Midterm 2 review

Lecture 22

Part I

Recursion: Divide and Conquer

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Binary search, Merge sort, quick sort, multiplication, median selection.

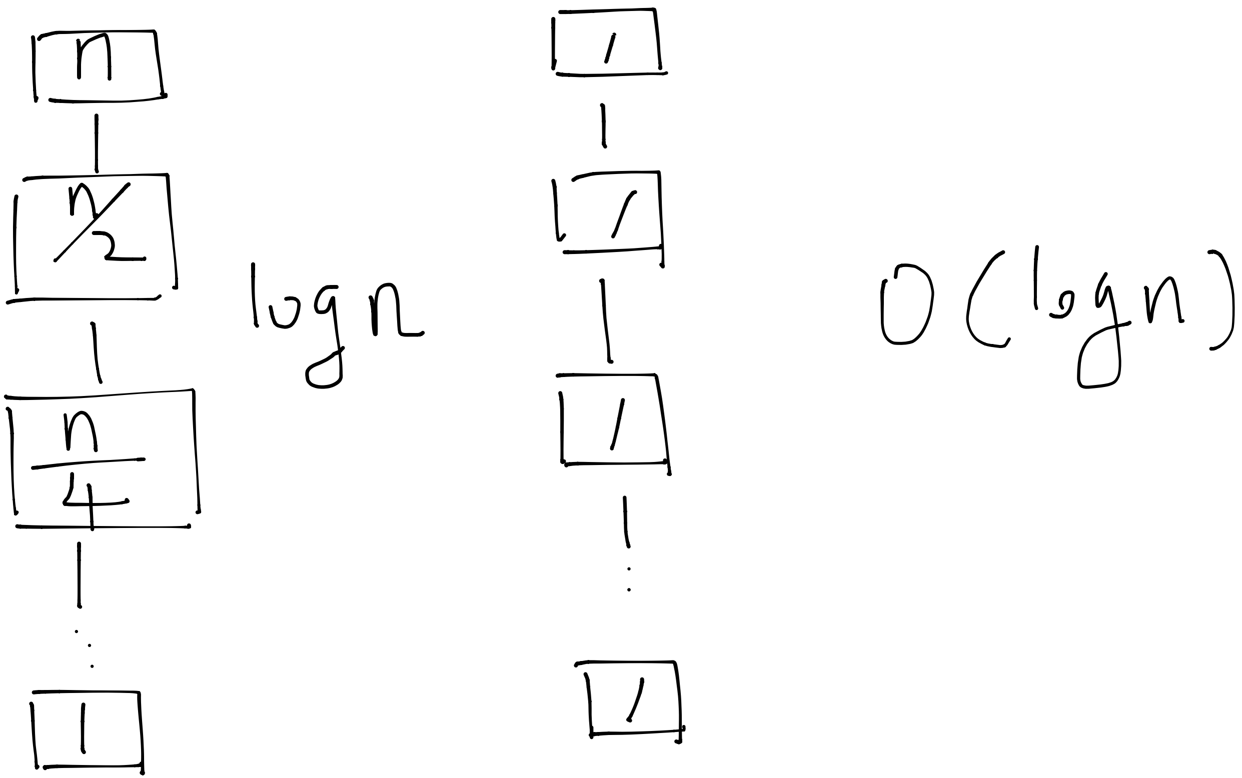
   Each sub-problem is a fraction smaller.

Binary Search

1. Discard half every time

Binary Search

1. Discard half every time
2. Recurrence tree



Binary Search

1. Discard half every time
2. Recurrence tree
3. Which condition to check?

Binary Search

Suppose you are given two sorted arrays $A[1 .. n]$ and $B[1 .. n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1 .. 8] = [0, 1, 6, 9, 12, 13, 18, 20]$$

$$B[1 .. 8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

your algorithm should return the integer **9**.

6

$A_1 < 4$ of elements

$A_1 <$ median

$\leq A_3$

$A_1 \quad B_1$

$A_2 < B_2$

$A_3 \quad B_3$

$A_1 \leq A_2 < B_2 \leq B_3$

Binary Search

Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20]$$

$$B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

your algorithm should return the integer 9.

Compare the two medians.

Binary Search

$\underline{\text{MEDIAN}(A[1..n], B[1..n]):}$

```
if n < 10^100
    use brute force
else if A[n/2] > B[n/2]
    return MEDIAN(A[1..n/2], B[n/2+1..n])
else
    return MEDIAN(A[n/2+1..n], B[1..n/2])
```

Binary Search

```
MEDIAN(A[1..n], B[1..n]):
  if n < 10^100
      use brute force
  else if A[n/2] > B[n/2]
      return MEDIAN(A[1..n/2], B[n/2+1..n])
  else
      return MEDIAN(A[n/2+1..n], B[1..n/2])
```

Because we discard the same number of elements from each array, the median of the remaining subarrays is the median of the original $A \cup B$.

