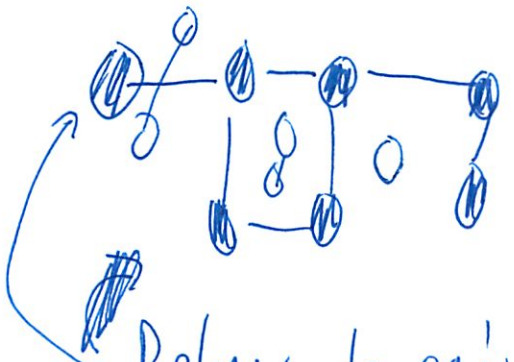
Connected Components

Lecture: undirected
Recitation: undirected

$\leftrightarrow$ Symmetric
$\circlearrowleft$ reflexive
$\rightarrow\rightarrow$ transitive

a relation that satisfies all 3 = equivilence relation

a equivilant class
- The maximal set such that every pair is related to each other
- maximal = can't add to it



Belongs to equiv class
7 nodes

others: 2, 2, 1 nodes

③

algorithm has inputs + outputs

↑ graph
Undirected

Mark each vertice in each class w/ diff symbols ~~the~~

- remember all symbols are #s
- So lable 1, 2, 3, 4

How get it?

Pick a node, do a search

- Did BFS 3 times
- Want to mark
- Shortest path - distance b/w me and the people I can reach

M(arked)

M[1 ... n]

M[i] ∈ {1, ..., k} # of connected components

(4)

(0) $\forall i \quad M[i] = 0$

Pick a node

Start $\leftarrow$ 1st vertex

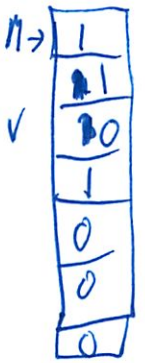
$i \leftarrow 1$

(1) BFS from Start

mark "i" all that are reached

We want all

(2)



Start $\leftarrow$ V i $\leftarrow$ i+1 etc

You only march through array once $O(1)$

(3) When nothing else is marked → halt

(5)

Complexity $O(n-m)$

Things are not $O(n\ m)$

? ? ensure what said

Start

its n' m' in the connected component

are partitions

You don't double count

$$\sum_i O(n^i + m^i) = O(n+m)$$

Must learn to count

Things are not as pessimistic as it seems

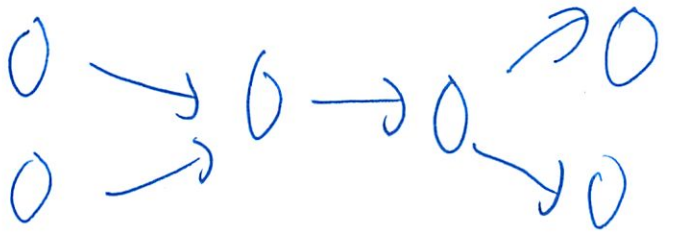
We ~~m~~ saw this in heaps

- we did a very ~~fw~~ careful count

(6)

Topological Sort

directed graph



represent a time constrain
priority order of stuff to do

is a number/renumbering t_s of the nodes
such that $u \to v$ implies
$t_s(u) < t_s(v)$

$\exists$ ~~t~~ t_s for all directed graphs
↳ No, not if there is a cycle

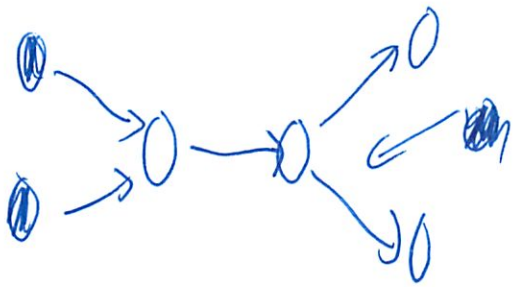
7

DAG

there are no cycles

So will be a topological graph

Canidates



no nodes pointing to them = source

---

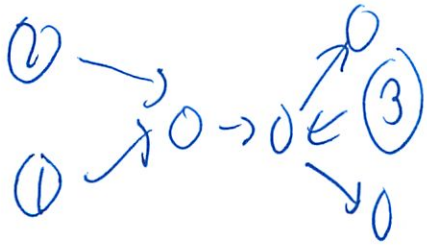
topological sort must have that

---

if found one we did see before → is a cycle

How # the sources?

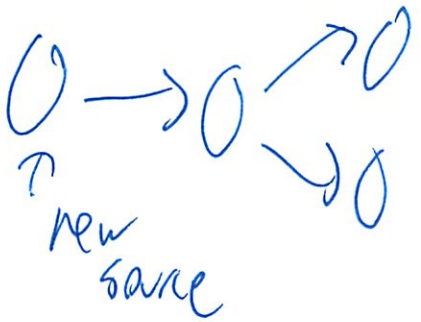
it doesn't matter



(8)

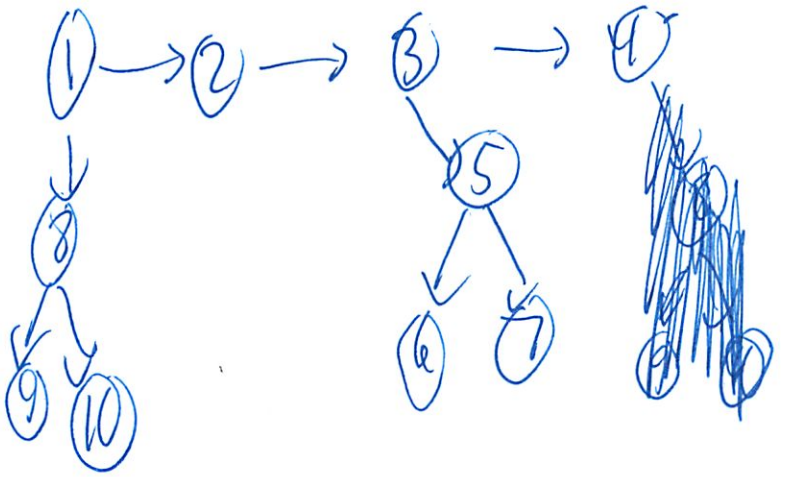
Remove ~~[illegible]~~ sources

Re-topological sort



$O(nm)$

(this lecture is very confusing)



Topological reverse

As

A in reverse

$u \to v$

$t\,R(u) \to t\,R(v)$

$n - t\,R(u) = t\,S(u)$

if we can find topological reverse, we are done

Can we have an edge

$3 \dashrightarrow 7$

So backtrack

How to backtrack w/o exploring

(10)

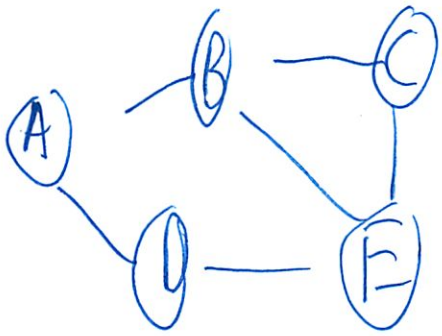
Find <u>sinks</u> – ~~edges~~ nodes w/ no edges out

6.006 Recitation

3/23

(3 min late)

	Undir	dir
BFS		
DFS		



Push vertices to list to process

Pop a vertex

~~All~~ all the vertices that have not been processed

Set ([ ])

BFS: Push right, pop left (queue)

[A]

DFS: push right, pop right (stack)

(same as 6.01)

②

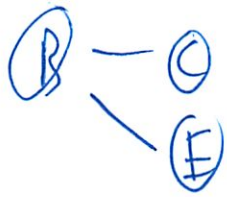
A B D

~~A~~ B D

↓

A B D C E

~~A~~ ~~B~~ ~~D~~ C E



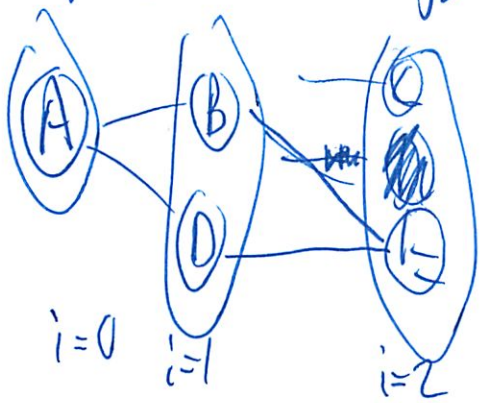
↓

A B D C E

~~A~~ ~~B~~ ~~D~~ ~~C~~ ~~E~~

Append to list
But don't delete from left - use pointer

Split up into things in same level



(3)

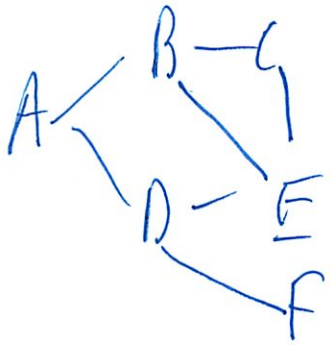
Structural Theorm for undirected BFS
every non tree edge is in 1 level b/w
two adj levels

Order you add does not matter
- in lecture they showed a clockwise order

DFS

Set (A)

[~~A~~ B ~~D~~]



↓

A B D E F

~~A~~ B ~~D~~ E F

↓

A B D E F C

~~A~~ B ~~D~~ ~~E~~ ~~F~~ C

(4)

A B C D E F C

A B D E F C

2nd 1st

This is different than lecture

Structural theorm does not hold

6.01 ← taught in

DFS is actually fairly hard

Structual theorm (that should hold) [undirected]

All edges go b/w vertex and an ancestor

B → E

He does not know how to code the in class version of DFS

(5)

Diff way to code DFS

- behaves like the lecture version

Set ([ ])

Current path [A]

↓

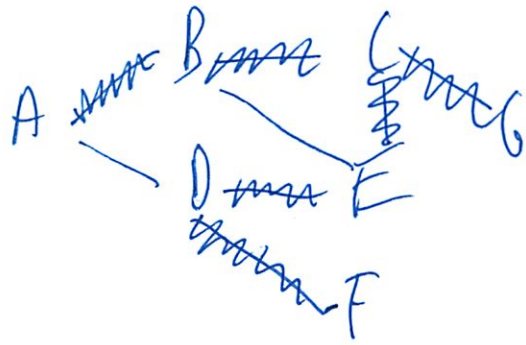
A B C E

A B C E

↓

A B C E

A B C E D F

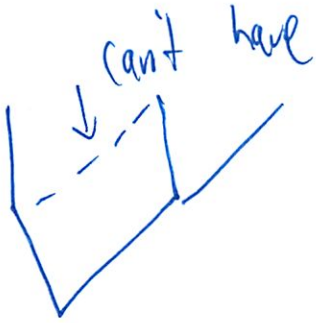


A B C D F G

A B C G

Theorem: All edges go b/w a vertex and its ancestors

(6)



Or the tree would have look very differently

Directed

BFS

can have all dotted lie edges

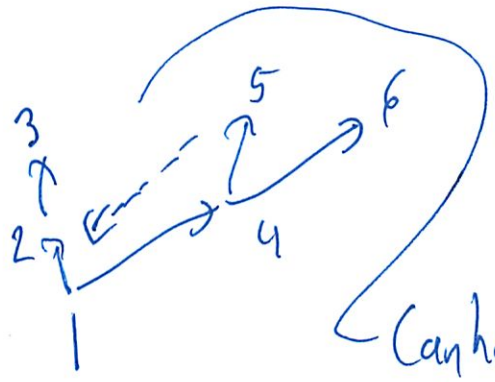
is new edge you are allowed to have

Directed BFS Theorm

- edges can go back arbitrary many levels
- edges can go forward up to 1 level

DFS

You can't get order from looking at the tree. It matters how the tree was built. Any edge is legal as long as it respects the exploration order



Can have dotted line since going backward

Topological Sort

At end

Where order always goes forward

Here every non tree edge goes backward

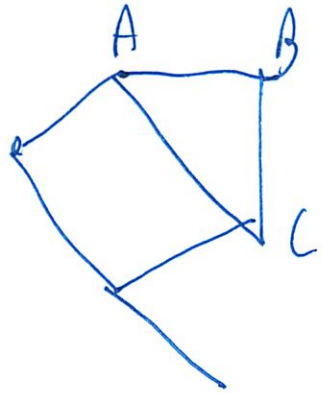
8

Can change graph, but must sort

Can turn into top. sort w/ "post" order

Harder problems related to graph

Find 3 vertices in a triangle
as an adj list

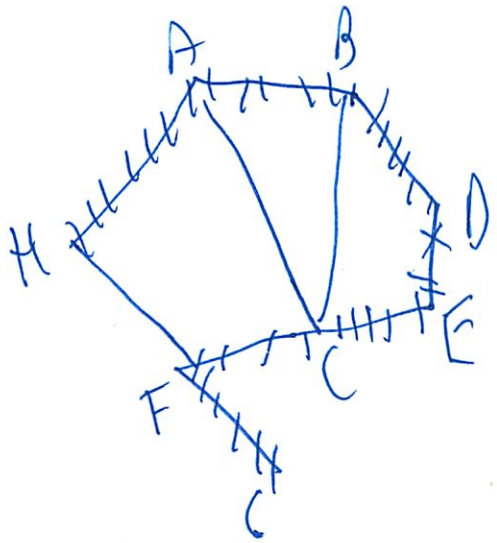


~~Use a DFS to go through nodes~~

DFS - vertices close to ye

But that is not ness. what get

9



; Find when backtracks 4
; might work

New idea:

make lists → adj lists

A [B C H A]
B [D C A B]
C [F A B E C]

Very slow run time

$V^3 \cdot V = V^4$
; # of neighbors

(10)

Another idea 2 nodes into layer

All possible triples - see if triangle
- w/ adj list

If they are all neighbors

So instead adj sets
adj hash tables

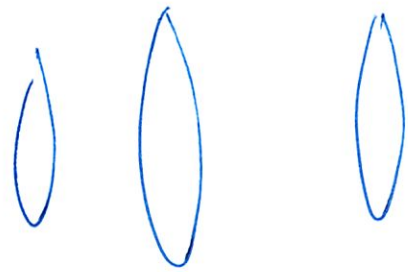
V^3 - Turn adj lists into adj hash tables
Check each triple

or the whole adj matrix
└ useful here

11

One more (ans)

BFS



triangle has 1 in 1 level

2 in other level
nodes

if hash carefully → ve time

Problem is actually really hard

fastest known

$$O\left(\min\left(V^{2.36}, E^{1.4}\right)\right)$$

- One for sparse, one for dense

A^2 = things reached w/ path len 2

A^3 " 3

6.006 Graphs

3/28

Reading

graphs are all over CS

graph-searching → one of the most basic operations

Representations $G = (V, E)$

adj list

best when sparse $|E| << |V|^2$

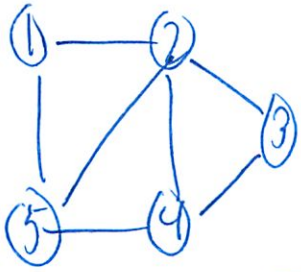
most algo. in this book use

adj matrix

best when dense $|E| \approx |V|^2$

or to see if edge connecting 2 vertices

Undirected



adj list

1 → 2 → 5
2 → 1 → 5, 3, 4
3 → 2, 4
4 → 2, 5, 3
5 → 4, 1, 2

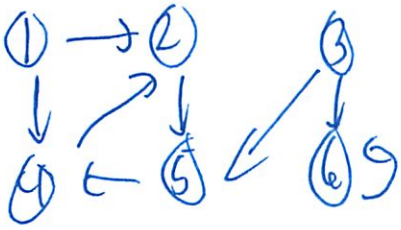
linked list

adj matrix

	1	2	3	4	5
1					
2	0	1	0	0	1
3		0	1	1	1
4			0	1	0
5	symetric			0	1
					0

symmetric

Directed



1 → 2 → 4
2 → 5
3 → 6, 5
4 → 2
5 → 4
6 → 6

	1	2	3	4	5	6
1						
2	0	1	0	1	0	0
3	0	0	0	0	1	0
4	0	0	0	0	1	1
5	0	1	0	0	0	0
6	0	0	0	1	0	0
	0	0	0	0	0	1

②

<u>adj list</u> sum is $|E|$ directed
$2|E|$ undirected

Can adapt to add weights

<u>adj matrix</u> $\Theta(V^2)$ memory
but undirected → can cut almost in half

adj list are easy at least as much space as adj matrix

<u>BFS</u>

One of the simplest algorithms

give a start point s

build <u>breath first tree</u>
has shortest path from s

directed and undirected

Colors white – un touched
gray – discovered
black – discovered <u>and</u> searched all imediate children

don't really need

(3)

first to discover a node → parent

first in - first out queue

Shortest path ↓
- will be on tree
(Hopefully we don't need to be adept at the proofs...)

tree: Predecessor Subgraph G_π

can use to print shortest path
runs in Θ(depth)

DFS

goes to farthest node
backtracks
explores next

predecessor subgraphs may be several trees
since search can repeat from several places
not the same as BFS!

9

Search may repeat from multiple places

?hmm

forms <u>depth first forest</u> w/ several dfs trees

Same colors

each vertex in 1 DF tree

trees may be disjoint

this may be the difference - but why?

Record 2 timestamps

- when first discovered (greyed) v.d
- when adj list searched (blackened) v.f
- v.d < v.f
- timestamps 1 ⇔ 2|V|

edges are explored

Can have different orders of visiting neighbors

↳ not a problem in pratice

(5)

Runtime

$\Theta(V)$ for finding each time

$\sum_{v \in V} |Adj[v]| = \Theta(E)$ for DFS Visit

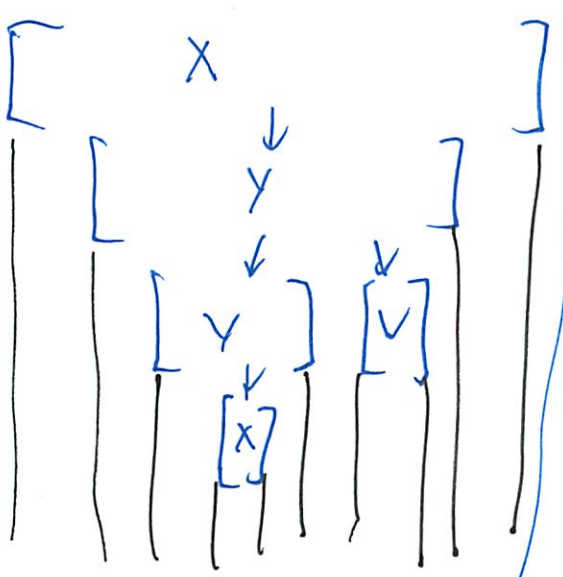
So $\Theta(V + E)$

(forest seems to be from original starting points
but how does BFS not have it
or is for directed only?)

"search may be repeated from multiple places"

Parenthesesis structure

Shows when start / stop looking



(x(z(y(x x) y) (w w) z) x)

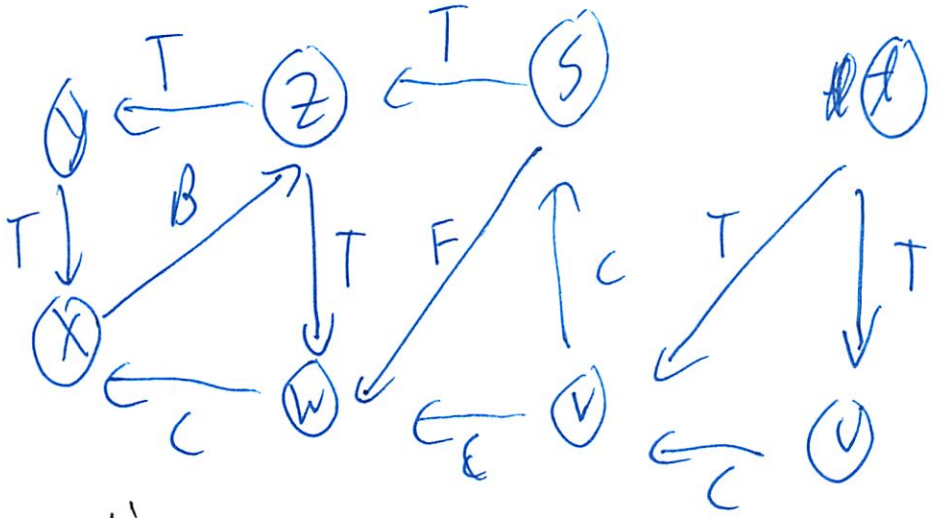
= time steps

Classification of Edges

DFS Can classify edges by type

1. Tree edges - edges in DF forest G_π
   - 1st time it was discovered
2. Back edges - edges (u, v) connecting a vertex u to an ancestor v in a DFS tree (goes back to an ancestor)
   inc self loops
3. Forward edges non tree edges (u, v) connecting a vertex u to a decendent v in a DFS tree
   (an example?)
4. Cross edges All other edges
   Go b/w vertices that are not ancestors

7



correct:

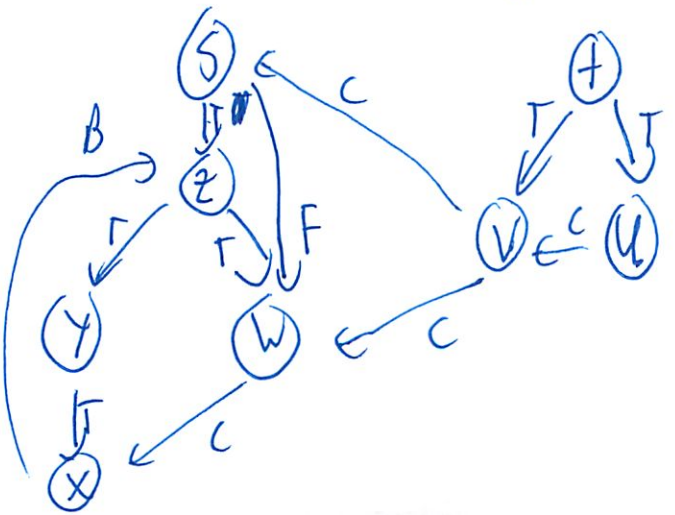
So we do T first (DFS)

B = back to an ancestor

F = goes to a decendent we already discovered a different way

C = all other edges (not decendents/ancestors) like peer (u to v)

We can redraw any graph so



T, F downward

B upward

C anyway

(8) Rules don't seem 100% clear!

forward edge $u.d < v.d$
cross edge $u.d > v.d$) clearer

Undirected graphs → first type of classification that applies

(u, v) or (v, u) → whatever is seen first

Can only be forward or back

↑ seen first ↑ other way

Topological Sort

of a DAG
└ directed, acyclic graph
- gives precidence info

horizontal sort where everything $l \to r$

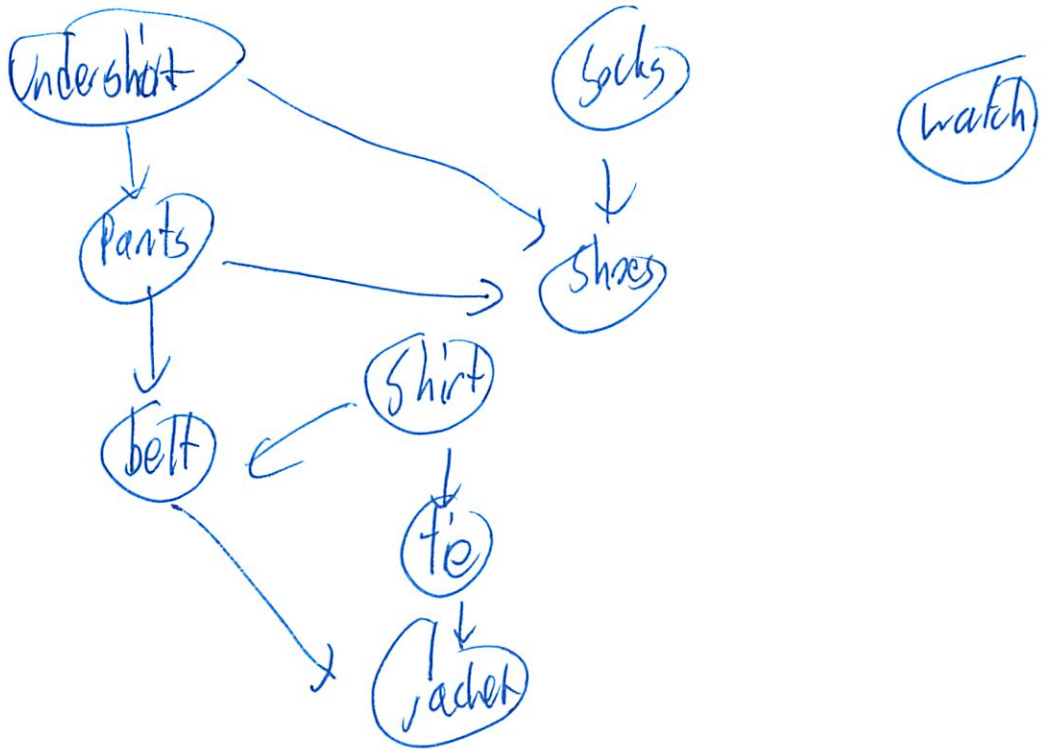
(no backward edges)

$\Theta(V + E)$
↑ for DFS

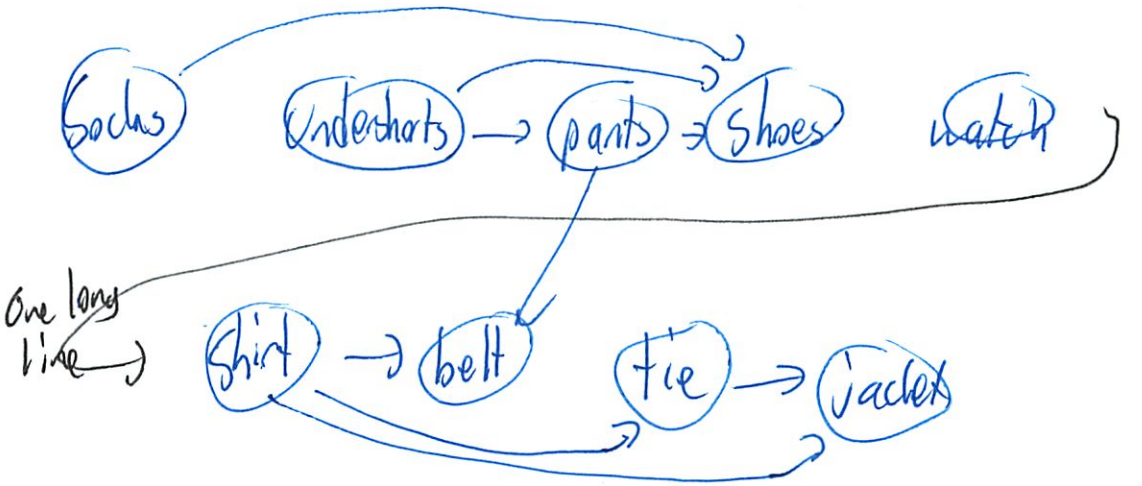
$O(1)$ for each V entry

9 (? why didn't they introduce DAG seperatly ?)

