CMSC 451 - Algorithm Design
Lecture 1 - Introduction

About this course -

- Second course in algorithm design (after 351)
- Fundamental elements of algorithm design
  - design techniques (greedy, divide + conquer, ...)
  - proving correctness
  - analyzing running times
- Theoretical focus - no programming projects

Overview:

- Graph basics, DFS, + shortest paths
- Greedy algorithms
- Dynamic programming
- Network flows
- NP-Hardness + Approximation algorithms

Prerequisites - (CMSC 351)

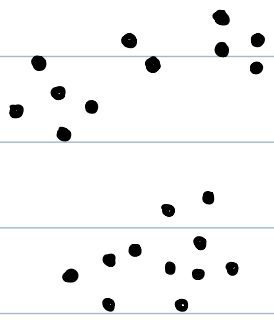
- Basic programming
- Discrete math - induction, sets, probability, ...
- Sorting + basic data structures
- Math - logs + exponentials, calculus, linear algebra

Algorithm Design:

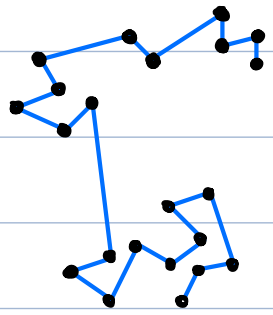
Given a well defined computational problem design an efficient procedure for solving it.

Example 1: Given a set of points in the plane compute a path of min length that visits all the points.

Input:

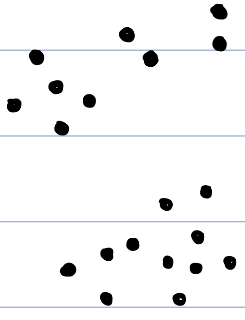


Output:

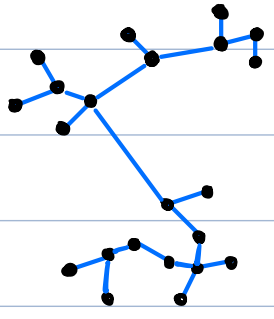


Example 2: Given a set of points in the plane compute a network of min length that connects all the points.

Input:



Output:



These are superficially quite similar, but computational complexities are quite different.

Example 1: Euclidean Traveling Salesman Problem
- NP-Hard
- Can be efficiently approximated

Example 2: Euclidean Minimum Spanning Tree
- Solvable exactly in $O(n^2)$ time
- Practical approximation in $O(n \log n)$ time

Measures of computational complexity -
- Running time
- Space

... as a function of input size (denoted n)

But many inputs of same size...
- Worst-case - max over all inputs of size n
- Average-case - expected case for inputs of size n from some distribution

→ More realistic, but a lot harder!

Asymptotic Notation:

- Simplify complex functions
- Focus on trend for large n
- Ignore constant factors

Example:

$$T(n) = 3.9n + 4.17 \cdot n\log n + 3.5n^2$$

$$\cong 3.9n + 4.17 \cdot n\log n + \boxed{3.5n^2} \quad \text{- large } n$$

$$\cong \cancel{3.9n} + \cancel{4.17 \cdot n\log n} + 3.5\boxed{n^2} \quad \text{- ignore const factors}$$

$$\cong \cancel{3.5n^2}$$

$T(n)$ is $O(n^2)$ "on the Order of"

↳ Usually written $T(n) = O(n^2)$ but not formally correct. (But we'll do it anyway)

"O" is shorthand for asymptotically "$\leq$"

Also:

Θ	↔	"$=$"	Big theta
o	↔	"$<$"	Little o
Ω	↔	"$\geq$"	Big Omega
ω	↔	"$>$"	Little omega

Important Complexity Classes - For some constant $c > 0$

$\leq (\log n)^c = \log^c n$ - **Polylogarithmic** time

$\leq n^c$ - **Polynomial** time

$\leq c^n \quad (c > 1)$ - **Exponential** time

Asymptotic Ordering - for any $a, b, c > 0 \quad (c > 1)$

$$\log^a n \leq n^b \leq c^n$$

Summary:

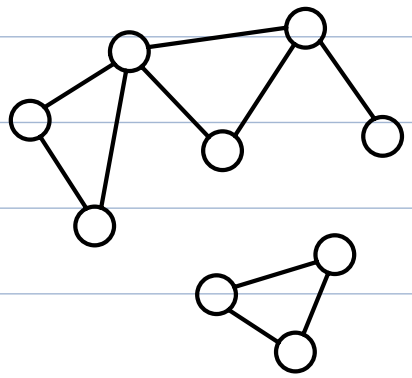
- Basics of algorithm design
- Asymptotic Notation

CMSC 451 - Algorithm Design
Lecture 2 - Graph Basics

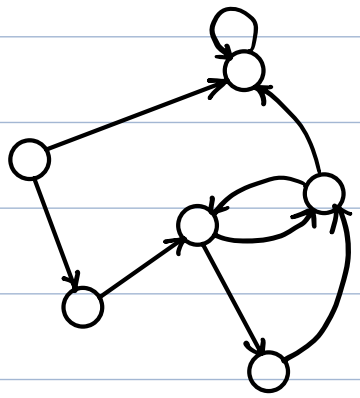
Graph - A discrete structure representing nodes (vertices) joined by links (edges)

Edges may either be:

undirected - or - directed



(Undirected) Graph



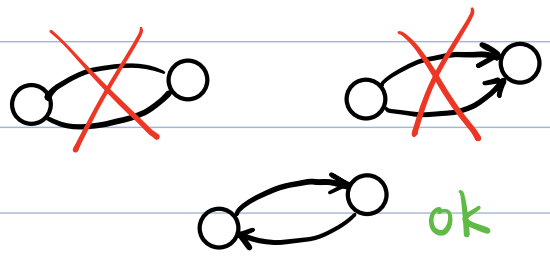
Directed Graph (Digraph)

Definition:

(Undirected) Graph: $G=(V,E)$ is a finite set V of vertices and a set E of unordered pairs of vertices, called edges

Directed Graph (or digraph) $G=(V,E)$ is a finite set V of vertices and a set E of ordered pairs of vertices, called edges

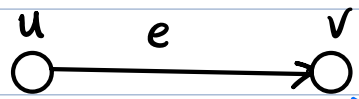
This definition rules out:



- multiple instances of the same edge

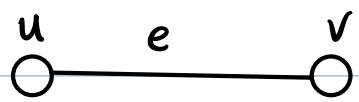
- self loop edges for undirected graphs

Terminology:



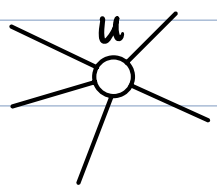
e's origin

e's destination



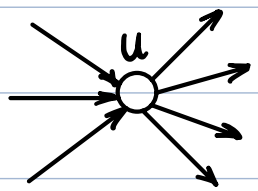
- e is incident on u & v
- u is adjacent to v

Graph degree:



degree = 5

Digraph degree:



out-degree = 4

in-degree = 3

Identities:

$$\sum_{u \in V} \text{in-deg}(u) = \sum_{u \in V} \text{out-deg}(u) = |E|$$

$$\sum_{u \in V} \deg(u) = 2|E|$$

How many edges? $n = |V|$ $m = |E|$

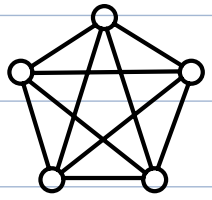
Digraph: $0 \le m \le n^2$

Graph: $0 \le m \le \binom{n}{2} = \frac{n(n-1)}{2}$

A graph/digraph is **sparse** if $m = O(n)$

$\Rightarrow$ **average vertex degree** is $O(1)$

Complete graph $m = \binom{n}{2}$

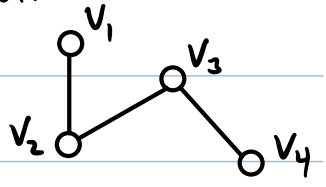


K_5

Paths & Connectivity:

Path - Sequence $v_1, v_2, \ldots, v_k$, s.t. $(v_i, v_{i+1}) \in E$

Length = num. of edges = k-1



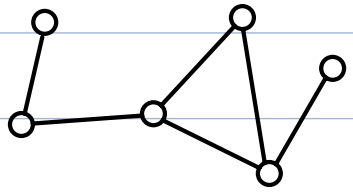
Cycle - $v_1 = v_k$

Simple - No **vertex or edge** is repeated

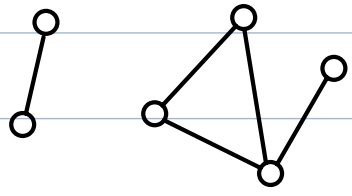
u is **reachable** from v if

there is a **path from v to u**

Graph is **connected** if all vertex pairs reachable

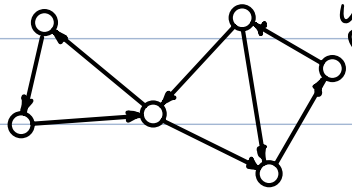


yes

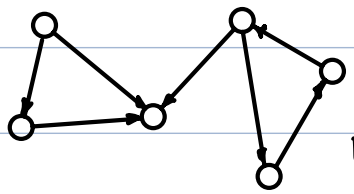


no

Digraph is **strongly connected** if $\forall u, v \in V$ $u \rightsquigarrow v$ + $v \rightsquigarrow u$



yes



no

Q: What is the **min. number of edges** in a **connected graph** on n vertices?

Q: What is the **min. number of edges** in a **strongly-connected digraph** on n vertices?

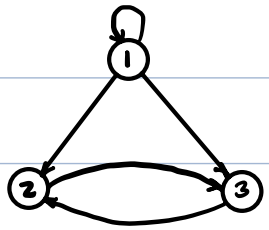
Representations: Let $G = (V, E)$ be a graph/digraph

$n = |V|$ and $m = |E|$ $\quad V = \{1, 2, \ldots, n\}$

Adjacency Matrix: An **n×n matrix** A

$$A[v,w] = \begin{cases} 1 & \text{if } (v,w) \in E \\ 0 & \text{o.w.} \end{cases}$$

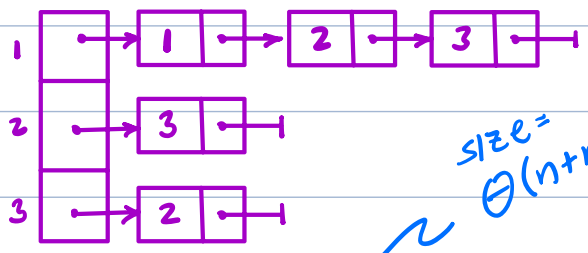
size = $\Theta(n^2)$



A:

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

Adj:



size = $\Theta(n+m)$

Adjacency List: Array Adj[n] where Adj[v] is **head of a linked list** of v's **neighbors**

Depth-First Search:

- A systematic process for visiting the vertices and edges of a graph.
- Induces a tree structure on the graph
- Intuition - Go vertex to vertex, backtracking only when all neighbors have been visited
- Additional information to guide search:

  for $u \in V$:

  mark[u] - undiscovered → visited → finished

  d[u] - when was u first discovered?

  f[u] - when was u finished?

  p[u] - the predecessor, the vertex that discovered u

```
DFSVisit(Vertex u)
    mark[u] ← visited      // u is discovered
    d[u] ← ++time
    for each (v ∈ Adj[u])  // check neighbors
        if (mark[v] = undiscovered) // first time
            pred[v] ← u    // set predecessor
            DFSVisit(v)    // ... then visit
    mark[u] ← finished     // done with u
    f[u] ← ++time
```

DFS(G): time ← 0

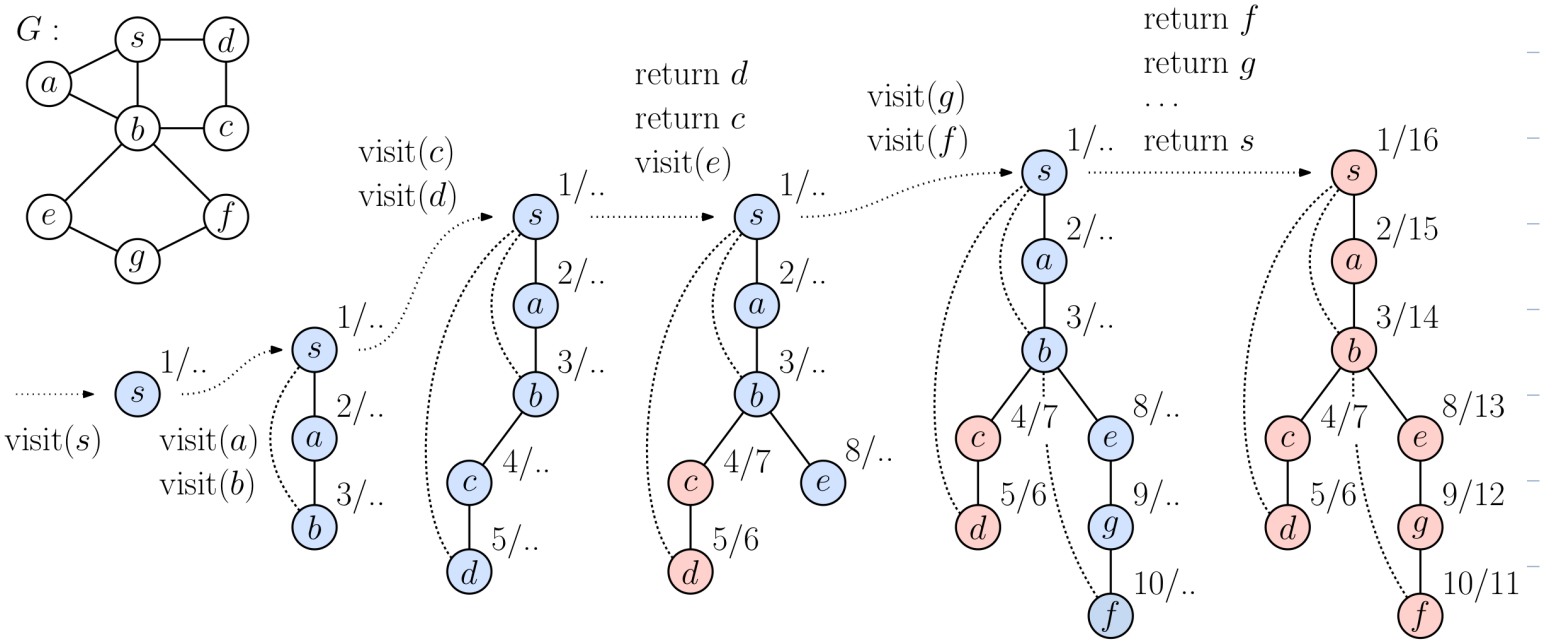
mark all vertices undiscovered

as long as there is an undiscovered vertex v

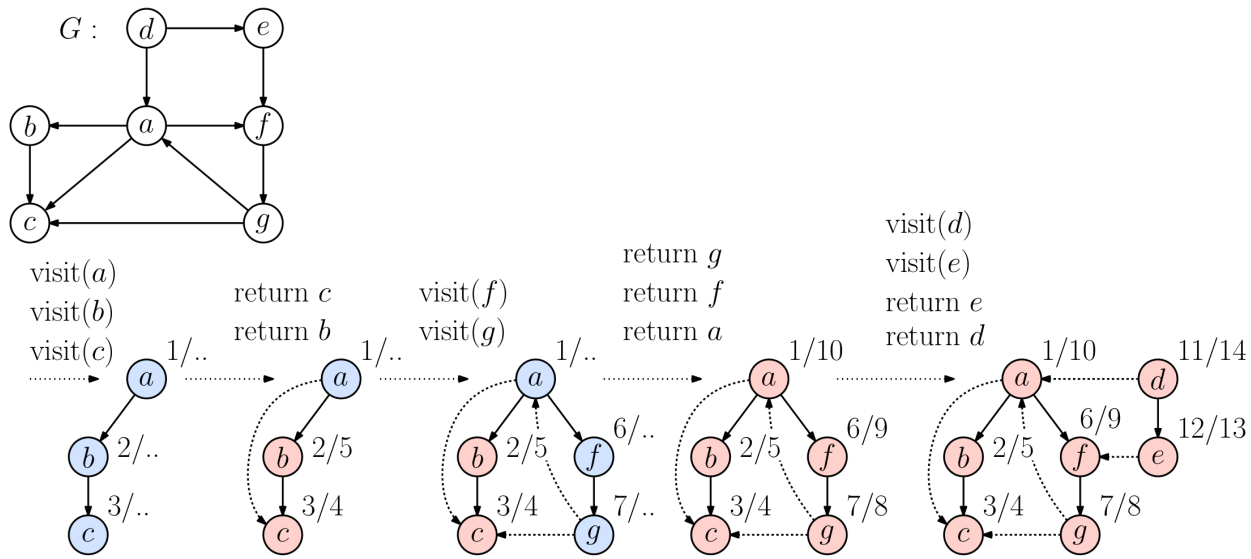
DFSvisit(v)

Example: (Undirected)

○ discov/finish



Example (Directed)



Analysis:

DFS runs in time $O(n+m)$ $[n=|V|, m=|E|]$

- We hit each vertex once: $O(n)$
- We visit each neighboring vertex: $O(\deg(u))$
  - Total:

$$\sum_{u \in V} \deg(u) = 2|E| = O(m)$$

- Total: $O(n+m)$

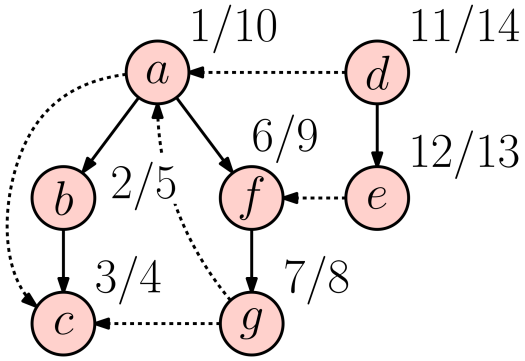
Hierarchical Structure:

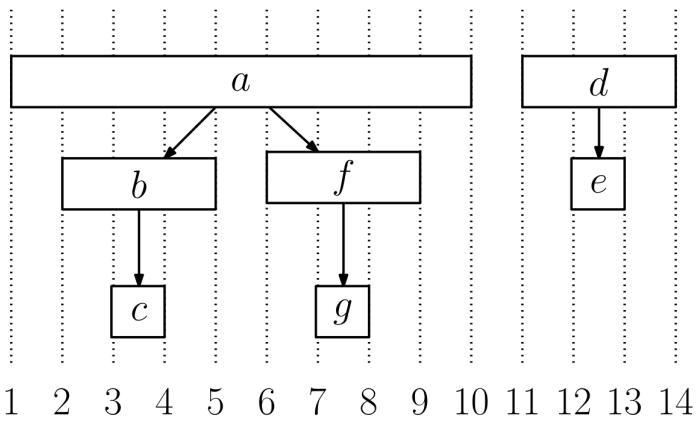
$d[u]$ = discovery time
$f[u]$ = finish time

Parenthesis Lemma:

For all $u, v \in V$ in DFS tree:

- u is descendant of $v \Leftrightarrow [d[u], f[u]] \subseteq [d[v], f[v]]$
- " " ancestor " " " " " $\supseteq$ "
- otherwise $[d[u], f[u]]$ is disjoint from $[d[v], f[v]]$



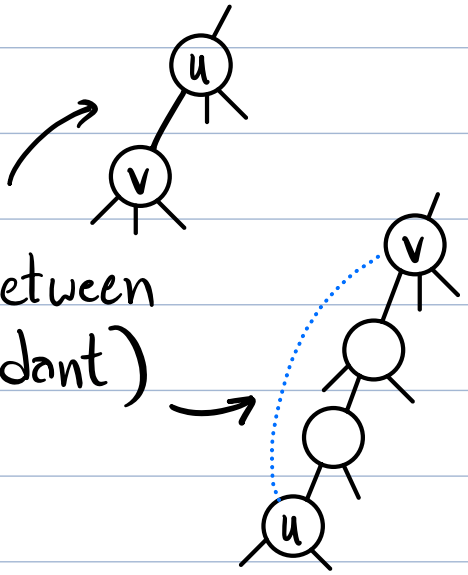


Edge Classification: Tree structure induces distinct edge types. Edge (u, v)

Undirected Graph:

Tree edge: u discovers v

Back edge: Anything else (between ancestor + descendant)



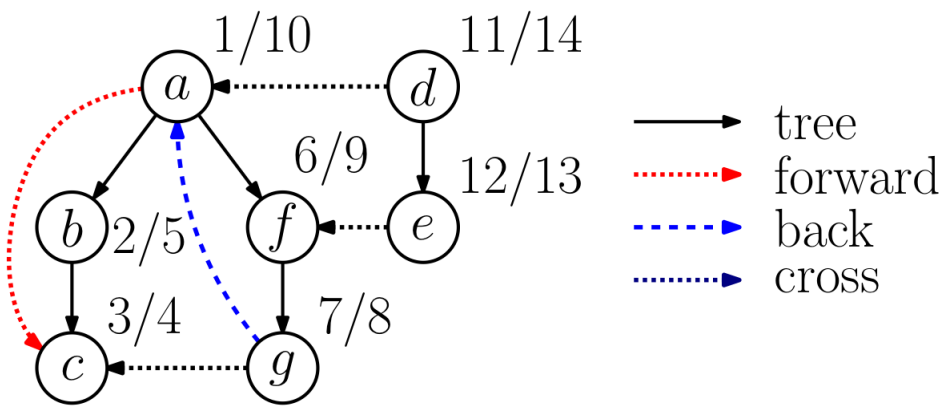
Directed Graph:

Tree edge: u discovers v

Back edge: v is an ancestor of u (includes self loops)

Forward edge: v is a non-child descendant of u

Cross edge: Anything else

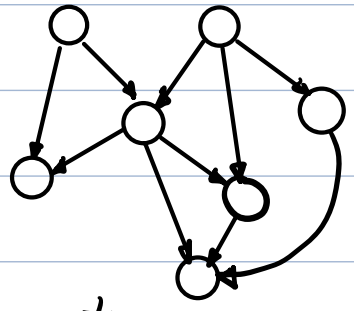


Q: Classify on edge (u,v) as tree, cross, back, forward based on discovery, finish, pred values

Summary:

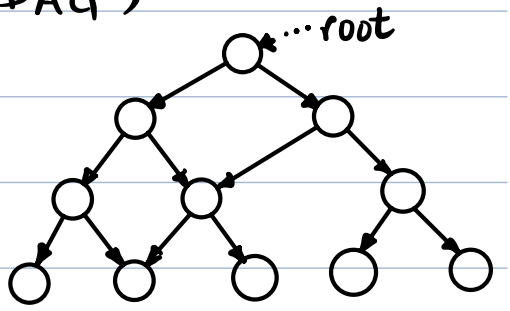
- Basic graph concepts, definitions, props.
- Depth-First Search
  - Discovery / Finish times
  - Edge classification

CMSC 451 - Algorithm Design
Lecture 3 - Cycles + Strong Components

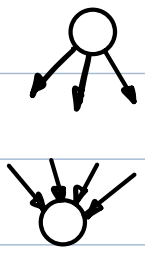


Directed Acyclic Graphs (DAG):

- Arises in:
  - scheduling precedence constraints
  - data structures (rooted DAG)
  - ...