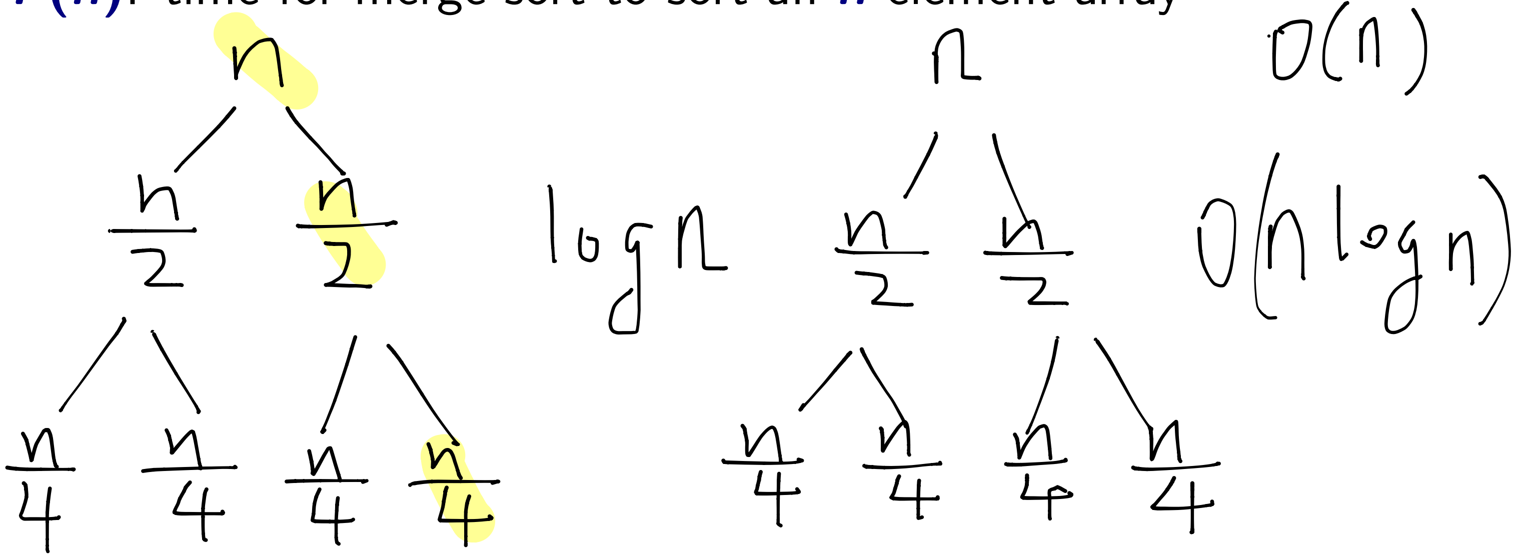
Sorting

1. Divide into two halves. Together takes $O(n)$ time.

Sorting

1. Divide into two halves. Together takes O(n) time.
2. Recurrence tree

$T(n)$: time for merge sort to sort an n element array



Sorting

1. Divide into two halves. Together takes O(n) time.
2. Recurrence tree

$\boldsymbol{T(n)}$: time for merge sort to sort an $\boldsymbol{n}$ element array

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

Karatsuba's Algorithm

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Karatsuba's Algorithm

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Karatsuba's Algorithm

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means

Karatsuba's Algorithm

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

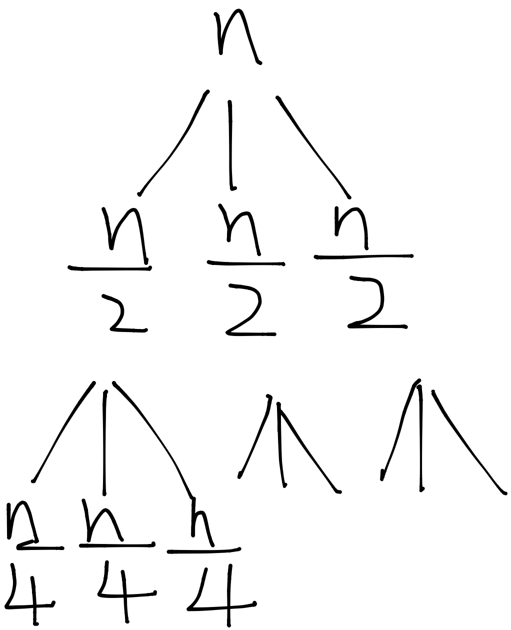
Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Recursion tree analysis