DAG



Topological Sort



Topological sort (G)

1. Call DFS(G) to compute finishing time v.f for each vertex v
   ↑ I has reading on this!
2. As each vertex is finished, insert at front of linked list
3. Return linked list of vertices

(10)

Strongly Connected Components

a classic app of DFS

Using 2 DFSes

Many algorithms decompose into connected components

Process on each

recombine results

Strongly connected for every pair of vertices

u and v we have $u \to v$

$v \to u$

G^T uses E^T which is all arrows reversed

└ time to create $O(V+E)$

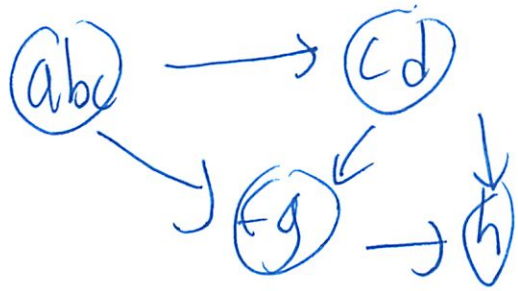
└ has <u>same</u> strongly connected components

Component Graph G^{SCC}

Connects all of the SCC.

is a DAG

(or else that part would be a SCC)



Minimal Spanning Tree

Often must connect several nodes (n) together

can do in n-1 wires

but could we use less?

want to use the least # or Edges

$$W(T) = \sum_{(u,v) \in T} w(u,v)$$

2 algorithms

Kruskal $O(E \lg V)$
Prim $\quad \parallel \quad$ or $O(E + V \lg V)$
↑ better if $|V| << |E|$

Both are greedy - best at the moment
(so this is out of scope)

Shortest Path

how to plot the shortest path from here to here?
Could look at all routes?
all routes w/o cycles?
Sometimes have weights $w(p)$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

↑ sum of ind. weights

δ = shortest path weight
↳ if no path

(13)

BFS works when all weights = 1

have a given <u>single source</u>

want shortest path to <u>all</u> destinations

- Or the reverse all source, single dest.
- Or, of course, a given <u>pair</u>

* a shortest path will contain other shortest paths inside of it *

Dynamic Programming - Split a program into smaller pieces

<u>NOT</u> Dynamically updated...

if weights are $\ominus$, shortest path weights <u>NOT</u> well defined

So take $\ominus$ cycles

So do that ∞ get $\delta = -\infty$

(14)

No $\ominus$ or $\oplus$ cycles
or 0

└ just no cycles. Period.

Maintain a list of <u>predecesors</u>
So can read backward $v \to s$ to get $s \to v$

So have G_π "shortest path tree"
like a BFS tree
but w/ weights

<u>Trees are not nessarily unique</u>

~~Relax~~
<u>Relaxation</u>

initalize w/ $\Theta(V)$ algo

then test if we can improve the shortest path
to v so far by going ~~through~~ through u
"<u>relax</u>"

Relax (u, v, w):

if $v.d > u.d + w(u,v)$ ← if we should update

update $\begin{cases} v.d = u.d + w(u,v) & \leftarrow \text{weight} \\ v.\pi = u & \leftarrow \text{parent} \end{cases}$

Have some properties

- triangle inequality

$$\delta(s,v) \le \delta(s,u) + w(u,v)$$

- upper bound property

We always have $v.d \ge \delta(s,v)$ for all vertices $v \in V$ and once $v.d$ achives $\delta(s,v)$ it never changes

- No path property

if no path $v.d = \delta(s,v) = \infty$

16

Convergence property

(not writing)

Path relaxation property

Predecessor subgraph property

Bellman-Ford Algorithm

Can find shortest path w/ Θ costs

returns true/false if Θ weight cycle from source

(¿whats this obsession w/ Θ cost stuff?)

if no cycle - shows shortest weight + paths

initalize $\Theta(V)$

relax each edge $|V|-1$ of $\Theta(E)$

Check for Θ cycles $\Theta(E)$

(17)

(I guess I'll see in lecture as this example does <u>not</u> make much sense ...)

Single-Source shortest path in DAGs

allways well defined
topological sort helpful to compute shortest path

make 1 pass over topo sort and relax each

$\Theta(V+E)$
↑ study why (+)

Dijkstra's Algorthm

Solves on a weighted, directed graph
where all edges $\neq \ominus$

better running time than Bellman Ford

maintains set S of (final) shortest-path weights
Selects $u \in V-S$ adds it to S
relaxes all edges leaving u

(18)

Q = priority queue

(notice counts updating in the graphic - like 6.01)

it does find the shortest path

(this section is much more wordy!)

<u>running time</u>

depends on how implement priority queue

- binary heap
- fib. heap

(end of what coveing in class..)

Reviewing Notes

3/28

Reviewing representations

4 of them in lecture

- adj list
- adj matrix

] saw in book

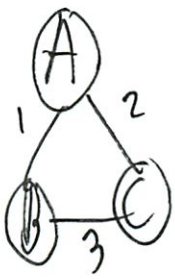
- incidence list

list edges not neighbors

↑ or object

WP: store edges that contain referencing vertex

So



A → B, C
B → A, C
C → A, B

the adj list

A → 1, 2
B → 1, 3
C → 2, 3

incidence matrix

or

A → (A,B), (B,C)
B → (B,A), (B,C)
C → (C,A), (C,B)

incidence matrix

↑ can also have cost

②

implicit representation

WP: vertices or edges are not explicit objects - but are determined algorithmicly

like for Rubix cube can easily build adj list on the fly - pre computing it all would be too much work

Notes Adj(v) can be found on the fly

Adj matrix

A^2 is # of lenght 2 paths between vertices

---

directed (i, j)

undirected $\{i, j\}$

(I so hate the no PPT lectures)

So I think I pretty much understand the lectures (even if I can't decode notes - textbook better

Not sure if can simulate DFS - but should be ready for HW

Think got now - nice chart ~~Greedy~~ start/stop

600 Lecture

4/3

(PPT is back! :))

We were discussing graphs: directed and undirected

Adj lists

a -> c -> b
b -> c
c -> b

Good for sparse graphs
both sides if undirected

BFS

Start vertex v
List its neighbors
Than the neighbors for it
O(n + m)
↑ linear - look in both directions

Creates Spanning Tree
Can have edge from 1 row to other
and in same level
but not jumping multiple levels

②

DFS is like exploring a maze
move from one to other
till have to backtrack
impatent people who don't want to see

leaves a <u>DFS tree</u>

root at bottom
visit 1st on left) 6.0[illegible] convention

are other edges besides tree edge

- back edge (b/w you and an ancestor)
- but not all edges - where you would have ~~backtrack~~ ~~on~~ traveled on it if it exists

(I need to study this more)

ie between node and non ancestor

- Gives you an ordering
  - Crucial part of it

Can have a <u>directed DFS tree</u>

w/ arrows everywhere

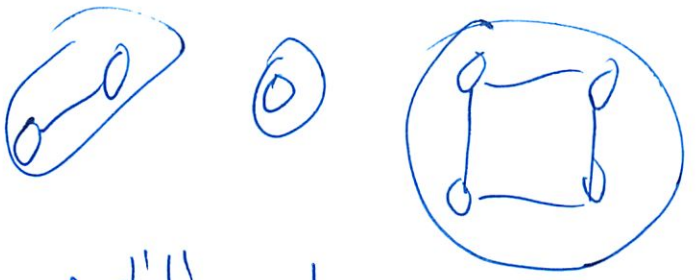
③

* You never leave a node w/o exploring all its parents

Some edges only exist in directed

- <u>forward edges</u> (not a good name)

Connected Component



partition the graph

Can do in ~~non~~linear time w/ good counting

I like making heap from scratch

Topological Sort

number the vertices on a DAG such that

if $u \to v$ then $TS(u) < TS(v)$

Topological Reverse

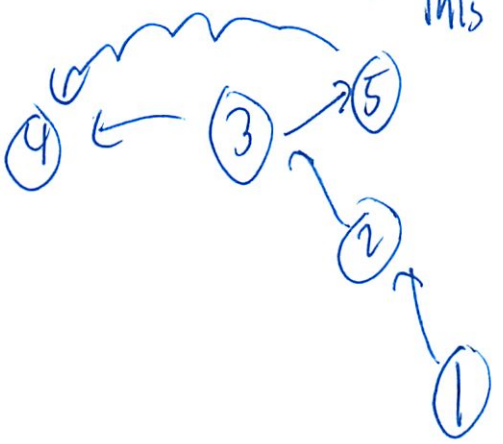
Topological un-sort

Never helped anyone - define good as opposite of bad

$$TS(x) = n - TR(x)$$

but could do for efficiency!

- its easy to find sinks w/ DFS
- this allows us to <u>recycle</u> stuff



- Remember we are in a DAG
- will be a sink at some point
- Must backtrack at some point

DFS for free gives you topol. reverse

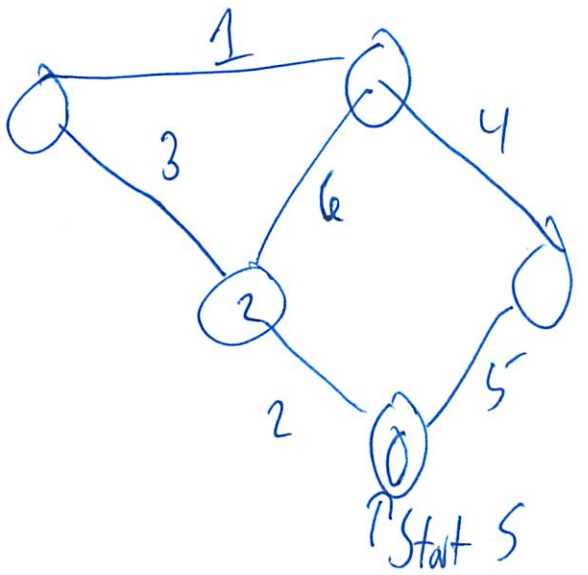
Can snap it

More General Shortest Paths for given node s

Undirected graphs w/ non neg edge lenght

$G = (V, E)$ $\quad l: E \to [\ , +\infty)$

Dijstra's Algorithm



triangular inequality

$A \to B + B \to C$ means $A \to C$
no more than

distance S to S = 0

We see 2,5 edges out of S

We ~~lable~~ take the lowest value edge and lable that node 2

(6)

Have 2 sets of vertices (lables)

T = Temporary

P = Permanant

(he forgot a line on his slide) (or some other mistake)

V = smallest labled element in T (but not S)

Take it up from T

after exploring all nodes

3
7
11
2
14

← Order (N) to scan for min

will do N times

$= O(N^2)$

$O(N+M)$ subsumed by $O(N^2)$

but we could keep a min heap

(7)

Cycles:

Don't worry us

Since edge lenght is never ⊖

Directed graph

Yes

Cycles still don't matter

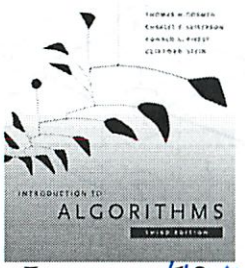
⊖ Distance/Cost:

Causes trouble

It will go around the ⊖ for ever

For next time i figure out what you should do here

6.006- *Introduction to Algorithms*



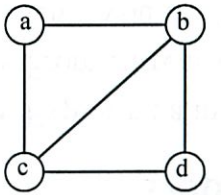
Lecture 13 14

Prof. Silvio Micali

Graphs

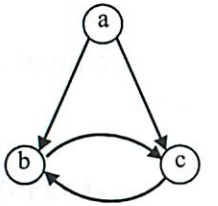
Undirected

- V={a,b,c,d}
- E={{a,b}, {a,c}, {b,c}, {b,d}, {c,d}}



Directed

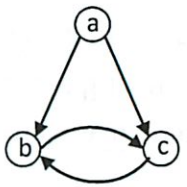
- V = {a,b,c}
- E = {(a,c), (a,b) (b,c), (c,b)}

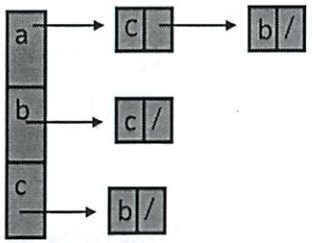


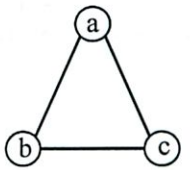
Computer Representation

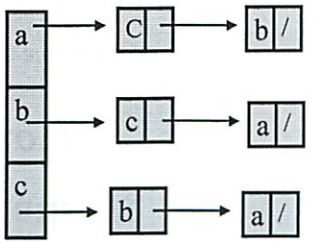
Four representations with pros/cons

Adjacency lists (of neighbors of each vertex)









Breadth First Search

- Start with vertex v
- List all its neighbors (distance 1)
- Then all their neighbors (distance 2)
- Etc.

Augmented Breadth First Search =Shortest Path Alg

$(Pseudo^2)$

Initially, s is marked 0, all other vertices are marked ∞

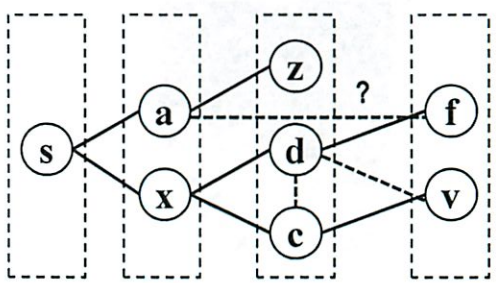
1. $i \leftarrow 0$
2. Find all neighbors of at least one vertex marked i. If none, STOP.
3. Mark all vertices found in (3) with $i + 1$.
4. $i \leftarrow i + 1$

Thm: Every vertex is marked with its distance form s

Complexity: $O(n + m)$

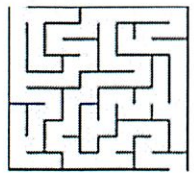
BFS Tree Structure

- Spanning Tree with Lots of Structural Information



Depth First Search

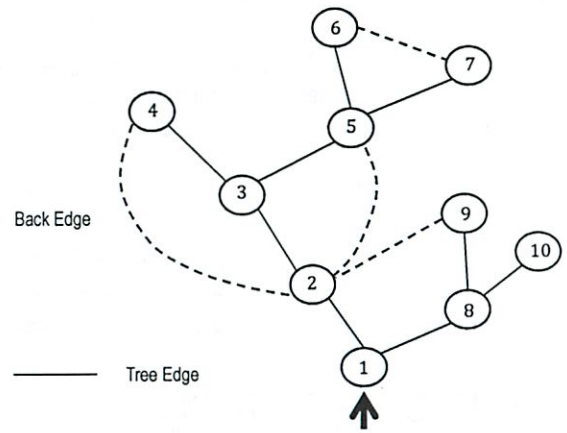
- Exploring a maze
- From current vertex, move to another
- Until you get stuck
- Then backtrack till you find the first new possibility for exploration



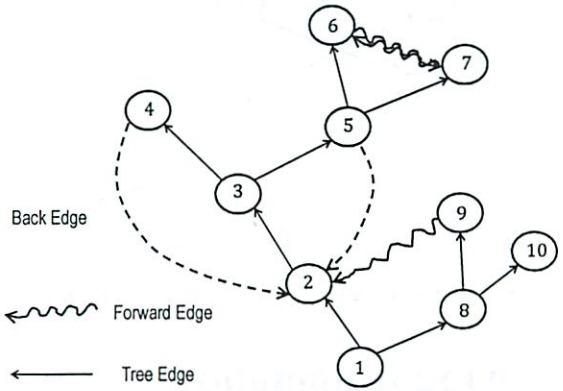
DFS

0. Mark all edges "unused". For all $v \in V, \#(v) := 0$. Let $i := 0$ and $CoA := s$.
1. $i \leftarrow i + 1$ $\#(CoA) \leftarrow i$
2. If CoA has no unused edges, go to (4)
3. Choose an unused edge $CoA \overset{e}{\leftrightarrow} u$. Mark e used. If $\#(u) \neq 0$ go to (2). Else $F(u) \leftarrow CoA$ $CoA \leftarrow u$ and go to (1)
4. If $\#(CoA) = 1$ HALT
5. $CoA \leftarrow F(CoA)$ and go to (2)

DFS Tree

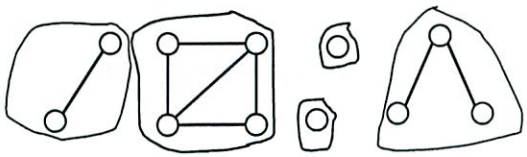


DFS Tree Directed Case



Connected Components

An equivalence relation

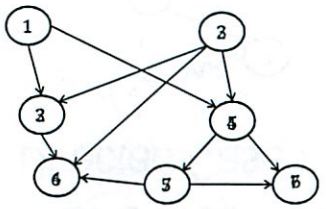


Linear with Good Counting!

Topological Sort

$\boldsymbol{TS}$: numbering of the vertices of a directed acyclic graph (DAG) such that

if $u \to v$ then $TS(u) < TS(v)$



Idea:

Topological Reverse

$\boldsymbol{TR}$: numbering of the vertices of a directed acyclic graph (DAG) such that

if $u \to v$ then $TR(u) < TR(v)$

$$TS(x) = n - TR(x)$$

More General Shortest Paths for a given node s

Undirected (Directed in recitation!) graphs with non-negative edge length

$$G = (V, E) \qquad \ell: E \to [, +\infty)$$

Picture

Dijstra's Algorithm

λ: label. If $\lambda(v) = x$, then there is a path from s to v of length x, not necessarily minimum

T: Set of temporarily labeled vertices

P: Set of permanently labeled vertices

0. $\lambda(s) \leftarrow 0 \qquad T \leftarrow \{s\} \qquad P \leftarrow \emptyset$

1. *While* $T \neq \emptyset$ do:
- Choose $v \in T$ with minimum label
- $T \leftarrow T \setminus \{v\} \qquad P \leftarrow P \cup \{v\}$
- $\forall\, v \xrightarrow{e} u$ *do*
  - *if* $u \in T$, *then* $\lambda(u) \leftarrow \min\{\lambda(u), \lambda(v) + \ell(e)\}$
  - *Else, if* $u \notin P$ *then* $\lambda(u) \leftarrow \lambda(v) + \ell(e)$ & $T \leftarrow T \cup \{u\}$

Analysis

Cycles?

Problem Set 4

This problem set is due **Wednesday, April 4** at **11:59PM**.

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **An assortment of sorts (10 points)**

   (a) Merge sort on n integers in the range $\{1, \ldots, n^3\}$ requires time $\Theta(n^c \log n)$.
   What is c?

   (b) Counting sort on n integers in the range $\{1, \ldots, n^3\}$ requires time $\Theta(n^c)$.
   What is c?

   (c) Radix sort on n integers in the range $\{1, \ldots, n^3\}$ (with optimal choice of parameters) requires time $\Theta(n^c)$.
   What is c?

2. **Median of two arrays (20 points)**

   Let X and Y be two arrays, each containing n ordered values already in sorted order. Give the most efficient algorithm you can to find the median of all $2n$ elements in arrays X and Y. Prove correctness of your algorithm and analyze its running time.

3. **Cycle testing (20 points)**

   Design and analyze an algorithm for detecting if an undirected graph has an odd cycle. (A cycle of length k is a sequence of k distinct vertices $v_1, \ldots, v_k$ such that there are edges between v_1 and v_2, between v_2 and v_3, etc, and also an edge between v_k and v_1.)

4. **BFS or DFS? (10 points)**

   For each of the following problems, answer `'B'` if the most appropriate search algorithm is BFS, or `'D'` if the most appropriate search algorithm is DFS

   (a) You are a mouse who is trapped in a maze with no cycles. You have no memory, but you know left from right. Your escape strategy is closest to which search algorithm?

   (b) You are a pirate looking for hidden treasure on an island. You are at the location marked X on the map, but the maps is slightly inaccurate, so you believe the treasure to be at a nearby location. How do you determine the order in which to search the locations on the island?

   (c) You are Google Maps. Which search algorithm do you use to get driving directions?

   (d) Which search algorithm explores a graph in a manner reminiscent to a BST in-order traversal?

   (e) Which search algorithm is good at keeping track of shortest distances from the start node?

5. **True/False (30 points)**

   (a) Let G be an undirected graph. If we have a back edge when we run DFS on G, then the graph has a cycle.

   (b) Let G be a directed graph. If we have a cross edge when we run DFS on G, then the graph has a directed cycle.

   (c) The running time of insertion sort can be reduced to $O(n \cdot \log(n))$ if we use binary search when inserting each element into its appropriate position of the array instead of traversing the array backwards.

   (d) The running time of BFS is $O(V + E)$ irrespective of the graph representation.

   (e) Let G be a connected undirected graph, let v be a vertex in G, and let D be a directed graph obtained by orienting the edges of G arbitrarily. Then it is always the case that a DFS in D starting from v will explore the entire graph.[1]

   (f) If an undirected graph has vertices v_1, v_2, and v_3 in a triangle, then when performing BFS, AT LEAST two of v_1, v_2, and v_3 must be at the same level.

   (g) If an undirected graph has vertices v_1, v_2, and v_3 in a triangle, then when performing BFS, EXACTLY two of v_1, v_2, and v_3 must be at the same level.

   (h) If an undirected graph has vertices v_1, v_2, and v_3 in a triangle, then when performing DFS, no two of them can be on the same level. (We define "level" as the length of the path taken from the source in the DFS tree.)

   (i) Suppose that in the "Awkward Sort of Party" problem from Problem Set 2, each of the n people are assigned a vertex in a directed graph G. DFS is run on the graph, and a person arrives at the party when his vertex is first explored by the search, and leaves the party when his vertex is finished processing ("colored black" in the terminology of CLRS). True or False: At the conclusion of the party, no one will become a Twitter follower of anyone else.

   (j) A strongly connected component in a directed graph G is a maximal subset of vertices such that there is a directed path from any vertex in the set to any other vertex. True or False: Let C and D be two (distinct) strongly connected components of a directed graph G, and suppose that there is a directed edge from some vertex in C to some vertex in D. Then any depth-first search will either explore no vertices in D or will finish processing all vertices in D before it finishes processing all vertices in C. (By "finish processing," we mean, in the notation of CLRS, that the node has been "colored black.")

[1] The DFS does not restart from other vertices when the first search finishes

6. **Breadth-First Search (30 points)**

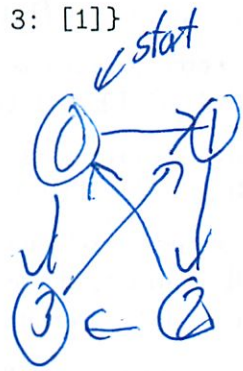
One way of representing a graph in Python is as a dictionary `edges` mapping node numbers to lists of adjacent node numbers. The vertex set of the graph is the set of keys of the dictionary, that is, `edges.keys()`. A key `k` has a directed edge outwards to each key in the list `edges[k]`. This representation is basically an implemention of the adjacency lists discussed in class.