- Source - in-degree = 0
- Sink - out-degree = 0



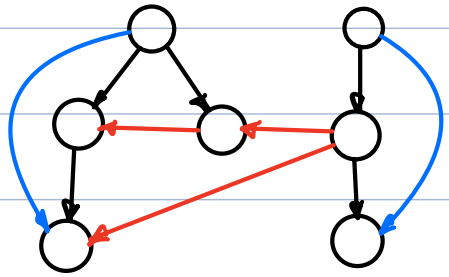
Acyclicity Test:

Given a directed graph, does it have a cycle?

Thoughts:

Idea 1: Repeatedly delete sinks (or sources)

Idea 2: DFS? How to detect cycles?



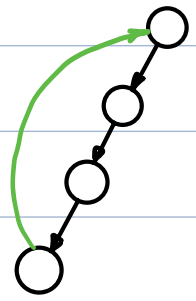
→ tree edge

→ forward edge

→ cross edge

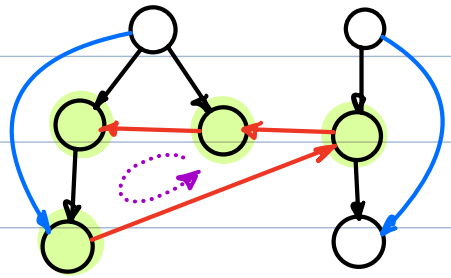
No cycles!

Back edge ⇒ cycle



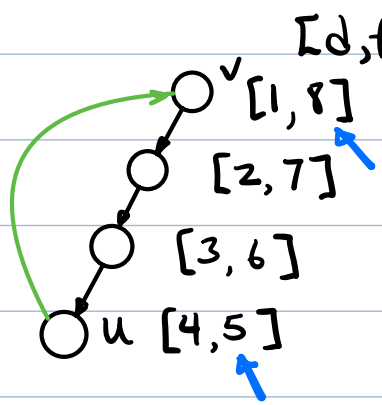
→ Back edge

Huh? Can't cross edges create cycles?



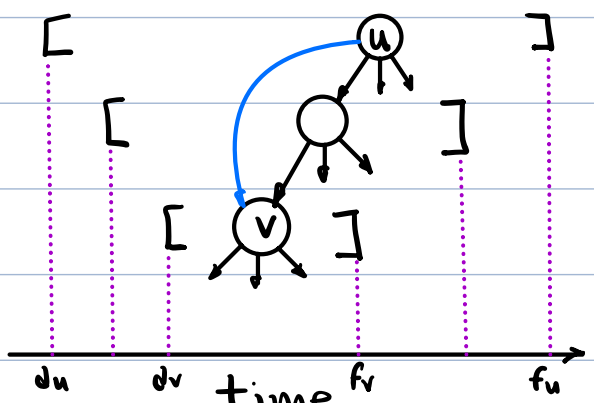
No! Try DFS on this graph + you'll see why

How to detect a back edge?



Claim:
(u,v) is back edge iff $f[u] \leq f[v]$

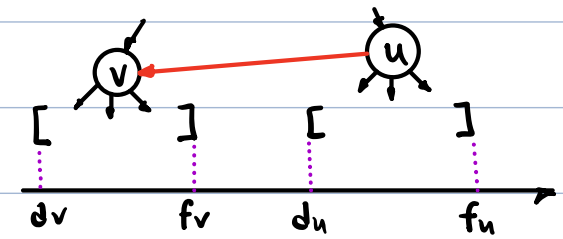
↳ why "="?



If (u,v) is tree or forward

$[d[v], f[v]] \subset [d[u], f[u]]$

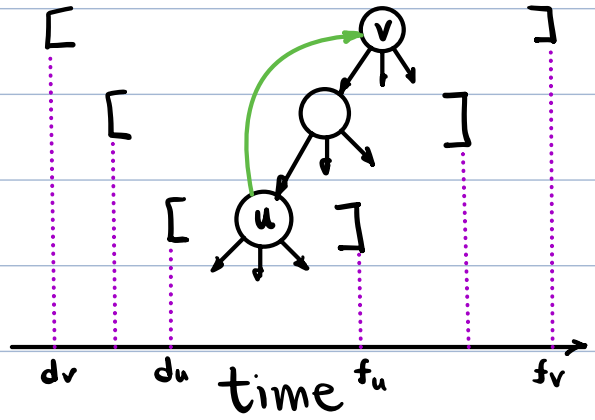
$\Rightarrow f[u] > f[v]$



If (u,v) is a cross edge

$[d[v], f[v]] < [d[u], f[u]]$

$\Rightarrow f[u] > f[v]$



If (u,v) is **back edge**

$[d[u], f[u]] \subseteq [d[v], f[v]]$

$\Rightarrow f[u] \leq f[v]$

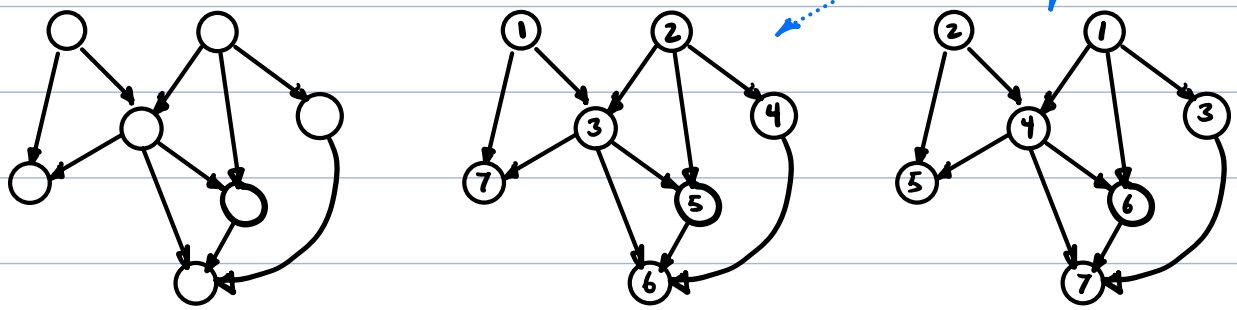
(See latex notes for formal proof)

Topological Sorting (via DFS):

Fact: Given any DAG $G = (V, E)$, there exists a **linear ordering** of the vertices that **respects** edge directions

$(u,v) \in E \Rightarrow u < v$

Possibly many



How to compute?

Idea 1: Repeatedly find a **sink** (or **source**) v
Put v at **end** (or **start**) of order
Remove v

Time: $O(n(n+m))$ - trivial
$O(n+m)$ - more clever

Topological Sort:

Idea 2: DFS

- Order vertices in reverse order of finish times.
- DFS on DAG has no back edges
  by Claim ⇒ $\forall (u,v) \in E, f[u] > f[v]$
- Implementation:
  - Push vertices on stack when finished
  - Pop stack at end to reverse order

```
topSort(G=(V,E)):
    mark all vertices undiscovered
    S ← empty stack
    for each (undiscovered u∈V) topVisit(u)
    pop S & output

topVisit(u):
    mark u visited
    for each (v ∈ Adj[u])
        if (v undiscovered) topVisit(v)
    push u onto S    // finished with u
```

Same DFS structure - eliminated start, finish, pred - unneeded

Time: $O(n+m)$ - standard DFS

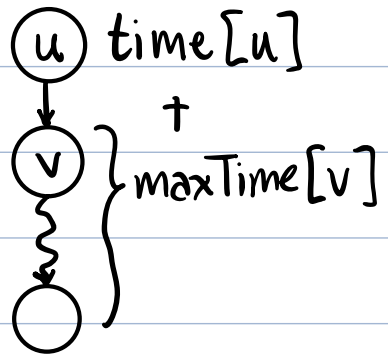
Longest Path in a DAG:

Vertex = task
Label = time to perform task

- Given a DAG where each vertex u stores its duration time[u]
  Compute the path that maximizes sum of durations.
- This is shortest completion time assuming:
  - Edges are precedence constraints
  - Allowing maximum parallelism

- Approach: Use DFS
  - Maintain maxTime[u] = length (time) of longest path starting at u.
  - For u, visit each neighbor v:

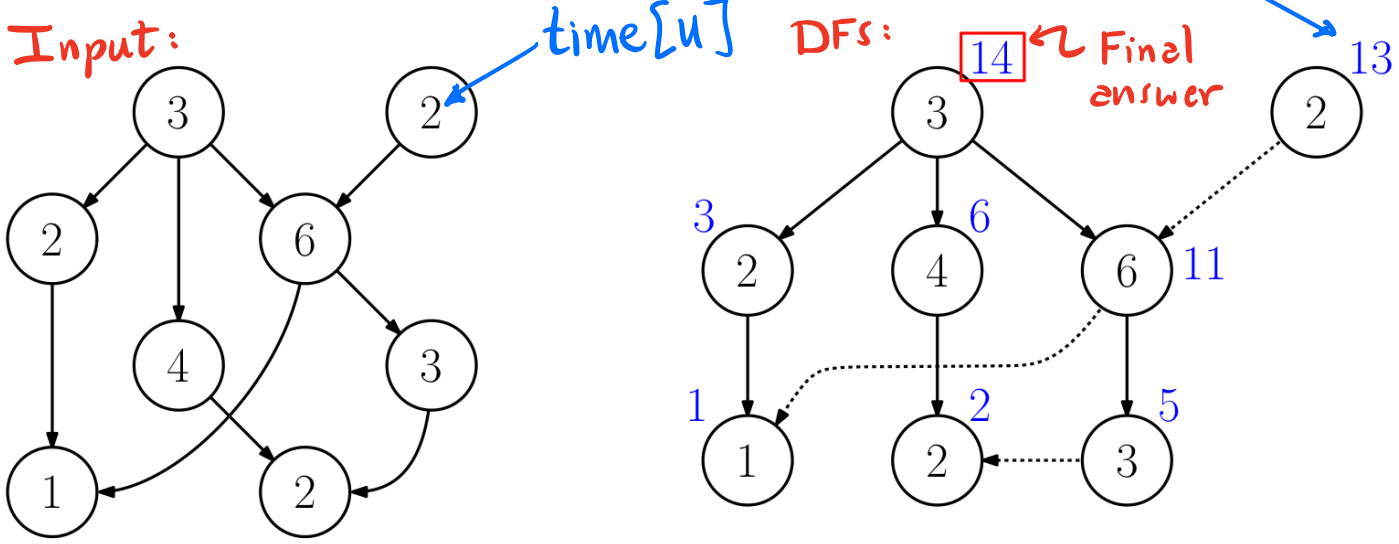
$$\text{maxTime}[u] \leftarrow \max \begin{cases} \text{maxTime}[u] \\ \text{time}[u] + \text{maxTime}[v] \end{cases}$$



```
LongPathVisit(u)
    mark[u] ← discovered ; maxTime[u] ← time[u]
    for each v ∈ Adj[u]
        if (v undiscovered) LongPathVisit(v)
        maxTime[u] ← max { maxTime[u],
                           time[u] + maxTime[v] }
```

On termination, output $\max_{u \in V}$ maxTime[u]

Example:

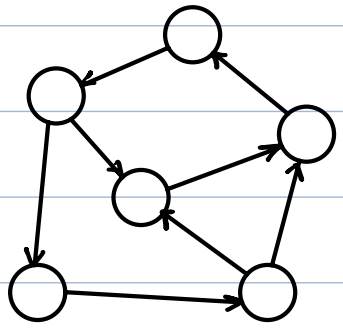


Does this only work on DAGs?
Can we use this to compute the longest simple path (no repeats) in any directed graph?

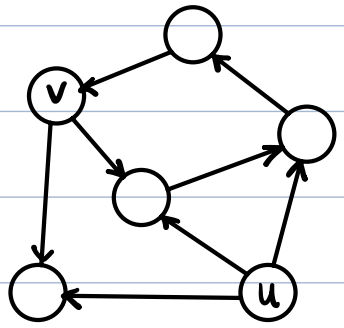
Ans: No - But why not?

Strong Components:

Def: A digraph is **strongly connected** if for every $u, v \in V$, there is a **path from u to v** and **from v to u.**

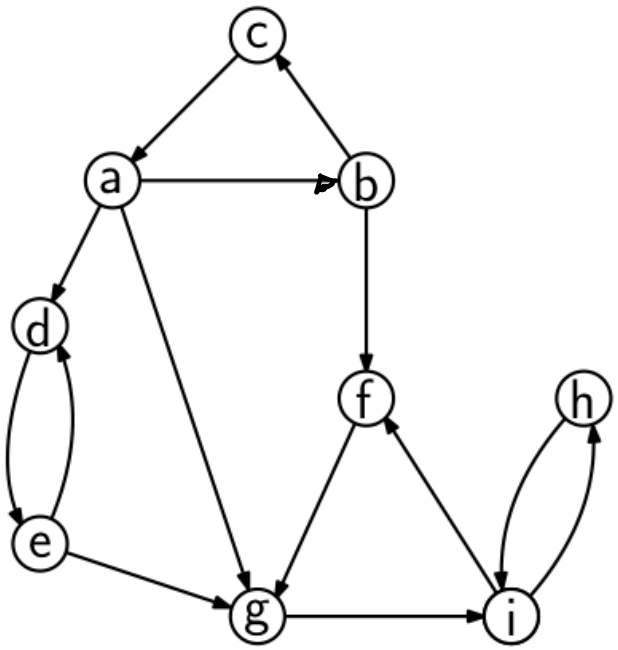


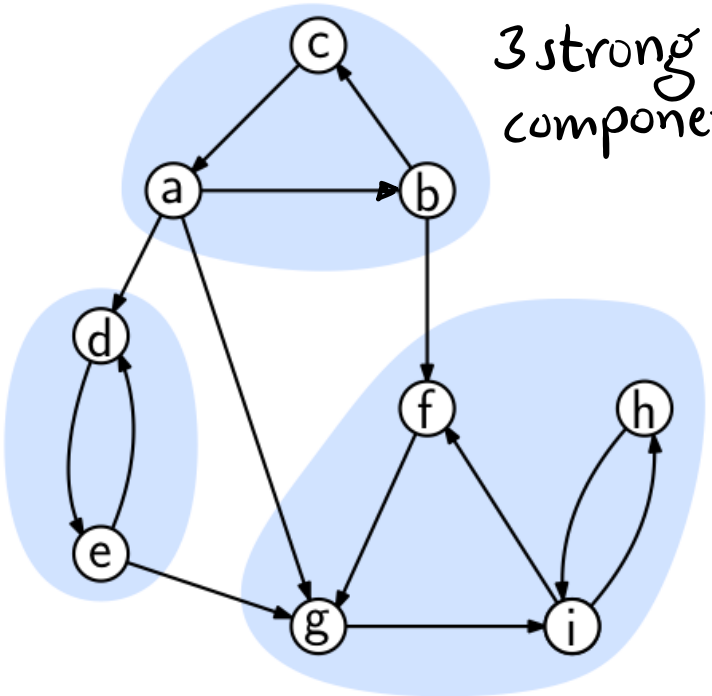
strongly connected



not
(No path from v to u)

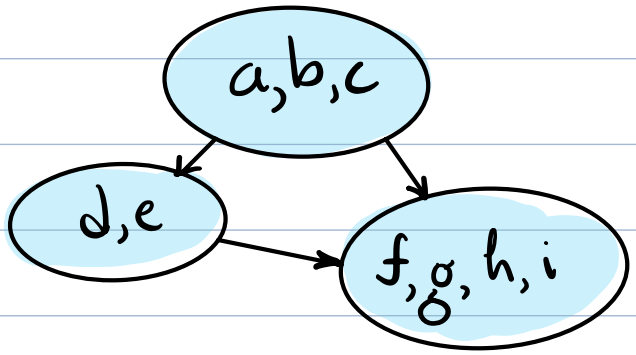
Def: Two vertices u + v are in the same **strong component** if **$\exists$ path u to v + v to u**





3 strong components

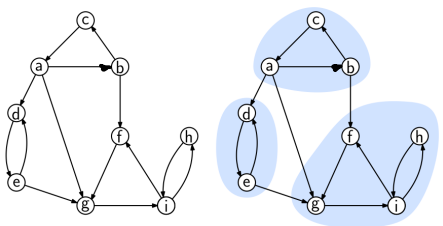
Def: If we "collapse" all the vertices in each strong component, we obtain the **component digraph.**



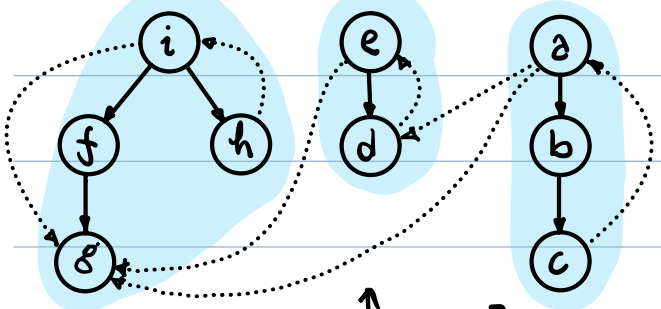
Claim: The **component digraph** is a **DAG**

Computing the Strong Components in a digraph $G=(V,E)$

- Use **DFS** → $O(n+m)$
- **Idea** - If we visit vertices in the **magic order** the components appear in **separate trees** of DFS.

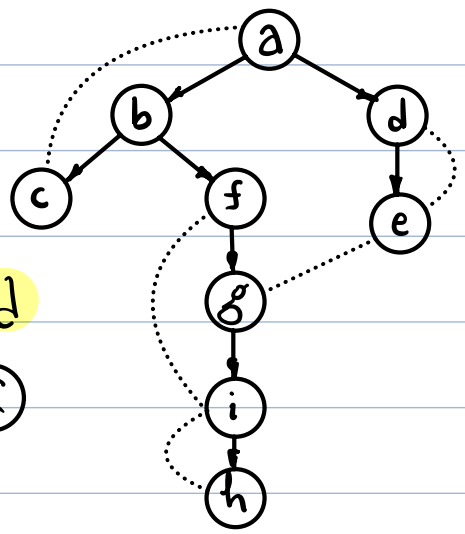


Good order: i...e...a



Each component a **separate tree** ☺

Bad order: a...