log n

$$\sum_{i=0}^{\log n} \frac{3^i}{2^i} n$$

↑ increasing

$$O\left(\left(\frac{3}{2}\right)^{\log n} n\right)$$

$$\frac{3^{\log n}}{n} n = 3^{\log n} = 2^{(\log 3)(\log n)}$$
$$= n^{\log 3}$$

Selecting in Unsorted Lists

1. One-armed Quick-sort

Selecting in Unsorted Lists

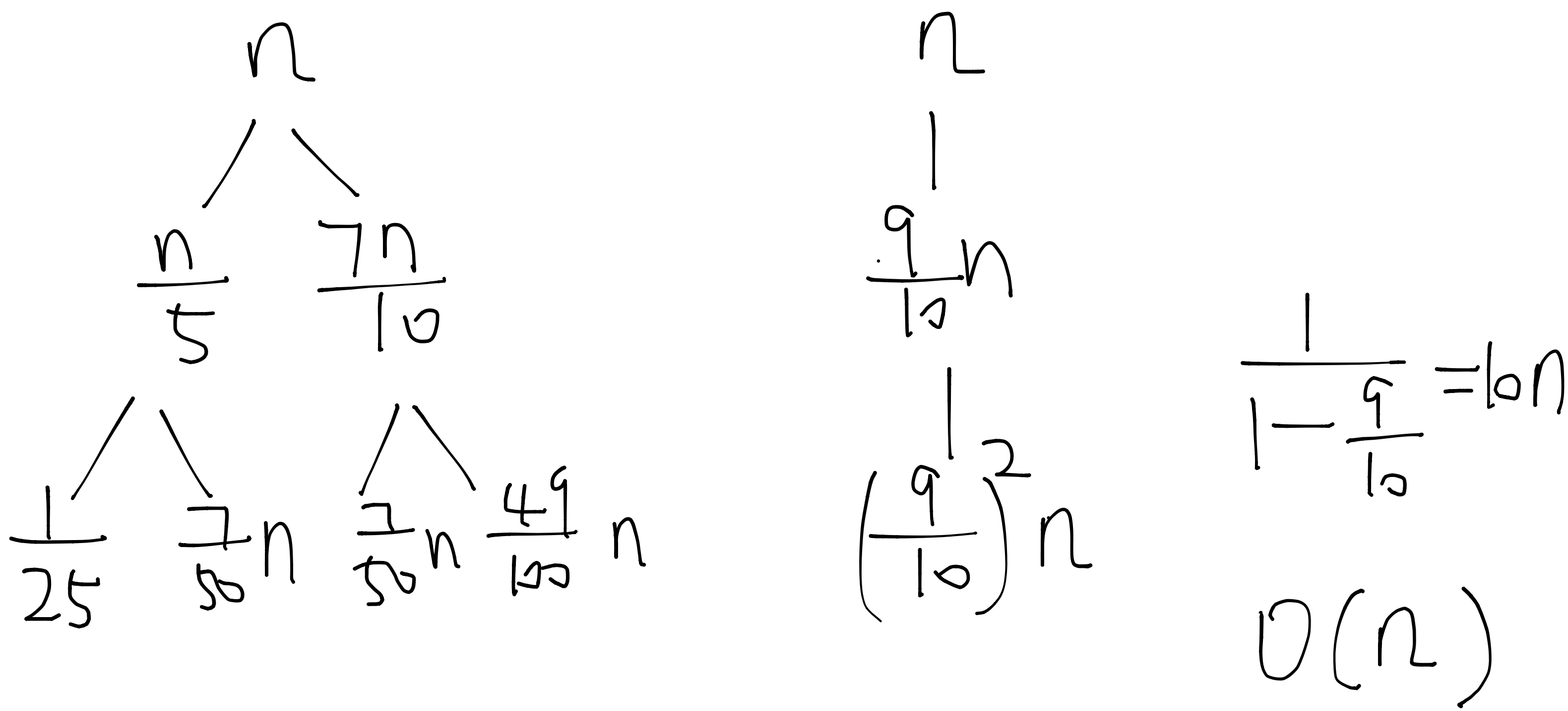
1. One-armed Quick-sort
2. With a good pivot (median of the medians)

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + O(n)$$

and

$$T(n) = O(1) \qquad n < 10$$

Recursion tree analysis



Part II

Dynamic programming

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

   Each sub-problem is a fraction smaller.

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

   Each sub-problem is a fraction smaller.

2. **Backtracking**: A sequence of decision problems. Recursion tries all possibilities at each step.

   Each subproblem is only a constant smaller, e.g. from n to $n - 1$.

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

   Each sub-problem is a fraction smaller.

2. **Backtracking**: A sequence of decision problems. Recursion tries all possibilities at each step.

   Each subproblem is only a constant smaller, e.g. from n to $n - 1$.

3. **Dynamic Programming**: Smart recursion with memoization

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Recursion!

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Recursion!

Text segmentation: All possibilities for next word

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Recursion!

Text segmentation: All possibilities for next word

LIS: Two possibilities: Include the current number or not

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Recursion!

Text segmentation: All possibilities for next word

LIS: Two possibilities: Include the current number or not

Edit distance: Three possibilities: align the two letters, or each align with a gap

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Recursion!

Text segmentation: All possibilities for next word

LIS: Two possibilities: Include the current number or not

Edit distance: Three possibilities: align the two letters, or each align with a gap

Max-Weight Independent Set in Trees: Two possibilities: Include the root or not

How to design DP algorithms

1. Find a “smart” recursion (The hard part)
   1. Formulate the sub-problem
   2. so that the number of distinct subproblems is small; polynomial in the original problem size.

How to design DP algorithms

1. Find a “smart” recursion (The hard part)
   1. Formulate the sub-problem
   2. so that the number of distinct subproblems is small; polynomial in the original problem size.

2. Memoization
   1. Identify distinct subproblems
   2. Choose a memoization data structure
   3. Identify dependencies and find a good evaluation order
   4. An iterative algorithm replacing recursive calls with array lookups
   5. Further optimize space

Which data structure?

- Text segmentation, suffix, 1-D array
- Longest increasing subsequence, suffix+index, 2-D array
- Edit distance, two prefixes, 2-D array
- Max-Weight Independent Set in Trees, tree

Part III

Graphs

Path and cycle

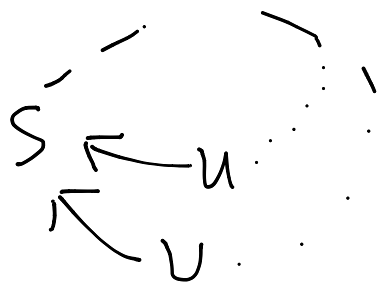
A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path) and the path is from v_1 to v_k.
Note: a single vertex u is a path of length 0.

Path and cycle

A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path) and the path is from v_1 to v_k.
Note: a single vertex u is a path of length 0.

A cycle is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ and ~~$\{v_1, v_k\}$~~ $\in E$. Single vertex not a cycle according to this definition.