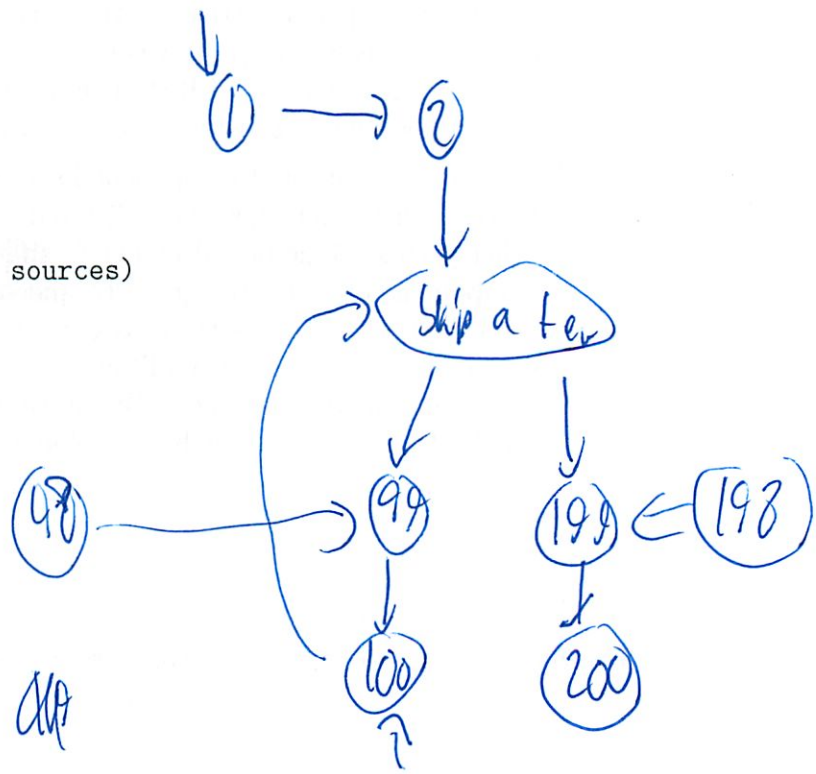
Write a function `find_distances` that takes two arguments: an dictionary, `edges`, and a list of vertices, `sources`. It should return a dictionary `dist` which records the minimum distance from ANY source to each vertex of the graph, or `None` if it is unreachable.

Your function should pass the following test cases:

```
graph = {0: [1,3], 1: [2], 2 : [0, 3],  3: [1]}
sources = [0]
dist = find_distances(graph, sources)
dist[0] == 0
dist[1] == 1
dist[2] == 2
dist[3] == 1

graph = {1: [2],
         2: ['skip a few' ],
         'skip a few' : [99, 199],
         98: [99]
         99: [100],
         100: ['skip a few']
         198: [199]
         199: [200]
         200: [] }
sources = [1, 100]
dist = find_distances(graph, sources)
dist[1] == 0
dist[2] == 1
dist['skip a few'] == 1
dist[98] == None
dist[99] == 2
dist[100] == 0
dist[198] == None
dist[199] == 2
dist[200] == 3
```





6.006 PS4

4/3

1. Assortment of Sorts

a) Merge Sort on n integer in the range $\{1, \ldots, n^3\}$ requires time $\Theta(n^c \log n)$

What is c?

Shri Always $n \log n$

So $c = 1$

b) Counting Sort $\{1, \ldots, n^3\}$ $\Theta(n^c)$

$O(k + n)$

normally $k = O(n)$

So $\Theta(n)$

So n^3

$c = 3$

(2)

c) Radix sort $\{1, \ldots, n^3\}$

w/ optimal choice of parameters

$\Theta(n^c)$

$b = O(\log n)$

$r = \lfloor \lg n \rfloor$ ← what is this again

$\Theta(n)$ r is just one value

Or each pass $\Theta(n+k)$

So $\Theta(d(n+k))$

d = constant

$k = O(n)$

~ $\lceil \log N \rceil$ is # of digits something has

k - can be n^3 possible values

(3)

Oh haha problem 1.3 - Y

$C = 1$

---

2. Median of two arrays

X, Y each w/ n sorted

What to find median in X, Y

Shit avg of X, Y

let me think [1, 2, 3]

[4, 5, 6]

$\frac{2+5}{2} = 3.5$ ✗

or find nth element ✗ works here

[1 2 3]

[1 2 3]

but how combine

- can't just simply append

(4)

$$\begin{bmatrix} 1 & 2 & 4 & 7 & 9 & 11 \\ 2 & 5 & 18 & 21 & 24 & 25 \end{bmatrix}$$

So median is

$$[1\ 2\ 2\ 4\ 5\ 7\ \underset{\uparrow}{9}\ 11\ 18\ 21\ 24\ 25]$$

avg is $\frac{5.5 + 19.5}{2} = 12.5$ (2)

does not work out

; Something about sorting

---

1. Find middle of first
2. Find the next one larger in that in 2nd list
3. See what offset that is
4. How much it differs from $n/2$
5. Advance half of that (half of $n/2$) on both
6. Compare again - which is ~~larger~~ smaller

(5)

7. Repeat

looking for the next largest in the other row
Check offsets

8. If we go over, backtrack

9. Do until offset ≥ n total

---

3. Cycle Testing

Odd cycle: just a cycle w/ an odd # of nodes or edges

Median Q is Google able

Geeks for Geeks

3 methods
- count while merge sort
- compare medians
- binary search for median

Cal the medians

if $m_1 > m_2$ than

[////]

[////]

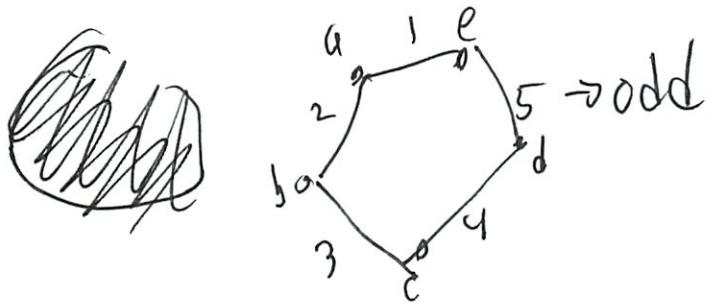
(6)

Or other way around

Repeat till find

Ignoring some special cases

Odd Cycle



? First need to find if cycle?

When it comes back in visited path

[a b c d e a]

5 4 3 2 1 then cant

↑ odd

Doing DFS

(is that just silly?)

(This class got a lot harder)

0

This is the nieve way

Hash table
└ or in memory table of each node

As well as linked list of items

Check O(1)
Then O(difference) can be max $n = O(n)$

---

4. BFS or DFS

~~a) Mose~~

a) Mouse trapped in maze w/ no cycles
DFS

b) Pirate looking for treasure
BFS - since nearby

c) Google Maps driving directions
~~DFS~~ w/ costs
BFS - incrementing distance

(8)

d) Search algorithm → BST in order traversal

DFS

e) Shortest distance from start node

BFS

---

5. True/False

a) 6 undirected cycle

---

3. What about DFS itself?

---

5. a) Undirected graph w/ back edge has a cycle

└ True

b) directed cross edge

└ False Can have cross edge from 1 pt to other

c) Insertion sort $O(n \log n)$ if use binary search

Bounds true - but couldn't prove

9

d) BFS is $O(V+E)$

True, in the slides

On wp

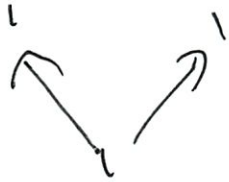
e) Orience arbitrally

? could create ~~arbitrarily~~ unconnected

? if start here, wont

False

f)



So true?

? but if already saw a node ?

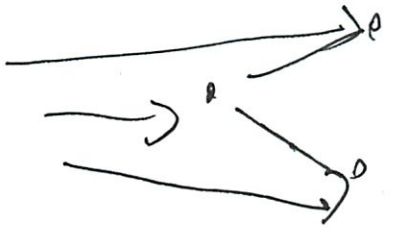
└ then not in the tree

Or would have picked up next time

(10)

but same level

- Yes I think that could work



g) False

h) DFS

No - theres a node

must come since triangle

True

i) (I am having the hardest time understanding this!!!)

So want people who were explored before they were but finished before they wer

- can't do w/ DFS - ~~false~~ true

(11)

That took me forever to decide

j) Maximal subset - ind set that is not a subset of any other ind set

Strongly Connected → path from every vertex to

(Oh they were defining)

Only from C → D

So true

(I'm becoming better at search)

G. BFS Any Source to Each Vertex

---

3. On Google

Color black and white

G. Someone mentioned hashing

need to do it in very little time

(12)

Use distance from one for the other
└ Dijstra

But heard its not fast enough on its own

Hash sources to see if list of sources
Run BFS
If something is in table - look up and use that

Go through each node
Start w/ a source
Go to next in list
if already calc - recall what call an add to it

* Given a smaller list of sources
Are only 2 test cases
But very strict in time

(13)

BFS on all at same time

look at 1st level neighbors for each, then move on

hash table that stores distances, is Visited

? really just need this

Want list of paths

Precomputed results

initialize dict w/ None

(Have same BFS from (0,0)?)

Yied = return next

After visit - take it out

Min distance from any source

I did wrong BFS

- need to store count

(so tired!)

(19)

(✓) Pass small test case 1!

Did something else

~~Need Nov~~ only save it <

Need Novo?

(✗) Yes you need

And need to check the table

(✓) Pass small test cases

But way too slow

No printing

(✗) Still limit exceeded

Need to check table

~~Ge~~ Ge built it all

(✓) works

(15)

Try to streamline

Has no table of things to look up

just goes to each source

Oh it was never correct!

(X) Limit exceeded

└ it can't be!

It was actually not right

- Should have made it right 1st

Go slow + figure out

My count is wrong

Thats why he had extra layer

Ahh I figured it out - sources not source!

Guess I didn't understand it before!

(16)

(✓) Pass all test cases/ full credit!

DS4 Review w/ Shri

4/4

Disagree on 5e, g, j

↳ He now agrees

He thinks C is False

I'm not sure on this

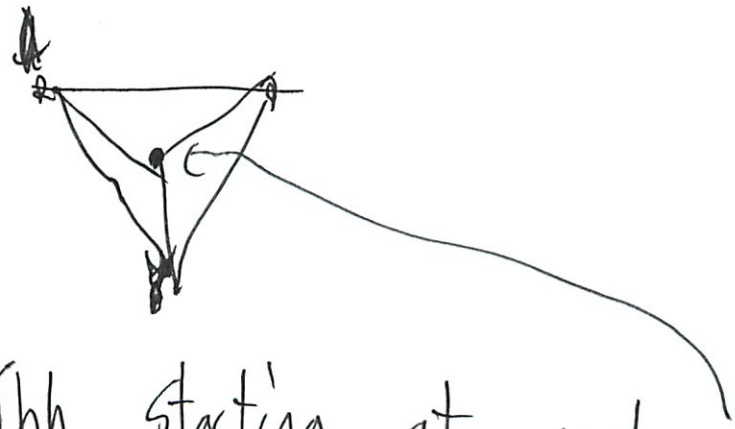
When insert must move all elements over

which costs n

So is n^2 ~~False~~ I agree - False

j) False

Since



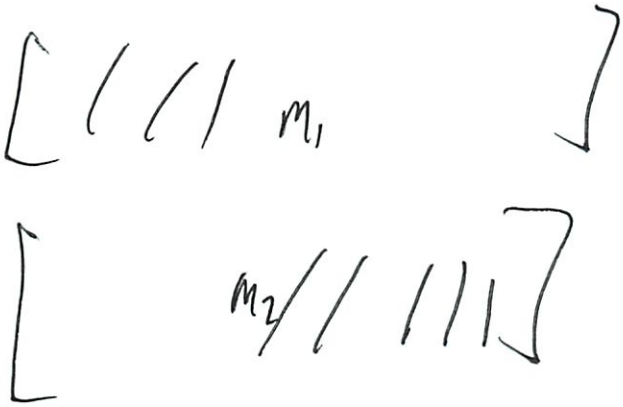
Ohh starting at center

False ← same as what we have

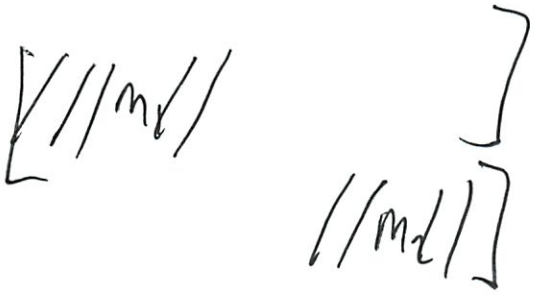
but I forget why we had said that...

Fix up

Haha "This algorithm is clearly correct"

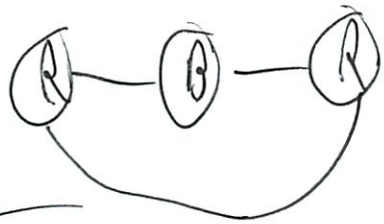


$m_1 > m_2$



$m_1 > m_2$

Red black



= odd

(taking forever - why?)

very worried

```
import random
from collections import deque
collaborators = 'Shri, Arianna, Crystal, Web resources'

# Enter some numerical value for each part of problem 1.
answer_for_problem_1_part_a = 1
answer_for_problem_1_part_b = 3
answer_for_problem_1_part_c = 1

# Enter your answer to problem 2 here.
answer_for_problem_2 = '''
First get the medians of the two sorted arrays a1 and a2 and then compare them.
1) Calculate the medians m1 and m2 of each array.
2) If m1 and m2 both are equal then we are done, return m1 (or m2)
3) If m1 is greater than m2, then median is present in one
   of the below two subarrays.
    a)  From first element of a1 to m1 (a1[0...|_n/2_|])
    b)  From m2 to last element of a2  (a2[|_n/2_|...n-1])
4) If m2 is greater than m1, then median is present in one
   of the below two subarrays.
   a)  From m1 to last element of a1  (a1[|_n/2_|...n-1])
   b)  From first element of a2 to m2 (a2[0...|_n/2_|])
5) Repeat the above process until size of both the subarrays
   becomes 2.

This is correct because we always go towards the array with the larger median.  We hill
climb up, with the range getting smaller and smaller, until we are left with the median.

This is hillclimbing O(logn).

O(log n) is pretty good, and you can't do better than that.  (Not very rigerous)

'''

# Enter your answer to problem 3 here.
answer_for_problem_3 = '''
Create a table with every node O(n)
Start DFS at an arbitrary start node.
Wach time you progress, you check the table if you have already been there O(1)
and append it to a visited list O(1)
Color nodes alternatingly red-black to record odd/even.  Record the color in the table as
well.
When you come across a node that you have already visited AND is the same color as the node
that you are currently on, the graph has a cycle!

Redo on each connected component.  (Still O(n) since still n nodes either n connected
components of 1 or 1 connected component of n)

So O(n)

This is correct because the colors store odd/even.  The table stores where we have already
visited, so we can eaisly O(1) check where we have been before.  The list of visited places
```

```
lets us return the nodes.

'''

# Your answer to each part should be the character 'B', or the character 'D'
answer_for_problem_4_part_a = 'D'
answer_for_problem_4_part_b = 'B'
answer_for_problem_4_part_c = 'B'
answer_for_problem_4_part_d = 'D'
answer_for_problem_4_part_e = 'B'

# Your answer to each part should be a boolean.
answer_for_problem_5_part_a = True
answer_for_problem_5_part_b = False
answer_for_problem_5_part_c = False
answer_for_problem_5_part_d = True
answer_for_problem_5_part_e = False
answer_for_problem_5_part_f = True
answer_for_problem_5_part_g = False
answer_for_problem_5_part_h = True
answer_for_problem_5_part_i = True
answer_for_problem_5_part_j = True

# Fill in the function here, for problem 6
def find_distances(graph, sources):

    #pre-generate Nones for all
    distance = {}
    for node in graph:
        distance[node] = None

    #the distance from all sources is 0
    for source in sources:
        distance[source] = 0


    count = 1
    while sources:
        nextsources = []
        for source in sources:
            #print "exploring node "+str(source)
            neighbors = graph[source] #get neighbors
            for neighbor in neighbors: # process each neighbor
                    if distance[neighbor] == None:
                        #print "neighbor "+str(neighbor)+" with count "+str(count)
                        distance[neighbor] = count
                        nextsources.append(neighbor) #add it to the queue
        count = count +1
        #print "start of count "+str(count)
        #print nextsources
        sources = nextsources
```

```
    return distance

#small test cases
graph = {0: [1,3], 1: [2], 2: [0, 3], 3:[1]}
sources = [0]
print find_distances(graph, sources)


graph = {1: [2], 2: ['skip a few' ], 'skip a few' : [99, 199], 98: [99], 99: [100], 100: [
'skip a few'], 198: [199], 199: [200], 200: [] }
sources = [1, 100]
print find_distances(graph, sources)
```

```
import random
collaborators = 'Your collaborators here'

# Enter some numerical value for each part of problem 1.

answer_for_problem_1_part_a = 1 # Merge sort is always Theta(n log n)
answer_for_problem_1_part_b = 3 # There are Theta(n^3) buckets
answer_for_problem_1_part_c = 1 # Use counting sort 3 times to reduce it to Theta(n) time

# Enter your answer to problem 2 here.
answer_for_problem_2 = '''

DESCRIPTION

Suppose m is the median value.

Then if there are exactly i elements less than m in X, there must be exactly n-i elements
less than m in Y.
So X[i-1] <= m <= X[i] and Y[n-i-1] <= m <= Y[n-i]

Since X and Y are sorted, we have that:
  - If k > i, then X[k] >= m >= Y[n-k-1]
  - If k < i, then Y[n-k] >= m >= X[k-1]
Thus it is easy to check if k is too high or low, by comparing X[k-1], X[k], Y[n-k-1],
and Y[k].

Thus we can binary search to find i, which then lets us recover the median.

PSEUDOCODE (OPTIONAL)

start, end = 0, n - 1
while True:
  k = (start + end) / 2
  if X[k] > Y[n-k-1]:
    end = k
    continue
  elif Y[n-k] > X[k-1]:
    start = k
    continue
  else:
    return (max(X[k-1], Y[n-k-1]) + min(X[k], Y[n-k])) / 2

ANALYSIS

Correctness:  Follows from the discussion above

Runtime:  Is the same as the runtime of binary search, Theta(log n).
We cut down the interval of possible i values by half each time,
doing constant work each iteration.

'''
```

```
# Enter your answer to problem 3 here.
answer_for_problem_3 = '''

DESCRIPTION

We do a modified BFS on each connected component.  The modification is simply that when we
explore a non-tree edge, we check that the parity of the distances to the endpoints is
different.
If not, we immediate return YES.  Otherwise, if all the BFSes complete, return NO.

ANALYSIS

Correctness:

Suppose there is an odd cycle.  Then, there are two adjacent vertices with equal parity
(otherwise, all adjacent vertices have different parity, but if the cycle has length
(2k+1),
we see that v_1 has the same parity as v_3, v_5, ... v_(2k+1).  This is a contradiction,
since v_1 connects to v_(2k+1)).  Since a BFS looks at all edges, we will find this edge
between two vertices of equal parity, and return YES, as desired.

Suppose we return YES.  So we found an edge e = (v, w) between vertices with equal parity
of distance from the source s. The distances must be equal, since the edge e means the
distances differ by at most one.  Consider a shortest path from s to v, and one from s to
w.
Neither path contains e = (v, w), since the paths are SHORTEST paths of equal length.
Let t be the furthest (away from s) vertex at which the paths intersect (possibly equal
to s).
Then the path from t to v, and from t to w, are disjoint, and are the same length (since we
removed identical sub-paths).  So combining the two gives an even length path from v to w.
And since it doesn't use e = (v, w), adding the edge e shows that the graph had an odd
cycle.

Thus we return YES if and only if there is an odd cycle.

Runtime:

The runtime is simply that of BFS, O(E+V), since our check takes constant time.

'''

# Your answer to each part should be the character 'B', or the character 'D'
answer_for_problem_4_part_a = 'D'
answer_for_problem_4_part_b = 'B'
answer_for_problem_4_part_c = 'B'
answer_for_problem_4_part_d = 'D'
answer_for_problem_4_part_e = 'B'

# Your answer to each part should be a boolean.
answer_for_problem_5_part_a = True
answer_for_problem_5_part_b = False
answer_for_problem_5_part_c = False
```

```
answer_for_problem_5_part_d = False
answer_for_problem_5_part_e = False
answer_for_problem_5_part_f = True
answer_for_problem_5_part_g = False
answer_for_problem_5_part_h = True
answer_for_problem_5_part_i = True
answer_for_problem_5_part_j = True

def find_distances(graph, sources):
  distances = {node: None for node in graph}
  visited = set(sources)
  i = 0
  while sources:
    newsources = []
    for v in sources:
      distances[v] = i
      for w in graph[v]:
        if w not in visited:
          visited.add(w)
          newsources.append(w)
    i += 1
    sources = newsources
  return distances
```

6.006 Recitation

4/4

OH for test before lecture tomorrow

Yesterday Dijstra

Today: Review that

BFS provides us a guarentee of finding shortest path
- pushed on queue in shortest distance from origin
- only works for = cost paths

Dijkestra' will do the same w/ weighted path

The smallest weighted path, is the shortest distance
if have a list of "estimated distances"

So keep graph
At each step, turn est. dist. into actual dist

②

Estimated distances hash table

We will build

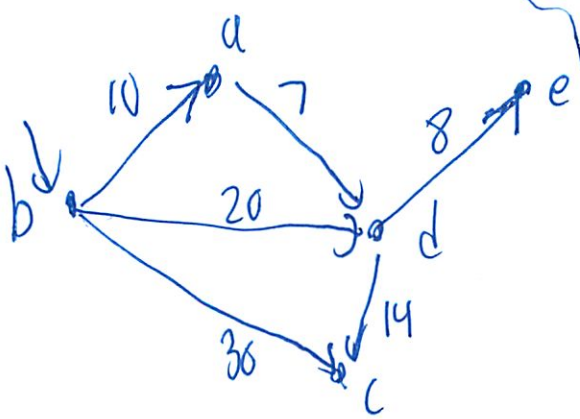
a	∞
b	0
c	∞
d	∞
e	∞

← upper bound (always ≥ actual)

Actual distance

a	
b	
c	
d	
e	

Graph



Algorithm

1. Initialize est distances table to
   start = 0
   everything else = ∞

   Can have multiple starts
   ↳ just much harder than w/ BFS

2. a. Find entry v: d_v in est_dist w/ minimal dist
   ↑ smallest row in est table

   b. Delete from est table

   c. Add d_v to actual

   d. For every edge (v,w), neighbor if $d_v + d(v,w) < d_w$, set d_w to $d_v + d(v,w)$
   ↑ found shorter than current est
   update est

3. Repeat 2

③

So example

1. Remove b from est table
2. Add b, 0 to actual
3. Go through all of b's neighbors
   Update estimated table

a	10
c	30
d	20
e	∞

4. Repeat. Take a into actual table
5. Update a's neighbors
   For d 17 < 20, so update

d	17

6. D is next smallest est
   Take into perm table

(4)

7. Look at its neig

c	30
e	25

← don't update 31 < 30 (No!)

8. Etc

9. Final table

b	0
a	18
d	17
e	25
c	30

We could also track edges traversed to find shortest path

- track in estatmated
  - ~~set~~ have a col of w's parent

Then track backwards when add to actual dist table

(5) $O(V^2)$

What data structure to make this fast?