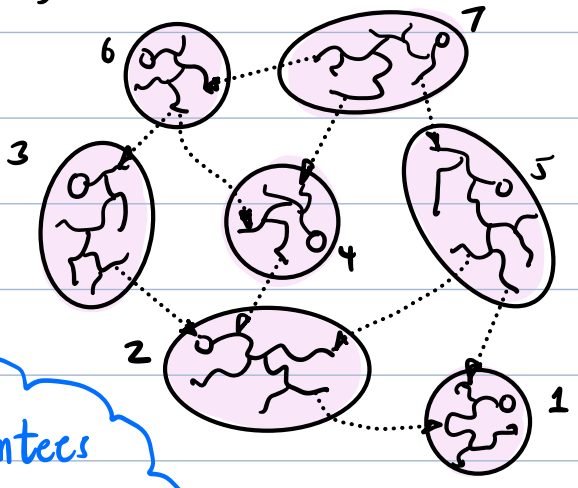
All smooshed together ☹

What makes an order good?

Key: When DFS visits a vertex, it does not finish until all reachable vertices are visited

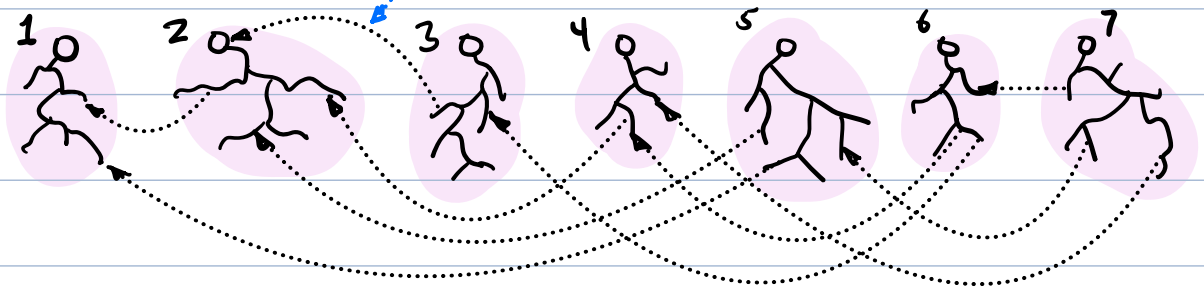
★ - Idea is to visit strong components in reverse topological order of component digraph.

DFS(G):



Shows DFS trees and edges between components

This order guarantees that edges between components are cross edges



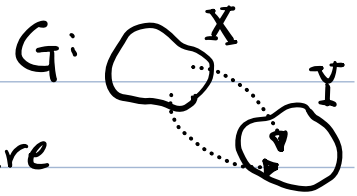
Huh? How can you order the components without knowing the strong components ??

Insanely Clever Trick:

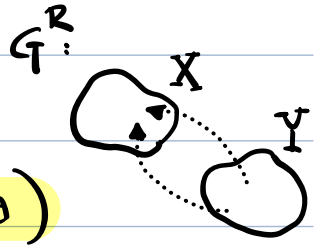
- Compute reverse graph G^R (reverse directions)
- DFS(G^R)
- Sort vertices inversely by finish times

Claim: This is the *magic order*
(Doesn't require knowledge of components)

Why?



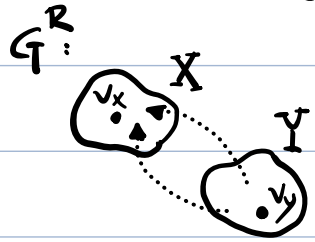
- Consider components X + Y in G, where X can reach Y (but not vice versa)
- In G^R, these are strong components, and Y can reach X (but not vice versa)



- Let v_x + v_y be the last vertices to finish in X+Y, respectively (in DFS(G^R))

Lemma: v_x must finish before v_y

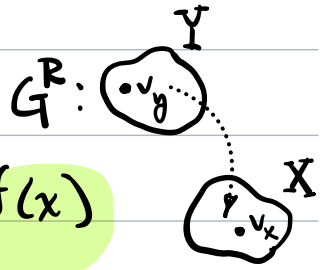
Why? If DFS hits X first, it visits all of X before starting Y. $\Rightarrow f(v_y) > f(v_x)$



If DFS hits Y first, it will leak into X, visit/finish everything in X, and then return + finish Y.

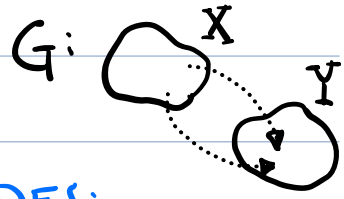
$\Rightarrow f(v_y) > f(v_x)$

From this we have:

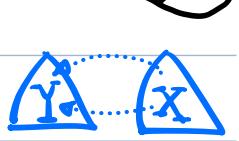


$\Rightarrow$ In DFS(G^R): $\max_{y \in Y} f(y) > \max_{x \in X} f(x)$

- Run DFS(G) but favor high f-values whenever starting a tree



$\Rightarrow$ Component Y will be visited before X in DFS(G)



$\Rightarrow$ DFS(G) visits strong components in reverse topological order of component digraph.

Final algorithm:

Strong Components ($G = (V, E)$)

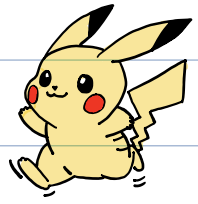
- Compute G^R
- Run DFS(G^R) + record finish times
- Sort vertices reversely by finish times
- Run DFS(G)
  - Whenever we start a new tree, use the above order
- Output DFS trees as Strong Comps.

Correctness: Follows from previous observations. (See latex notes for details)

Time: Let $n = |V|$, $m = |E|$

- Compute G^R: $O(n+m)$ (exercise)
- DFS(G^R): $O(n+m)$ (G^R has same size)
- Sort vertices: $O(n)$ (radix sort)
- DFS(G): $O(n+m)$

Total: $O(n+m)$



Summary:

- DAGs + Acyclicity Testing
- Topological Sorting (via DFS)
- Longest Path in a DAG
- Strong Components

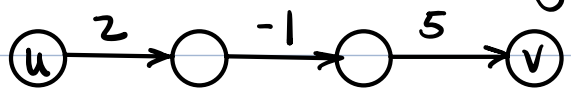
CMSC 451 - Algorithm Design
Lecture 4 - Shortest Paths - Bellman-Ford + Dijkstra

Problem: Given a digraph with numeric edge weights, compute **shortest paths**

Notation: $G = (V, E)$ - the digraph $n = |V|$, $m = |E|$

$w(u,v)$ - **weight** of edge $(u,v) \in E$

cost of a path = **sum** of edge weights



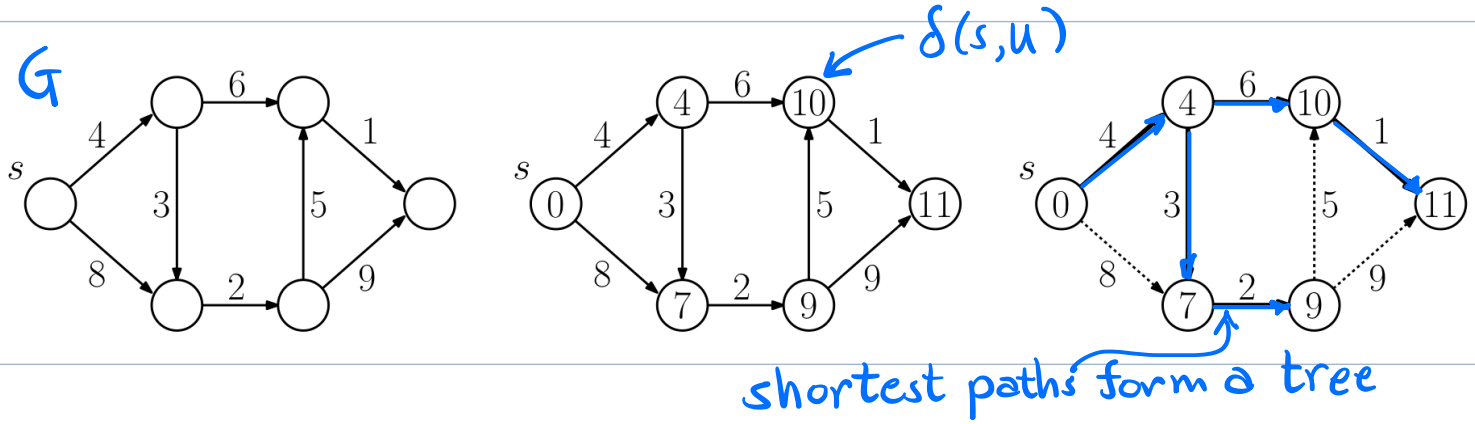
cost = 2 - 1 + 5 = 6

distance from u to v is **minimum cost** of any path from u to v (∞ if no path)

Denoted $\delta(u,v)$. Note: $\delta(u,u) = 0$

single-source: for a given source $s \in V$, compute $\delta(s,u)$ for all $u \in V$

(Also, encode shortest path info.)

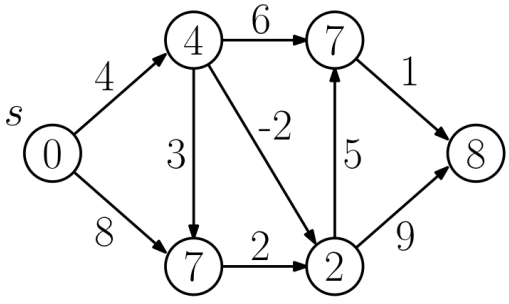


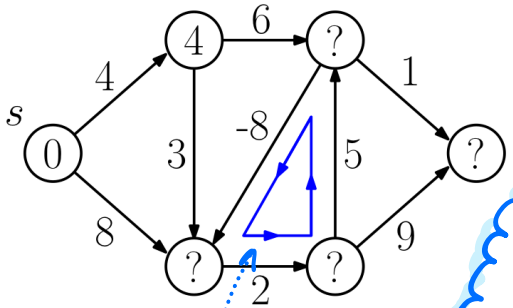
Preliminaries:

If edge weights are **uniform** (e.g. $w(u,v) = 1, \forall (u,v) \in E$)
fastest algorithm is **breadth first search (BFS)**
$O(n+m)$ time.

Negative edge weights?

- Used, e.g., when edges model **financial transactions** - buy + sell
- Shortest paths are well defined ($\delta(s,u)$ may be negative) provided there are no **negative-cost cycles**.

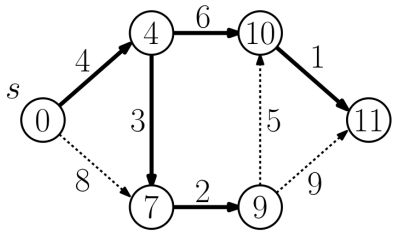


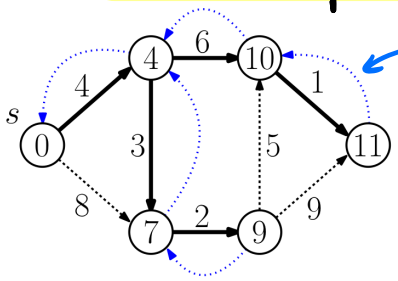


The more times you go around this loop, the lower the cost
$\delta(s,u) \to -\infty$

Paths?

pred[u] points to **prior vertex** on path from s to u. (pred[s] = null)
Follow these back = **reverse path**.





pred links

Bellman-Ford Algorithm -

- Solves the single source shortest path prob. for arbitrary edge weights - no neg. cost cycles.
- Invented 1955. Very simple
- Runs in time $O(n+m)$

Basics: Maintains, for all $v \in V$

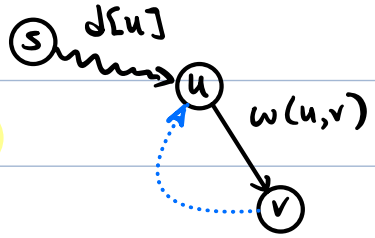
$d[v]$ = current distance estimate

There is a path $s \rightsquigarrow v$ of this cost, but might not be shortest $d[v] \geq \delta(s,v)$

$pred[v]$ = predecessor based on d-value

If $u = pred[v]$ then

$d[v] = d[u] + w(u,v)$

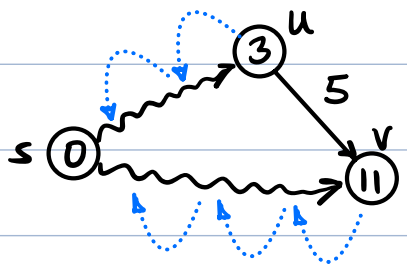


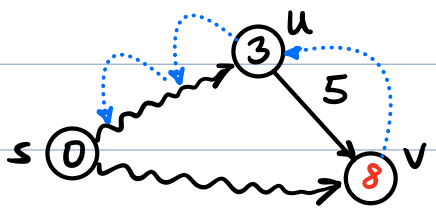
Relaxation:

- Propagates shortest paths forward one edge at a time

relax(u,v):

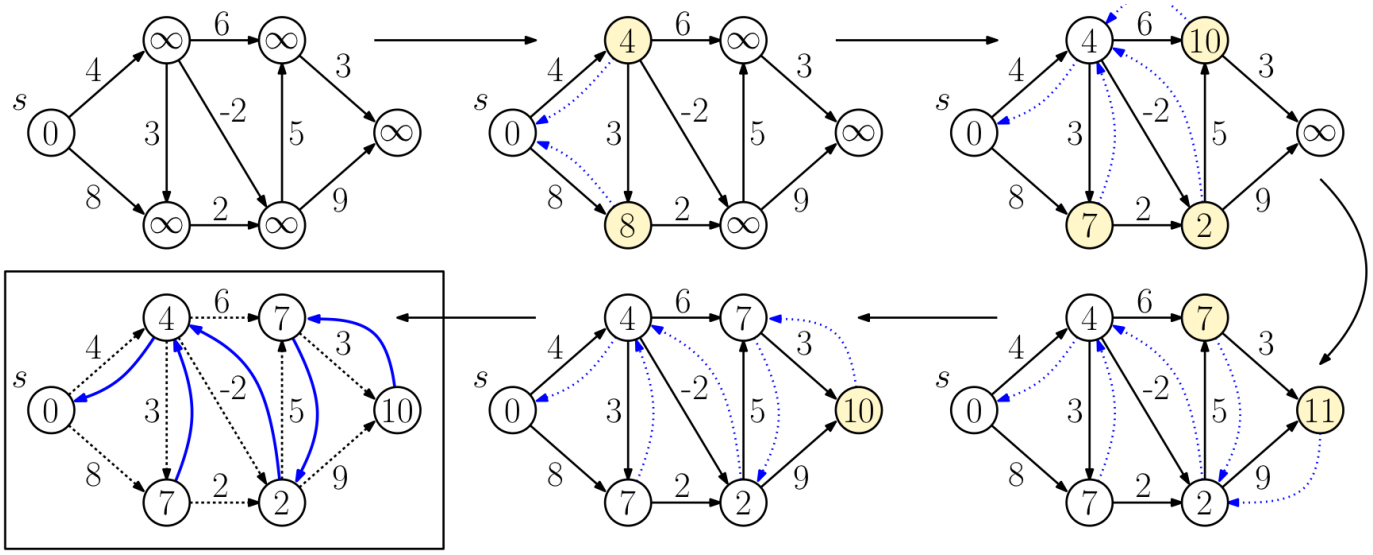
if
$d[u] + w(u,v) < d[v]$
then
$d[v] \leftarrow d[u] + w(u,v)$
$pred[v] \leftarrow u$





Idea: relax (u,v), $\forall (u,v) \in E$. Repeat until convergence

Example: (Each step involves relax (u,v) on all edges)




```
bellman-ford (G=(V,E), w, s)        ← source, weights
    for each (v ∈ V)
        d[u] ← ∞ ; pred[u] ← null   ← init
    d[s] ← 0
    repeat                          // relax all edges until converge
        converged ← true
        for each ((u,v) ∈ E)
            relax(u,v)
            if d[v] changed         // distance changed?
                converged ← false
    until (converged)
    [pred links define an inverted s.p. tree]
```

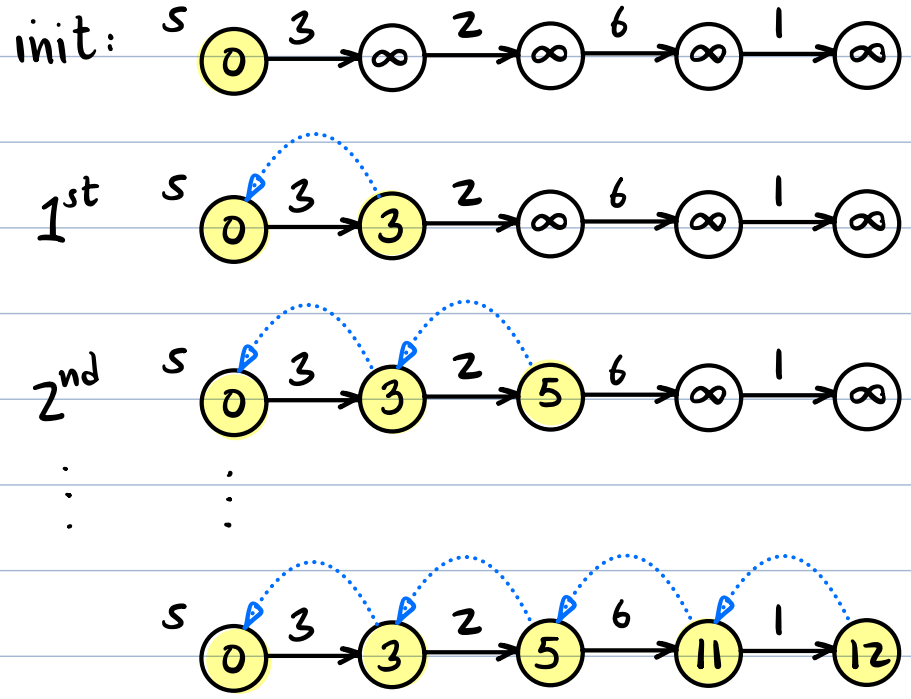
Running time: $n = |V|$ $m = |E|$

- Each repeat loop takes $O(m)$ time
- Will show convergence within n-1 iterations
- Total time: $O(nm)$

Correctness:

Key - Consider any shortest path

- Each iteration of the repeat loop propagates distances one more edge



Lemma: If G has no neg. cost cycles, Bellman-Ford will terminate within $n-1$ iterations + $d[v] = \delta(s,v)$

Proof sketch: (see latex notes for details)

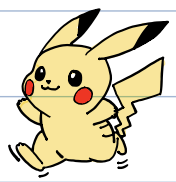
- By induction: After k^{th} iteration, all vertices whose shortest path has $\leq k$ edges have $d[u] = \delta(s,u)$

- If no neg. cost cycles, any shortest path has $\leq n-1$ edges (no repeats)

$\Rightarrow$ Bellman-Ford converges with correct distances in $\leq n-1$ iterations.