$\{v_k, v_1\}$



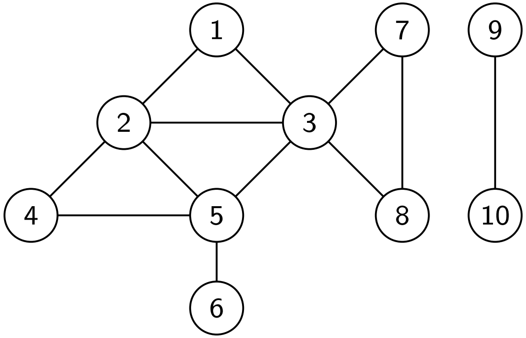
$d(s \to u) + w(u \to s)$

$d(s \to v) + w(v \to s)$

min

Connectivity on Undirected Graphs

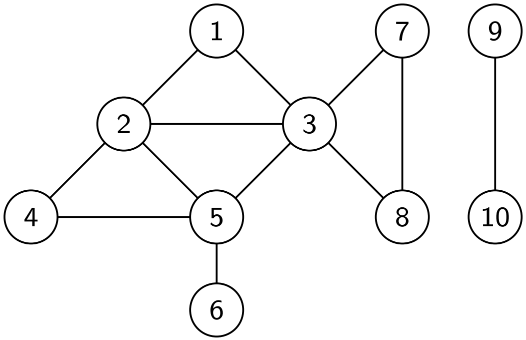
Given a graph $G = (V, E)$:



A vertex u is connected to v if there is a path from u to v.

Connectivity on Undirected Graphs

Given a graph $G = (V, E)$:

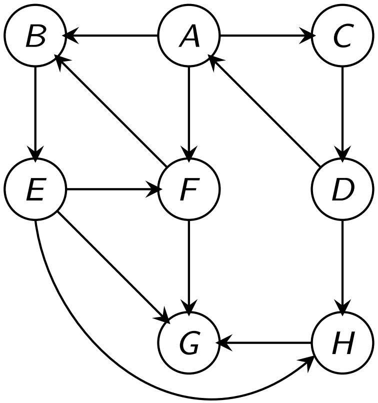


A vertex u is connected to v if there is a path from u to v.

The connected component of u, $\text{con}(u)$, is the set of all vertices connected to u.

Directed Connectivity

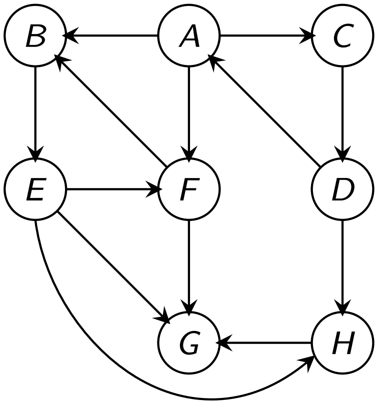
Given a graph $G = (V, E)$:



A vertex u can reach v if there is a path from u to v.

Directed Connectivity

Given a graph $G = (V, E)$:



A vertex u can reach v if there is a path from u to v.

Let $\mathbf{rch}(u)$ be the set of all vertices reachable from u.

Asymmetricity: D can reach B but B cannot reach D

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to v.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to v.

Proposition

C is an equivalence relation, that is reflexive, symmetric and transitive.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

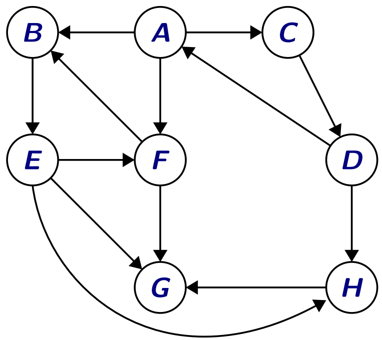
Define relation C where uCv if u is (strongly) connected to v.

Proposition

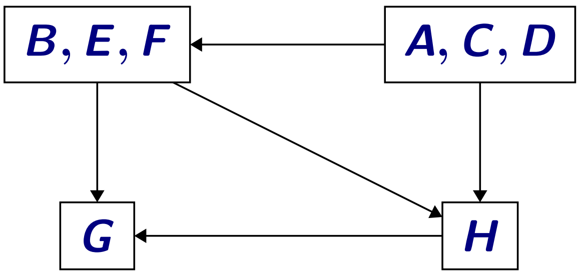
C is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of C: *strong connected components* of G.
They *partition* the vertices of G.
$\mathrm{SCC}(u)$: strongly connected component containing u.

Structure of a Directed Graph



Graph G



Graph of SCCs G^{SCC}

Reminder

G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph G^{SCC} is a DAG.

DAG Properties

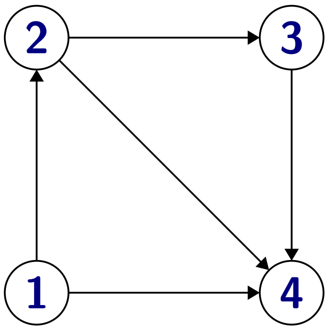
Proposition

Every DAG G *has at least one source and at least one sink.*

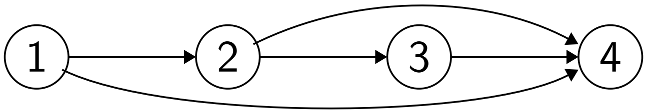
Proposition

A directed graph G *can be topologically ordered iff it is a* DAG.

Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition

A **topological ordering**/**topological sorting** of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.

Informal equivalent definition:

One can order the vertices of the graph along a line (say the x-axis) such that all edges are from left to right.

DAGs and Topological Sort

What does it mean?

DAGs and Topological Sort

What does it mean?

Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

DAGs and Topological Sort

What does it mean?

Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

Case 1: DAG. Heat a pizza $\longrightarrow$ eat the pizza, have a Coke.

DAGs and Topological Sort

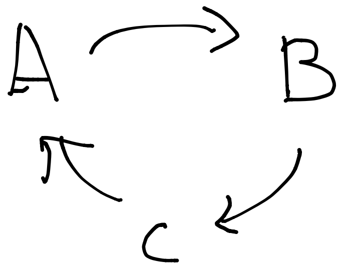
What does it mean?

Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

Case 1: DAG. Heat a pizza $\longrightarrow$ eat the pizza, have a Coke.
Case 2: Circular dependence.



DAGs and Topological Sort

What does it mean?

Consider a dependency graph.

Topological ordering

Find an order of events in which all dependencies are satisfied.

Case 1: DAG. Heat a pizza $\longrightarrow$ eat the pizza, have a Coke.
Case 2: Circular dependence.

Application: Given pairwise ranking, find an overall ranking that satisfies all pairwise ranking.

Part IV

Graph Search

Basic Search

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

```
Explore(G,u):
    array Visited[1..n]
    Initialize:  Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List:  ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
    Output S
```

Running time: O(n+m)

Properties of Basic Search

Proposition

On an undirected graph, **Explore**(G, u) *terminates with* $S = con(u)$.

Proposition

On a directed graph, **Explore**(G, u) *terminates with* $S = rch(u)$.

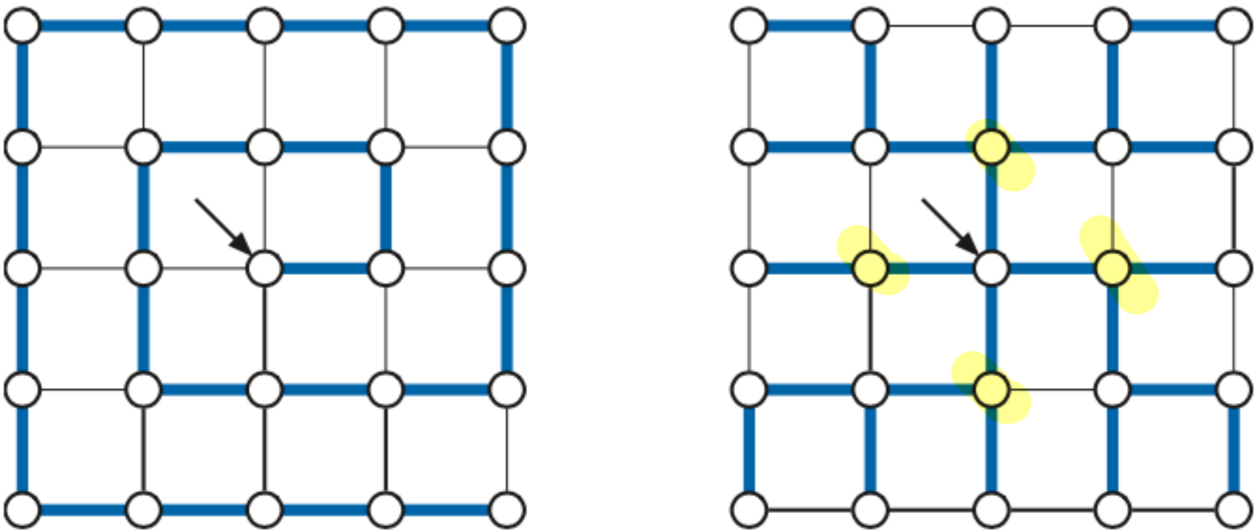
Properties of Basic Search

DFS and **BFS** are special case of BasicSearch.

1. Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***
2. Breadth First Search (**BFS**): use queue data structure to implementing the list ***ToExplore***

Spanning tree

A depth-first and breadth-first spanning tree.



Algorithms via Basic Search-II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

Definition (Reverse graph.)

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Algorithms via Basic Search-II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

Definition (Reverse graph.)

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\text{rch}(u)$ in G^{rev}!

1. **Running time:** $O(n + m)$ to obtain G^{rev} from G and $O(n + m)$ time to compute $\text{rch}(u)$ via Basic Search.

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$

Hence, $\mathrm{SCC}(G, u)$ can be computed with $\mathit{Explore}(G, u)$ and $\mathit{Explore}(G^{rev}, u)$. Total $O(n + m)$ time.

Algorithms via Basic Search - IV

1. Is G strongly connected?

Algorithms via Basic Search - IV

1. Is G strongly connected?

Pick arbitrary vertex u. Check if $\mathrm{SCC}(G, u) = V$.

DFS with Visit Times

Keep track of when nodes are visited.

```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
    T is set to ∅
    time = 0
    while ∃unvisited u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each uv in Out(u) do
        if v is not marked then
            add edge uv to T
            DFS(v)
    post(u) = ++time
```

An Edge in DAG

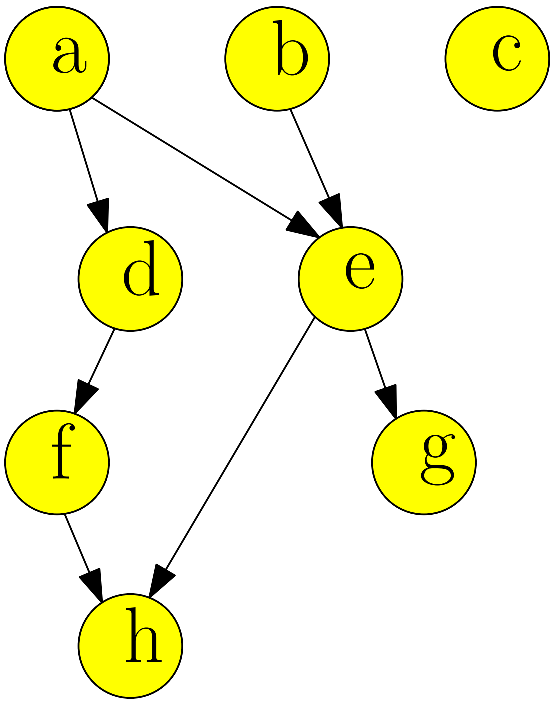
Proposition

If G is a DAG *and* $\mathrm{post}(u) < \mathrm{post}(v)$, *then* (u, v) *is not in* G.
i.e., for all edges (u, v) *in a* DAG, $\mathrm{post}(u) > \mathrm{post}(v)$.