<u>find min</u>

need to do this V times

don't want it to be linear

<u>min-heap instead</u>

V times $\log V$ to find

but need to update

$\log V$ time

must do E times

So $O(V \log V + E \log V)$

which is

$O(E \log V)$

can actually be worse

but good for normal queues

<u>Fibinichi queue</u> special heap for this

$O(\log n)$ to extract, $O(1)$ to update

don't need to know

(6)

People don't care about since
many algo don't have E
in pratice

Building Data Structures

For P-set

Want LIFO

Today: write a deque
pronounced "deck"

lets you push/pop on both sides in O(1)

Class My Deque (object):

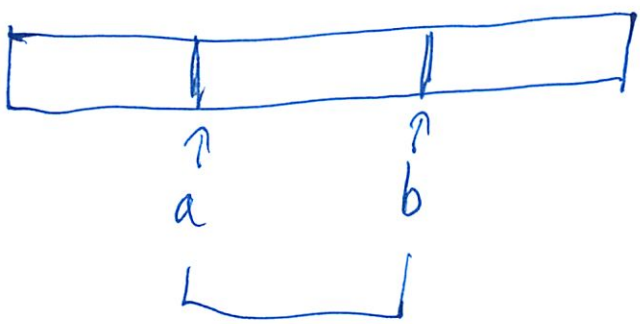
Could do a doubly link list
but a lot of overhead
- lots of pointers all over memory

7

So use a list
└ which uses ammoritized runtime

Extend off both ends
Expand it when it gets too small

Deque



actually compares
what we care about

push right = move b →

push left = move a ←

But what about if when we run out:
double the size of the list

(8)

When resize - can decide how many elements on deque
So if always moving $l \to r$
put less ~~[illegible]~~ free space on the l

```
__init__(self):
    self.data = []
    self.a = 0
    self.b = 0
```

Well we'd rather start it w/ some values

```
self.data = 3*[0]
self.a = 1          ≤ a
self.b = 1          > b
```

Must always make sure we have space

9

```
def push_left(self, val):
    self.a -= 1
    self.data[self.a] = val
    if self.a == 0:
        self._rebalance()

def _rebalance(self):
    size = self.b - self.a
    new_data = 3 * size * [0]
    new_data[size : 2 * size] = self.data[self.a : self.b]
    self.data = new_data
```

(we can use the whole size since its still O(1) until it fills-up)

```
    self.a = size
    self.b = 2 * size
```

(10)

```
push_right (self, val):
    self.data[self.b] = val
    self.b += 1
    if self.b == len(self.data):
        self._rebalance
```

This assumes both sides will be freq used ← general case
If have more data, could do a more specific job

```
pop_left(self)
    self.a += 1
    return self.data[self.a - 1]
```

a is ~~[illegible]~~ at 1st el
b is 1 after the last el

(11)

```
pop_right(self):
    self.b -= 1
    return ...
```

* Update the pops to not return anything if the list is empty

6.006 L4

4/5

Shortest Path in Graphs w/ Θ edges

$\ell(e)$ edge e

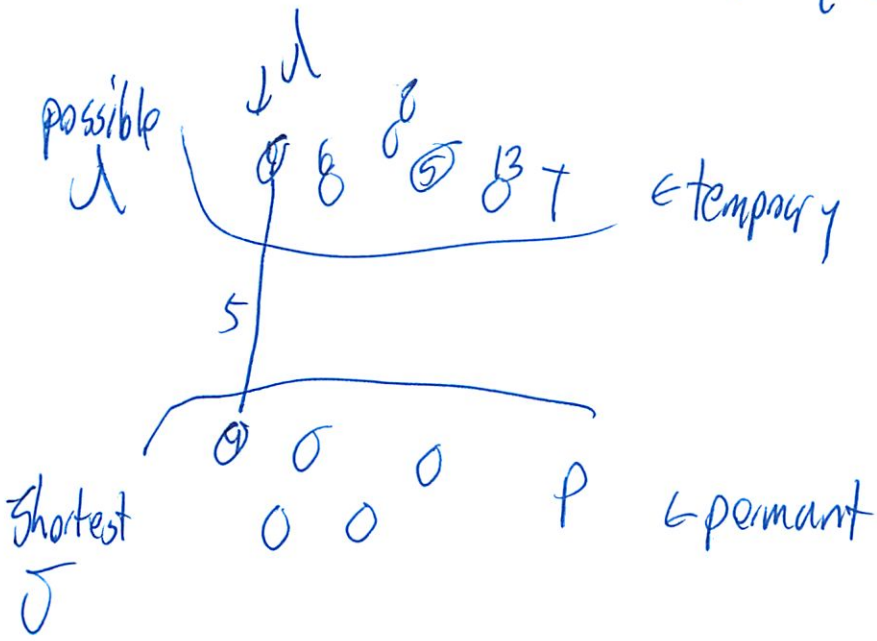
$\ell(p)$ path p

$d(v)$ You can get from s to v in $d(v)$ length
↳ might not be shortest

$\delta(v)$ distance/len of shortest path

Recall Dijstra

Distances from s where $\ell : E \to [0, \infty]$



Then ~~[illegible]~~ grab a T into P

(2)

Explore from here

This works for non neg

$O(n+m)$

look at each twice

But today $l_i \in (-\infty, \infty)$

in digraph

makes sense when G has no neg cycles!

Add

$\pi(v)$ predecessor of v on the best path so far

initially $\pi(s) = s$

$\pi(v) =$

missed

(3)

Generic Start

$d[s] \leftarrow 0$
$\pi[s] \leftarrow s$

for each $v \in V - \{s\}$
do $d[v] \leftarrow \infty$
$\pi[v] \leftarrow nil$

initialization

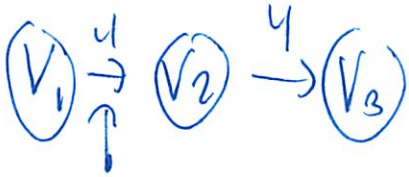
While there is an edge $(u,v) \in E$
$d[v] > d[u] + \ell(u,v)$
do select arbitrarily one such edge
Set $d[v] \leftarrow d[u] + \ell(u,v)$
$\pi[v] \leftarrow u$
end while

relaxation/improvement

<u>but</u> won't stop if neg cycle

(4)

So assume no neg cycles



Remember can pick any arbitrary one
(see slide animation)

<u>Analysis</u> # of relaxations / improvements

$T(n)$; n = # of vertices

$$T(n) = 3 + 2T(n-2)$$

look at 3 vertices

do work in box twice
Once relaxing
2nd come from path following

$$T(n) = \Theta(2^{\frac{n}{2}})$$

(5)

But exponential is bad!

Combinatorics
- find object w/ given property

Would prefer polynomial time

So instead be careful how you relax
So find sol faster
Need new algorithm

<u>Bellman Ford</u>

arbitrally fix an ordering of the edges $e_1, \ldots, e_m$
↳ not "random"

0. $\lambda(s) \leftarrow 0$
1. Until no improvements found

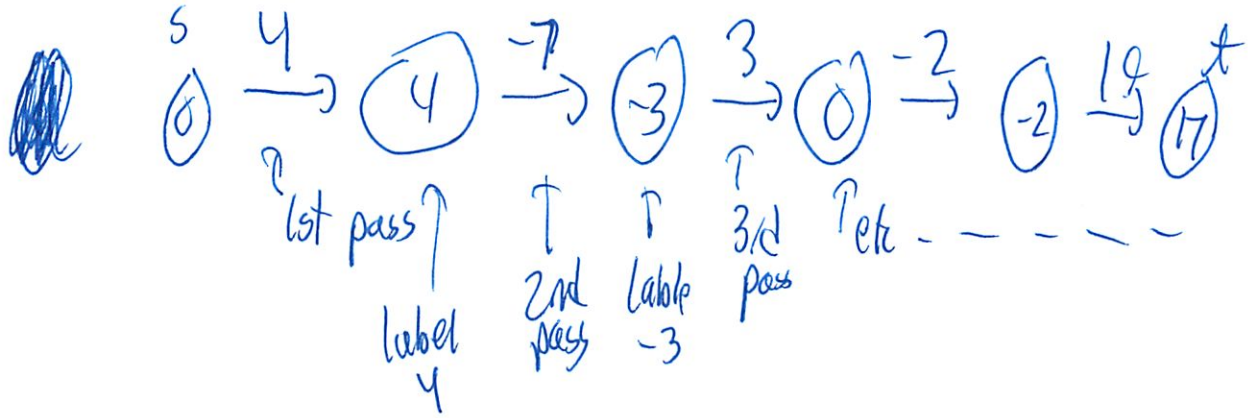
pass
For $i = 1$ to m do
If $u \xrightarrow{e_i} v$ is such that $\lambda(v) > \lambda(u) + \ell(e_i)$
Then $\lambda(v) \leftarrow \lambda(u) + \ell(e_i)$

6

Cost of 1 pass $O(m)$

How many passes? n

$n < m$ if graph is connected



Ford's total complexity $O(n\,m)$

n times, each pass

Any sub path of shortest path should still be shortest

"Spectacular algorithm"

- simple
- very local improvements = global min

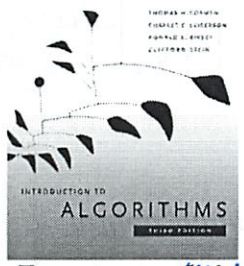
7

If $G(V, E)$ has cycles?
it goes forever
$-\infty$

Big ideas
Optimal substruction
Triangle inequality
Combinatorics vs Combinatorial optimizations

(Confused - need to read book)

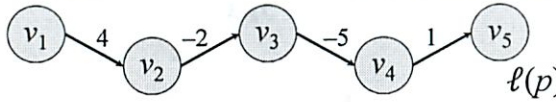
6.006- *Introduction to Algorithms*



Lecture ~~14~~ 15

Prof. Silvio Micali

Shortest Paths in a Graph



$\ell(p) = -2$

- $\ell(e)$ length of edge e
- $\ell(p)$ length of path p

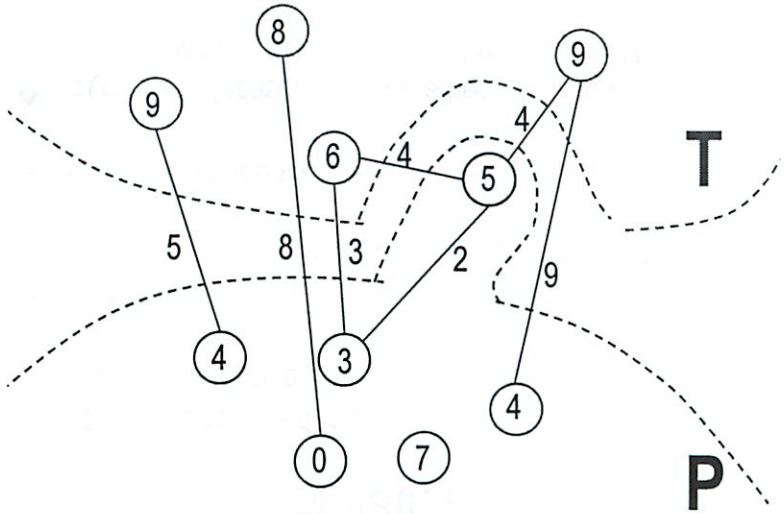
Given a "source" s:

- $\lambda(v)$ $\exists$ path p from s to v with $\ell(p) = \lambda(v)$
  You can get from s to v in $\lambda(v)$ *length*

- $\delta(v)$ distance from s to v
  Length of **a** shortest path from s to v

Recall: Dijstra's Algorithm

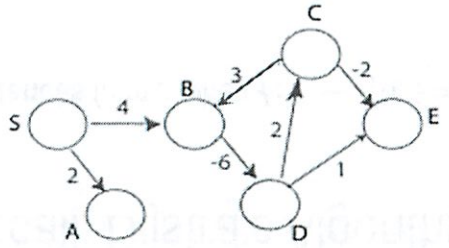
Distances from s when $\ell: E \rightarrow [0,+\infty]$



Today: Shortest Paths (from s) in Digraphs with General Edge Length

$$\ell: E \to (-\infty, +\infty)$$

Makes sense when G has no negative cycles!



Notation

- $\ell(e)$ length of edge e
- $\ell(p)$ length of path p

Given a "source" s:

- $\lambda(v)$ $\exists$ path p from s to v with $\ell(p) = \lambda(v)$
- $\delta(v)$ distance from s to v
- $\pi(v)$ predecessor of v on a best path so far (initially, $\pi(s) = s$, and $\pi(v) = NIL$ $v \neq s$

A generic start

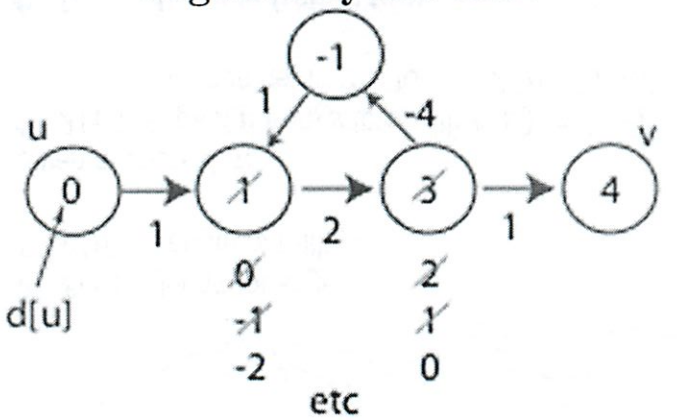
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ s
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ nil

} ***initialization***

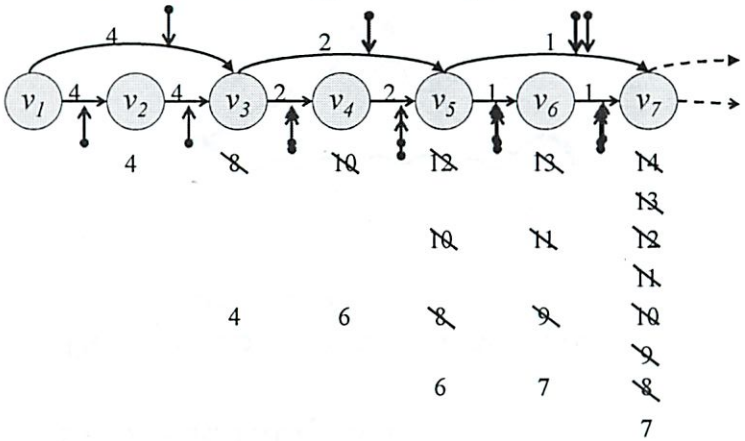
while there is an edge $(u, v) \in E$ s. t.
  $d[v] > d[u] + \ell(u, v)$ **do**
  select arbitrarily one such edge
  set $d[v] \leftarrow d[u] + \ell(u, v)$
    $\pi[v] \leftarrow$ u
endwhile

} ***Relaxation (Improvement) Step***

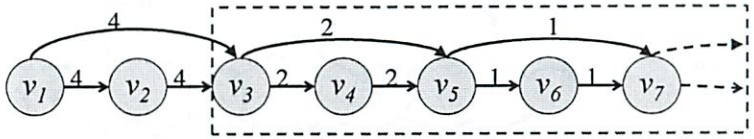
Of course, it will not stop when negative cycles exist



What if no negative cycle ....



What if no negative cycle ....



Analysis = # of relaxations

$T(n)$? $\qquad n$ = number of vertices

$T(n) = 3 + 2T(n-2) \qquad \Rightarrow T(n) = \Theta(2^{\frac{n}{2}})$

Need to be **careful** how you relax!

HOW? (Bellman Ford)

- Arbitrarily fix an ordering of the edges: $e_1, \ldots, e_m$

0. $\lambda(s) \leftarrow 0 \qquad \forall x \neq s: \ \lambda(x) = +\infty$

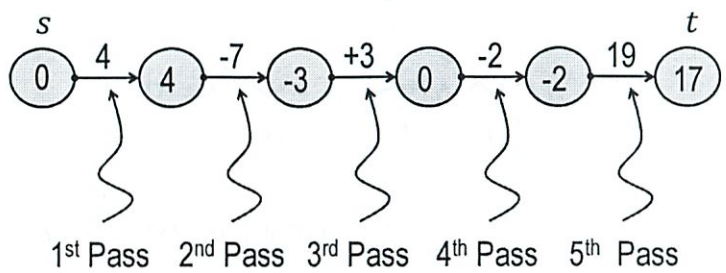
1. Until no improvement found do:

PASS$\stackrel{\text{def}}{=}$

$For\ i = 1\ to\ m\ do$:
IF $u \xrightarrow{e_i} v$ is such that $\lambda(v) > \lambda(u) + \ell(e_i)$
Then $\lambda(v) \leftarrow \lambda(u) + \ell(e_i)$

Cost of one PASS = $O(m)$ How many PASSES ?

Let this be shortest path from s to t



Ford's Total Complexity = $O(nm)$

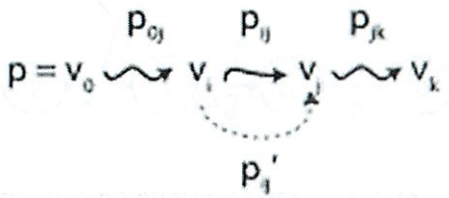
And if G=(V,E) had cycles?

Take Homes

Optimal substructure

Theorem. A subpath of a shortest path is a shortest path.

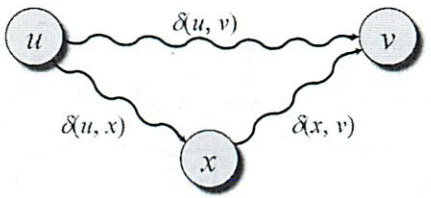
Proof. By contradiction ...



Triangle inequality

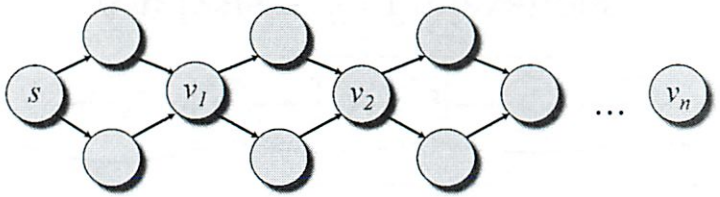
Theorem. For all $u, v, x \in V$, we have
$$\delta(u, v) \le \delta(u, x) + \delta(x, v).$$

Proof.



□

Combinatorics vs. Combinatorial Optimization



"See you laters alligaters"

6.006 Recitation

4/6

1. Given a directed G w/ (maybe neg) edge weights. Find the shortest path comprised of
   a) 2 edges
   b) 3 edges

2. Given a DAG w/ (maybe neg) edge weights and two vertices s and t
   Find the longest path from s to t

1a. So from any start
   Nieve is start from every node
   Wait - shortest path w/ 2 edges is 2?
   - have weights

(2)

I'm not really familar w/ the primities of weighted paths

Answer

Linear is possible

Ways to think

1. Find Nieve Sol

$\Theta(E^2)$ every pair of edges

Gets you 10% of the grade

Look at each node - look at all in and out ~~edge~~

- better E^2

Greedy won't get you the ans

- almost always for shortest path

(3)

$\Theta(V^3)$ - every triple of vertices

- must find lenght in constant time
- can't do w/ adj list
  - would be V^4

$$V \longrightarrow W \longrightarrow X$$

- replace w/ adj hash tables or adj matrix

V: [(a, 10), (b, 100), (w, 5)]

hash table pairs

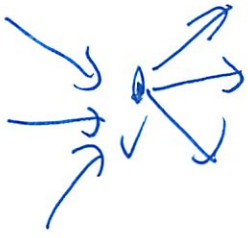
a → 10
b → 100
w → 5

1 table per node

(4)

Another brute force ✓

(~~saw~~ wrote earlier)



each pair

no more than E^2

but better than since every pair of edges is connected

Worst case - still E^2 ?
- all one node
- no

but every edge can only be paired w/ V other edges going out

So if E edges going in, V going out

So $O(EV)$

(5)

$= V \cdot \text{indegree } v$

$= V \cdot \sum \text{indegree}_v$

$= VE$

But we can still do better

have a bunch of edges going in →
just pick shortest

For each node
find smallest in + smallest out

But how iterate through node in?
- need reverse adj
- must build it in our time

H/W is same

(6)

This takes $V + E$

Didn't use any previous ideas at all

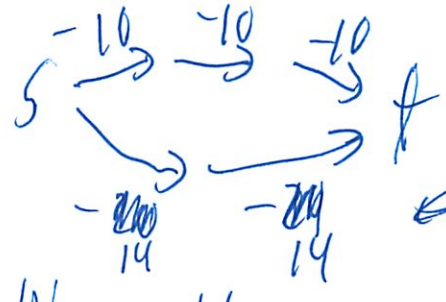
Perhaps use them in b)

- That algo we just did would be useful

- Dijstra - no - only non neg
  - but can turn edges non neg
  - add a constant to all the edges

- BFS
- Bellman Fords
  - good since Θ

* (Since paths are a fixed lenght



← ~~bottom~~ top is best

~~← bottom is best~~

Now add 14 to each (now bottom best)

Since we added more to <u>longer</u> paths

So can't combine

7

Nieve for b

E^3
V^4) counterpats to old ones

Now similar thing to last

out of time

Linear

$\longrightarrow$

before Consitered center of vertex

now consider edge
best edge in
and best edge out
iterate over edges

(you need to see the tricks...)

8

If more displined about how think of this
would not do

Would do Bellman Ford to 3rd cand

6.006 L16

4/10

Shortest Path 3

Today
- Bellman Ford on a DAG
- Dijkstra for non-neg

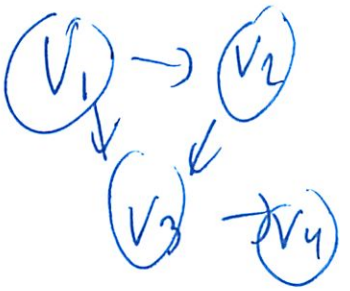
Bellman Ford DAG

Can't use dijstra (didn't hear why -neg?)

Relax edges till can't relax anymore

1st: Topological support
- relabled labels

Bellman Ford



Fix the order

- lexigraphically by head of edges - do all the ones leaving from that edge
- so $O(V+E)$

(2)

Then relax those edges in order

But can we do this faster

That last step is <u>not</u> needed
since no cycles

Only 1 iteration of relaxation

Must show correct after 1 (not V) iteration

O(n) initialization
O(m) relaxation) O(n+m)

Sorting in order
- as topologically sort <u>thats</u> when get
lexographic order of edges

Can't hope to do better
other things are also O(n+m)

(3)

So why?

No neg cycles

topo. sort implies a linear ordering of vertices

So every ~~edge~~ path sorted in same way

So can only relax each edge right

Proof of correctness

t = arbitrary vertex, $d[t]$

$s = s_0, s_1, s_2, \ldots, s_k$ shortest path^s to t

Induction → compute $d[s_i]$ correctly

$d[s_{i-1}]$ is correctly computed by our hyp.

$[s_{i-1}, s_i]$ relaxed AFTER $d[s_{i-1}]$ computed

[no more edges that go in

or would have processed them earlier

(4)

Review of Dijkstra

Proof of correctness

Edge weights non neg

Compute $\delta(s,v)$

Can be cyclic

Basically: Greedy iterative approach

List S whose shortest distance is known

Add estimates to all its neighbors

(wasn't there a temp and perm list?)

Pseudocode on slide

Initialization: Set values to default

Extract min from Q

Add to S

Check if <, then update d[v]

(yes temp → Q, perm is S)

(5) (This lecture much better - back to simple how-tos)

Example

(I understand this better than before)
(They should have queue info on slides...)

If the same i unchanged preprocessor

Then get shortest path tree

Correctness

initialization establishes $d[v] \geq \delta(s,v)$

This is maintained over relaxation steps
- upper bound
- holds before → holds after

When terminates

$d[v] = \delta(s,v)$

it is δ ~~exactly~~ at the time when its added to the set

but then we never change the d value

(6)

Proof by contradiction

Showed previously $d \geq \delta$

Case 1: $y = v$

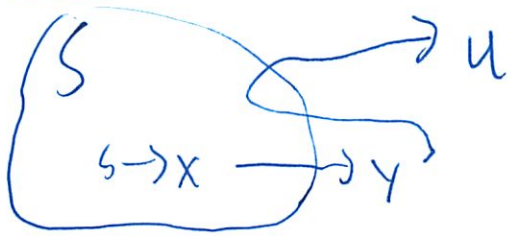
Since assume v is 1st thing that violates property

but when add a vertex in, you relax all of its edges

$d[v] = \delta(s, u)$

Contradiction: $d[v]$ is never red

Case 2 $y \neq u$



Since v is 1st vertex violates invariant

$d[x] = \delta(s, x)$

Subpath of shortest path is itself a shortest path

$\delta(s, x) + w(x, y) = \delta(s, y)$

7

So $d[y]$ is computed correctly

Contridiction

~~shortest path~~ $d[y] = \delta(s,y) \le \delta(s,u) < d[v]$

Analysis

$|V|$ times to do extract min

and then degree(v) neighbors

the decrease-key on the min priority queue

$$\text{Time} = \Theta(n) \cdot T_{\text{extract min}} + \Theta(m)\, T_{\text{Decrease key}}$$

at most once per edge

But we can have diff queues

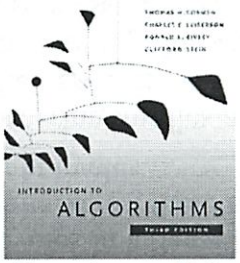
Q	$T_{\text{extract min}}$	$T_{\text{decrease key}}$	Total
array	$O(n)$	$O(1)$	$O(n^2)$

actually $O(n+m)$ but max m is n^2

8

binary heap	$O(\lg n)$ have to rearrange	$O(\lg n)$	$O(m \lg n)$
Fibonacci heap not covered in class	$O(\lg n)$ amortized	$O(1)$ amortized	$O(m + n \lg n)$ worst case

6.006- *Introduction to Algorithms*



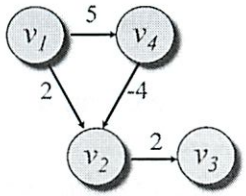
Lecture 16

Alan Deckelbaum

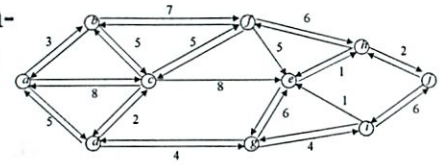
Lecture overview

Shortest paths III

- Bellman-Ford on a DAG (CLRS 24.2)

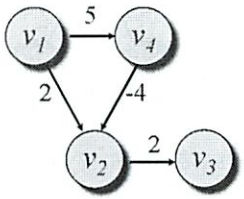


- Dijkstra algorithm for the case with non-negative weights (CLRS 24.3)

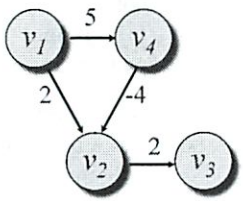


This graph has a special structure: DAG. How to use it within Bellman-Ford?