Running time:

- Each relax takes $O(1)$ time
- Each iteration takes $O(m)$ time
- $\leq n-1$ iterations $\Rightarrow$ Total time = $O(n \cdot m)$



Dijkstra's Algorithm:

- Simple greedy algorithm for single-source s.p.
- Requires edge weights to be ≥ 0
- Discovered in 1956 by famous Dutch scientist Edsger Dijkstra
- The best from a worst-case perspective.
  More practical → A*-search

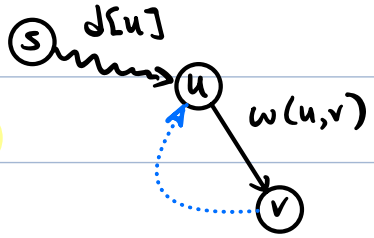
Basics: Maintains, for all $v \in V$

$d[v]$ = current distance estimate

There is a path $s \rightsquigarrow v$ of this cost, but might not be shortest $d[v] \geq \delta(s,v)$

$pred[v]$ = predecessor based on d-value

If $u = pred[v]$ then $d[v] = d[u] + w(u,v)$



Recall:

```
relax(u,v)
    if (d[u] + w(u,v) < d[v])
        d[v] ← d[u] + w(u,v)
        pred[v] ← u
```

Overview:

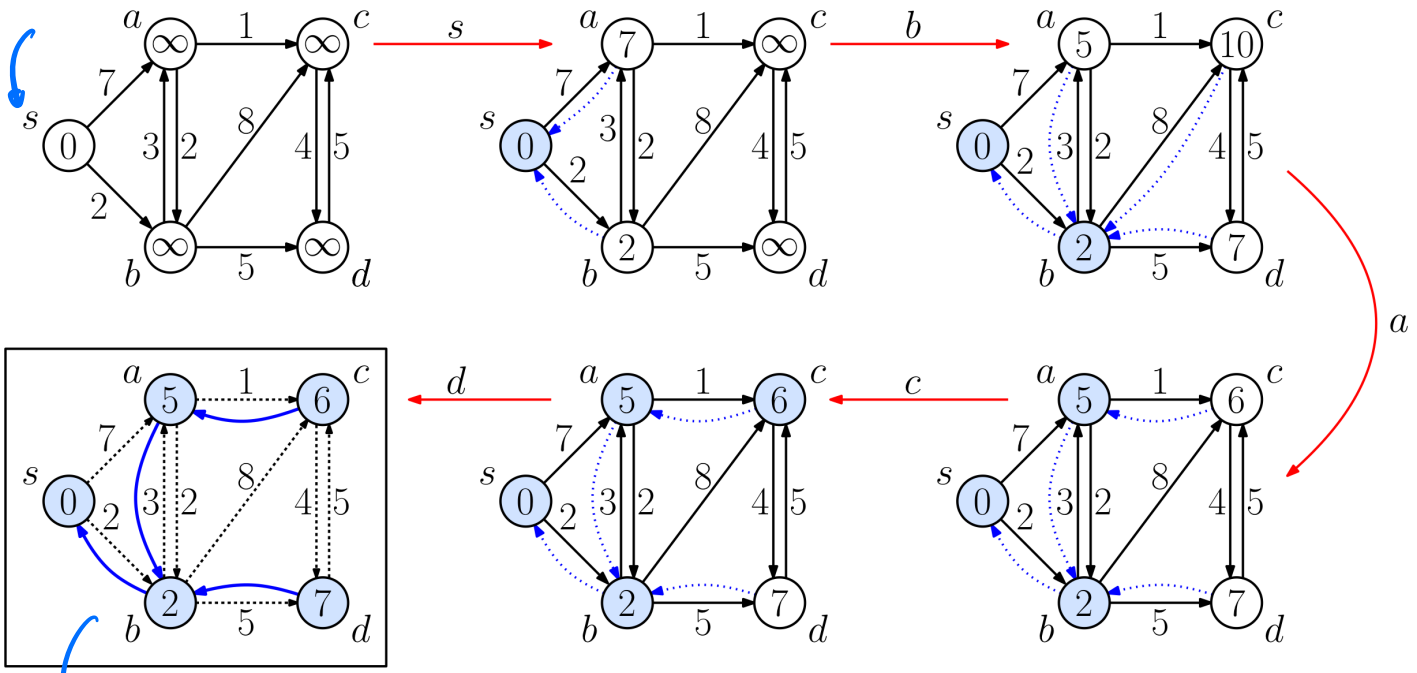
- Init: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty$, o.w.
- Repeat:
  - Select unprocessed vertex with min d-value
  - Apply relax on all outgoing edges

Details:

- Store unprocessed vertices in a priority queue, sorted by d-values
- Priority queue operations:
  - Build initial - $O(n)$
  - Extract min - $O(\log n)$
  - Decrease key - $O(\log n)$

```
dijkstra ( G=(V,E), w, s )        → source, → weights
    for each (v ∈ V)
        d[u] ← ∞ ; pred[u] ← null
    d[s] ← 0
    Q ← priority queue sorted by d-values
    while (Q is not empty)
        u ← Q.extractMin()
        for each (v ∈ Adj[u])
            relax (u,v) + update Q if necessary   ← since d[v] may have changed
    [pred links define an inverted s.p. tree]
```

Example: source = s



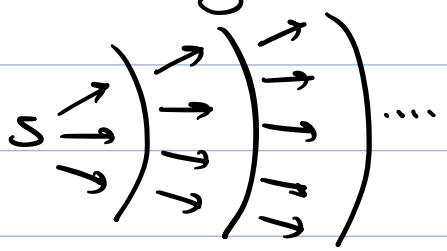
Pred links yield reverse of shortest path

E.g. c ⇢ a ⇢ b ⇢ s

⇒ path to c is s → b → a → c

Correctness:

Intuitively, Dijkstra's algorithm propagates costs forward in increasing distance order from s.



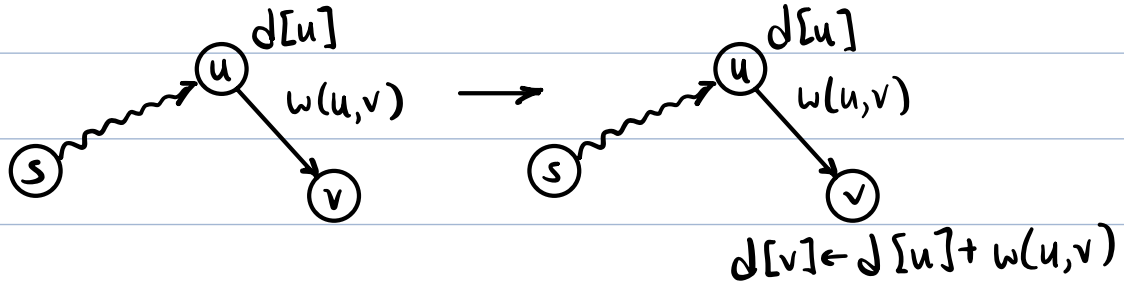
Safe because later vertices cannot affect distances to closer ones. (No neg. weight edges)

Correctness follows from the next lemma.
[for $u \in V$, $d[u]$ = estimate $\delta(s,u)$ = true dist.]

Lemma: For all $u \in V$,
(1) If $d[u] \neq \infty$, there exists a path of this cost
(2) After u is processed, $d[u] = \delta(s,u)$

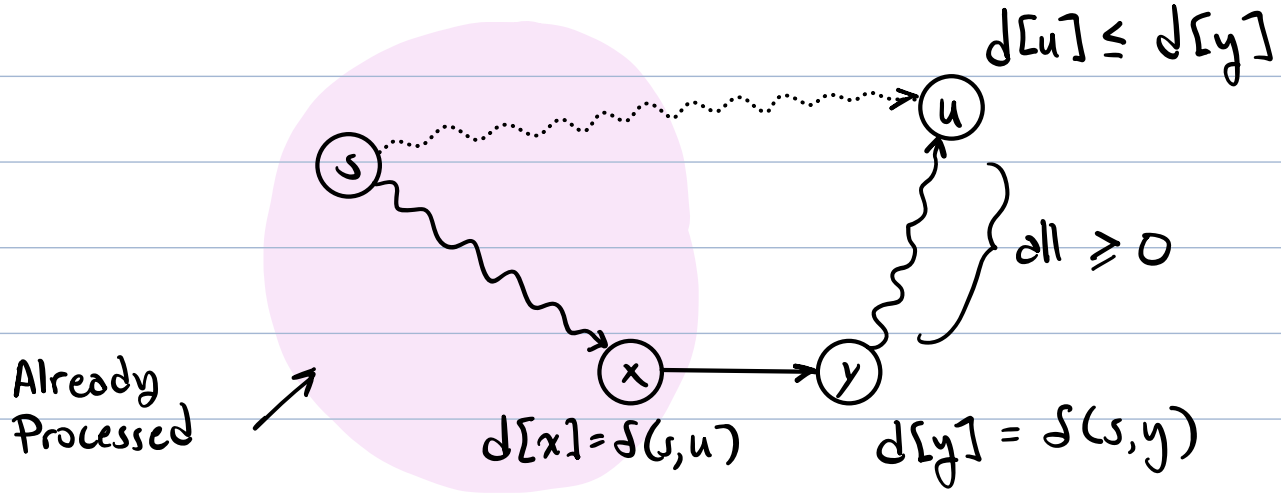
Proof:

(1) Follows by induction + fact that d-values are defined by relax op.



(2) Suppose not. Consider the first vertex u, where $d[u] \neq \delta(s,u)$ after processing u. By (1), $d[u] > \delta(s,u)$ @

- Let S be the set of processed vertices prior to u.
- The true shortest path $s \rightsquigarrow u$ must first jump outside of S - let (x,y) be this edge. (possibly $x = s$, $y = u$)



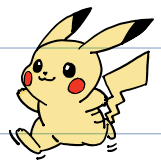
- Since **no errors** up to now
  $d[x] = \delta(s,x)$ (b)
- Since **x was processed**, relax(x,y) sets
  $d[y] \leftarrow d[x] + w(x,y)$
  $\overset{(b)}{=} \delta(s,x) + w(x,y)$
  $= \delta(s,y)$ (c) [since this is the true shortest path]
- Since **u is processed next**, we know
  $d[u] \leq d[y]$ (d)
- Since edge **weights ≥ 0**, $\delta(s,u) \geq \delta(s,y)$ (e)
- Putting this together:

$$\delta(s,u) \overset{(a)}{<} d[u] \overset{(d)}{\leq} d[y] \overset{(c)}{=} \delta(s,y) \overset{(e)}{\leq} \delta(s,u)$$

contradiction!

QED.

Running time: $n = |V|$ $m = |E|$



- Outer loop - n times
  - extract min - log n time
- relax - once for each edge - m times
  - update key value - log n time
    - Fibonacci heap - O(1) amortized

Total: $O(n \log n + m \log n)$

Fibonacci heap: $O(n \log n + m)$

Summary:

Single-Source Shortest Paths in Digraphs

- Nonnegative Weights
  - Dijkstra - $O(n \log n + m)$
- Neg. weights (no neg. cycles)
  - Bellman-Ford - $O(n \cdot m)$

CMSC 451 - Algorithm Design
Lecture 5 - Greedy Algorithms for Scheduling

Discrete Optimization:

Compute a **discrete structure** of a given class (e.g., subset, tree, path, partition) to **maximize/minimize** a given **objective function** (e.g., cost, distance, size) subject to a given set of **constraints** (e.g., disjointness, connectedness, completeness).

A **feasible solution** satisfies all **constraints**
An **optimal solution** is feasible and max/minimizes the **objective function**

Example:

Min. Spanning Tree: Given a connected, weighted graph compute:

- **structure** – **subset of edges**
- **objective** – min. **total weight**
- **constraints** – **connected**, **acyclic**, **cover** all vertices

Common Strategies:

- **Brute-force search** - Try all possibilities - **Slow!!**
- **Local search** - Find an init. feasible solution + repeatedly make **small improvements**
- **Dynamic programming** - (Future lectures)
- **Greedy** - Build a solution by **repeated additions** (never revoked/reversed) each based on **best choice** subject to constraints

  E.g., **Kruskal's MST algorithm** (add **min. weight edge** that doesn't cause a cycle)
- ...

This lecture: Three scheduling problems

- **Interval scheduling**
- **Interval partioning**
- **Schedule to minimize lateness**

Interval Scheduling:

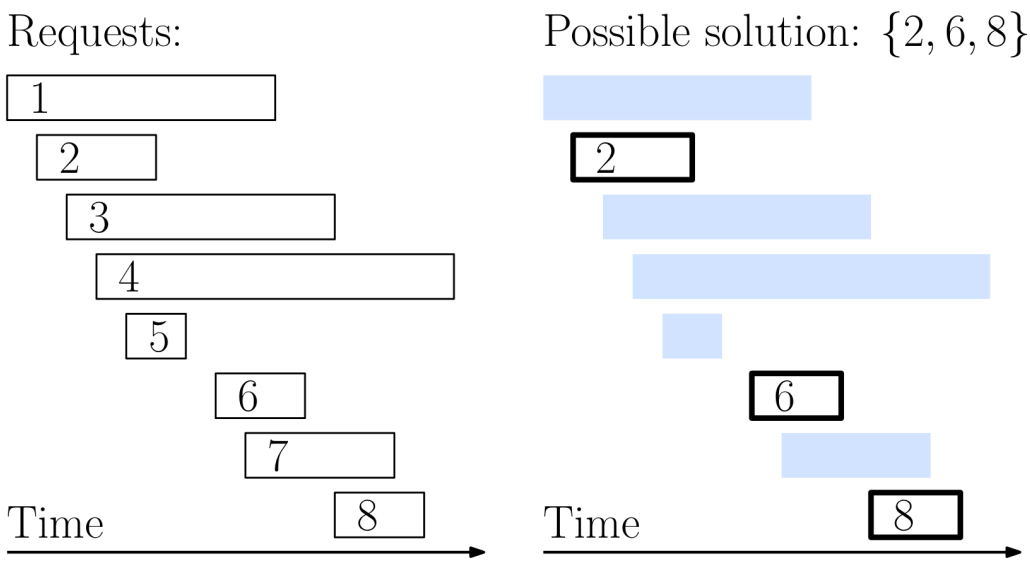
Given a set $R = \{r_1 \dots r_n\}$ of **requests**, each being an **interval** $r_i = [s_i, f_i]$ compute a subset of **non-overlapping requests** of **max. cardinality**

Application: Scheduling events at a facility

Example:

3 requests satisfied

Also: {5,6,8}



Greedy Approach - Select a request that **does not conflict** with prior selection + **min. some measure:**

- **Earliest Start** - request with smallest **start, s_i**
- **Earliest Finish** - " " " **finish, f_i**
- **Shortest Duration** - " " " **duration, $f_i - s_i$**
- **Min conflicts** - **overlaps fewest** among remaining requests

Optimal?

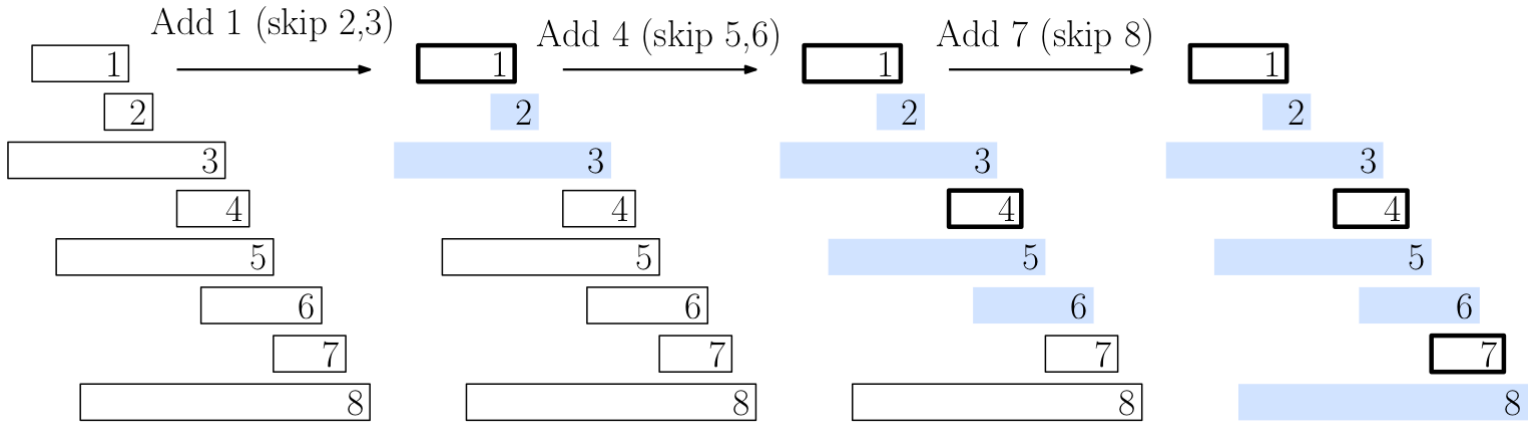
- **Earliest Start** - ✗
- **Earliest Finish** - ✓
- **Shortest Duration** - ✗
- **Min conflicts** - ✗

Exercise: Find counterexamples

Earliest Finish First:

```
greedy-interval-sched(s,f)
    sort requests by f-times
    S ← ∅                          // init empty schedule
    prevFinish ← -∞
    for (i ← 1 to n)
        if (s[i] > prevFinish)     // no conflict?
            append i to S          // ... schedule it
            prevFinish ← f[i]
```

Example: (Presorted by finish times)



Running time:

$O(n \log n)$ - sort

$O(n)$ - remaining processing

} $O(n \log n)$ total

Correctness:

Must show: **Feasibility** - A **valid** schedule (no conflicts)
Optimality - **Maximizes** no. of requests

Feasibility: **Easy** - No request is **scheduled** until after **prev** request **finishes**

Optimality: Not so easy - Let's do this rigorously.

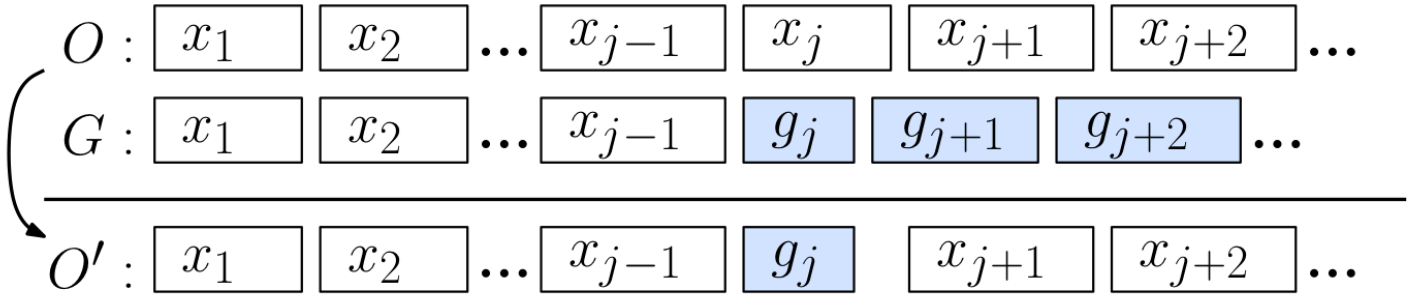
Let $O = \langle x_1, \ldots, x_k \rangle$ be **any optimal schedule** (may be many)

Let $G = \langle g_1, \ldots, g_{k'} \rangle$ be **EFF greedy schedule** ($k' \leq k$)

If $G = O$, we're done!

otherwise, let **j be smallest index** s.t. $x_j \neq g_j$

- By definition of EFF, $f[g_j] \leq f[x_j]$
- Form a **new schedule O'** by replacing **x_j by g_j** — Still **feasible** + **same size**



Repeat (hidden induction) until $O''''' = G$ ⇒ **G is optimal**

□

Next problem:

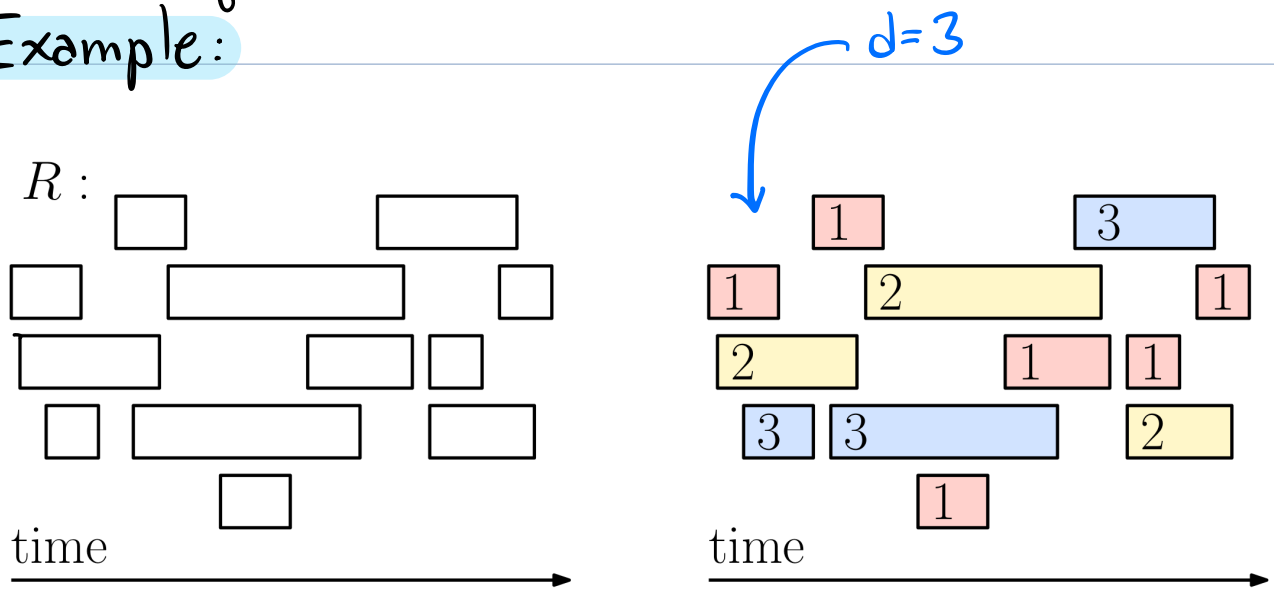
Interval Partitioning: Suppose we have ∞-many resources, but they cost $$$

Want to satisfy all requests with fewest resources

Problem:

Given a set $R = \{r_1, \ldots, r_n\}$ of requests, each with start/finish times s_i / f_i, partition R into the smallest num. of sets $R_1, \ldots, R_d$ such that all requests in R_i are pairwise non-conflicting

Example:

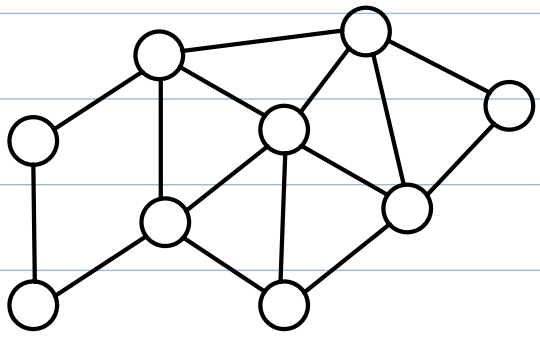


This is an example coloring -

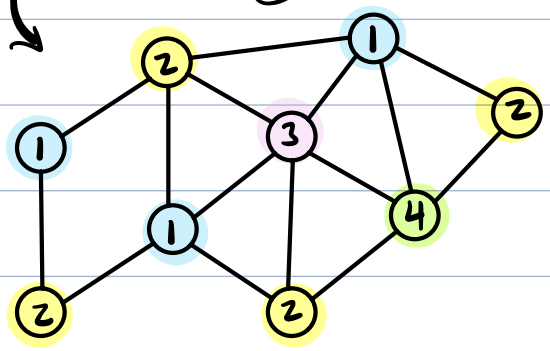
Given a set of objects + a conflict relation, partition into smallest number of conflict-free subsets.