post $>$ $>$ $>$ $>$

u_1 u_2 u_3 $\ldots$ u_n

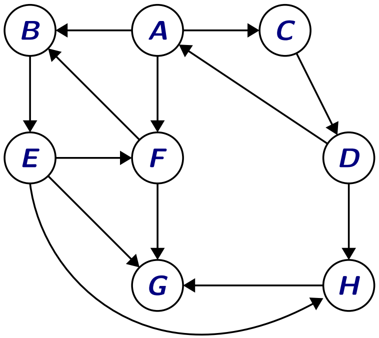
Reverse post-order is topological order



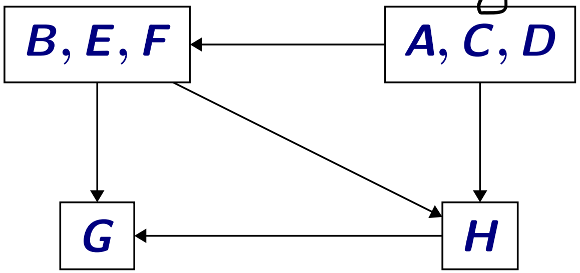
Sort SCCs

The SCCs are topologically sorted by arranging them in decreasing order of their highest post number.

highest post



Graph G



Graph of SCCs G^{SCC}

DFS post.

Linear Time Algorithm

...for computing the strong connected components in **G**

```
do DFS(G^rev) and output vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Theorem

Algorithm runs in time $O(m + n)$ and correctly outputs all the SCCs of G.

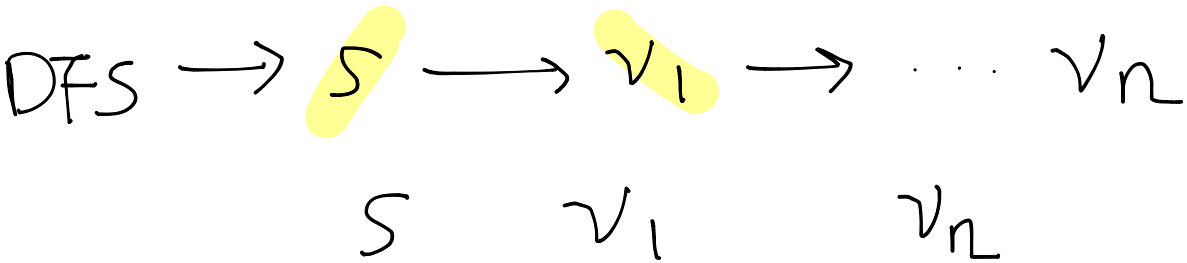
Using DAG and SCC

A node u is good if it can reach every node in V. Describe a linear-time algorithm to find if there is a good node in G.

Using DAG and SCC

A node u is good if it can reach every node in V. Describe a linear-time algorithm to find if there is a good node in G.

1. First consider a DAG.



Using DAG and SCC

A node u is good if it can reach every node in V. Describe a linear-time algorithm to find if there is a good node in G.

1. First consider a DAG.
2. For any directed graph, construct the meta-graph G^{SCC}, which is a DAG.

Using DAG and SCC

A node u is good if it can reach every node in V. Describe a linear-time algorithm to find if there is a good node in G.

1. First consider a DAG.
2. For any directed graph, construct the meta-graph G^{SCC}, which is a DAG.
3. The good node, if exists, has to be in the source SCC.

u

Part V

Shortest Path in Graphs

Breadth First Search (BFS)

Overview

(A) **BFS** is obtained from **BasicSearch** by processing edges using a data structure called a **queue**.

(B) It processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex).

BFS finds *shortest distance* starting from s on unweighted graphs.

Non-negative edge length: Dijkstra

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = {s},
for i = 2 to |V| do
    (* Invariant:  X contains the i − 1 closest nodes to s *)
    Among nodes in V − X, find the node v that is the
            i'th closest to s
    Update dist(s, v)
    X = X ∪ {v}
```

Dijkstra's Algorithm using Priority Queues

```
Q ← makePQ()
insert(Q, (s, 0))
for each node u ≠ s do
    insert(Q, (u, ∞))
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d′(s, u) is shortest path distance from s to u
     using only X as intermediate nodes*)
X ← ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {v}
    for each u in Adj(v) do
        decreaseKey(Q, (u, min(dist(s, u), dist(s, v) + ℓ(v, u)))).
```

Running time: $O((m+n)\log n)$ with heaps and $O(m + n\log n)$ with advanced priority queues.

One negative edge: Use Dijkstra

Compute the shortest path from s to t on a graph with exactly one negative edge $x \rightarrow y$.

One negative edge: Use Dijkstra

Compute the shortest path from s to t on a graph with exactly one negative edge $x \to y$.

1. Detect if there is a negative length cycle.

One negative edge: Use Dijkstra

Compute the shortest path from s to t on a graph with exactly one negative edge $x \to y$.

1. Detect if there is a negative length cycle.
    1. Remove the negative edge: G'.

One negative edge: Use Dijkstra

Compute the shortest path from s to t on a graph with exactly one negative edge $x \to y$.