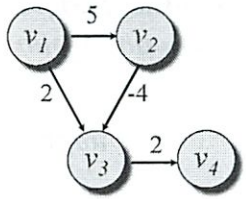
E={(v_1,v_2); (v_1,v_4); (v_2,v_3); (v_4,v_2)}



... first use topological sorting ...



E={(v_1,v_2); (v_1,v_4); (v_2,v_3); (v_4,v_2)}



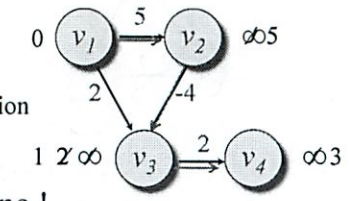
E={(v_1,v_2); (v_1,v_3); (v_2,v_3); (v_3,v_4)}

4/10

... Bellman-Ford ...

$E=\{(v_1,v_2); (v_1,v_3); (v_2,v_3); (v_3,v_4)\}$

end of first iteration



and we are done !

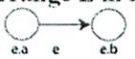
the shortest paths from v_1

Bellman-Ford algorithm on DAG

topologically sort the vertices V

($f: V \rightarrow \{1, 2, \ldots, |V|\}$ *such that* $(u,v) \in E \Rightarrow f(u) < f(v)$)

arrange E in lexicographical order of $(f(e.a), f(e.b))$ $\quad O(n+m)$



$d[s] \leftarrow 0;\ \pi[s] \leftarrow s$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty;\ \pi[v] \leftarrow \text{nil}$ } ***initialization*** $\quad O(n)$

do for each edge $(u, v) \in E$
  do if $d[v] > d[u] + w(u, v)$
    then $d[v] \leftarrow d[u] + w(u, v)$
      $\pi[v] \leftarrow u$ } ***one iteration of relaxation steps*** $\quad O(m)$

for each edge $(u, v) \in E$
  do if $d[v] > d[u] + w(u, v)$
    then report a negative cycle } ***final steps not needed***

... why does this work? ...

- there are no cycles in a dag => even with negative-weight edges, there are no negative-weight cycles ...
- topological ordering implies a linear ordering of the vertices; every path in a dag is a subsequence of topologically sorted vertex order; processing vertices in that order, an edge can't be relaxed more than once ...

Proof of Correctness

- Let t be an arbitrary vertex. Suffices to show that we compute d[t] properly.
- Let $s=s_0, s_1, s_2, \ldots, s_k=t$ be a shortest path to t. Show by induction that we compute each $d[s_i]$ correctly.
- $d[s_{i-1}]$ computed correctly by inductive hypothesis.
- (s_{i-1}, s_i) relaxed AFTER $d[s_{i-1}]$ computed.

Review of Dijkstra (Non-negative Edge Weights)

Problem: Given a directed graph $G = (V, E)$ with edge-weight function $w : E \rightarrow \mathsf{R}+$, and a node s, find the shortest-path weight $\delta(s, v)$ (and a corresponding shortest path) from s to each v in V.

Greedy iterative approach

1. maintain a set S of vertices whose shortest-path distances from s are known.
2. at each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal.
3. update distance estimates of vertices adjacent to v.

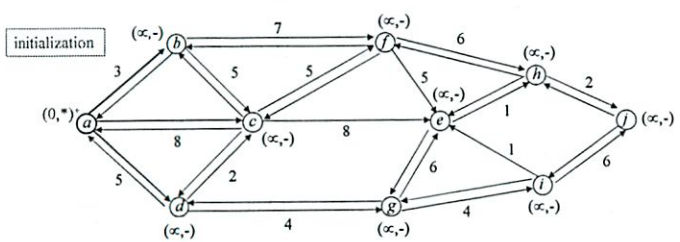
Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$
} ***initialization***

while $Q \neq \varnothing$ (Q min-priority queue maintaining $V - S$)
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$
} ***relaxation steps***

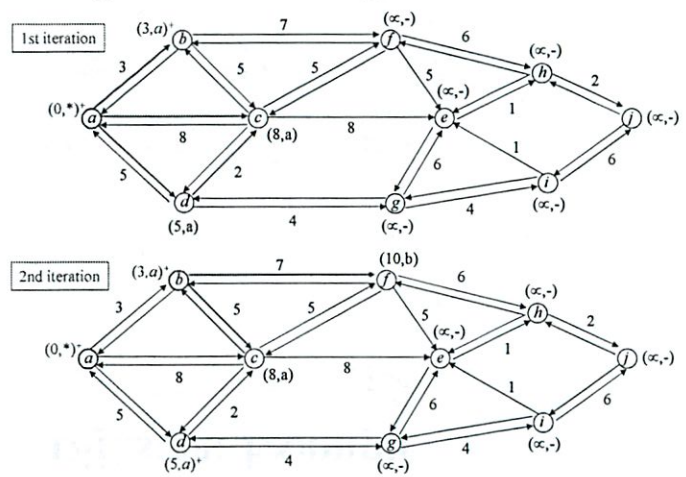
(Implicit DECREASE-KEY)

Dijkstra: Example

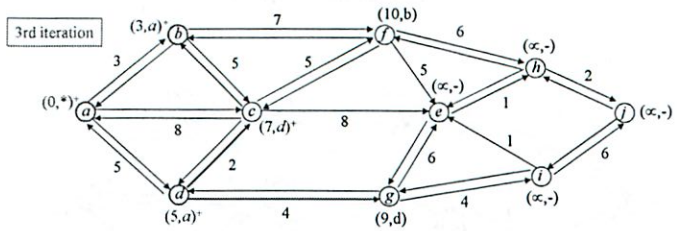


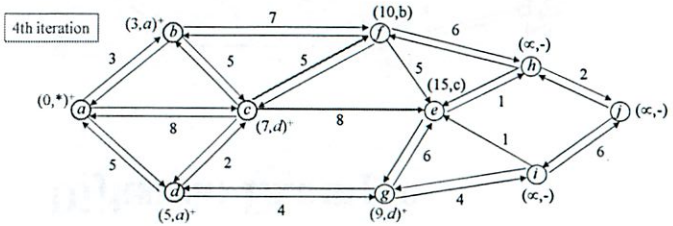
$Q = V$, a = EXTRACT-MIN(Q)

Dijkstra: Example

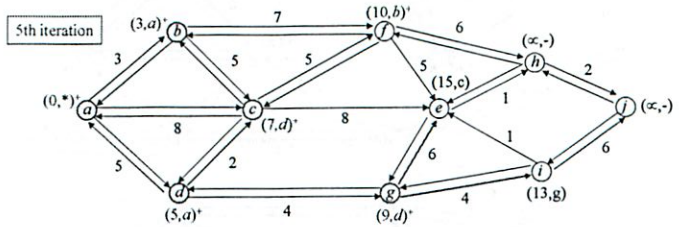


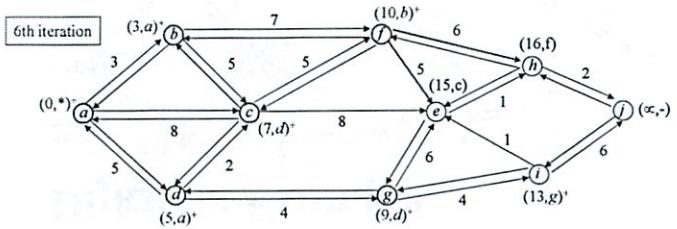
Dijkstra: Example



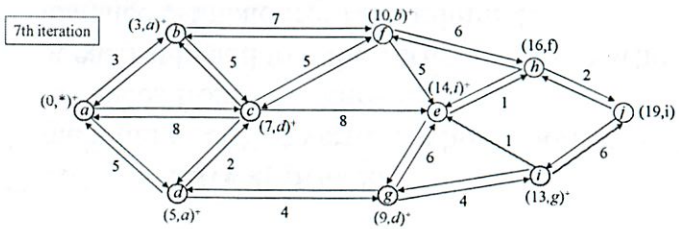


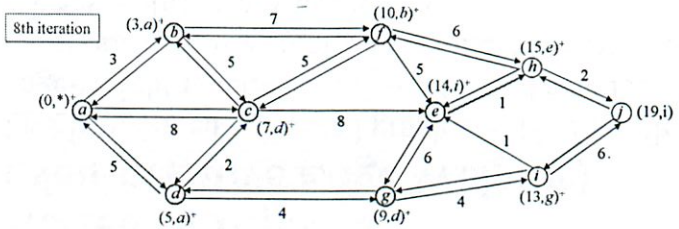
Dijkstra: Example



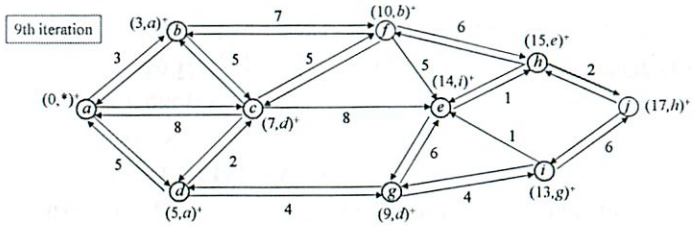


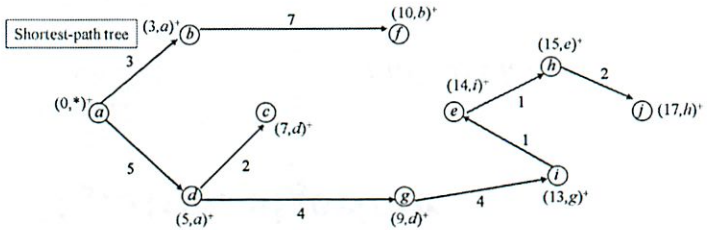
Dijkstra: Example





Dijkstra: Example



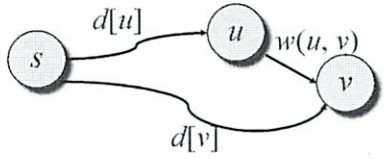


Correctness — Part I

Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof. Recall relaxation step:

if $d[v] > d[u] + w(u, v)$ **set** $d[v] \leftarrow d[u] + w(u, v)$

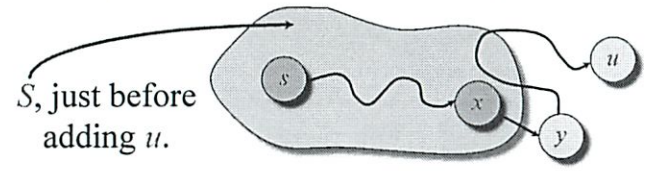


Correctness — Part II

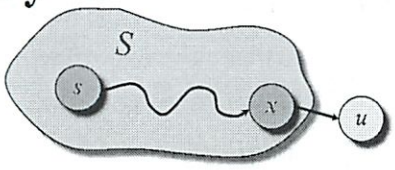
Theorem. Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof.

- It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to S
- Suppose u is the first vertex added to S for which $d[u] > \delta(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor:

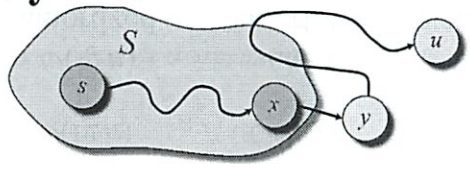


Correctness — Part II (continued)
Case 1: y =u



- Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$ at the time x was added to S.
- Just after x was added to S, we therefore set $d[u] = \delta(s, u)$
- This is a contradiction, since $d[u]$ is never increased by edge relaxation.

Correctness — Part II (continued)
Case 2: y != u



- Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$
- Since subpaths of shortest paths are shortest paths, it follows that $d[y]$ was set to $\delta(s, x) + w(x, y) = \delta(s, y)$ just after x was added to S
- Consequently, we have $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$
- But, $d[y] \geq d[u]$ since the algorithm chose u first => a contradiction

Analysis of Dijkstra

$|V|$ times:

while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    $degree(u)$ times:
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$

DECREASE-KEY

$$\text{Time} = \Theta(n) \cdot T_{\text{EXTRACT-MIN}} + \Theta(m) \cdot T_{\text{DECREASE-KEY}}$$

Analysis of Dijkstra (continued)

$$\text{Time} = \Theta(n) \cdot T_{\text{EXTRACT-MIN}} + \Theta(m) \cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\lg n)$	$O(\lg n)$	$O(m \lg n)$
Fibonacci heap	$O(\lg n)$ amortized	$O(1)$ amortized	$O(m + n \lg n)$ worst case

6.006 Recitation

4/11

Shortest Path on a DAG

Nieve Bellman Ford $O(VE)$
└ best for ~~gen~~ gental graph

Other possibilites **BFS** - seperates graphs into levels
DFS - Can work w/ Topo Sorts but does do weights ⊗
Dijstra - Only works ⊕ edge weights ⊕
Bellman Ford Looked at earlier ⊕

So must Topo Sort ← usually do on DAG
Then do edge relaxation

Know need to do better since extra info (ie is a DAG)

Remember can only make ⊖ → ⊕ if = # of edges

② Warmup problem: Standing Pokemon problem

$n \times n$ matrix type effectiveness T

$$T_{ij} = \begin{cases} 1 & \text{if } i \text{ is super effective vs } j \\ 0 & \text{otherwise} \end{cases}$$

Goal is to find a triple of types such that

$$\begin{tikzcd} i \arrow[r] & j \arrow[dl] \\ k \arrow[u] & \end{tikzcd}$$

What is T^3 again?

The # of lenght 3 paths from one to another

can we use that

want lenght 1 $i \to j$

~~of those that are~~

(3)

Then A^2 for $i \to j \to k$

make sure mentioned earlier

but automatic

A^3 make sure $i \to j \to k \to i$

or just A from before

TA: actually what I was thinking was wrong

Nieve: All triples V^3

We basically have an adj matrix

Can check all $i \to j \to k$

Adj matrix tells you lines

T3 - look for non zero entries on a diagonal

so simple than I thought

Matrix multiplication $\Theta(n^{2.31})$

gives you a starting point - then have to find but easish - use BFS

(4)

So think about possibles
- make the list of stuff you learned

Unweighted - so no reason to use Dijstra or Bellman-Ford

As said before → BFS to find triangles up to level 3

$O(V(V+E))$

Thats a reasonable runtime

Depends on how many V vs E

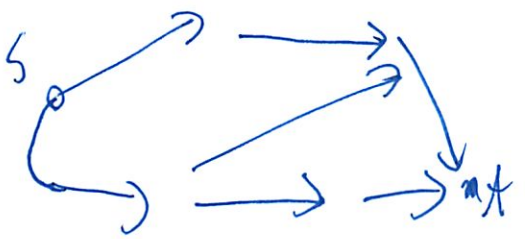
TA: I suspect n^2 is possible

(5)

New problem Number of Shortest path

Given unweighted G, find # of shortest paths $s \to t$.

We know we can calc shortest path in lin time. But now we want # of shortest paths



3 shortest paths

just do BFS fully. See which level t first shows up
w/ repeats

Then count # of times t appears

TA said this is reasonable
said problem hard to answer at all

(h)

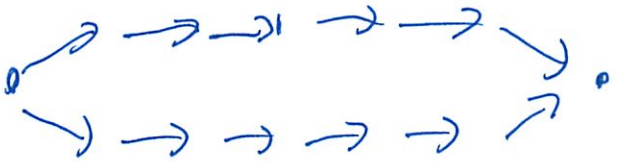
BFS one occurance of each <u>path</u> not each vertex

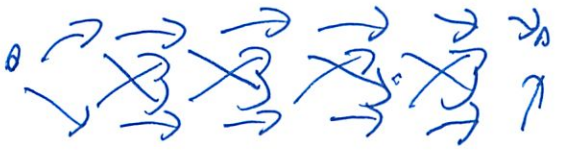
times appear is # of ~~time~~ inputs to a path

So use that info to calculate running time

shortest paths = $\Theta(E)$

What about





So could be as bad as $O(V^V)$

exponential time

(7)

Think about our choices

BFS

DFS

Adj matrix ← looks pretty good

~~Topo sort~~ not a DAG

~~Dijstra~~ no weights

~~Bellmanfort~~ no weights

Remember it tells us the # of paths of lenght k from i to j $= (A_k)_{ij}$

time

$V^2 \rightarrow$ Compute A

$V \cdot V^{2.32} \rightarrow$ Compute $A^2, A^3, \ldots A^i$

until i such that $(A^i)_{s \to t}$

$1 \rightarrow$ return $(A^i)_{st}$

$V^{3.23}$

(8)

Don't need to find each A^i

Just the max one

Use binary search on i

Repeated square

$$A \to A^2 \to A^4 \to A^8 \to A^{16}$$

to compute $A^k \to \left(A^{\lfloor k/2 \rfloor}\right)^2 \cdot A^{k \bmod 2}$

So $A^{15} = (A^7)^2 \cdot A$

$A^{16} \to (A^8)^2$

So we are doing this recursively (?)

$T(k) = T(k/2) + O(1)$

So Cost $\log^2 V \cdot V^{2.32}$

So overall $V^{2.32} \log^2 V$ ← is reasonable

(9)

Could solve in linear w/ Dynamic Programming

(will learn later in 6.006

t distance d

t has predecessors at distance $d-1$

shortest paths to t

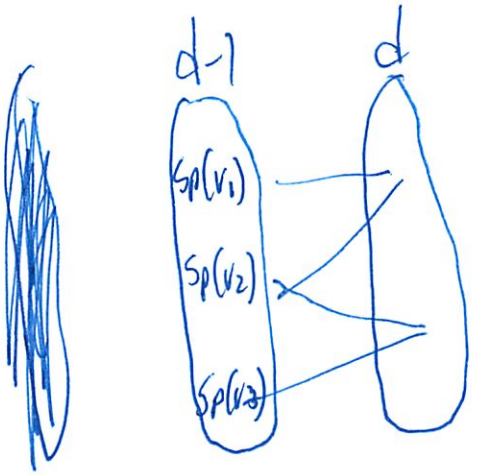
$$S_p(t) = \sum_{\text{pred } p_i} S_p(p_i)$$

(reduce shortest path problem
into finding ✓ other problems

bad recurrance
but small depth → d
Vertices used are $d-1$
No cycles - like DFS

(10)

# of shortest paths for all vertices



So just about # shortest paths at each level

(I was not really following)

TA: You are not expected to come up w/ this yet

Hard also to understand

6.006
L17 Heuristics for Faster Search

4/12

(Kishore lecturing)

Dijkstra ~ linear

But that's too slow for Google Maps

- edges of roads
- 10^{10} vertices
- 10^{11} edges

Dijkstra would be Θ(1 min)

So come up w/ some heuristics
- no worst case guarentee
- but help in practice
- all proofs handwaved

Use extra info to help
- Random
- Planar graph

) 2 sep cases
stuff works for other graphs

②

Random Graphs

lots of diff random choices to make

every vertex has d random edges

So d vertices from s

$\sim d^2$ " " " distance 2

$\sim d^k$ " " " " k

Since can be overlap, so approx

Can be cycles (head light)

BFS in a Random graph

Most vertices are in last few levels

So $\sim \log_d n$ levels

Most time is on last levels

How can we improve?

- explore end better

(3)

So go $s \to t$ and $t \to s$ simultaneously

Stop when have vertex from ~~both~~ both s and t

Can cut our runtime in half!

Can ~~the~~ be added to s first
or t first

But it can be tricky!

Ends after $\frac{\log_d n}{2}$ levels

Explores $\sqrt{n}$ vertices

Sublinear time!

↑ much less than half work

Since each level increasing geometrically

Works very well in pratices

And in non random graphs

(4)

Like the rubix cube graph

Lots of choices at each level

$n \to \sqrt{n}$

Bi-directional Dijstra

A bit diff

Can't alt. levels

Are no levels!

2 min heaps

- from s
- to t

Can do lookups forward + backwp

Pick smallest from either heap

Terminate when add a node to each

But the shortest path may not run through v
does go through something in S and something in T

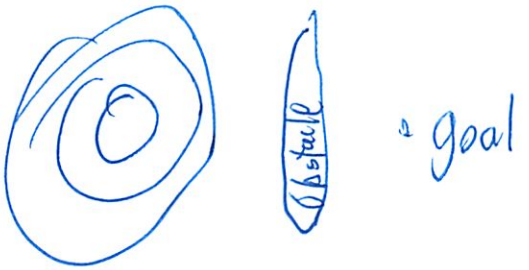
So loop over edge from a vertex x in S to
a vertex y in T
Find paths $d(s,x) + \ell(x,y) + d(y,t)$
If any path is shorter, return it
Must check all
Slower than you expect
But works well on graphs

Planar-weighted graph

- every vertex is on a plane
- can measure distance b/w points on a plane
- ie Google Maps - or a map in general

6
But

Dijstra moves in directions in discremetly
it ~~grows~~ grows in a circle



We want to grow towards the goal

Bidirectional helps us by factor of 2
- Compare circle areas
- worse improvement than random

So <u>A*</u>

Have a potential $\lambda()$

Reweight towards low potential

$\ell'(u,v) = \ell(u,v) - \lambda(u) + \lambda(v)$

But some flaws - must make sure non neg.

7

Proof: It will still find shortest path
(see slide)
Also shows no ⊖ cycles
↳ if there were none before

Consistent Heuristic

How Choose $h(u)$
- Can do as crow flies distance
(I've heard this twice before 6.01, 6.033)

Both non neg and points us towards
So cost of edges modified
- ~~towards~~ directly towards goal = no difference
- away from goal - grows

Can choose other $h()$ as long as
~~any~~ inequality $l(u,v) - h(u) + h(v) \geq 0$

Examples of A*
(see slides)

Basically you often use A* over Dijstra

Bi-dir A* is very complicated
- must use same huristic
- often times they are different

Other ideas
- precompute shortest paths for certain pairs
- incremental - use data from prior searches
- Only return approx shortest path

Lots of varients of A*
tuned for certain circumstances

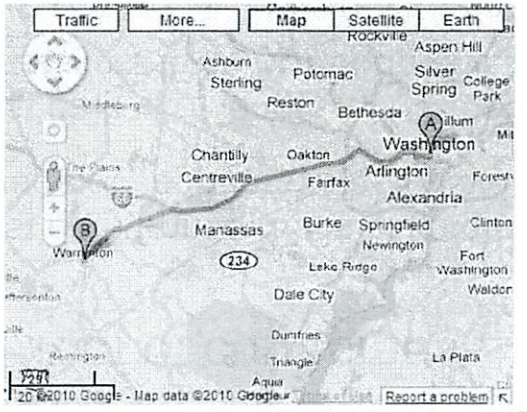
Can't be provably better than linear
but in practice

6.006 - Introduction to Algorithms

Lecture 17:
Heuristics for Faster Graph Search

Linear time is too slow...

- Google Maps: ~10^{10} locations, 10^{11} edges
- Dijkstra's would take Θ(1 minute)

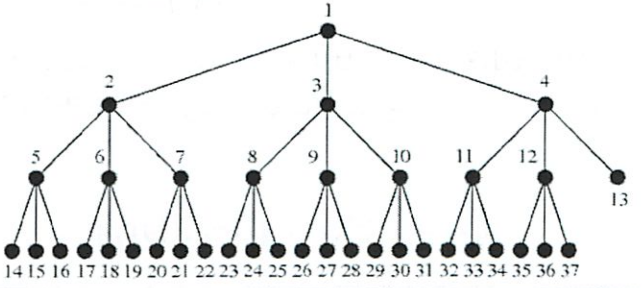


Today's goals

- Develop heuristics for shortest path searches
  - Preserve correctness
  - Improve runtime in practice, not in theory
- Consider special classes of graphs:
  - Random graphs
  - Planar-weighted graphs

Part 1: "random" graphs

- Every vertex has d random neighbors
- Consider the neighborhood of a vertex s
  - Number of vertices at distance 1: d
  - Number at distance 2: $\sim d^2$
  - ...number at distance k: $\sim d^k$



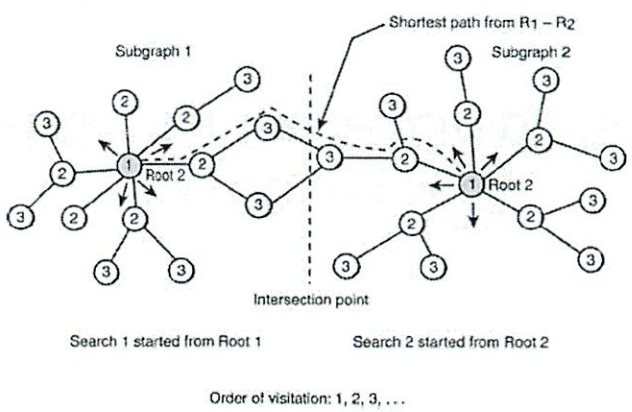
BFS in random graphs

- G is a random graph (n vertices, degree d)
- Suppose we search for a path from s to t in G
  - Almost all vertices are at levels $\sim\log_d n$
  - Almost all time spent at the last levels
- How can we improve our runtime?