Number of subsets = coloring number

Graph coloring:



4-coloring



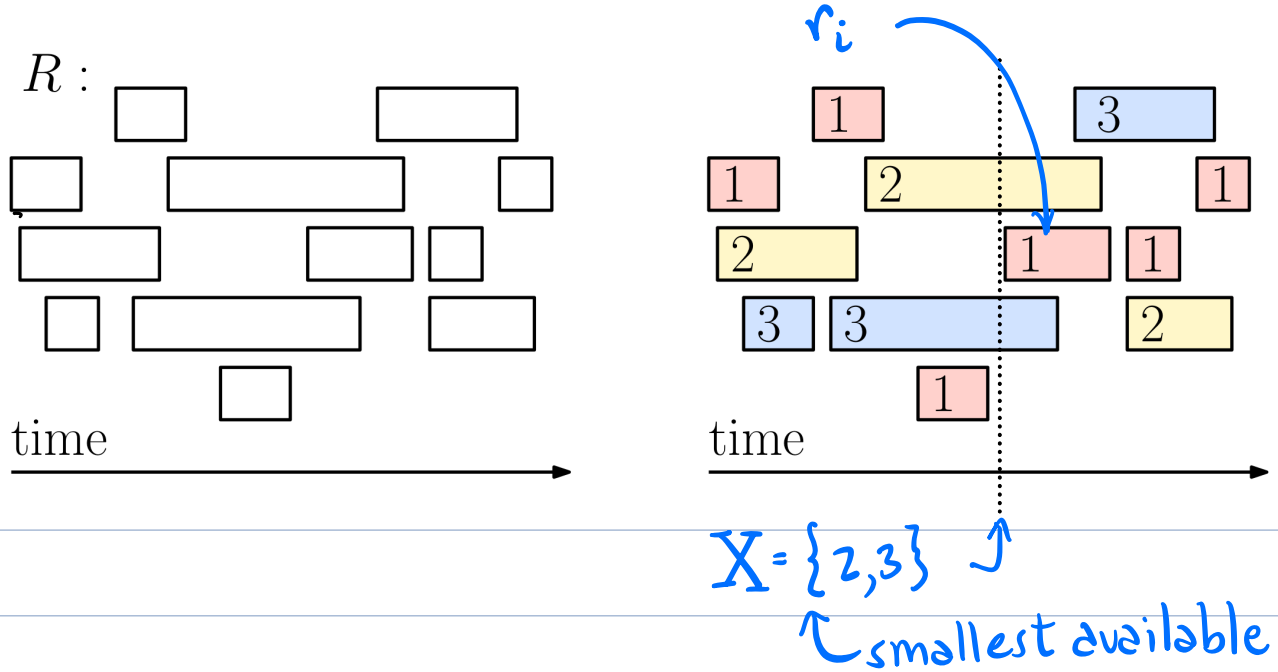
Graph coloring is NP-hard, but interval coloring is much easier.

Greedy Strategy for Interval Partitioning

Lowest Available Color

```
greedy-interval-partition (s, f)   // s = start
    sort by start times            // f = finish
    for (i ← 1 to n)
        X ← ∅                      // X = excluded colors
        for (j ← 1 to i-1)
            if ([s_j, f_j] overlaps [s_i, f_i])
                add color[j] to X
        color[i] ← smallest color not in X
    return color array
```

Example:



Running Time:

- Sorting - $O(n \log n)$
- 2 nested loops $1 \ldots n$ & $1 \ldots i-1$

$$= \sum_{i=1}^{n} (i-1) = O(n^2)$$

} $O(n^2)$ total

> There is smarter approach that runs in $O(n)$ time after sorting - See lecture notes.

Correctness: Need to show

- Feasibility - Obvious (Avoid conflicting colors)
- Optimality - ??

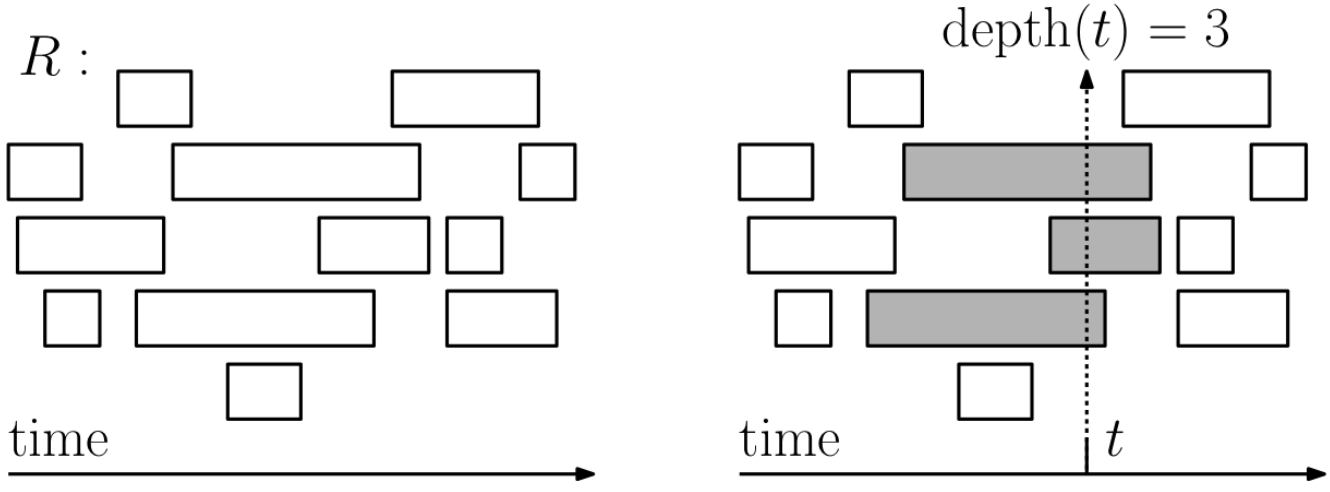
Optimality - Approach

- Define a statistic - depth
- Lower bound - Any solution must use $\geq$ depth colors
- Upper bound - Greedy algorithm uses $\leq$ depth colors

$\Rightarrow$ Greedy is optimal

Depth - Given request set $R = \{r_1, \ldots, r_n\}$, $r_i = [s_i, f_i]$ + time t define

$$\text{depth}_R(t) = \text{num. intervals overlap } t = |\{i \mid t \in [s_i, f_i]\}|$$



and: $\text{depth}(R) = \max_{t \geq 0} \text{depth}_R(t)$

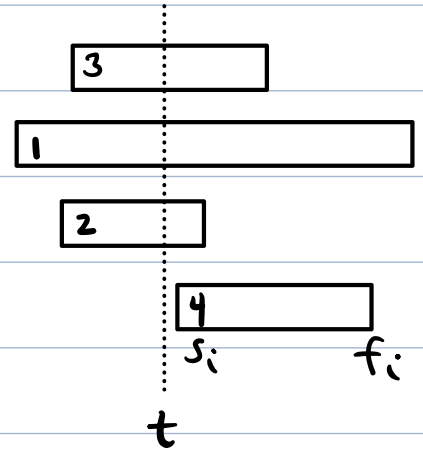
Clearly, all requests contibuting to depth conflict...

Lemma: For any d-coloring of R, $d \geq \text{depth}(R)$

The following implies that greedy is optimal

Lemma: The greedy algorithm generates a d-coloring, where $d \leq \text{depth}(R)$

Proof: Suppose towards a contradiction that there is a first time s_i where greedy uses more than $\text{depth}_R(s_i)$ colors.



Consider time t just prior to s_i

$$\text{depth}_R(s_i) = \text{depth}_R(t) + 1 \quad (a)$$

Let d = num. greedy colors at time t. Since s_i is the first error:

$$d \leq \text{depth}_R(t) \quad (b)$$

Thus, there are d excluded colors in X when s_i is processed. Implies that greedy uses

$$d + 1 \overset{(b)}{\leq} \text{depth}_R(t) + 1 \overset{(a)}{=} \text{depth}_R(s_i)$$

Contradicting hyp. that error at s_i □

Q: If requests were sorted by a different criterion, would the algorithm still be optimal?

Q: Can we modify this to color graphs? (NP-hard!)

Scheduling to Minimize Lateness

- Tasks rather than requests - $X = \{x_1, \ldots x_n\}$
  - Execution time - t_i
  - Deadline - d_i
- Application - Homework assignments

Objective:

- Compute start times $S = \{s_1, \ldots, s_n\}$ s.t.
  - Tasks do not overlap $[s_i, f_i] \cap [s_j, f_j] = \emptyset$
    where $f_i = s_i + t_i$
  - Minimize lateness - max. deadline excess

$$\ell_i = \max(0, f_i - d_i)$$

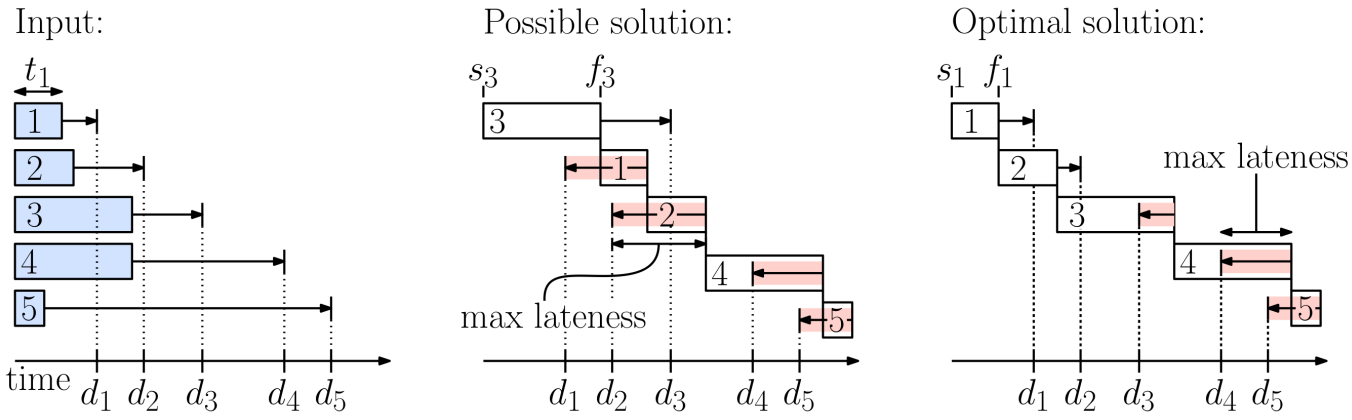
How far beyond the deadline did you finish?

max. lateness:

$$L(S) = \max_{1 \le i \le n} \ell_i$$

your worst deadline excess (not sum)

Example:



Greedy approach

- Schedule based on some criterion

- Shortest duration first - Sort by t_i ✗
- Earliest deadline first - Sort by d_i ✓
- Smallest slack first - Sort by $d_i - t_i$ ✗

```
greedy-lateness-sched(t, d)
    sort by deadlines (d_1 ≤ ... ≤ d_n)
    prevFinish ← 0          // when prev task finished
    for (i ← 1 to n)
        s_i ← prevFinish    // start after prev finish
        prevFinish ← f_i ← s_i + t_i   // update finish
        l_i ← max(0, f_i - d_i)        // lateness
    return [s_1, ..., s_n]  L = max_i l_i   // return start times
```

Running Time:

- Sorting - $O(n \log n)$
- Processing - $O(n)$

} Total: $O(n \log n)$

Correctness: Need to show

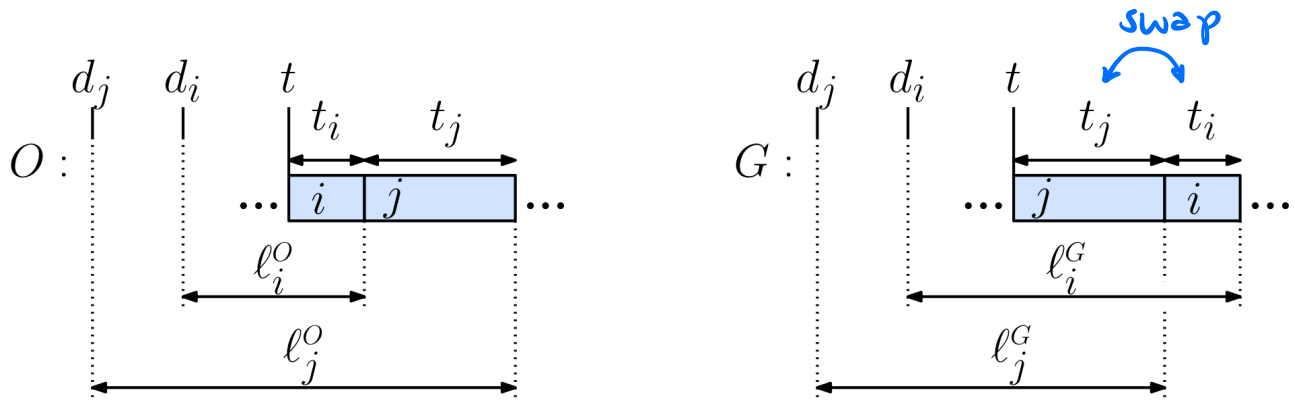
- Feasibility - Easy - No conflicting tasks
- Optimality - ??

> **Lemma:** The greedy algorithm minimizes max. lateness.

Proof: We may limit consideration to schedules that are "slack-free"

- no gaps between tasks. (Greedy is slack-free)

- Let O be any lateness-optimal, slack-free schedule.
- If O is deadline sorted - we're done! $O = G$
- O.w. let x_i + x_j be first consecutive pair not in deadline order $d_i > d_j$
- We'll swap them + show that max lateness can only decrease.



- Let $\ell_i^\sigma + \ell_j^\sigma$ be latenesses before swap + $\ell_i^G + \ell_j^G$ " " after swap
- These are the only latenesses affected
- Want to show:

$$\max(\ell_i^G, \ell_j^G) \leq \max(\ell_i^\sigma, \ell_j^\sigma)$$

↳ swap improves max lateness

- Let t be current time
- For simplicity, assume $d_i + d_j$ passed: $d_j < d_i \leq t$

(Excercise - Fix this)

↳ We can ignore max

- Observe:

$$\ell_i^\sigma = (t + t_i) - d_i \qquad \ell_j^\sigma = (t + t_i + t_j) - d_j$$

Since $d_i > d_j + t_j \geq 0$

$$\ell_j^\sigma = (t + t_i) + t_j - d_j > (t + t_i) - d_i = \ell_i^\sigma$$

$$\Rightarrow \max(\ell_i^\sigma, \ell_j^\sigma) = \ell_j^\sigma$$

- Also:

$$\ell_i^G = (t + t_i + t_j) - d_i < (t + t_i + t_j) - d_j = \ell_j^\sigma$$

$$\ell_j^G = (t + t_j) - d_j \leq (t + t_i + t_j) - d_j = \ell_j^\sigma$$

- Thus:

$$\max(\ell_i^G, \ell_j^G) \leq \max(\ell_j^\sigma, \ell_j^\sigma) = \max(\ell_i^\sigma, \ell_j^\sigma)$$

□

Q: Does earliest deadline first minimize total (sum of) lateness?
- If not, what does?

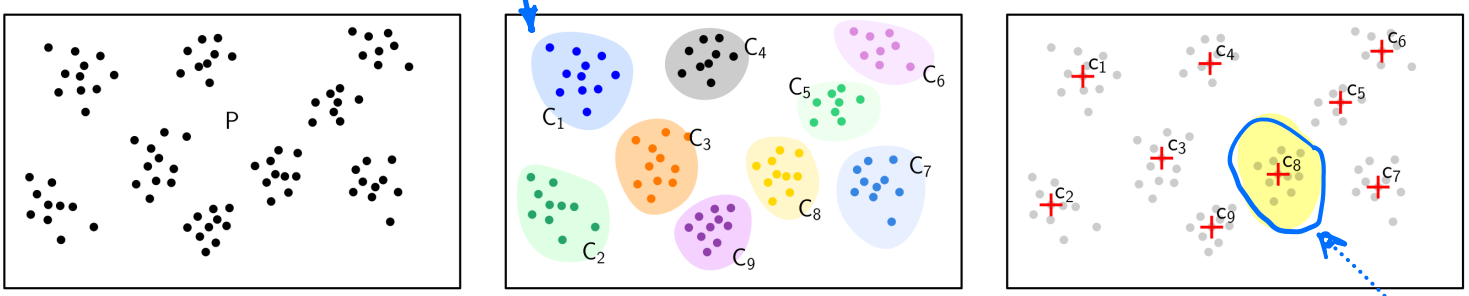
Summary: 3 examples of greedy solutions to simple scheduling problems.

Interval scheduling - Earliest finish first
Interval partitioning - Smallest available color
Minimizing Max Lateness - Earliest deadline first
↳ All $O(n \log n)$

CMSC 451 - Algorithm Design
Lecture 6 - k-Center Clustering + Gonzalez's Algorithm

Greedy algorithms often used to **approximate** **NP-hard problems**

Clustering - Given a set of **points** P + **distance** function, partition it into similar groups, called **clusters** $\{C_1, \ldots, C_k\}$



Center-based clustering -

Compute a set of **cluster centers** $\{c_1, \ldots, c_k\}$ and clusters are implicitly defined by **distance**

$N(c_i)$ = **subset of P closest to c_i**

Two varieties -

- Centers **must** be chosen **from P** (discrete clustering)
- Centers can be **any point in space**

Three Common Center-Based Clusterings:

Compute **k center points** to **minimize** ...

- **sum of** → **k-median**
- **sum of squared** → **k-means**
- **max of** → **k-center**

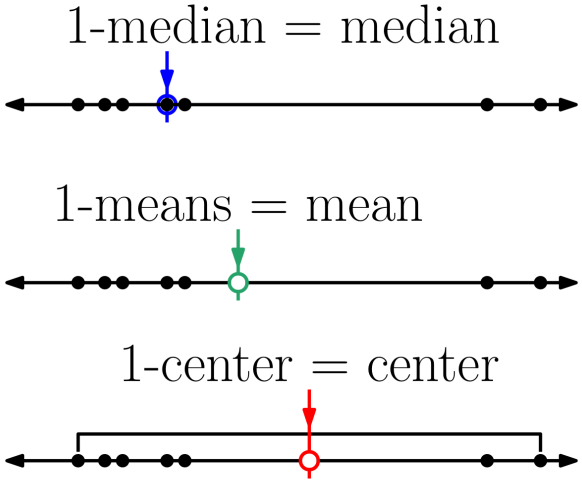
... **Euclidean distances** to **nearest center**

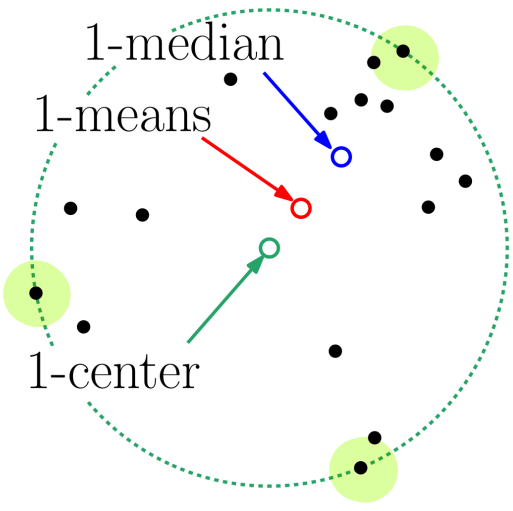
Which is best? Depends on **application** (e.g., sensitity to **outliers**)

Helps to understand the **single-cluster** case (**k=1**)

1-median - **1-D** → **median**

d-D → hard! **Fermat-Weber problem**





1-means - 1-D → mean

d-D → centroid (center of mass)

Easy to compute in any dimension!