1. Detect if there is a negative length cycle.
    1. Remove the negative edge: G'.
    2. Compute the shortest distance $y \to x$ on G'.

One negative edge: Use Dijkstra

Compute the shortest path from s to t on a graph with exactly one negative edge $x \to y$.

1. Detect if there is a negative length cycle.
   1. Remove the negative edge: G'.
   2. Compute the shortest distance $y \to x$ on G'.
2. Suppose no negative length cycle, find shortest distance by

$$dist(s,t) = \min \left\{ \begin{array}{c} dist'(s,t) \\ dist'(s,u) + w(u \to v) + dist'(v,t) \end{array} \right\}$$

Negative-length edges: Bellman-Ford Algorithm

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

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

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

Running time: $O(mn)$

Bellman-Ford: Negative Cycle Detection

Check if distances change in iteration n.

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

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

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

Algorithm for DAGs

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for DAGs

Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

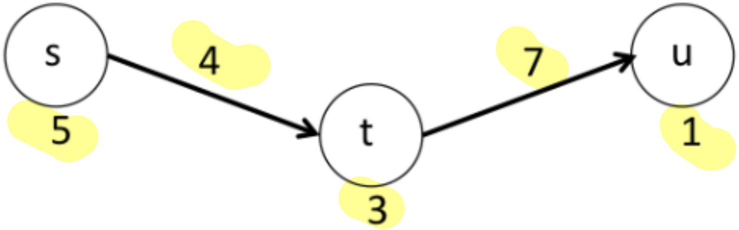
```
for i = 1 to n do
        d(s, v_i) = ∞
d(s, s) = 0

for i = 1 to n − 1 do
        for each edge (v_i, v_j) in Out(v_i) do
                d(s, v_j) = min{d(s, v_j), d(s, v_i) + ℓ(v_i, v_j)}

return d(s, ·) values computed
```

Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

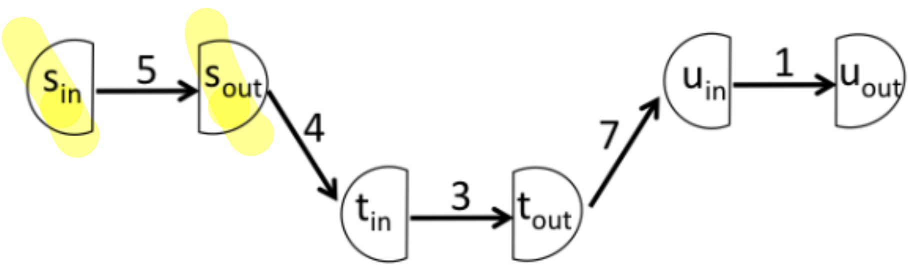
Part VI

Graph reduction and tricks

Split nodes



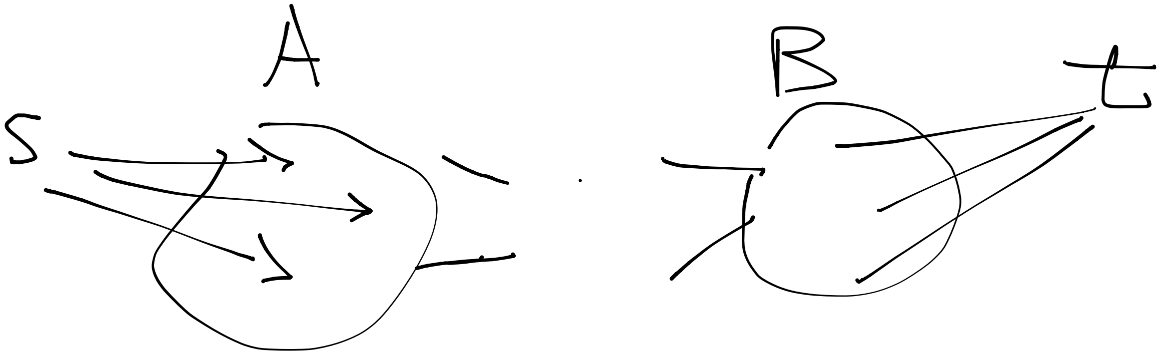
original graph
with vertex weights



new graph
with only edge weights

Add nodes

Given a graph $G = (V, E)$ and two disjoint sets of nodes $A, B \subset V$, is there a path from some node in A to some node in B?



Add nodes

Given a graph $G = (V, E)$ and two disjoint sets of nodes $A, B \subset V$, is there a path from some node in A to some node in B?

Connect s to each node in A, and t to each node in B. This becomes the basic $s - t$ reachability problem.

DP on graphs

Q: How to compute the shortest distance between s and t with at most k hops?

DP on graphs

Q: How to compute the shortest distance between s and t with at most k hops?

Ans: We arrived at Bellman-Ford by considering the shortest distance with at most k hops.

$d(u, k)$

DP on graphs

Q: How to compute the shortest distance between s and t with at most k hops?

Ans: We arrived at Bellman-Ford by considering the shortest distance with at most k hops.

Q: A subset of risky ~~nodes~~ edges $E' \subset E$. Find shortest path from s with at most h risky edges.

Ans: Use Bellman-Ford style DP. Consider which $u \to v$ edge to include for each v.

$d(s, u_1) \to v \quad + w(u_1, v)$

$d(s, u_2)$

$d(s, u_3)$

$\vdots$

DP on graphs

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

Ans: Use Bellman-Ford style DP. Consider which $u \rightarrow v$ edge to include for each v. Remove the risky nodes to form G'.