Bidirectional BFS

- Idea: instead of running a BFS from s to t, run BFS from s to t and from t to s simultaneously
  - For each level i:
    - Compute vertices at distance i from s
    - Compute vertices at distance i from t
  - Stop when a vertex v has been found from both s and t
  - Shortest path from s to t runs through v

Example of bidirectional BFS



Proof of correctness

- If shortest path from s to t is of length $2k$, then middle vertex v_k appears in both level ks
- If shortest path is of length $2k+1$, then vertex v_{k+1} appears in s-level $k+1$ and t-level k
- Is this too easy?

"Analysis" on random graphs

- Bidirectional BFS expands $(\log_d n) / 2$ levels, instead of $\log_d n$
  - Explores about $\sqrt{n}$ vertices
  - Graph search in sublinear time!
- Performs well on many non-random graphs

Bidirectional Dijkstra

- Run Dijkstra simultaneously forwards from s and backwards to t
- Keep vertices in two min-heaps:
  - First sorted by distance from s
  - Second sorted by distance to t
- Pop the smaller of the two minimums
  - From s heap: add it to a set S
  - From t heap: add it to T
- Repeat till we add a vertex v to both sets

Subtleties in bidirectional Dijkstra

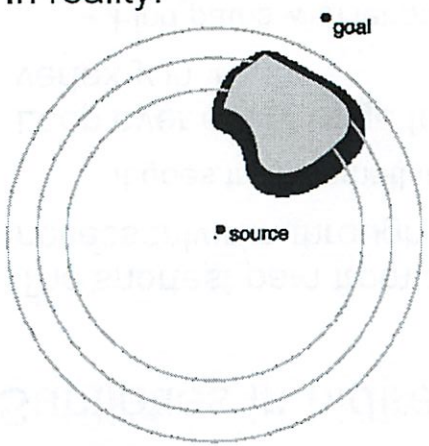
- The shortest path from s to t does not necessarily run through the vertex v...
  - It goes from something in S to something in T
- Loop over every edge from a vertex x in S to a vertex y in T
  - Find paths with lengths $d(s, x) + l(x, y) + d(y, t)$
  - If any of these paths is shorter than the path through v ($d(s, v) + d(v, t)$), return it instead

Part 2: planar-weighted graphs

- In a planar-weighted graph, vertices are points
- Edge length $l(u, v)$ is the distance from u to v
- We've seen this before:

Dijkstra on planar-weighted graphs

- In reality:



- In an ideal world:

Goal-directed search: A*

- Idea: use extra information to guide search from s to t
- Assign each vertex v a potential $\lambda(v)$
  - t should have potential $\lambda(t) = 0$
  - Vertices close to t should have low potential
- Try to search toward low potential
  - Modify edge costs: $l'(u, v) = l(u, v) - \lambda(u) + \lambda(v)$
  - Run Dijkstra?

Edge modification preserves paths

- New edge costs: $l'(u, v) = l(u, v) - \lambda(u) + \lambda(v)$
- Claim: the shortest path from u to v is preserved by edge modification
  - Let $(u, v_1, v_2, \ldots v_k, v)$ be a path from u to v
  - New path length:

$$l'(u, v_1) + l'(v_1, v_2) + \ldots + l'(v_k, v)$$

$$= l(u, v_1) - \lambda(u) + \lambda(v_1) + l(v_1, v_2) - \lambda(v_1) + \lambda(v_2) + \ldots + l(v_k, v) - \lambda(v_k) + \lambda(v)$$

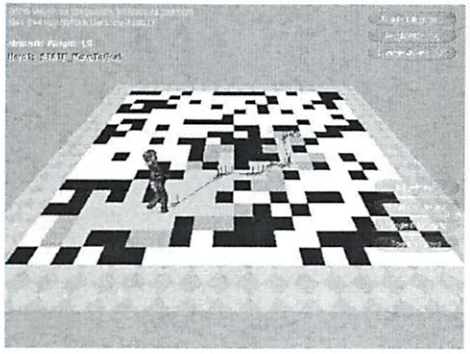
$$= [l(u, v_1) + l(v_1, v_2) + \ldots + l(v_k, v)] - \lambda(u) + \lambda(v)$$

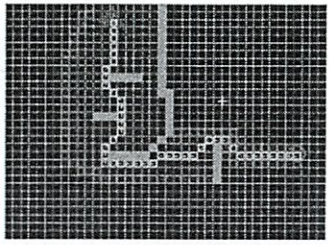
- New path length = old path length – $\lambda(u) + \lambda(v)$

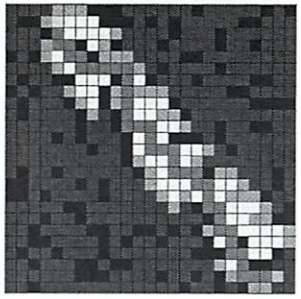
Consistent heuristics

- Edge modification preserves paths
- We can use Dijkstra if $l'(u, v) \geq 0$ for all u, v
  - As long as $l(u, v) - \lambda(u) + \lambda(v) \geq 0$
- How to choose $\lambda(u)$?
  - Suppose graph is planar-weighted
  - Use distance to t as potential: $\lambda(u) = d(u, t)$
  - Triangle inequality: $l(u, v) + d(v, t) \geq d(u, t)$
- Other graphs – other potentials

Results of A*







A* has been called one of the top ten algorithms of the last century!

Other ideas to speed up search...

- Precompute shortest paths for some pairs...
- "Incremental": use data from prior searches...
- Only return approximate shortest paths...
- ...

6.006 Recitation

4/13

Last time: Problem: # of shortest paths from $s \to t$ in an unweighted graph

Adj matrix kinda answers this

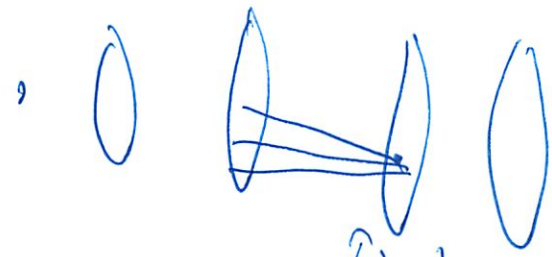
$(A^k)_{s,t}$

Leven worse than quadratic

$n^{2.32} \log n$

↑ # of times to multiply

But could be solved in linear time



↑ just sum of # of paths to ancestors ~~in the parents~~

(dir or undir)

(2)

Since you have to be at the d-1 nodes

Doing nsp() on all previous nodes as well

$$nsp(V) = \sum_{\uparrow prev} nsp(V_i)$$

Since each node can only be in 1 level

Only visit each node once

Each edge is included at most once

$$O(V+E)$$

Each recurrence is V time

Can do this in forward BFS

by tracking # of shortest paths to each vertex

(3)

<u>Today</u> # of shortest paths $s \to t$

in a weighted graph w/ non neg weights

<u>Possibilities</u>

~~Adj~~ - Weighted edges

BFS - might be useful

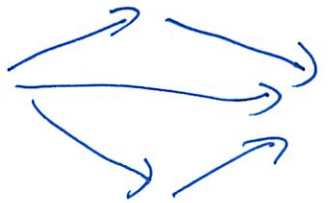
~~DFS~~ - for topological sort (used in)

→ Dijstra - ~~on~~ non neg weighted edges (✓ Oh yes)

Bellman Ford

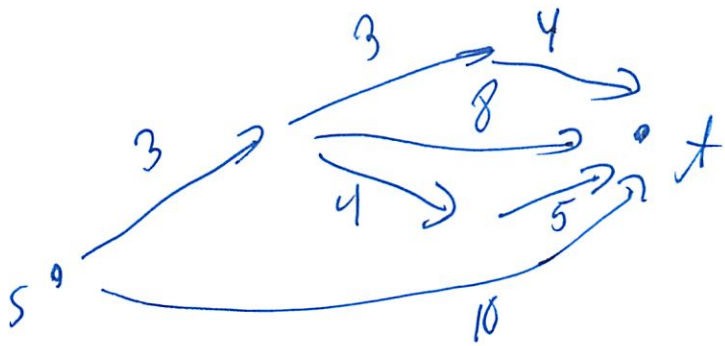
keep in mind → Results from last problem

(need to study this stuff ...)

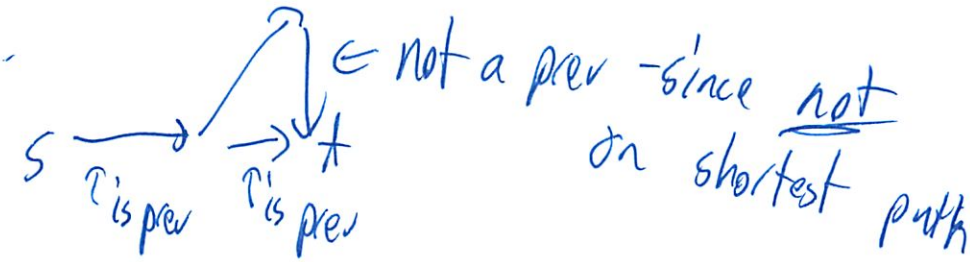


track previous nodes

Want to define the set of previous nodes



Can't just point to t
even in ~~[illegible]~~ unweighted



So instead look at the weights

<u>Def</u> Previous nodes of a vertex t
nodes that could be in the second to last vertex on a shortest path from s to t

$$nsp(t) = \sum_{\substack{\text{prev nodes} \\ p_i}} nsp(p_i)$$

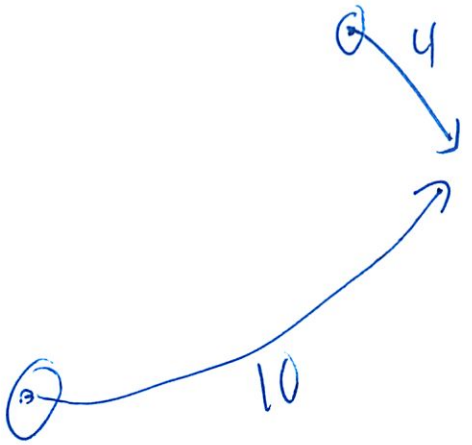
(5)

by inspection

We know the 2 prev ~~n~~ nodes are the ones w/ lenght 10 leading in

but we don't really know that

by algorithm



So ~~have~~ to find: basically keep going backwards

Here conditions are harder

- before: leads into A

on the same level

Levels before were distances to a vertex from s

Levels $\Leftrightarrow$ $d(s,v)$

same distance = same level

(6)

Using dijstra can calc $\delta(s,v)$ for all vertices

u is a prev node of v if $\delta(s,u) + \ell(u,v) = \delta(s,v)$

└ For every node we can check this

So can do a list in <u>linear</u> time

↑ for all vertices

Use dijstra $O(V \log V + E)$

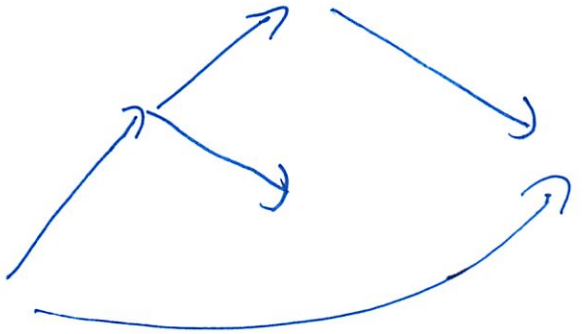
Make list $O(V + E)$

So how to eval recurrence?

Prev node graph

└ edge from u to v if u is a prev node of v

7



Finding # of paths, not shortest path

No consistent order to build off of

But don't know which order to go in

How to get a consistent order to eval nsp

So if we - Sort vertices by $\delta(s,v)$

- Dijkstra does automatically

- nsp value of t depends only on nsp's of vertices closer to s than t
  - using recurrences
  - Same properties of BFS $\Rightarrow$ going by level

(8)

W/ 0 cycle → ∞ # of shortest lenght cycle
↳ can detect cycle

So complete algo → $O(V \log V + E)$

Could also fold that stuff into dijstra

Work forwards both times
→ Dijstra
→ nsp from s

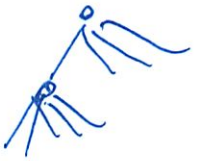
Tricki Alt to finding an order
↳ as long as there is an order

As long as there is <u>an</u> order that works
We can do computation

lookup = {s : 1}

```
def nsp(t):
    if t in lookup:
        return lookup[t]
    if t == s: return 1
    lookup[t] = sum(nsp(p) from p in prev nodes[t])
    return lookup[t]
```

but problem is runtime



exponential

V^n

Could call several times for same vertex

So save your results in a lookup table

So only computes nsp() once per vertex

So linear time

* This is the key example of dynamic programming

(10)

Challenge: # shortest paths in a DAG w/ arbitrary weights

Problem Set 5

This problem set is due **Wednesday, April 18** at **11:59PM**.

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **Graph Transformations (15 points)**

   You are given a directed graph $G = (V, E)$ with positive or negative weights $w(i, j)$ and no negative cycles. Your job is to find a transformation from weights $w(i, j)$ to new weights $w'(i, j)$ that eliminates the negative edges but does not change the sequence of vertices for each shortest path between any two vertices. Call such a transformation "good" if all shortest path vertex sequences in the graph with weights w' are the same as the shortest path vertex sequences in the graph with weights w.

   For each part below, answer whether the transformation is good. That is, answer `True` if the transformation is good, and `False` if it is not good.

   (a) Replace each weight with its square so that $w'(i, j) = w^2(i, j)$.

   (b) Add a large constant C to each edge weight, so that the weights $w'(i, j) = w(i, j) + C$ all become nonnegative.

   (c) Suppose it is possible to find a value $d(v)$ assigned to each vertex v of the graph such that $w(i, j) + d(i) - d(j) \geq 0$ for each edge (i, j).[1] Take $w'(i, j) = w(i, j) + d(i) - d(j)$.

[1] It is possible to compute such d values by using a variant of the Bellman-Ford algorithm: Make a new source vertex s, connect s to every vertex by a weight-0 edge, run Bellman-Ford starting from s, and let $d(v)$ be the length of the shortest path from s to v.

2. **Topological Sort (25 points)**

Consider the DFS code for directed graphs from CLRS. (This code iterates through all vertices in the graph, and runs DFS starting from this vertex if the vertex has not yet been visited in a prior search.) One can use this DFS to obtaining a topological sort of a directed acyclic graph (DAG) G. (In a topological sort, your goal is to obtain an ordering of the vertices such that all directed edges go from a vertex to a vertex later in the ordering.)

For each of the below proposals, answer `True` or `False` to the following: Running the algorithm on a DAG necessarily produces a topological sorting.

(a) Run the DFS code from CLRS and order the vertices in increasing order of their start time.

(b) Run the DFS code from CLRS, where we start DFS only from sources (vertices with no incoming edges) and sort in increasing order of the start time.

(c) Run the DFS code from CLRS on the reverse of the graph (where we reverse the direction of all directed edges), and where we start DFS only from sources of the reversed graph (vertices with no incoming edges), and sort in decreasing order of the start time.

(d) Run the DFS code from CLRS, and order vertices in decreasing order of their finishing time.

(e) Run the DFS code from CLRS on the reverse of the graph, and order vertices in increasing order of their finishing time.

3. **Making Unlimited Money (40 points)**

You decide to use your MIT education play the stock market. Being an ambitious 6.006 student, you desire not just to make large amounts of money, but to make *unlimited money* through a sequence of financial transactions. We will model this problem by a walk on a directed graph, where each node represents a state of the stock market. If there is a directed edge (i, j) in the graph, then it is possible, by making some financial decision, to move from state i to state j.

Each edge (i, j) has an associated nonnegative real number value, denoted $m(i, j)$, representing the *multiplicative* change in your total cash assets as you move from i to j. (If you have d dollars at state i and take the edge (i, j), you will have $d \cdot m(i, j)$ dollars in state j. Thus, a m value greater than 1 denotes an increase in money, while a value less than 1 denotes a decrease in money.)

Your goal is to design an efficient algorithm to determine if it is possible, starting with 1 dollar and beginning from a start vertex s, to obtain arbitrarily large amounts of money by making a series of financial transactions. At no intermediate step is your cash balance allowed to be below some threshold b (with $b > 0$), since if you do so your broker will not allow you to play the market further.

Formally, your task is to determine (yes/no) whether the graph has the following property:

For any positive integer N, there is a sequence of steps beginning from s with 1 dollar such that you have at least N dollars at the end of the sequence and at no point in the sequence did your balance become less than b.

Design an efficient algorithm for this problem, argue its correctness, and explicitly state its asymptotic running time in terms of $|V|$ (the number of states in the graph) and/or $|E|$ (the number of edges).

4. **Shortest paths on expanders, in sub-linear time (30 points)**

Suppose we construct an undirected graph in the following way: Fix some small even value d, and for each of the n vertices, choose $\frac{d}{2}$ random neighbors and create those edges. Such a graph is an example of an **expander graph**, with expansion d. For a graph like this, the number of nodes within distance k of a node is roughly d^k, for $k < \frac{\log(n)}{2\log(d)}$ (i.e. when the square of the neighborhood size, d^{2k}, is less than the number of nodes, n). For example, for an expander with expansion factor $d = 10$ and $n = 10^{100}$, a node will have about 10 neighbors, 100 nodes within distance 2, and 10 billion nodes within distance 10.

We've seen how to use breadth-first search starting from s to find the shortest path from s to t on an undirected (unweighted) graph. But in the special case of expanders, one can actually do much better than $\Theta(E) = \Theta(nd)$ in the average case. Your job will be to design and code a function `find_distance(graph,s,t)` which quickly returns the shortest path from s to t on an expander, or `None` if there is no path (though this is extremely unlikely to happen if $d > 1$).

Your code should pass the following test case (where $d = 2$):

```
graph = {1: [4, 5],
         2: [3, 4, 5, 6],
         3: [2, 6],
         4: [1, 2, 5, 6],
         5: [1, 2, 4],
         6: [2, 3, 4]}

find_distance(graph, 1, 1) == 0
find_distance(graph, 1, 2) == 2
find_distance(graph, 1, 3) == 3
find_distance(graph, 3, 4) == 2
find_distance(graph, 4, 5) == 1
find_distance(graph, 5, 6) == 2
```

6.006 P-Set 5

4/16

1. directed
   + or − weights
   no neg cycles

   $w \to w'$ to get rid of neg edges
   but does not change shortest path

Oh they give us suggestions

I was going to say increase by

~~# nodes / ~~ ~~in lenght of path~~

lenght / total lenght of path

No thats not it either

---

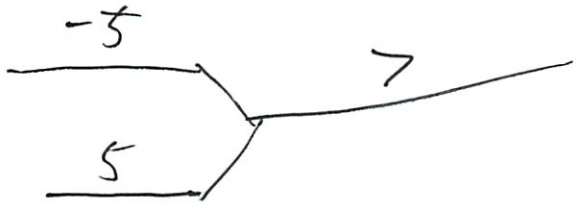
make each path fraction
but the neg!

(2)

a) Square

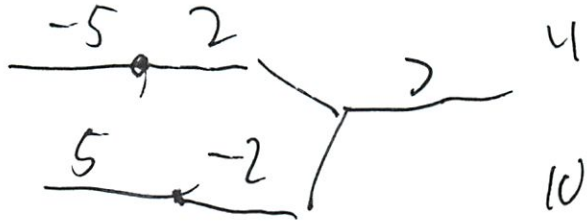
Sounds pretty good

but what if non unique shortest path

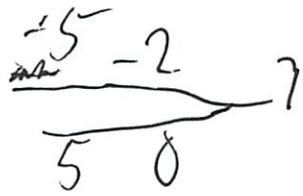


25 ~ 44 ← now more shortest
25

but



25 4 44 again both same
25 4



0 vs 12

25 4 7
5 0 7 ← now smaller False

③

Didn't TA say this is not possible

b) Add C

~~We~~ One we talked about in class thats wrong

c) Sounds complicated

V = vertex value

Bellman ford

- ~~Tell~~ start each ∞
- ~~add~~ explore an edge
- see if path better
- fill in value
  - value is best way to that pt

?? every pass does every edge

└ look seems

Thought it was just the ones you are on

9

Watched video online → every edge every time

Abibitrary ordering of edges

(un through same each time

V-1 times (repeat)

So $O(VE)$

E edges V times

But what is this $d(v)$

Bellman cord s to v

bet δ length path s to every vertex

?V?

?? everywhere

But similar to heuristic??

But this is planar

(5) ¿But it could be neg?

Pizza: Arbitrary values that fit the eqn

So its some soft function

└ I guess that

But foot note crazy

Thinks l_c is good

└ I'm not so sure

¿

Mapping

¿ slide in L17

└ edge multiplication preserves paths

which is part of A*

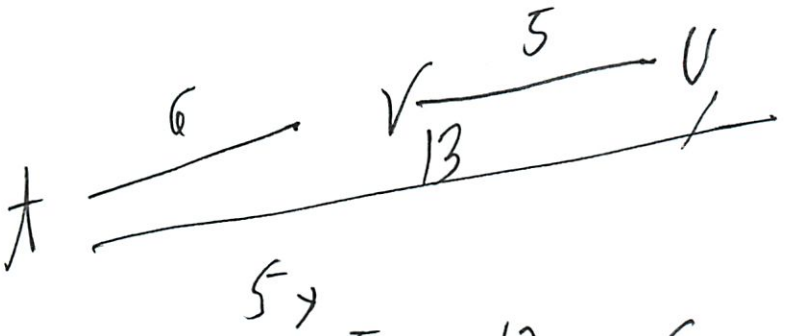
¿ but relies on heuristic

f

g

(4)

So $\ell(u,v)$ is actual distance



So $5 - 13 + 6$

$5 \to -2$

gets us closer so ⊖

Choose $\mathcal{H}$ as crow flys distance

Proof that this shortest path is preserved

¿I don't get

But is general case

So say yes - edge modification preserves True

¿? But pratical challenge in picking paths ¿

(7)

2. Topo Sort

def Review: Only on a DAG

linear ordering

$O \to O \to O \to O \to O$ etc

One after the other

How to find: Call DFS(G) to find

V.f for each V

↑ finishing time → finish processing all its child nodes

when done on front of linked list

return list

↑ last to finish → ↑ early to finish

(So simple!)

[I often don't think of finishing as usable)

(8)

initalize each white, nil
π

DFS

for each vertex
set grey
~~explo~~ for each adj edge
~~grey~~ explore
if white
set π to be v (time)
visit that → turn

?
not really precise

when finished, set black
set v.f as time

→ time = # of nodes finished
white → ~~un~~ unreached
gray → exploring
can have ≥1 be gray

black - returned back from it

π = parent

if not white → don't visit

(9)

Ok questions → Does this produce a topo sort

a) Start time

no its end time

but does this also work?

but won't be correct?

Underpants → shoes → pants → belt →

Jacket →

(Where do you start it multiple times?

just start somewhere random

→ socks → shirt → tie

no Jacket before tie - wrong

↳ must finish - clearly seems better

(10)

b) Only start at sources (no incoming edges)

?sounds good

↑ order start time

Still no – I actually did that last time

c) Reverse the graph

Run on sources of that

sort ↓ order on start time

jacket → tie → shirt → belt →

pants → underwear →

→ shoes → socks

read

No ~~underwear~~ shoes is before ~~shoes~~ underwear

No

(11)

d) ↓ order finishing time

Is that it

last → first to finish

no thats backwards

False

e) Reverse the graph

Order ↑ finishing time

I feel this will be wrong

~~Jacket → tie → shirt~~

run → shirt → tie → ~~belt → pants underwear →~~

~~shoes → socks~~ underwear → pants → belt → jacket

socks → shoes →

that actually works I believe

True

Matches Crystal a e

(2)

3. Making Unlim $

"not just a large amt

Directed graph

each node = state of stack market

edge $i \to j$ if can move there

↑ Some non neg multiplier

>1 = ↑ in $

<1 = ↓ in $

Can you get arbitrarily large $

↑ Sounds like a loop of >1s

Can't ever fall $<b$

↑ ie get into a <1 loop

But its the multiplication of all values in the loop

Can the graph do this?

(13)

So basically find an algo that detects cycles > 1 ↑ return true instantly

and < 1 ↑ return false instantly

So cycle detection is what?