Take mean coord. value in each dim.

k-Means is very popular - Lloyd's Algorithm

1-center - 1-D → midpoint of min & max

d-D → center of min enclosing ball

(Can compute in $O(n)$ time,

but tricky algorithm - take CMSC 754)

Metric Space:

Distance function $\delta : P \times P \to \mathbb{R}^{\geq 0}$

- $\delta(p,q) \geq 0$ & $\delta(p,p) = 0$ - Positive
- $\delta(p,q) = \delta(q,p)$ - Symmetric
- $\delta(p,r) \leq \delta(p,q) + \delta(q,r)$ - Δ-Inequality

k-Center Problem:

Given point set P in a metric space and $k \geq 1$, compute $C \subseteq P$ of size k to minimize max distance to closest center in C.

Note: Centers must be drawn from P

More formally - Given $C \subseteq P$, define objective fn.

$$\Delta_P(C) = \max_{p \in P} \min_{c \in C} d(p,c)$$

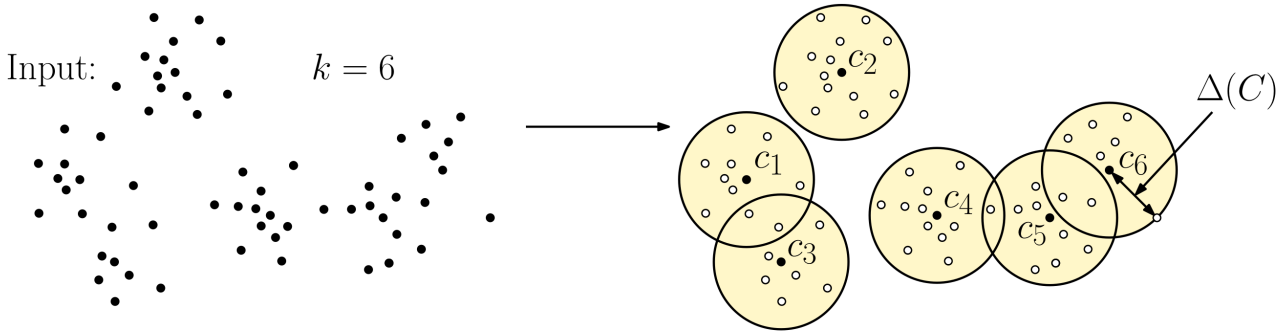
Problem - Compute a k-element set to minimize this:

$$\min_{\substack{C \subseteq P \\ |C|=k}} \Delta_C(P)$$

Geometric interpretation -

Cover all pts of P

- k balls (centered at pts of P)
- minimum radius $r = \Delta_P(C)$



Gonzalez's Algorithm -

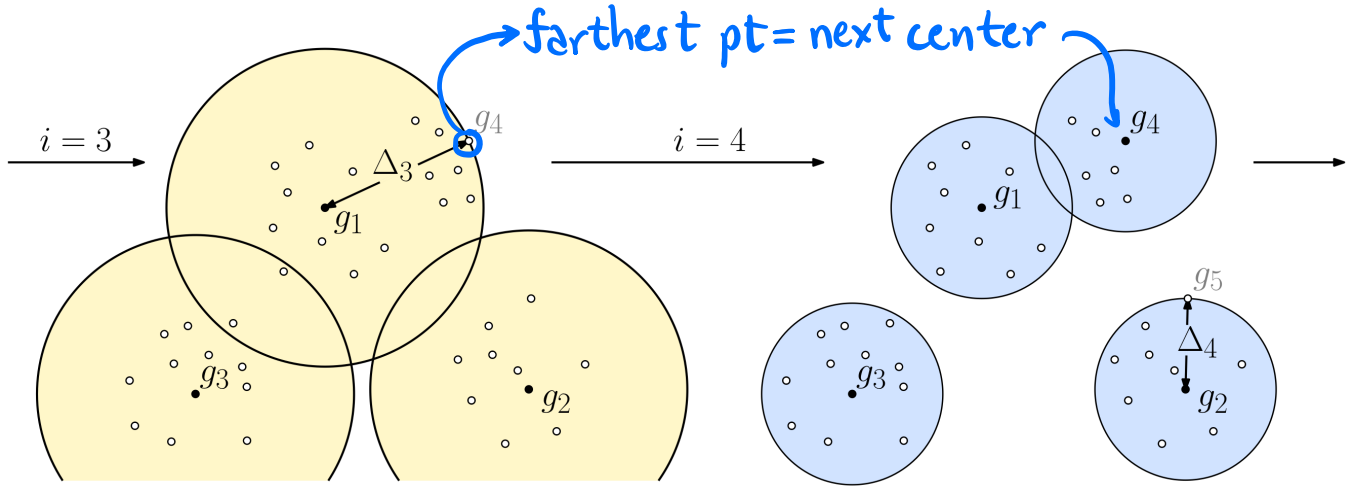
- Greedy + very simple
- 2x-approx. to k-center
- $O(k \cdot n)$ time

Intuitive Explanation:

Repeatedly add the point that is farthest from its closest center

```
gonzalez(P, k)        // Gonzalez's k-center
    G ← ∅
    for each (p ∈ P) d[p] ← +∞        // init. dists
    for (i ← 1 to k)
        p ← pt of P s.t. d[p] is max        // farthest pt
        add p to G                          // ... is next center
        for each (q ∈ P)        // update dist to closest
            d[q] ← min(d[q], dist(p,q))
    return G        // final centers
```

Example - $i = 4$ Δ_i = max distance to closest center = ball radius



Correctness -

Feasibility - Clearly the algorithm returns a valid clustering (provided $|P| \geq k$)

(Approx.) Optimality - Will show that our final radius $\leq 2 \cdot$ opt radius

Given any set $C \subseteq P$, recall that obj. fn. is

$$\Delta_P(C) = \max_{p \in P} \min_{c \in C} d(p, c)$$

Let G = output of Gonzalez

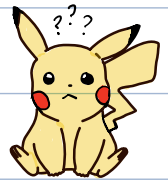
O = opt. k-center solution

Thm: $\Delta_P(G) \leq 2 \cdot \Delta_P(O)$

we'll drop this subscript

At first glance this seems hopeless

- k-center is NP-hard
- We cannot know what $\Delta(O)$ is!



Strategy -

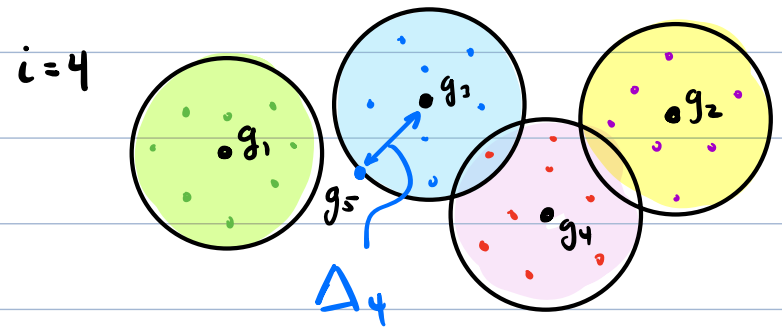
- Derive an easily computable estimate, Δ_{min}
- Show: $\Delta(O) \geq \Delta_{min}$
- Show: $\Delta(G) \leq 2 \cdot \Delta_{min}$

$\Rightarrow \Delta(G) \leq 2 \cdot \Delta_{min} \leq 2 \cdot \Delta(O)$ ✓

Notation:

$G_i = \{g_1, \ldots, g_i\}$ - the first i greedy ctrs.

$\Delta_i = \Delta(G_i)$ - farthest dist to these ctrs.



Imagine that we ran one additional iteration to get $k+1$ centers G_{k+1}

The theorem follows from 3 claims:

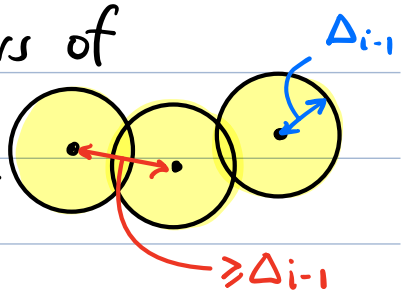
Claim 1: (Greedy distances decrease)

For $1 \le i \le k+1$, $\Delta_{i+1} \le \Delta_i$

Pf: As we add more centers, the dist to each pt's closest ctr. can never increase.

Claim 2: (Greedy centers are never close)

For $1 \le i \le k+1$, every pair of centers of G_i are at dist $\ge \Delta_{i-1}$



By setting $i = k+1 \Rightarrow$

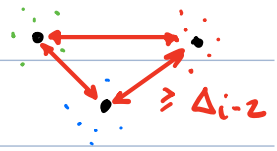
Corollary: $g, g' \in G_{k+1} \Rightarrow d(g,g') \ge \Delta_k = \Delta(G)$ @

Pf. (of Claim 2) By induction on i.

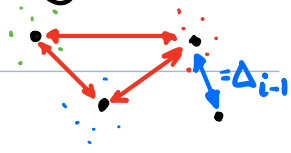
- At stage $i-1$, by induction, all ctrs. are sep. by dist $\geq \Delta_{i-2}$.
- by Claim 1, $\Delta_{i-2} \geq \Delta_{i-1}$ ✓
- Gonzalez choice $\Rightarrow$ i^{th} center at dist $= \Delta_{i-1}$ from it closest ctr

$\Rightarrow$ All centers at dist. $\geq \min(\Delta_{i-2}, \Delta_{i-1}) = \Delta_{i-1}$ from each other $\square$

Stage $i-1$:



Stage i:



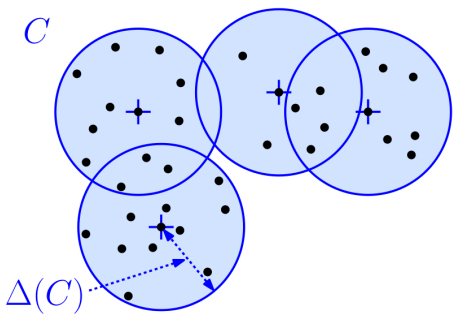
Define: $\Delta_{min} = \Delta_k/2 = \Delta(G)/2$

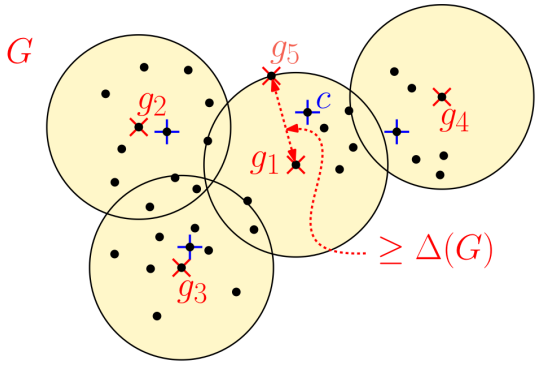
Important: Be sure you understand

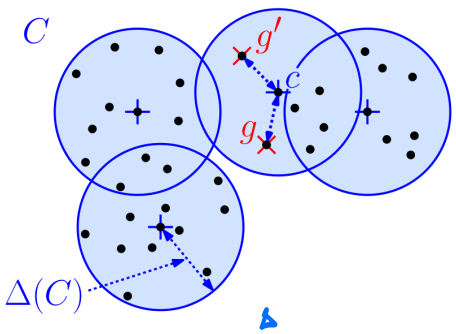
Claim 3: (Δ_{min} is a lower bound)
For any set $C \subseteq P$ of size k, $\Delta(C) \geq \Delta_{min}$

Pf: By def, every pt of P lies within dist $\Delta(C)$ of some pt of C.

- Since $G_{k+1} \subseteq P$, every pt. of G_{k+1} is within dist $\Delta(C)$ of some pt of C.







- Since $|G_{k+1}| = k+1$ & $|C| = k$, at least two pts of G_{k+1} are within dist $\Delta(C)$ of the same pt of C.

Pigeonhole principle

$\Rightarrow \exists\, g, g' \in G_{k+1}$ $c \in C$ s.t.

$$\delta(g,c) \leq \Delta(C) \quad + \quad \delta(g',c) \leq \Delta(C) \quad (b)$$

- We have:

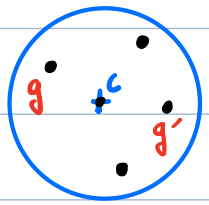
$$\Delta(G) \overset{(a)}{\leq} \delta(g,g')$$

$$\leq \delta(g,c) + \delta(c,g') \quad (\Delta\text{-inequality})$$

$$\leq \delta(g,c) + \delta(g',c) \quad (\text{symmetry})$$

$$\overset{(b)}{\leq} \Delta(C) + \Delta(C)$$

$$= 2\Delta(C)$$



by Def of Δ_{min}

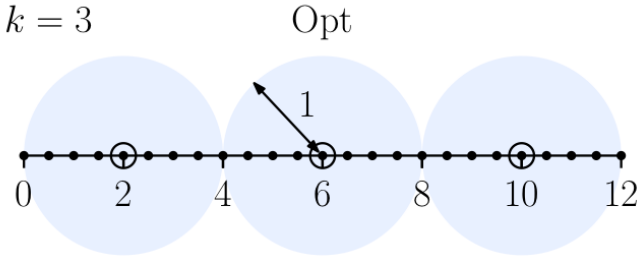
$$\Rightarrow \Delta(C) \geq \Delta(G)/2 = \Delta_{min}$$ ✓

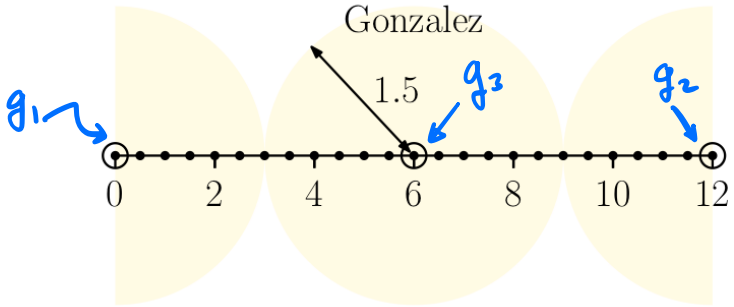
In conclusion: Applying Claim 3 to opt, O,

$$\Delta(G) = 2 \cdot \Delta_{min} \leq 2 \cdot \Delta(O)$$

$\therefore$ Greedy is within factor 2 of opt. □

Example (in 1D)





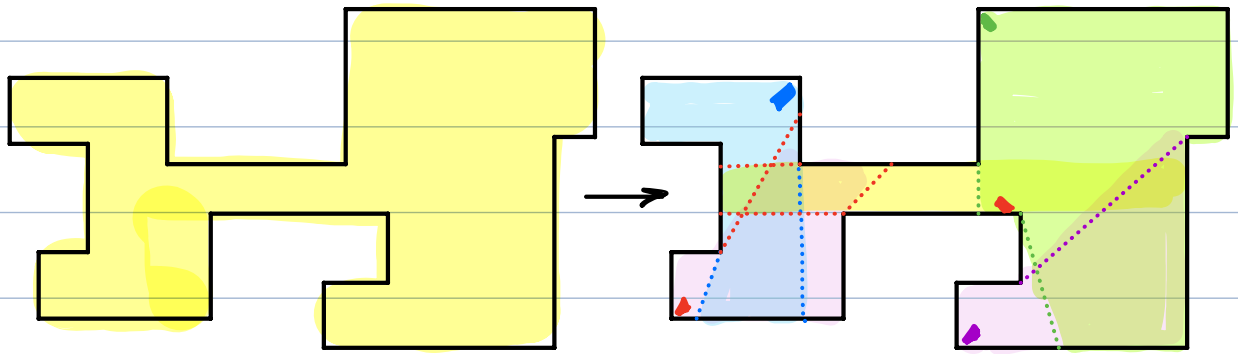
Summary - **k-center** - NP-hard clustering problem
- **Gonzalez** - Greedy alg. for k-center
- Factor **2 approx.** to optimum

CMSC 451 - Algorithm Design
Lecture 7 - Greedy Approximation: Set Cover

Set Cover - Cover a domain with a minimum number of sets

Applications:

Surveillance - Place a min. number of cameras in art gallery.



Wireless/Cellular Coverage - Place min. number of wireless routers/cell towers to cover some region.

Workforce Scheduling - Given when/where employees can work, schedule to cover all times/locations.

Set System: $\Sigma = (X, \mathcal{S})$ where:

$X = \{x_1, \ldots, x_n\}$ domain of things to be covered.

$\mathcal{S} = \{s_1, \ldots, s_m\}$ collection of sets that can be used to build the cover.

Throughout: $n = |X|$, $m = |\mathcal{S}|$

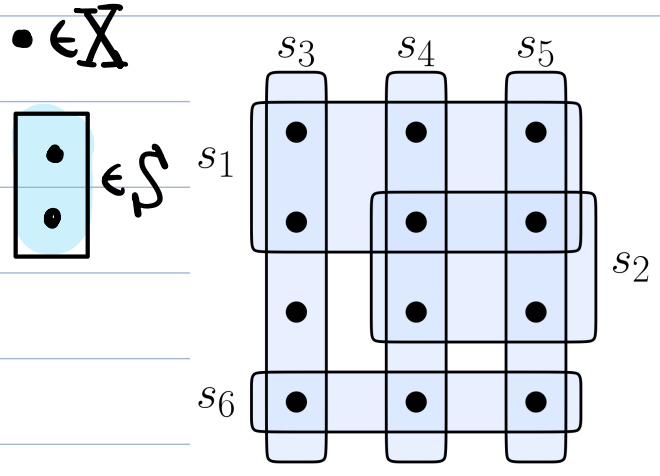
We assume: $s_1 \cup s_2 \cup \ldots \cup s_m = X$ - $\mathcal{S}$ covers X.

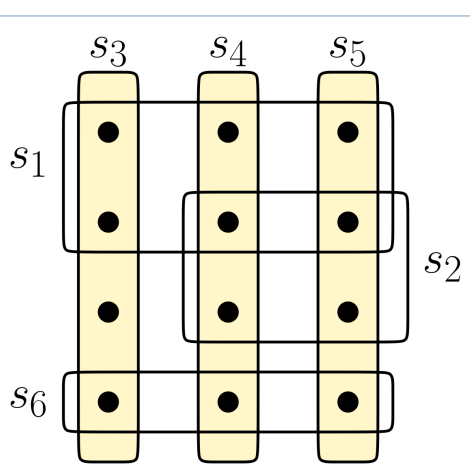
Set Cover Problem - Given a set system $(X, \mathcal{S})$ find smallest number of sets that cover all elements of X.

fewest num. of sets

i.e., Find $C \subseteq \{1, \ldots, m\}$ of min. size s.t.

$\bigcup_{i \in C} s_i = X$ ← cover entire domain





$C = \{3,4,5\}$

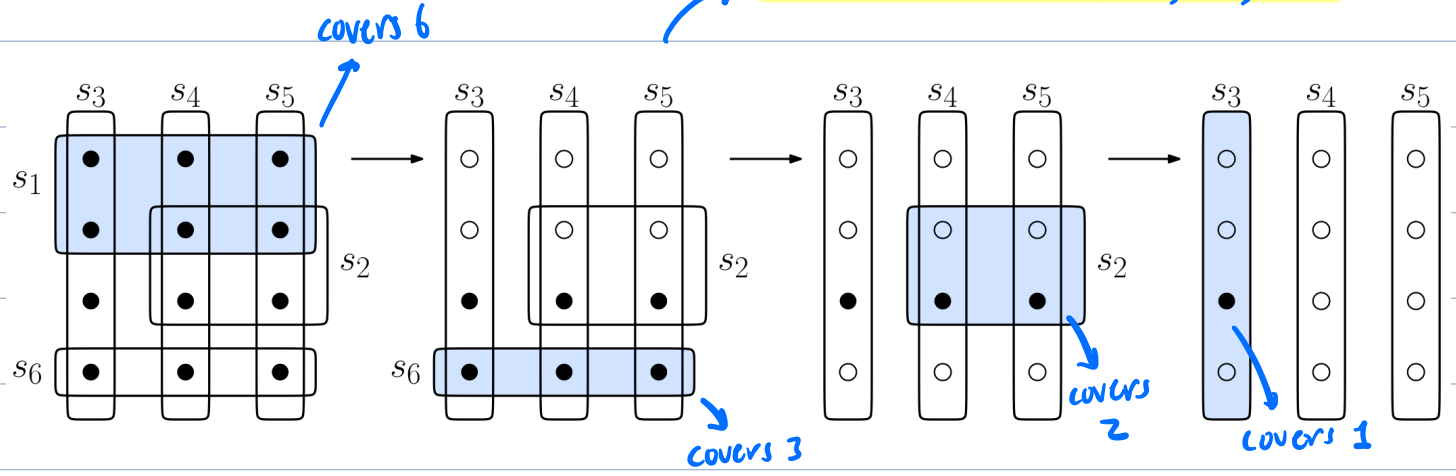
Bad news: Set cover is NP-hard.