DP on graphs

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

Ans: Use Bellman-Ford style DP. Consider which $u \to v$ edge to include for each v. Remove the risky nodes to form G'.

$$d(v,i,j) = \min \begin{cases} d(v,i-1,j) \\ d(v,i,j-1) \\ \min_{(u,v)\in E'} \; d(u,i-1,j-1) + \ell(u,v) \\ \min_{(u,v)\in E-E'} \; d(u,i-1,j) + \ell(u,v) \end{cases}$$

risky

not risky

$u \to v$

DP on graphs

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

Ans: Use Bellman-Ford style DP. Consider which $u \rightarrow v$ edge to include for each v. Remove the risky nodes to form G'.

$$d(v,i,j) = \min \begin{cases} d(v,i-1,j) \\ d(v,i,j-1) \\ \min_{(u,v)\in E'} \; d(u,i-1,j-1) + \ell(u,v) \\ \min_{(u,v)\in E-E'} \; d(u,i-1,j) + \ell(u,v) \end{cases}$$

Base case: Use Bellman-Ford to compute $d(v,i,0)$, shortest distance on G' with no risky edge.
Running time: $O(mnk)$.

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

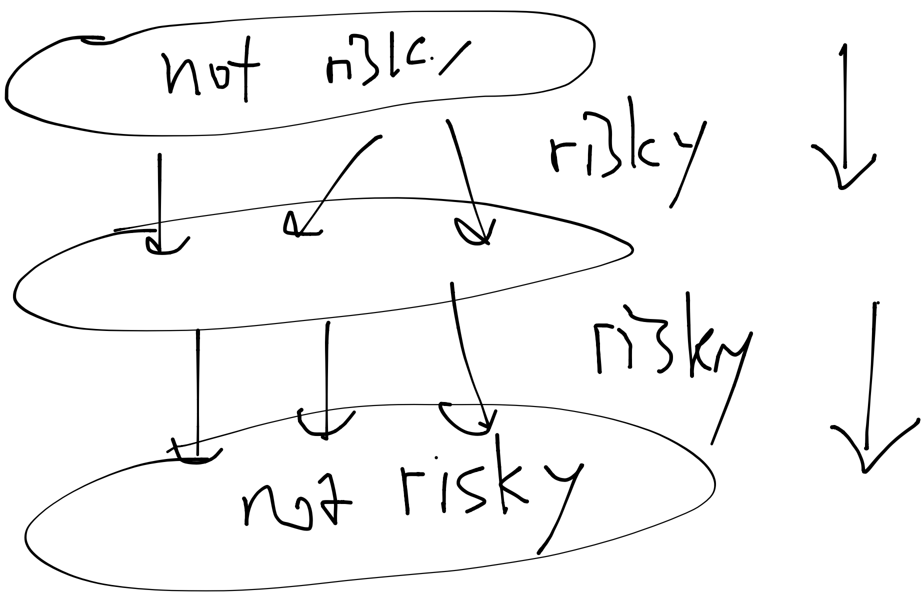
→ remove risky

1. Create $h + 1$ copies of G': $G_0, G_1, \ldots, G_h$

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

1. Create $h+1$ copies of G': $G_0, G_1, \dots, G_h$
2. Include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G.



Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

1. Create $h + 1$ copies of G': $G_0, G_1, \ldots, G_h$
2. Include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G.
3. The idea is that the only way a path can move from one copy of G' to the next is by traversing a risky edge.

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

1. Create $h+1$ copies of G': $G_0, G_1, \ldots, G_h$
2. Include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G.
3. The idea is that the only way a path can move from one copy of G' to the next is by traversing a risky edge.
4. Run Dijkstra's algorithm on this new graph, from vertex s_0, the copy of s in G_0, to $v_0, \ldots, v_h$ be the corresponding vertices in copies $G_0, \ldots, G_h$.

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

1. Create $h + 1$ copies of G': $G_0, G_1, \ldots, G_h$
2. Include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G.
3. The idea is that the only way a path can move from one copy of G' to the next is by traversing a risky edge.
4. Run Dijkstra's algorithm on this new graph, from vertex s_0, the copy of s in G_0, to $v_0, \ldots, v_h$ be the corresponding vertices in copies $G_0, \ldots, G_h$.
5. $d(s_0, v_i)$ is just the shortest path from s to v in the original graph G that uses exactly i risky edges.

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

1. Create $h + 1$ copies of G': $G_0, G_1, \ldots, G_h$
2. Include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G.
3. The idea is that the only way a path can move from one copy of G' to the next is by traversing a risky edge.
4. Run Dijkstra's algorithm on this new graph, from vertex s_0, the copy of s in G_0, to $v_0, \ldots, v_h$ be the corresponding vertices in copies $G_0, \ldots, G_h$.
5. $d(s_0, v_i)$ is just the shortest path from s to v in the original graph G that uses exactly i risky edges.
6. the distance from s to v in the original graph that uses at most h risky edges is just $\min_{0 \le i \le h} d(s_0, v_i)$.

Layering

Q: A subset of risky nodes $E' \subset E$. Find shortest path from s with at most h risky edges.

1. Create $h+1$ copies of G': $G_0, G_1, \ldots, G_h$
2. Include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G.
3. The idea is that the only way a path can move from one copy of G' to the next is by traversing a risky edge.
4. Run Dijkstra's algorithm on this new graph, from vertex s_0, the copy of s in G_0, to $v_0, \ldots, v_h$ be the corresponding vertices in copies $G_0, \ldots, G_h$.
5. $d(s_0, v_i)$ is just the shortest path from s to v in the original graph G that uses exactly i risky edges.
6. the distance from s to v in the original graph that uses at most h risky edges is just $\min_{0 \le i \le h} d(s_0, v_i)$.

Running time: $O(mk + nk \log(nk))$