BFS

DFS

Dij

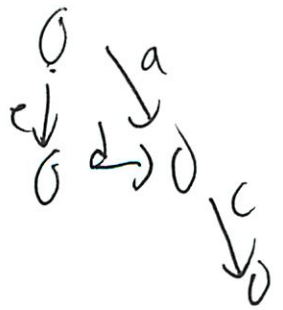
Bell-Ford

Piazza 436: Not just detecting cycle

Harder part is if can reach w/o losing too much $

Only must be possible

So if go under b

1. 

if $a \cdot c < b$

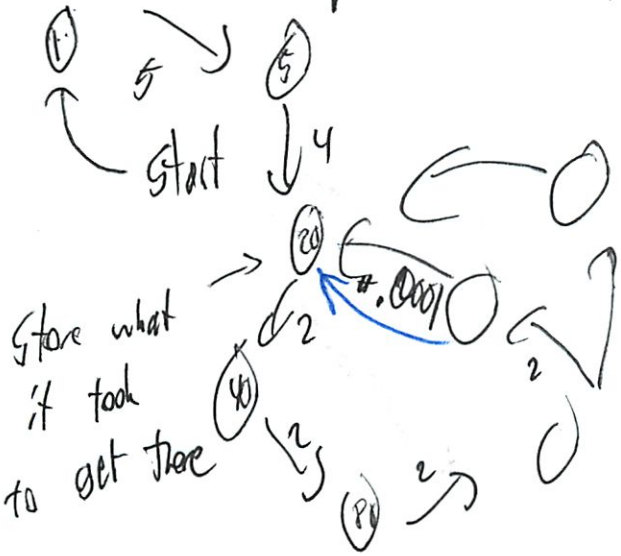
What went wrong?

(14)

Not looking for shortest path
└ but a cycle

DFS is cycle detection?
Can also do w/ BFS
"Did on last p-set"
└ the odd cycle coloring

DFS-best for cycle detection?
"keep track of factor



Store what it took to get there

want 32

20 · 32 so /out 20

So write current multiple at every pt in time

(15)

¿ Then what do we invalidate ?

if bal < b, back up | ¿ and take other path (ie DFS)

└ mark as black

(even though we did not fully explore its child nodes)

Always mark w/ current val

When cycle detected, divide the previously visited node

if > 1 we win → return true

< 1 ~~mark as black,~~ back up

if no other edges then mark as black

If get through whole graph (ie all black)

└→ return False

<u>Complexity</u>

DFS is $\Theta(V+E)$

↑ ¿ best you can do ?

(16)

DFS complexity

Visit each vertex $O(V)$

¿What is aggregate analysis

Chap 17.1 (have read before ...)

for all n, a seq of n ops takes worst case $T(n)$ in total

So ammortized cost $\frac{T(n)}{n}$

¿So some are more costly than others but it flattens out?

So if pop is $O(1)$

popping all n items is $O(n)$

But if had multipop (k) that pops k items

Then this is $O(1)$ k times $= O(k)$

(17)

Worst case multipop(n)
↳ $O(n)$

but if did multipop(n/2) → multipop($\frac{n}{2}$)
↑ $O(\frac{n}{2}) + O(\frac{n}{2}) = O(n)$

So can do multipop a max of n times if pop 1 each time → $O(1) \cdot n = O(n)$

So say aggregate cost $= O(n)$
↑ is a worst case bound

So DFS visit called once for each vertex ✓
↳ since then colored

So $\sum_{v \in V} |Adj[v]| = \Theta(E)$
↑ so we go on each edge

We visite every vertex once
try each edge once

The Θ in $\Theta()$ is confusing - don't see normally

(18)

BFS Time (just for fun)

→ initalize everything $O(V)$

Enqueue each vertex at most once

$\approx O(1)$ $= O(V)$

Scans each adj list once (when dequeued)

$O(E)$

So $O(V+E)$

↳ I guess flexible for whatever is bigger

¿ did we ever show $O(V+E)$ is the best you can possibly do ¿

Back to our regually scheduled proble

↳ I think its good - will discuss w/ others 1st

(19)

4. Shortest path in sublinear time

So from Crystal: bi directional BFS is slightly too slow
Bi directional Dijstra looks promising

One one hand graps seem easy
but on another hand they seem really complicated

Review bi directional
└ was latest lecture

Dijstra - find min where you are at now
then update/relax
$O(V \lg V + E)$ └ check if < then current vals

Differences	Bellman Ford	Dijstra
	weighted directed ⊖ supported	weighted directed ← the only 2 that support this
	no neg cycles	non neg (must be)
	slower (in general)	no neg cycles
	$O(VE)$	faster (in general)
		$O(V \lg V + E)$

Single shortest path in a DAG

O(V+E) (from DFS to topo sort)

always well defined (since no neg cycles)

1. Topo sort the DAG

   u must proceede v

2. just one pass over the topo sort

   from first to last

   (even if first before start)

   and "relax" each edge

   ↳ ie check if shorter, if it is, replace cost

3. Then each node has shortest lenght from start to there

Now Heuristics Lecture Review

Random graphs

each vertex d random edges

So d^2 vertexes within lenght 2

(21)

So explore from end first

Handwavy proofs

$\sqrt{n}$ vertices → sublinear times

Bi-dir Dij

- 2 min heaps
- do lookups forward + back
- pick smallest from either heap
- shortest path may not go through v
- but runs through something in s and something in t
- so all edges x in s to y in t

$$d(s,x) + l(x,y) + d(y,t)$$

- if any path shorter → return that
- check all

(22)

Planar

Use heuristic like crow flys distance

Bi dir A^* very complicated

Ok look at problem now

Some even value d

for n vertices create $\frac{d}{2}$ $\underbrace{\text{random}}_{\text{random!}}$ neighbors

expander graph

↳ wpi sparse graph - not near max # of edges

So # nodes $= d^k$

for $k < \frac{\log(n)}{2\log(n)}$ (only valid when this is true)

(23)

Used BFS to find $s \to t$

but can do much better than $O(E)$

So basically that whole intro is BS

Just do bi-dir Dij

Now how actually do?

(Nothing online - do old fashioned way - regular Dijstra lol)

non neg ✓

but not weighted → so bi dir BFS?

what is Dij w/ all weights = 1

└ DFS? No! - since it picks shortest from any node

Correct: So regular DFS does not find shortest path

BA Dij since picks from shortest path

(24)

What are all shortest path algs?

Dij - fastest

Bellman Ford - if Θ

A* - w/ heuristic

Ok so we're good on this idea

What is init-single-source

Set $v.d = \infty$

$v.\pi = \pi$ ← parent

$s.d = 0$

S = final value set

Q = min heap

(I forget heap stuff)

ignore for now, but remember at some point!

(25)

No heapq length!

└ no its the silly way Py does lengths

Queue needs length!

└ how do that?

tuple [len, item]

└ confuse me on interview

?2 data structures

On to heapify

One to quickly return distance?

?but how ~~can~~ update values on the heap

? push new values on?

just leave on it?

S is what?

(26)

Almost works

— but need to replace queue

And return on that value t

└ easy ...

But fix this lot " DECREASE-KEY

└ should take $O(\lg V)$

heapreplace()

Forum says bad → do $O(N)$ search

Fib heap better

Doing this complex lib...

node l is len 0

Still need edges decrease key

Need to store its node pointer

Done

└ Hopefully this is not dog slow

(27)

Does binary heap (regular) have delete by obj id?

Added return +

✓ Pass test cases

rev 100 for 6.006

This will be so slow!

Must fill in above ans ...

✓ Passes all the small cases

But fails the complex ones

in -- consolidate ...

Prob need to ditch this code ...

20 sec ...

Lots of None for some reason on graph??

↳ So insert fails

↳ creates a none ...

or ↓ key

(28)

I need to ditch this lib
but keep pointer

Oh can define a custom item

When its min → auto shortest

(✓) Woot it works!

Ran > 20 sec...

but is it right?

Try removing queue.remove ...

It passes the mini cases

? Try fixing remove

Then double eded...

(✓) Same objects

(✓) Does let me update w/ calling Heapify each time

└ Need heapify

(29)

4/17

Now double ended

need Psudo code

looked for paper - but also in slides

? delete it on other ?

L Think yes

Have to finished S and T

Now do the shared ✓

in both sets

Return path lenght

L can count lenght of pathsum

Since all weight 1

We never use parent...

Must be bidirectional graph...

L ✓ is undirected

(30)

Oh else if bug ... grr

If can't delete on other chars

- or don't do that?

No - don't think I should

- I think I got it

- but returning wrong

(✓) Works on test cases

Without the sublities ....

(Why no test cases for that?)

Still limit exceeded ...

(I think sometimes I am ~~then~~ finding "the old way")

Let me email

So if s has 1 2 5

t has 0 3 4 5

(31)

So ~~[illegible]~~ ~~[illegible]~~ might not be $s \to 5 \to t$

But try is there $1 \to 3$
a node $1 \to 4$

$2 \to 3$
$2 \to 4$

Return that instead

This is a correctness mistake - not a time
and all lenght is 1
emailed in

Its correct but way too slow
than to improve;
Unless its catically diff
Like no Dij - but that shald be it...

From some random paper

3 Theoretical Results

We model our network as a directed graph $G(V, E)$, where V is the set of nodes and E the set of directed *edges*. We refer to a *link* (i, j) as the pair of edges (i, j) and (j, i). We associate two values with every edge $(i, j) \in V$: a positive weight $w(i, j)$ and a nonnegative (bandwidth) capacity $b(i, j)$. Let $S \subseteq V$, $D \subseteq V$ be source and destination subsets of V, such that $\forall i \in S$, we associate $B_i \geq 0$ as the bandwidth needed by source i.

Define a multipoint connectivity structure (MCS) $\sigma(V', E')$ as a connected subgraph of $G(V, E)$ containing at least the nodes $S \cup D$ and having at least one path from each node $s \in S$ to every node $d \in D$. The bandwidth and weights associated with an edge in σ are those associated with the original edges of G.

Let $S(i, j) \subseteq S$ be the subset of sources contributing flow to edge (i, j). Then:

Definition 1 *An edge (i, j) is said to be underloaded if:*

$$b(i, j) \geq \sum_{p \in S(i,j)} B_p \tag{1}$$

otherwise, the link is said to be overloaded.

A link is underloaded if both edges comprising it are underloaded. An MCS is called *feasible* if all its edges are underloaded. The *weight* or *cost* of an MCS is the sum the weights of its edges.

In this paper we focus on searching and investigating low cost, feasible, MCSs.

3.1 The bidirectional connection

The simplest multipoint-to-multipoint (mtp-mtp) connection that one may conceive is a bidirectional unicast connection. In this case, $S = D = \{s, d\}$. If we allow the paths $p(s, d)$ and $p(d, s)$ to

be distinct, the problem of finding a minimum cost feasible connectivity structure can be reduced to computing two single source shortest-path problems with bandwidth constraint ([Wang95]). More formally:

Algorithm1:

1. Create the subgraph $G'(V', E')$ by pruning off edges $(i, j) \in V$ such that $b(i, j) < B_s$.
2. Compute a shortest path $m(s, d)$, between the source s and destination d.
3. Decrease the bandwidth of every edge $(i, j) \in m(s, d)$ by B_s.
4. Create the subgraph $G''(V'', E'')$ by pruning off edges $(i, j) \in V$ such that $b(i, j) < B_d$.
5. Compute a shortest path $m(d, s)$, between the source d and destination s.
6. Construct an MCS the includes all the nodes and edges comprising both $m(d, s)$ and $m(s, d)$.

Theorem 1 *Algorithm1 finds a feasible minimum cost MCS for $S = D = \{s, d\}$.*

Proof of Theorem 1 *If the MCS is formed by two disjoint paths, the Theorem follows by applying [Wang95]'s result for each "half" connection, as if they were separate problems. So, let's assume that minimum paths $m(s, d)$ and $m(d, s)$ share some edges(s). However, the underload condition of the edge guarantees that the referred edge will be present in the second shortest-path computation, which will find the second minimum path $m(d, s)$.*

Let us require now that the MCS be a single bidirectional path connecting (s, d). To proceed, we need the following simple definitions:

Definition 2 *The length $d^c(s, d)$ of a path $p(s, d)$ under cost function c is defined as:*

$$d_p^c(s, d) = \begin{cases} \sum_{(i,j) \in p(s,d)} c(i, j) \\ \quad \text{if } \forall (i, j) \in p(s, d), \qquad b(i, j) \geq B_s \text{ and } b(j, i) \geq B_d \\ \infty \\ \quad \text{otherwise} \end{cases}$$

i.e. the path length is the sum of its link lengths if its edge components are all underloaded. Otherwise, the path length is assumed to be ∞.

Definition 3 *The shortest path between (s, d) with respect to the cost function c is:*

$$\delta^c(s, d) = \min_{p \in P} d^c(s, d)$$

where P is the set of all paths $p(s, d)$.

We now propose a Dijkstra type of algorithm to solve the single bidirectional path min-cost problem. The pseudo-code follows:

3.1.1 BD-Dijkstra Pseudo-code

BD-DIJKSTRA G(V,E)

BD-Initialize(G,s); ← same

$S \leftarrow \{\}$, $Q \leftarrow V[G]$; ← same

while $Q \neq 0$ **do**

u= BD-Extract-Min(Q, d);

$S \leftarrow S \cup \{u\}$;

For each vertex $v \in Adj[u]$ **do**

BD-Relax$(B_s, B_d, u, v, b(u, v), b(v, u), c(u, v))$;

a lot longer!

BD-Initialize(G,s)

For each vertex $v \in V[G]$

Do $d[v] \leftarrow \infty$; $\pi[v] = NIL$;

$d[s] \leftarrow 0$;

"why d"

BD-Extract-Min(Q,d)

$d_{min} \leftarrow \infty$; $u \leftarrow NAN$;

For $i \in Q$ **do**

If $(d[i] < d_{min})$

$\{ d_{min} \leftarrow d[i]; u \leftarrow i;\}$

RETURN u;

"what here"

BD-Relax$(B_s, B_d, u, v, b(u,v), b(v,u), c(u,v))$

If $((B_s < b(u,v))\text{and}(B_d < b(v,u)))$

$\{w(u,v) = c(u,v);\}$

else

$\{w(u,v) = \infty;\}$

If $(d[v] > d[u] + w(u,v))$

$d[v] \leftarrow d[u] + w(u,v);$

$\pi[v] \leftarrow u;$

3.1.2 BD-Dijkstra analysis

As usual in Dijkstra type algorithms, S is a set of vertices whose current shortest path is maintained, and Q is a priority queue with vertices $i \in V - S$ with current distance $d[i]$. Each vertex u has a pointer $\pi[u]$ to its previous vertex in the current shortest path, which is initially set to NIL. BD-Extract-Min fetches the vertex outside S which is closest to the source, and BD-Relax updates vertices distances to the shortest ones, as is standard in Dijkstra's algorithm. The twist here is that BD-Relax tests if there is enough bandwidth on the link before consider it for relaxation. Notice that this is possible only because at this time, the direction in which the link will be used in the path is already defined. The reason why this link pruning is not done in advance is precisely because the direction in which the links could be used for connectivity is not known up until they are inserted in the path candidate, which is done by BD-Relax. So, BD-Dijkstra uses a cost function w which is defined during run time only.

We now prove the following theorem:

Theorem 2 *BD-Dijkstra algorithm computes $\delta^c(s,d)$ shortest path as defined above.*

Although we can prove the algorithm from scratch, we will rather build our proof on top of the correctness of the original Dijkstra algorithm [Cormen90]. In the course of the proof, we differentiate between w and c edge costs. We need the following lemmas:

Lemma 1 *BD-Dijkstra is a usual Dijkstra algorithm with respect to the edge cost function w. It, therefore, computes $\delta^w(s,d)$.*

Proof of Lemma 1 *The proof is based on the fact that each link is accessed by the algorithm only once*[1]*, by BD-Relax, whereby its cost value is determined and remains fixed throughout the rest of the computation. Therefore, after a first run of the algorithm, all edge costs w are determined (notice that w is non-negative, as required). Thus, one can easily see that the running of a regular Dijkstra algorithm on the edge costs just defined by BD-Relax is guaranteed to compute the same path as the one computed by BD-Dijkstra. From the regular Dijkstra algorithm, this path is the shortest path in w.*

Lemma 2 *Throughout the execution of the algorithm, for every vertex $v \in V$, $d[v]$, the current distance from the source s to vertex v, is non-increasing.*

Proof of Lemma 2 *The lemma follows from the regular Dijkstra algorithm.*

The last and most important lemma we need for the Theorem proof is:

Lemma 3 *During the execution of the algorithm, for every path $p(s,u)$ built by BD-Dijkstra, $d[u] = d_p^c(s,u)$.*

Proof of Lemma 3 *A path is built by sucessively calling BD-Relax, since this is the only place where $\pi[v]$ gets assigned. Using the previous lemma, it is easy to see that this assignment occurs only if, for each edge $(i,j) \in p$, $w(i,j) < \infty$. But then $\forall (i,j) \in p, w(i,j) = c(i,j)$. Summing up over all edges, we obtain $d[u] = d_p^c(s,u)$.*

The last lemma dictates that every path computed in w by BD-Dijkstra has identical length in c. Conversely, it is easy to see that any path with finite length in c has identical length in w. It remains to be proved that the minimum path computed in w by BD-Dijkstra is identical to the minimum path in c, or $\delta^w(s,d) = \delta^c(s,d)$.

[1] If a link could be relaxed more than once, even if in opposite directions, loops could be formed. But we know Dijkstra algorithm is loop free for non-negative edge costs

Proof of Theorem 2 *Suppose that the minimum paths, p_o^c, p_o^w for the two cost functions are different, thereby with different costs. We have:*

$$\delta^c(s,d) = \sum_{edge \in p_o^c} c(i,j) \tag{2}$$

$$\delta^w(s,d) = \sum_{edge \in p_o^w} w(i,j) \tag{3}$$

For sake of contradiction, assume:

$$\delta^c(s,d) < \delta^w(s,d)$$

By the previous lemma, however, path p_o^c has a cost given by:

$$\begin{aligned} d_{p_o^c}^c(s,d) &= \delta^c(s,d) \\ &= \sum_{edge \in p_o^c} w(i,j) \\ &< \delta^w(s,d) \end{aligned} \tag{4}$$

But this implies that there is a path, p_o^c, with lower cost in w than $\delta^w(s,d)$, which contradicts lemma 1.

The complexity of BD-Dijkstra algorithm is identical to the regular Dijkstra algorithm, and is $O(N \log N)$, where N is the number of vertices. However, it is worth noticing that the original Dijkstra algorithm outputs shortest paths from a source to all other network vertices, or a Shortest Path Tree, while BD-Dijkstra solves a *single shortest path* only. This is essentially due to the fact that the optimality principle may be violated in this problem. This principle basically states that subpaths of shortest paths are themselves shortest paths. More precisely, we can prove that:

Lemma 4 *In the bidirectional shortest path problem, subpaths of shortest paths are not necessarily shortest paths.*

Proof of Lemma 4 *Let $m(i,j)$ be the shortest path between vertices i and j, with respective bandwidth requirements B_i, B_j. Moreover, let k be an intermediate vertex on this shortest path, $k \in m(i,j)$, with bandwith requirement $B_k > B_j$. Let there be a link (r,q) on the subpath $p(i,k)$ of the shortest path $m(i,j)$ such that its edge bandwidth $b(q,r)$ is $B_k \geq b(q,r) \geq B_j$. Then, it is easy to see that path $p(i,k)$ is not even a feasible path connecting i and k.*

It is easy to see, therefore, that an all shortest bidirectional path with bandwidth constraints has $O(M^2 N \log N)$ complexity, where $M = |S \cup D|^2$.

3.2 Multicast Tree Problems

We now focus our attention to larger S and D sets. We are interested on a particular Multipoint Connectivity Structure (MCS), called multicast tree, which we now define:

Definition 4 *A multicast tree (mtree) $MT(E', V')$ is an* **acyclic** *MCS $\sigma(E', V'), E' \subseteq E, V' \subseteq V$ providing connectivity to every $m \in S \cup D$.*

One can easily see that an mtree is a Direct Acyclic Graph (DAG). Mtrees inherit the same feasibility definition as for any MCS. Regarding the construction of feasible mtrees, we may devise two problems:

Problem 1 *Construct a feasible mtree.*

Problem 2 *Construct a feasible mtree of minimum cost.*

Generic minimum cost tree problems are known as Steiner Tree problems. Steiner Tree problems are known to be NP-Complete. Problems of such nature with additional constraints are called constrained Steiner Tree problems. Our approach, therefore, is to provide polynomial time algorithms,

[2]We can prove that an all shortest bidirectional path with bandwidth constraints has $O(M^2 N \log N)$ complexity by using the known fact that a single shortest path has $O(N \log N)$ complexity plus lemma 4. A worst case analysis leads to the desired claim

Depth 1st BFS

4/18

So remove heap

just straight queue

queue nodes

queue distances

Still need to look up a nodes distance

"but from final only"

So both ordered and dict

Actually no ↓ distance!

"right

mark + enqueue

9/18
more

Ohh don't start w/ items on queue!

✓ works but way too slow
1.28 ~~ms~~ seconds → need 2ms

Add back math

✓ Now .048 sec → need .02

Oh just need dict for distance

Crystal i popping list is slow
so is keys of dict
just check it in

Removing keys .0089 ✓ woot!

Dict for distance, fix making
10089

②

Now .0084

¿Only 1 distance

~~add~~ ~~to~~ merge final and distance?

I think they are the same

seems to match

.0037 ✓ better

still too slow

---

for checking if in other

¿

---

kevin: check old ps4 sol

removed dup distance

.0035

But 2nd is now fast enough!

5 pts!

(3)

Got rid of a useless print

Still 5 pts

Mark can be distance

(So much cleaner w/o comments!)

Still 5 pts

.0023 now

So close!

Remove comments

Slow!, .003

So just cache same lookup!

Cache

.0023

(4)

Shi sent me 30/30 code

about levels ...