We'll prove greedy heuristic gives $(\ln |X|)$ approx.

Greedy Heuristic - Repeatedly add the set that covers the most uncovered items.

```
greedy-set-cover (X, S) {
    U ← X                          // U = uncovered items
    C ← ∅                          // C = indices of cover
    while (U ≠ ∅)
        i ← set s_i covers most elts of U
        add i to C
        U ← U \ s_i                // remove covered items
                                   (set subtraction)
    return C
}
```

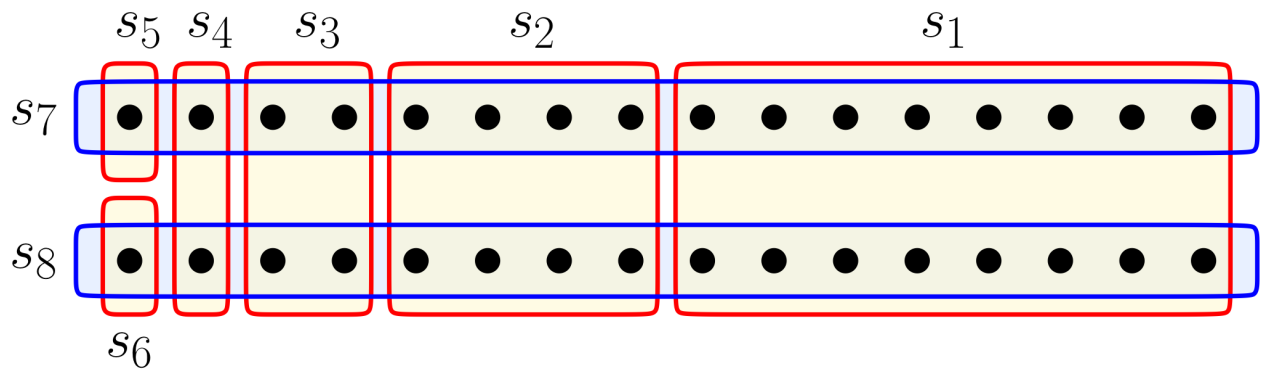
Final cover - $\{s_1, s_6, s_2, s_3\}$



Running Time: $O(n \cdot m)$ $n = |X|$ $m = |S|$

(see pdf notes for more info)

Bad Case for Greedy:



$|X| = 32 = 2^5$

Opt cover = $\{s_7, s_8\}$ - 2 sets

Greedy: s_1, s_7, s_8 all cover 16 → take s_1

s_2, s_7, s_8 all cover 8 → take s_2

s_3, s_7, s_8 all cover 4 → take s_3

⋮ "

s_4, s_5, s_6

Greedy takes 6 sets

We can generalize this to $|X| = n = 2^k$

Opt(O) takes 2 sets. Greedy (G) takes k+1.

Approximation ratio = $\frac{|G|}{|O|} \approx \frac{\lg n}{2} = O(\log n)$

↳ $\lg \equiv \log_2$

We'll show that approx. ratio $\leq \ln n = \ln |X|$ for any set system $(X, \mathcal{S})$

Thm: Given any set system $(X, \mathcal{S})$, let O = opt set cover, G = greedy set cover, then

$$\frac{|G|}{|O|} \leq \ln |X|$$

While true, we'll prove something slightly weaker

Proof: Utility lemma—

Lem: $\forall c > 0$,

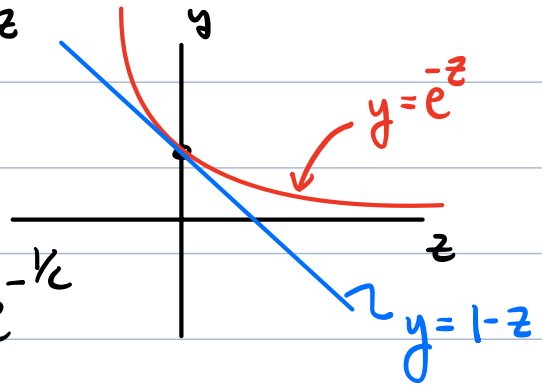
$$(1 - 1/c) \leq e^{-1/c}$$

base of ln $e \approx 2.718\ldots$

Pf: Fact: $\forall z,\ 1 - z \leq e^{-z}$

Setting $z = 1/c$

$$\Rightarrow 1 - 1/c \leq e^{-1/c}$$



Let $o = |O|$, $g = |G| - 1$ → cheat! (simplifies math)

We'll show $g \leq o \cdot \ln n$, $n = |X|$.

(Thus, we're showing: $|G| \leq 1 + |O| \cdot \ln |X|$)

Notation: Let X_i = items that remain uncovered at end of i^{th} iteration

$n_i = |X_i|$ (= |U| in algorithm)

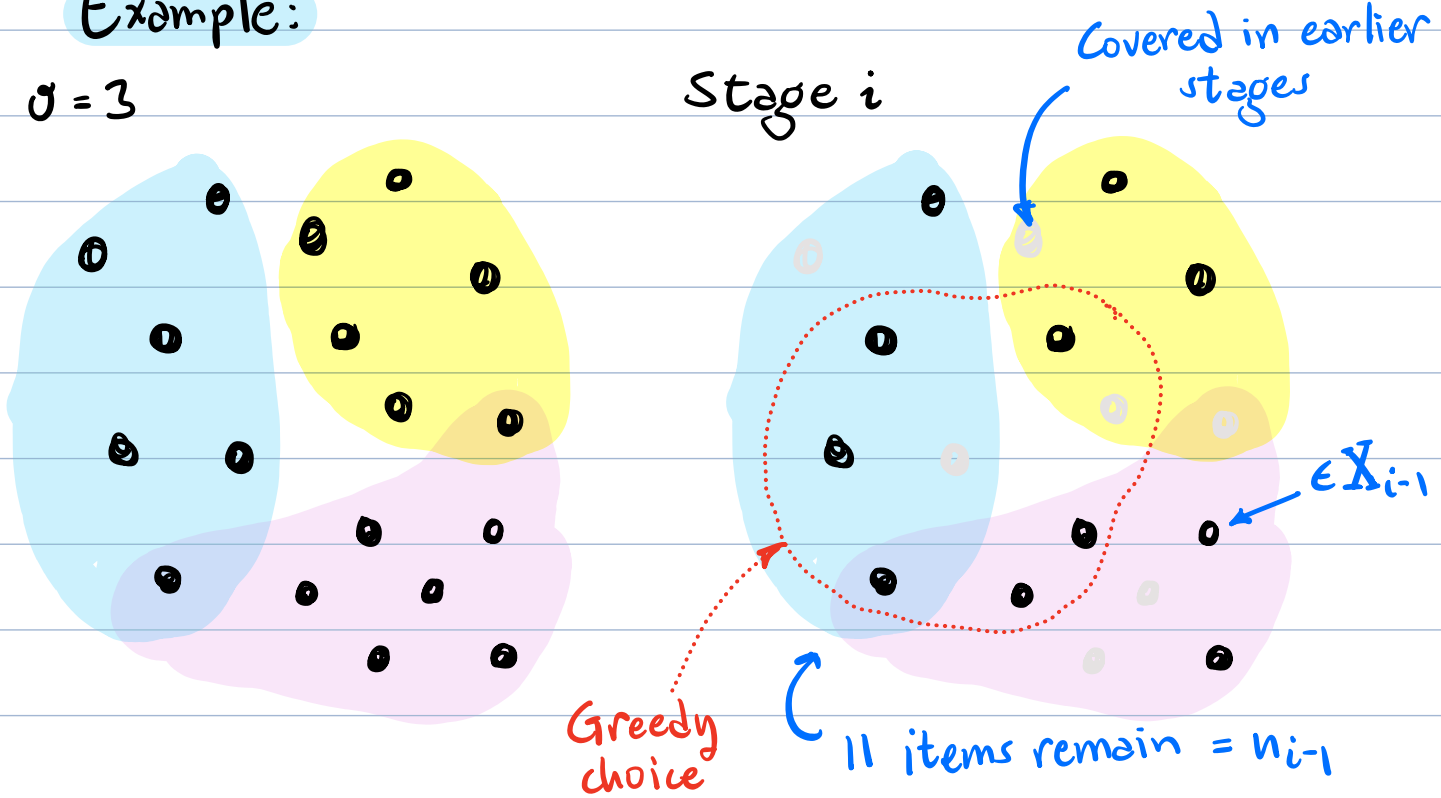
($n_0 = n$)

Observations:

- At start of stage i, n_{i-1} items remain
- $\exists$ cover of X of size o
  - $\Rightarrow \exists$ cover of X_{i-1} of size o
  - $\Rightarrow \exists$ set covers $\geq n_{i-1}/o$ items of X_{i-1}

(pigeonhole principle)

Example:



- Since we can cover all with $o = 3$ sets, some set of S covers $\geq \lceil 11/3 \rceil = 4$ elements.
- Greedy must cover at least this many

$\Rightarrow$ Greedy covers at least n_{i-1}/σ

$\Rightarrow$ $n_i \le n_{i-1} - (n_{i-1}/\sigma)$ remain

$= n_{i-1}(1 - 1/\sigma)$ ← Each stage reduces no. remaining by $*(1-1/\sigma)$

- By induction:

$$n_i \le n_{i-1}(1-1/\sigma) \le n_{i-2}(1-1/\sigma)^2 \le n_{i-3}(1-1/\sigma)^3$$
$$\le n(1-1/\sigma)^i$$

- After iteration g (Recall $g = |G|-1$), still have ≥ 1 item remain.

$\Rightarrow$ $1 \le n_g \le n(1-1/\sigma)^g$

- By Utility Lemma, $(1-1/\sigma) \le e^{-1/\sigma}$

$\Rightarrow$ $1 \le n(e^{-1/\sigma})^g = n \cdot e^{-g/\sigma} = n/e^{g/\sigma}$

$\Rightarrow$ $e^{g/\sigma} \le n$

$\Rightarrow$ $g/\sigma \le \ln n$ (take ln both sides)

$\Rightarrow$ $\frac{|G|-1}{|O|} \le \ln n$ since $g = |G|-1$ + $\sigma = |O|$

Ignoring the "-1", this is what we want! □

How bad is this?

- Ideally approx. ratio can be made arbitrarily small – $(1+\varepsilon)\cdot Opt$
- Next best - Approx ratio is a small constant e.g., $2\cdot Opt$, $3\cdot Opt$
- For set cover, approx ratio grows (slowly) with domain size - $(\ln |X|)\cdot Opt$

Summary:

- Set cover - A fundamental optimization problem.
- NP-hard (later this semester)
- Greedy - A simple heuristic
  (Take the set covering most remaining)
- Proved greedy $\leq$ opt $\cdot (\ln n)$ $n = |X|$

- Hope for better? No
  Can't beat $O(\log n)$ unless $P = NP$.

CMSC 451 - Algorithm Design
Lecture 8 - Dynamic Programming: Weighted Interval Sched.

Dynamic Programming -

- A fundamental **algorithm design principle**
- Involves **recursively** breaking a problem into **subproblems**
- **Optimal substructure** - Property of some optimization problems. To obtain a **globally optimal** result, all subproblems should be solved **optimally**

Isn't this obvious? No - Sometimes it is better to be suboptimal on one subproblem so you can do better on another subproblem

- Developed in 1950's by **Richard Bellman**
  - Bellman-Ford algorithm
  - Coined the term **"curse of dimensionality"**

Overlapping subproblems -

- Prevents methods like **divide-and-conquer**
- Can be solved **top-down** or **bottom-up**

Weighted Interval Scheduling -

Given a set of **n requests** to be **scheduled** on an exclusive resource.

Each request has:

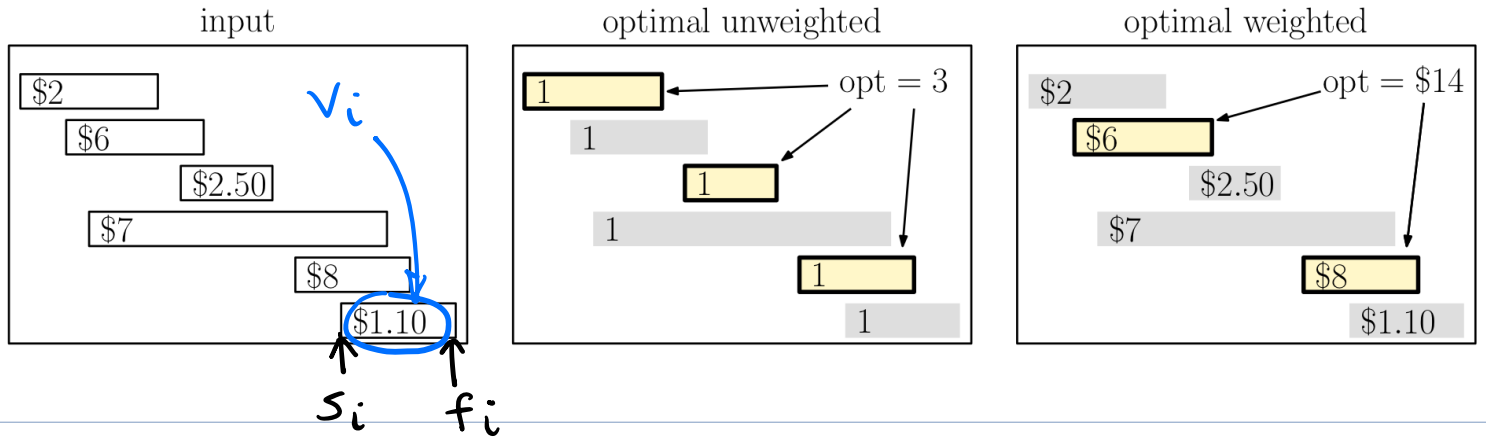
- **start-finish time** interval - $[s_i, f_i]$
- **weight or value** - v_i

Objective - Schedule a set of **non-overlapping** requests to maximize **sum of weights**.

Example: People make **bids** to use **picnic table** at local park

$[s_i, f_i]$ - **when** they want to use it

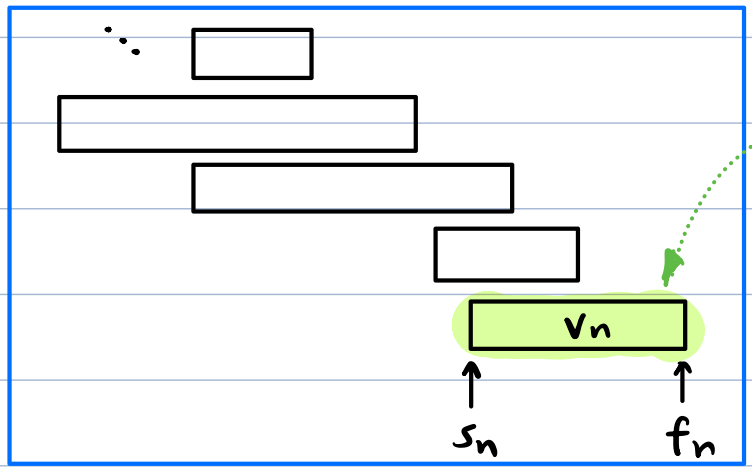
v_i - amount they'll **pay** for use



Note: Greedy **does not** work for **weighted** version.

Recursive Formulation:

- Assume requests sorted by finish times
- Consider the last request

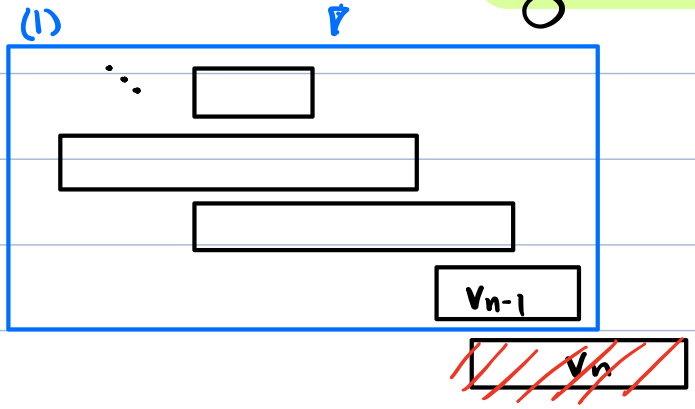


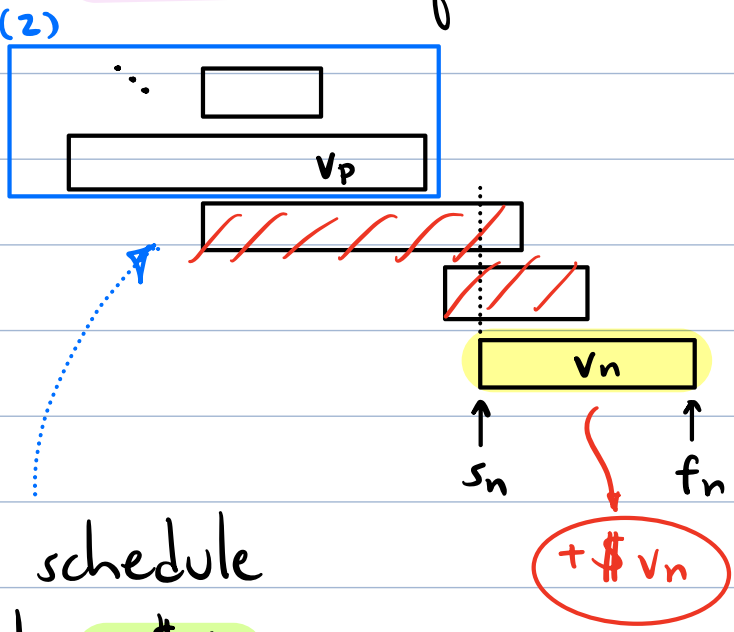
Why last, not first? Doesn't matter, but notation is a bit simpler.

- Two possibilities:

(1) $[s_n, f_n]$ not in opt schedule

- Ignore it + recurse on requests $1..n-1$

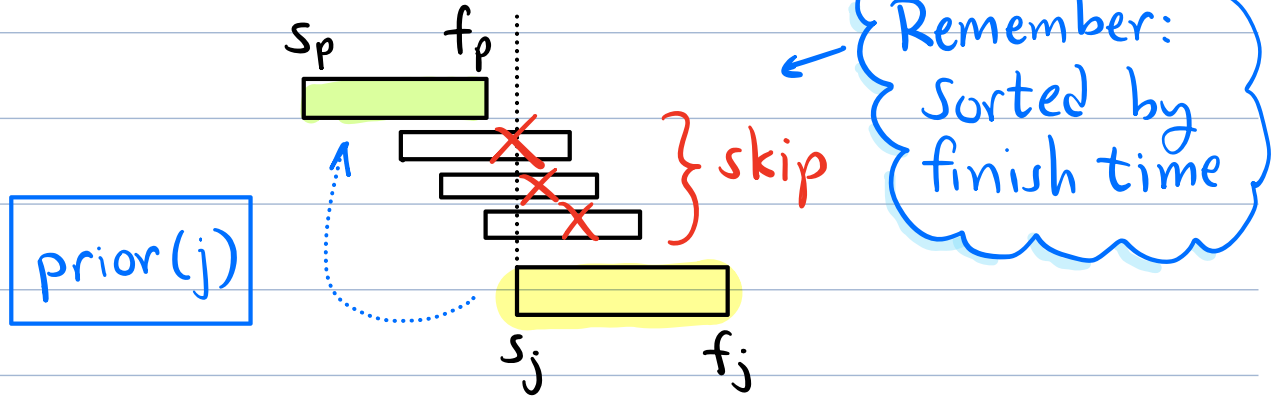




(2) $[s_n, f_n]$ is in opt schedule

- Add to schedule + $\$ v_n$
- Skip overlapping intervals $\{p+1, \ldots, n-1\}$
- Recurse on requests $1..p$

When we add a request j to schedule, how many should we skip?