So he just saves levels

- short

He processed both s and t but

does not save distances

Oh if go up a path is blocked

directed only

Oh - forget it

```
import collections

collaborators = 'Your collaborators here'


# Enter true or false for each part of problem 1.

answer_for_problem_1_part_a = False
# An edge of weight -2 becomes weight 4, while an edge of weight 1 becomes weight 1.

answer_for_problem_1_part_b = False
# Paths may be different lengths

answer_for_problem_1_part_c = True
# The weight of a path from s to t is simply changed additively by d(s) - d(t)


# Enter true or false for each part of problem 2.

answer_for_problem_2_part_a = False
# If a non-source is chosen initially, its parents will come later

answer_for_problem_2_part_b = False
# Consider the graph (1, 2), (3, 2).

answer_for_problem_2_part_c = False
# This is equivalent to b

answer_for_problem_2_part_d = True
# See CLRS

answer_for_problem_2_part_e = True
# This is equivalent to d

# Enter your answer to problem 3 here.
answer_for_problem_3 = '''

    The main idea for this problem it to perform a
    modification of Bellman-Ford.
    First, before we perform this modification, we will transform the
    weight function of the graph to be the following: if w : E -> R+ is
    the original weight function, our new weight function will be given
    by f : E -> R such that f(e) = - log(w(e)). This way, the multiplication
    factor now becomes an additive factor in our graph, and we want to find
    a negative weighted cycle in this new graph, since this implies a cycle
    that gives us infinite money. But this cycle has to be reachable, since
    we have to respect the constraint that we have to always have more than
    b dollars. This condition is translated to the new graph setting as:
    the sum of the weights of a path that I decide to take can never be
    greater than log(1/b). Therefore, we apply the following modification
    of Bellman-Ford to this graph:
    1) Initialize-single-source(G,s)
```

```
    2) for i = 1 to |V|-1:
        for each (u,v) in E
            if (v.d > u.d + f(u,v)) and u.d + f(u,v) < log(1/b)
            # need to check in the if statement if we can take the path
                v.d = u.d + f(u,v)
                v.pi = u
    3) for each (u,v) in E
        if v.d > u.d + f(u,v) and v.d < infinity
            return True (because we found a reachable negative weight cycle)
    4) return False


    This algorithm is correct because all the rechable vertices will
    be the ones through paths that have all partial sums of weights
    (from the beginning up to a vertex in the path)
    less than log(1/b) and we will only find a negative cycle if it is
    reachable. The proof of this fact is similar to the one given in
    CLRS for the original Bellman-Ford algorithm, the only modification
    in the analysis being that now we will discard the unreachable vertices.
'''

def find_distance(graph, s, t):
  if s == t: return 0
  slist, tlist = [s], [t]
  visited_s, visited_t = set(slist), set(tlist)
  i = 0

  def extend_level(oldlist, visited, other_visited, i):
    i+= 1
    newlist = []
    for old in oldlist:
      for new in graph[old]:
        if new not in visited:
          if new in other_visited: return (None, None, i)
          visited.add(new)
          newlist.append(new)
    return (newlist, visited, i)

  while slist and tlist:
    (slist, visited_s, i) = extend_level(slist, visited_s, visited_t, i)
    if not visited_s: return i
    (tlist, visited_t, i) = extend_level(tlist, visited_t, visited_s, i)
    if not visited_t: return i

  return None
```

6.006 Recitation

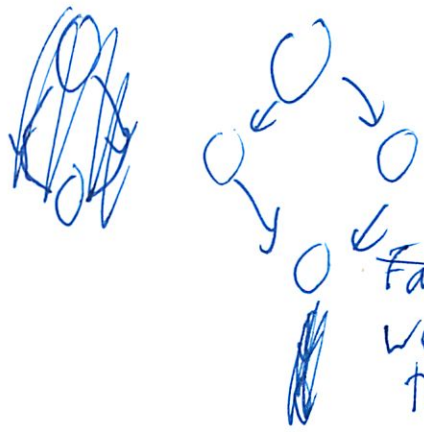
4/18

(1 on 1 P-set help)

Hint from other guys: start from last P-set code
Make double ended
↳ same check for repeats in both queues

note: no hashing tuples
instead keep 2 lists - node
- distance
well really 4 since both directions

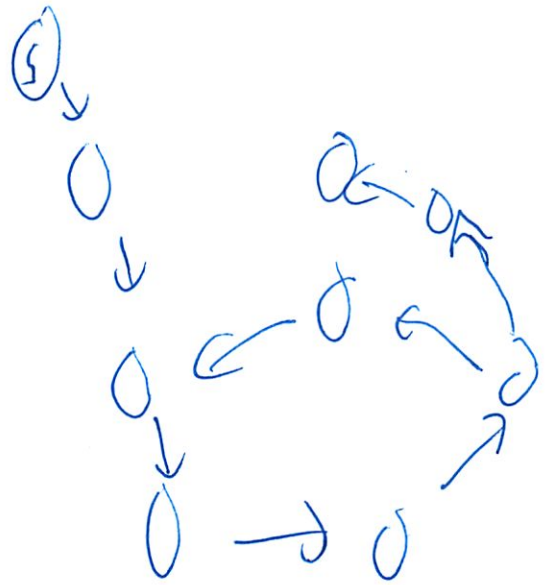
Gave my #3 hint to other guys
(they seemed to like -)
does it always find a cycle?
- if come across a different way?
- but isn't that a cycle



~~False - no cycle exists~~
would backtrack
then find

(2)

As you backtrack - erase the multiplier value
- set to False



TA: BFS - shortest path unweighted
DFS - ~~shortest p~~ topo sort

Are BFS/DFS perfectly substitutable?
Is $O(V+E)$ best you can do?
TA: Best worst case time

TA: Very hard to ~~the~~ modify DFS to use weights

2d trees as well

New material: <u>Dynamic programming</u>

- lecture tommorw

- technique for turning exponential time recurrences into polynomial time recurrences

- only 1 case when this works
  - when very small # of subproblems that ~~take~~ ~~it write~~ are called again + again
- essentially save results from sub-problems
  - use those results instead of recalculating
- "memoize"

Examples: Computing Fibinichi

```
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

nice sol

(4)

Runtime is proportional to # being returned
↳ is exponential in n

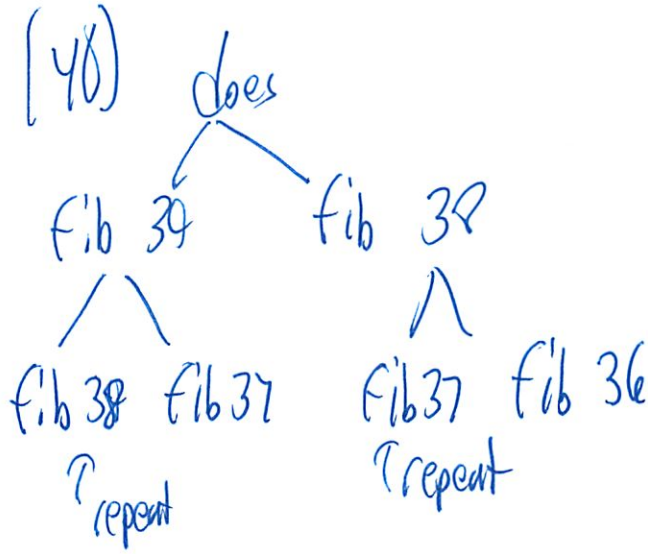
So its n sub problems for fib(n)

```
memo = {}
def fib(n)
    if n in memo
        return memo[n]
    if n < 2:
        return 1

    if n not in memo
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

It faster for every call
Repeat calls are even better

Since fib(40) does



This is linear

Old was exp. } asy faster

Runtine of a DP alg

$$O\left[(\#\text{subproblems}) * \left(\text{time to compute longest sub problem - assuming others are solved}\right)\right]$$

Can turn all DP alg into work-forward

Where it always grows/increases

Never runs out of stack space

(6)

Now: longest subsequence of ↑ #

- not ness. consecutive
- can have smaller ones in the middle

[10 5 3 4 9 7 8]

Nieve 2^n

- can either include or not include

But think DP!

Depth of recurrence is 2^n?

but only recurse on n^2 since prefix and <

If include 8, have

[5 3 4 7] ← take max(lis) + 1

← before 8 and less than 8

If don't include 8

[10 5 3 4 9 7] ← take lis

lis = longest ↑ subseq

How many subproblems?

This takes linear time per subproblem (compute lis)

next time $n \log n$

So total is n^3

6.006

Lecture 18

4/19

Midterm is Wed

See which room you are in

Today: Dynamic Programming

Fib #s

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

$\frac{F_n}{F_{n-1}} = \varphi$ (golden ratio)

↳ about beautiful things in nature

34 spirals on a sunflower clockwise

55 " counter

↑ two consec #s on Fib seq

How fast does F_n grow?

Master Theorem? – does not work here for divide + conquer

②

$F_n = F_{n-1} + F_{n-2} \geq 2F_{n-2}$

$= 2^{\Omega(n)}$ exponentially

But how quickly can we compute?

exponential → no we can do better later

fib-nieve(n)

base case

nieve(n-1) + nieve(n-2)

~~Exponential~~

$T(n) = T(n-1) + T(n-2)$

↑fib itself $O(F_n)$

↑exponential

But we are doing a <u>lot</u> of recomputing

So <u>memoize</u>!

```
memo = {}
fib(i)
    if i in memo, return memo[i]
    else compute w/ nieve fib
```

(3)

Only actually compute n times

So $O(n)$

Dynamic Programing

= Recursion + Memoization

- Works when the sol can be produced by combining sols of sub-problems
- the sol of subproblems can be produced by combing sols of sub-subproblems
- the totall # of subproblems is polynomial

$F_1, \ldots F_n$

Basically

Optimal substructure

Overlaping subproblems

Crazy 8s example

given a seq of cards $c[0] \ldots c[n-1]$
playing cards

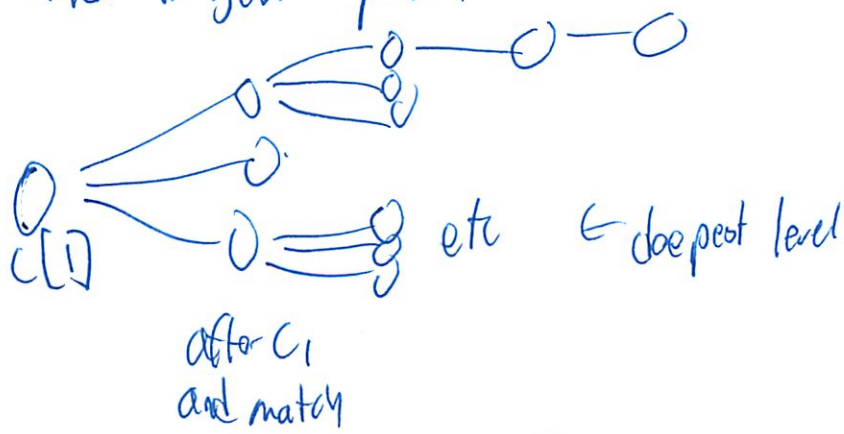
Find the longest trick subseq $c[i_1] \ldots c[i_k]$
where $i_1 < i_2 \ldots < i_k$

and must have same rank, suit, or 1 is an 8

skipping 7 hearts since can't jump to king
~~[illegible]~~ does not have to be consec.

(Similar to recitation - but I forgot what we did there...)

Could do via graph search BFS
- find longest path



(5)

Worst case BFS size

me: $n \cdot (n-1) \cdot (n-2) \cdot \ldots$

unsure

slides: $\geq 2^n$

Can DP save us from 2^n?

- are there subproblems? If yes, what are they?

trick(i) = lenght of ~~longest~~ longest trick that starts at c[i]

But how does this relate to trick(i+1) ... trick(n)

$$\text{trick}(i) = 1 + \max_{j > i,\ c[j] \text{ matches } c[i]} \text{trick}(j)$$

Max trick lenght

$$\max_i \text{trick}(i)$$

6

Recursive + Memoized

memo = { }

trick(i)

check memo

if i = n-1 return 1

else

f = 1 + Max$_{j>i}$ c[j] matches c[i] trick(j)

memo[i] = f

return f

n subproblems

n times to compute

$= n^2$

Iterative

no memo required - all in problem

for i = n-1 down to 0

memo[i] = 1 + max$_{j>i}$ c[j] matches c[i] memo[i]

- n subproblems n time each = $O(n^2)$

7

<u>Next time</u>: all-pairs shortest paths

- not just a single source
- no neg weight cycles

We could run Dij from every start

Have adj matrix

$d_{ij}(m)$ = weight of shortest path from $i \to j$ that uses <u>at most</u> m edges

Want $d_{ij}(n-1)$

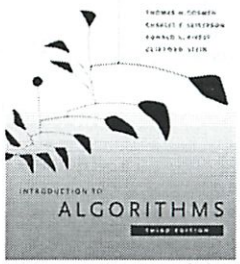
$d_{ij}(m) = \min_k \{ d_{ik}(m-1) + a_{kj} \}$

Picture in slides

But $d_{ij}(n-1)$ is $O(n^4)$

Will do better next time

6.006- *Introduction to Algorithms*



Lecture 18

Prof. Constantinos Daskalakis

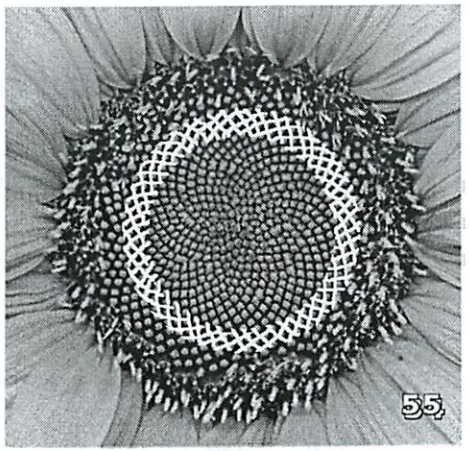
CLRS 15

Menu

- New technique: Dynamic Programming
  - Computing Fibonacci numbers – Warmup
  - "Definition" of DP
  - Crazy Eights Puzzle

Fibonacci Numbers

- Fibonacci sequence:
  - $F_0=0$, $F_1=1$
  - $F_n=F_{n-1}+F_{n-2}$
  - So $F_0=0$, $F_1=1$, $F_2=1$, $F_3=2$, $F_4=3$, $F_5=5$, $F_6=8$, $F_7=13,\ldots$
  - Interesting fact: $F_n/F_{n-1}\rightarrow \varphi$ (the golden ratio)
  - This is why if something looks beautiful in nature, chances are that it involves two consecutive Fibonacci numbers…



Clockwise Spirals: 34 Counter-clockwise Spirals: 55

34 and 55 are consecutive numbers in Fibonacci sequence…

Fibonacci Numbers

- Fibonacci sequence:
  - $F_0=0$, $F_1=1$
  - $F_n=F_{n-1}+F_{n-2}$
  - So $F_0=0$, $F_1=1$, $F_2=1$, $F_3=2$, $F_4=3$, $F_5=5$, $F_6=8$, $F_7=13,\ldots$
  - Interesting fact: $F_n/F_{n-1} \rightarrow \varphi$ (the golden ratio)
- How fast does F_n grow ?
  - $F_n=F_{n-1}+F_{n-2} \geq 2\,F_{n-2} \quad \Rightarrow F_n = 2^{\Omega(n)}$
- How quickly can we compute F_n?
  (time measured in arithmetic operations)

$F_n=F_{n-1}+F_{n-2}$

- Algorithm I: recursion

```
naive_fibo(n):
    if n=0: return 0
    else if n=1: return 1
    else:
      return naive_fibo(n-1) + naive_fibo(n-2)
```

- Time ? $T(n)=T(n-1)+T(n-2) = O(F_n)$
- Better algorithm ?

$F_n=F_{n-1}+F_{n-2}$

- Algorithm II: memoization

```
memo = { }
fibo(i):
    if i in memo: return memo[i]
    else if i=0: return 0
    else if i=1: return 1
    else:
        f = fibo(i-1) + fibo(i-2)
    memo[i]=f
    return f
return fibo(n)
```

- in the whole recursive execution, I will only go beyond this point, n times (since every time I do this, I fill in another slot in memo[])

- hence, all other calls to fibo() act as reading an entry of an array

- Time? O(n)

Dynamic Programming Definition

- DP ≈ Recursion + Memoization
- DP works when:
  - the solution can be produced by combining solutions of subproblems; $F_n=F_{n-1}+F_{n-2}$
  - the solution of each subproblem can be produced by combining solutions of sub-subproblems, etc;

moreover…. $F_{n-1}=F_{n-2}+F_{n-3}$ $\quad F_{n-2}=F_{n-3}+F_{n-4}$

  - the total number of subproblems arising recursively is polynomial. $F_1, F_2,\ldots, F_n$

Dynamic Programming Definition

- DP ≈ Recursion + Memoization
- DP works when:

Optimal substructure
The solution to a problem can be obtained by solutions to subproblems. $F_n=F_{n-1}+F_{n-2}$

moreover.... $F_{n-1}=F_{n-2}+F_{n-3}$ $F_{n-2}=F_{n-3}+F_{n-4}$

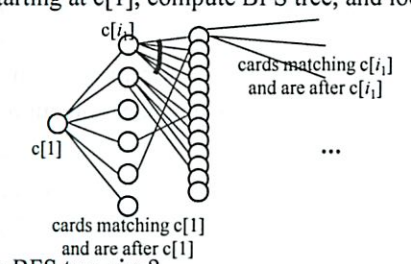
Overlapping Subproblems
A recursive solution contains a "small" number of distinct subproblems (repeated many times) $F_1, F_2, \ldots, F_n$

Crazy 8s

- **Input:** a sequence of cards c[0]…c[n-1].
- E.g., 7♣ 7♥ K♣ K♠ 8♥
- **Goal:** find the longest "trick subsequence" $c[i_1]\ldots c[i_k]$, where $i_1 < i_2 < \ldots < i_k$.
- For it to be a trick subsequence, it must be that: $\forall j$, $c[i_j]$ and $c[i_{j+1}]$ "match" i.e.
  - they either have the same rank,
  - or the same suit
  - or one of them is an 8
  - in this case, we write: $c[i_j] \sim c[i_{j+1}]$
- E.g., 7♣ K♣ K♠ 8♥ is the longest such subsequence in the above example

Crazy 8s via graph search

- Longest trick starting at c[1]?
- **Idea:** BFS was good for shortest paths in unweighted graphs. Let's try it for finding a longest path in the graph of matching cards.
- Do BFS starting at c[1], compute BFS tree, and look at deepest level.



- Worst case BFS tree size?
- e.g. 7♣ 10♥ 7♣ 2♣ 5♣ 7♣ 2♣ 5♣10♥ 7♣ 2♣ 5♣ 7♣ 2♣ 5♣…
- size $\geq 2^n$

DP Approach

- Identify subproblem:
- Let trick(i) be the length of the longest trick subsequence that starts at card c[i]
- **Question:** How can I relate value of trick(i) to the values of trick(i+1),…,trick(n)?
- Recursive formula:

$$\text{trick}(i) = 1 + \max_{j>i,\ c[j] \text{ matches } c[i]} \text{trick}(j)$$

- Maximum trick length:

$$\max_i \text{trick}(i)$$

Implementations

Recursive

- memo = { }
- trick(i):
  - if i in memo: return memo[i]
  - else if i=n-1: return 1
  - else
    - f := 1+max$_{j>i,\ c[j] \text{ matches } c[i]}$ trick(j)
    - memo[i] := f
    - return f
- call trick(0), trick(1),…,trick(n-1)
- return maximum value in memo

Implementations (cont.)

Iterative

memo = { }
for i=n-1 downto 0
  memo[i]= 1+max$_{j>i,\ c[j] \text{ matches } c[i]}$ memo[j]
return maximum value in memo

Runtime: $O(n^2)$

Dynamic Programming

- DP ≈ Recursion + Memoization
- DP works when:

Optimal substructure
An solution to a problem can be obtained by solutions to subproblems.
trick(i) = 1+ max$_{j>i,\ c[j] \text{ matches } c[i]}$ trick(j)

moreover….

Overlapping Subproblems
A recursive solution contains a "small" number of distinct subproblems (repeated many times)
trick(0), trick(1),…, trick(n-1)

Menu

- New technique: Dynamic Programming
  - Computing Fibonacci numbers – Warmup
  - "Definition" of DP
  - Crazy Eights Puzzle
  - Next Time: **all-pairs shortest paths**

All-pairs shortest paths

- **Input:** Directed graph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \longrightarrow \mathbb{R}$.
- **Output**: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

Assumption: No negative-weight cycles

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that uses <u>at most</u> m edges
- Want: $d_{ij}^{(n-1)}$

Claim: We have

$d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

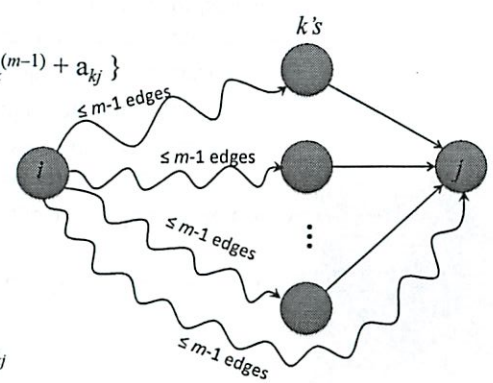
and for $m = 1, 2, \ldots, n-1$,

$d_{ij}^{(m)} = \min_k\{d_{ik}^{(m-1)} + a_{kj}\}$.

Proof of Claim

$d_{ij}^{(m)} = \min_k\{d_{ik}^{(m-1)} + a_{kj}\}$

for $k \leftarrow 1$ to n
  if $d_{ij} > d_{ik} + a_{kj}$
    $d_{ij} \leftarrow d_{ik} + a_{kj}$



"Relaxation" (recall Bellman-Ford lecture)

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that uses <u>at most</u> m edges
- Want: $d_{ij}^{(n-1)}$

Claim: We have

$d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

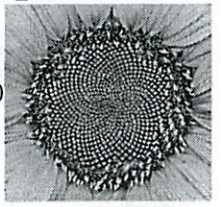
and for $m = 1, 2, \ldots, n-1$,

$d_{ij}^{(m)} = \min_k\{d_{ik}^{(m-1)} + a_{kj}\}$.

Time to compute $d_{ij}^{(n-1)}$? $O(n^4)$ - similar to n runs of Bellman-Ford

Something less extravagant? Next Lecture

Inventor of Fibonacci Sequence?



- Is it Fibonacci?
- where Fibonacci: Italian Mathematician (1170 1250)
- A: No. Fibonacci just introduced it to Europe.
- Sequence was known to Indian Mathematicians since the 6th century.
- So is it some Indian mathematician?
- That's more of a philosophical question.
- Same as question: Who invented the prime numbers some Greek, Egyptian or Babylonian?
- After all, these numbers play a role in natural systems that existed before humans…

6.006 Recitation

4/20

Longest increasing subsequence (cont)

[10 5 3 4 9 7 8 6]

O(exponential) ↑ answer heirse way

$O(n^3)$ DP way to solve

$lis(x) = \max(lis(y \text{ in } x[:-1] \text{ if } y < x[-1])) + 1$

↑ include or exclude last #

$lis(x[:-1])$

# of subproblems x lenght of subproblems

↑ not obvious

What is a subproblem?

Only request on special subseq – the ~~problem~~ points

you start ~~[illegible]~~ and the points < than that

So n^2 # of subproblems

②

Subproblems parametized by index i at which ~~we~~ we cut off

And by index j at which we filter for being $< x[j]$

Explain sublist as parameters

- n choices for parameters
- n places to filter

Generally working backwards

Then time to answer a subproblem $= n$
↳ time to ~~conca~~ combine the list

(still confused what we're doing - need to visualize step by step)

↳ Ashed

[10 5 3 4] left of 4, < 4

max(lis([3]) + 1) ← choose 4

or lis([10, 5, 3]) ← or skip it

③

$lis([3]) = 1$ ← base case

next step

$lis([10,5,3]) = \max(lis([\,]) + 1,$ ← empty case = 0 left of 3 and < 3

$+ lis([10,5]))$ ← will be 1

One more

$lis([10,5]) = \max(lis([\,]) + 1,$ ← left of 5, < 5 = 0

$lis([10]))$ ← 0 ← so 1

Important to paramitize ~~answers~~ subproblems
- helps you count

<u>Don't</u> create new elements of the problem
We should use previous answers
* Pass entire input, w/ coordinates
  ↳ better in theory

Like w/ Binary Search at start of year

(4)

~~So then time to answer 1 problem → 1~~ can be

But we need diff subproblems

~~[10 5 3 4 9 7 8 6]~~

[10 4 5 3 9 7 8 6]

What is subproblem?

Figure out recurrence later

$dp[i]$ = longest increasing subseq that ends w/ $x[i]$

Then solve for every i

$$= \max_{\substack{j \le i-1 \\ x[j] < x[i]}} \left(dp[j] + 1 \right)$$

~~#~~ time for subproblems n

subproblems n ← should try to ↓

(5)

The right way to do subseq problems

In general, longest subseq problems where you can check adj. condition elm by elm can be done in $O(n^2)$

$[a_1 \quad a_2 \quad a_3 \quad \cdots \quad]$

just local comparison

Iterate over all eliments

Take longest seq to that point

Special condition: increasing (recitation - just now)

Special condition: crazy 8 (lecture)

↳ check if same suit or rank

Since local comparison

↳ $O(n^2)$

(6)

For particular conditions, we can often speed up single problem loop

---

Often can answer subproblems in < $O(n)$

increasing $O(\log n)$

Crazy 8 $O(1)$

Involves some special data structure

heaps
BSTs
hashing
one value

BST - since dealing w/ comparison of ordered keys = els of the list

dp[i] does not just depend on dp[i-1]

7

Remember to the left and $< i$

$j < i$ $\quad x[j] < x[i]$

key → Insert $x[i]$ when I calc $dp[i]$

Need to be able to find maximal $dp[j]$

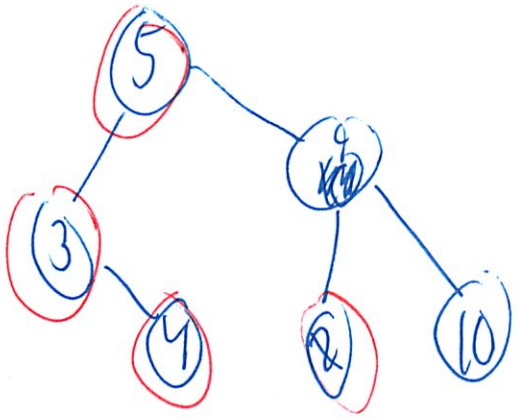
value → So augment each node w/ $dp[j]$ and w/ $\max(dp[k])$ for k in its subtree

Want largest value that is $<$ key

Can search in $O(\log n)$ time

So # subproblems n
time for each $\log n$ $\Big\}$ $O(n \log n)$

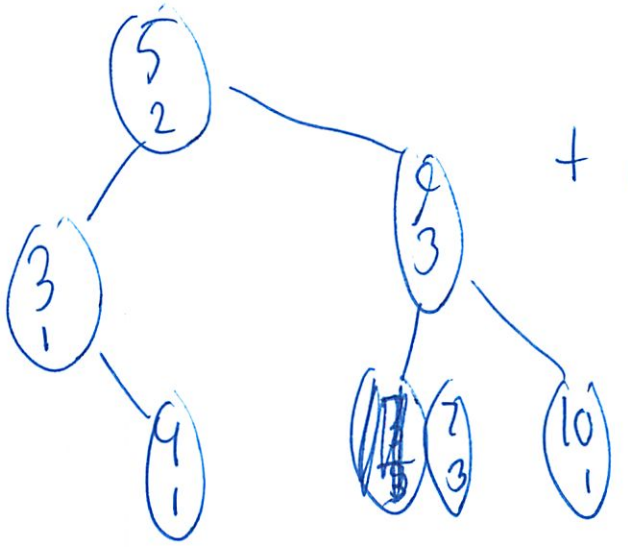
8



? so < 8

So finds max of values of these

So [10 4 5 3 9 7 8 6]
dp 1 1 2 1 3



+ augmentation - the max of itself, left, right subtrees

max < 8 = 3

↑ On a test, can treat this as a black box
Think about crazy 8 (use hashing)