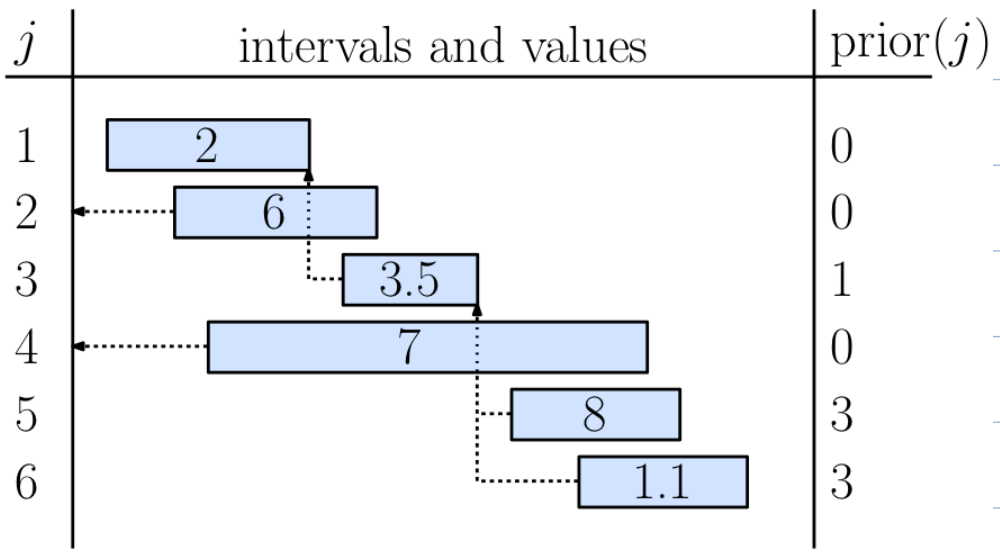


Define: for $1 \le j \le n$

$$\text{prior}(j) = \max_{p} \text{ s.t. } f_p < s_j$$

(or 0 if there is none)

Example of prior:



Optimal Total Value:

for $0 \le j \le n$, $W(j)$ = max value possible for requests $1, 2, \ldots, j$

$$W(j) = \begin{cases} 0 & \text{if } j = 0 \quad \text{(basis)} \\ \max \begin{cases} W(j-1) & \text{(reject)} \\ v_j + W(\text{prior}(j)) & \text{(accept)} \end{cases} & \text{if } j > 0 \end{cases}$$

skips requests that overlap j

Recursive implementation: (+ why this is bad!)

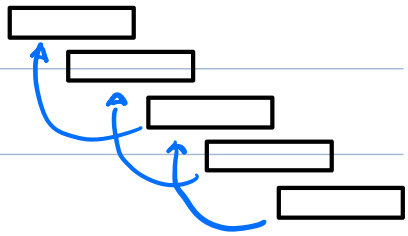
```
WIS(s[1..n], f[1..n], v[1..n])
    Sort requests by finish time
    Compute prior[j] (for 1≤j≤n)
    return rec-WIS(n)          // total value

rec-WIS(j)                     // value of 1..j
    if (j = 0) return 0
    else return max { rec-WIS(j-1)          // don't take
                    { v[j] + rec-WIS(prior[j])
                                            // take
```

Correct - Yes - just implements function W(j)

Too Slow! Why?

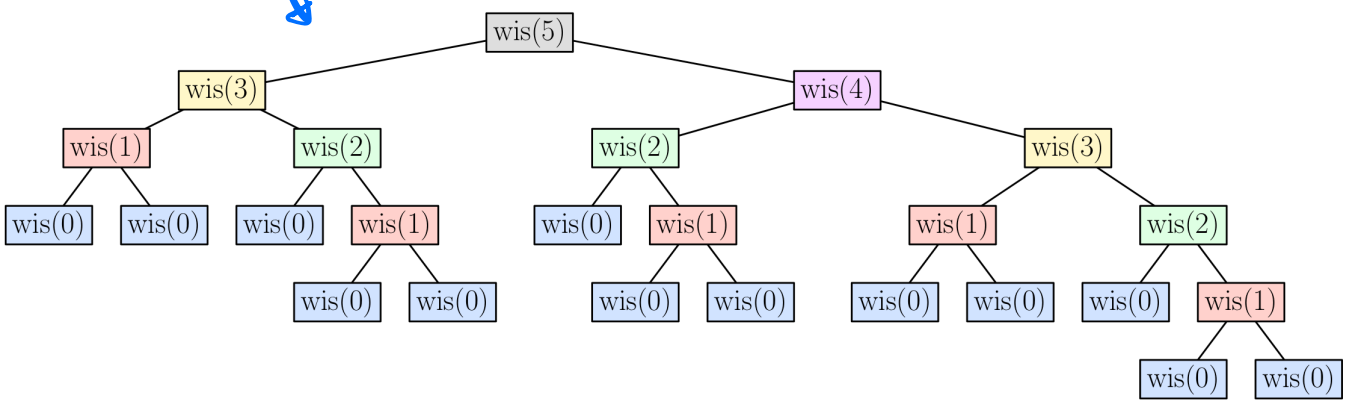
Let T(j) = num. of recursive calls to recWIS(0) arising from recWIS(j)

Suppose prior(j) = j-2 , ∀j



rec-WIS(j) calls:

- rec-WIS(j-1)
- rec-WIS(prior(j)) = rec-WIS(j-2)



⇒ T(j) = T(j-1) + T(j-2) } Fibonacci!

+ T(0) = 1

Grows fast!

j :	0	1	2	3	4	5 ...	30	50
T(j):	1	2	3	5	8	13 ...	2 Meg!	30 Gig!

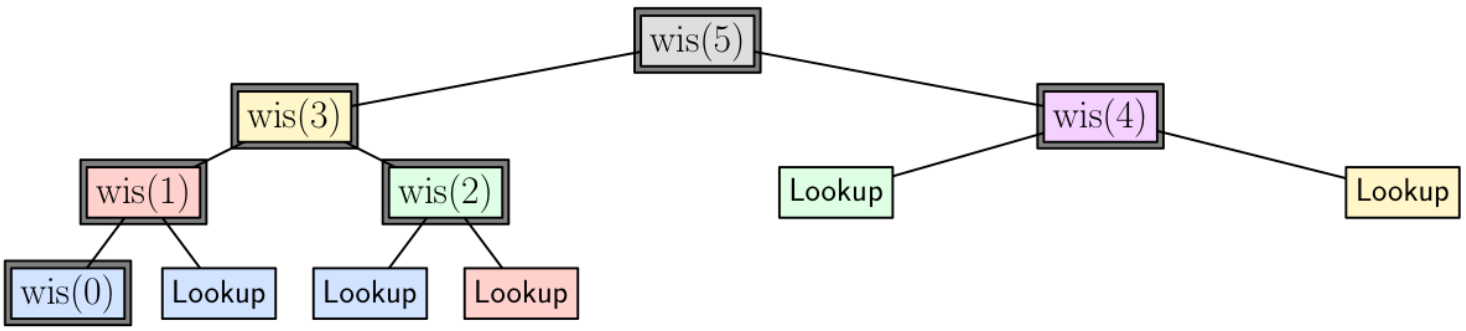
Fibonacci grows as ϕ^j, $\phi = \frac{1}{2}(1-\sqrt{5})$

How do we improve?

Memoization (a.k.a. caching)

Idea: After computing $W(j)$, save it in an array, say $W[j]$.

- Next time we need its value, look it up.
- Results in many fewer recursive calls, $O(n)$



Updated implementation:

- Sort by finish times } (as before)
- Compute prior[ ] array }
- Init: $W[j] = -1$ (means "undefined")
- Array: accept[1..n]

  accept[j] = True - accept request j

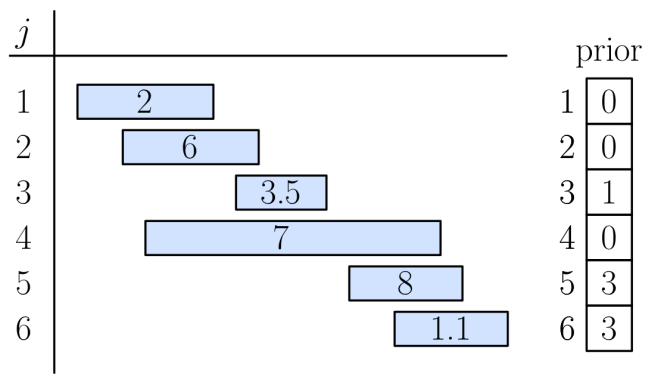
  False - reject " "

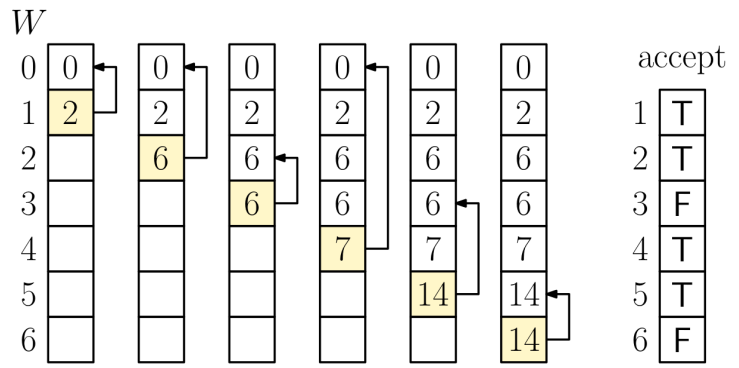
→ We'll use this to construct final schedule (later)

- return memo-WIS(n)

```
memo-WIS(j)                                  // memoized WIS
    if ( W[j] = -1 )                         // W[j] undefined?
        if ( j = 0 ) W[j] ← 0                // basis
        else
            rejVal ← memo-WIS(j-1)           // rej/acc values
            accVal ← v[j] + memo-WIS(prior[j])
            if ( rejVal > accVal )           // better to reject
                W[j] ← rejVal
                accept[j] ← false
            else                             // better to accept
                W[j] ← accVal
                accept[j] ← true
    return W[j]                              // return value
```

Example: (W-values are created bottom-up)





Running Time - $O(n)$ [n+1 rec. calls, each $O(1)$]

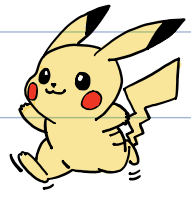
Bottom-Up Construction:

- In practice it is often more efficient to unravel the recursion, and build W bottom-up
- As before:
  - Sorted by finish times
  - prior[...] computed

```
bottom-up-WIS()          // bottom-up implementation
    W[0] ← 0             // basis
    for (i ← 1 to n)
        rejVal ← W[j-1]          // rej/acc values
        accVal ← v[j] + [W[prior[j]]
        if (rejVal > accVal)     // better to reject
            W[j] ← rejVal
            accept[j] ← false
        else                     // better to accept
            W[j] ← accVal
            accept[j] ← true
    return W[n]          // final value
```

Running Time - $O(n)$ (obvious)

[Assuming sorted + prior computed → $O(n \log n)$



Computing the Final Schedule:

- So far we only compute the final value, $M[n]$
- Use the accept[] array to guide us

Start at end ($j \leftarrow n$) + work back (until $j == 0$)

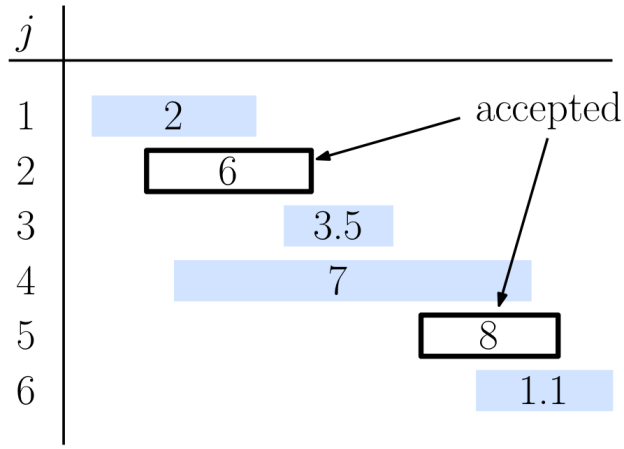
if accept[j] = True:
- add j to schedule
- continue with $j \leftarrow$ prior[j]

skip over overlapping requests

else:
- don't add to schedule
- continue with $j \leftarrow j-1$

```
get-schedule()          // get final schedule
    j ← n               // start at end
    sched ← ∅           // init empty sched.
    while (j > 0)
        if (accept[j])  // accepted j
            prepend [j] to sched
            j ← prior[j]
        else            // rejected j
            j ← j-1
    return sched        // final schedule
```

Example:



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

	accept	
1	T	
2	T	Accept 2
3	F	Reject 3
4	T	
5	T	Accept 5
6	F	Reject 6

Trace: $j \leftarrow 6$

accept[6] = F $j \leftarrow 6-1 = 5$

accept[5] = T add 5, $j \leftarrow$ prior[5] = 3

accept[3] = F $j \leftarrow 3-1 = 2$

accept[2] = T add 2, $j \leftarrow$ prior[2] = 0

$j = 0 \rightarrow$ terminate

Note: Even though
accept[4] = T
accept[1] = T
we never visit these

Summary -

- Intro. to dynamic programming
  - Recursive structure (subproblems)
  - Principle of optimality
- Weighted Interval Scheduling
  - Recursive (slow!), memoized, bottom-up

CMSC 451 - Algorithm Design
Lecture 9 - DP: LCS and Edit Distance

Strings - Used in **document processing** & computational **genomics**

This lecture - **Dynamic Programming** (DP) algorithms for two string processing problems:
- **Longest common subsequence** (LCS)
- **Edit Distance**

Notation - X is a **string** $= \langle x_1, \ldots, x_m \rangle$ over some **alphabet** Σ.

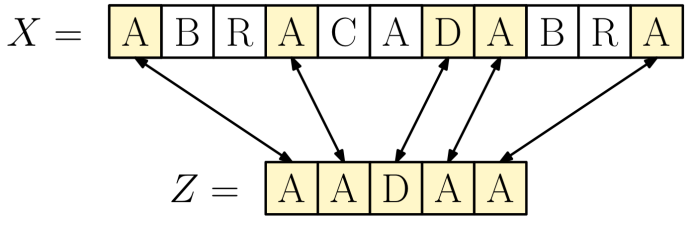
e.g. $\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{A, C, G, T\}$

$|X|$ = **length of** X

X_i = **prefix** $\langle x_1, \ldots, x_i \rangle$ $\quad X_0 = \langle\rangle$

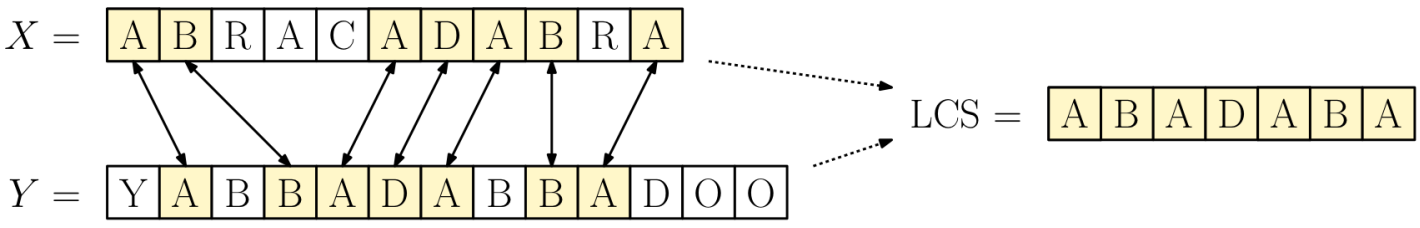
A string $Z = \langle z_1, \ldots, z_k \rangle$ is a **subsequence** of X if Z's characters appear **in order** in X.

E.g.



Given strings X + Y, their longest common subsequence (LCS) is a max length string that a subsequence of both

Example:



Note: The LCS is not unique

LCS($\langle ABC \rangle$, $\langle BAC \rangle$) = $\langle AC \rangle$ or $\langle BC \rangle$

DP Formulation for LCS:

- Decompose into subproblems (recursive)
- Priniciple of optimality will apply
  (subproblems should be solved optimally)

Define: For $0 \le i \le m$, $0 \le j \le n$:

lcs(i,j) = length of LCS for prefixes $X_i = \langle x_1 \dots x_i \rangle$ + $Y_j = \langle y_1 \dots y_j \rangle$

Eg. $X_5 = \langle ABRAC \rangle$ $Y_6 = \langle YABBAD \rangle$

lcs$(5,6) = 3$ ($\langle ABA \rangle$)

Basis: If $i = 0$ or $j = 0$ (empty string)
then LCS is empty

$$\Rightarrow \text{lcs}(i,0) = \text{lcs}(0,j) = 0$$

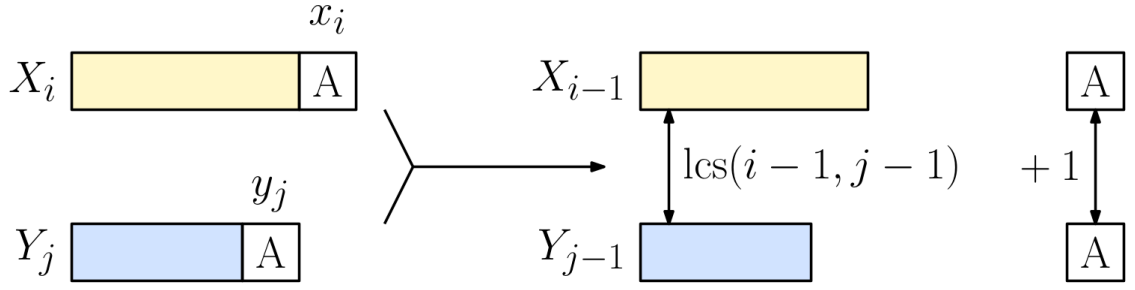
Last characters match: $x_i = y_j$

(Suppose $x_i = y_j = 'A'$)

Claim: LCS also ends in 'A'

Proof: Obvious. If not, we could extend it by appending an 'A'.

- Since LCS ends in 'A', we may as well assume it comes from mathing x_i with y_j.
  (There is no benefit from matching it earlier.)
- Once matched, x_i + y_j are elimated from further consideration.
- We should do our best with remainders
  $X_{i-1} = \langle x_1 \dots x_{i-1} \rangle$ + $Y_{j-1} = \langle y_1 \dots y_{j-1} \rangle$

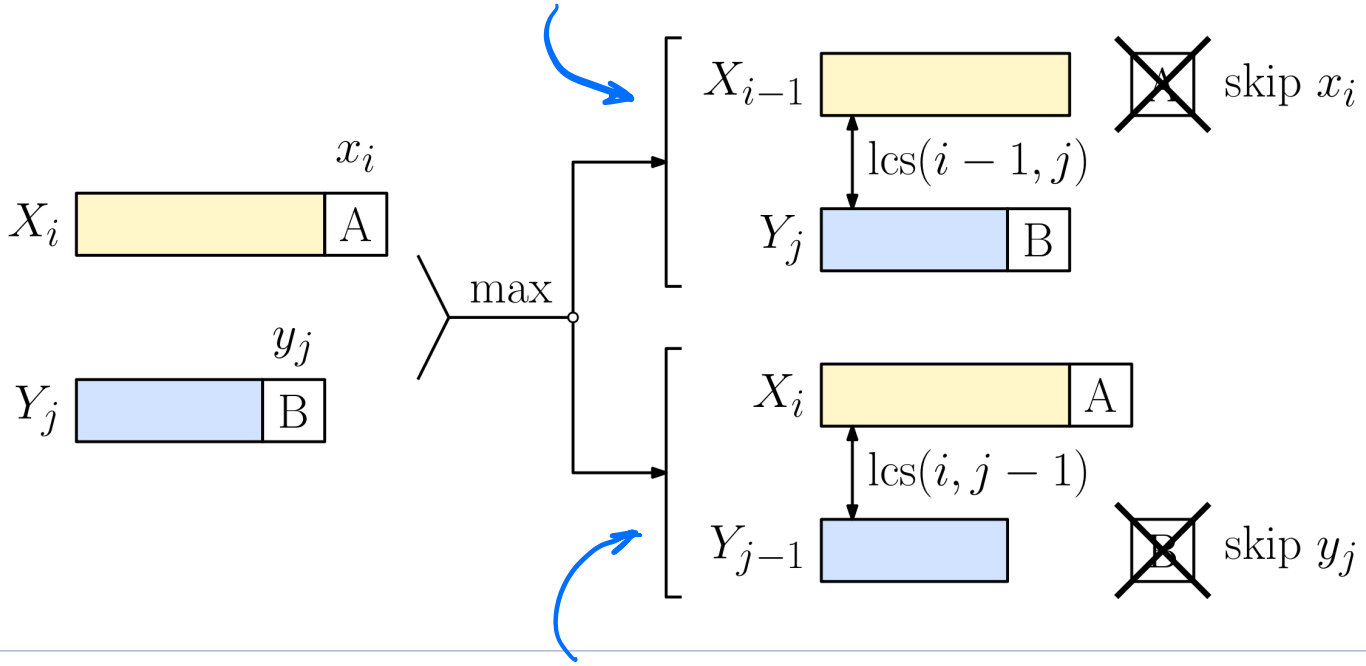


$$\Rightarrow \text{if } (x_i = y_j)\ \text{lcs}(i,j) = \text{lcs}(i-1, j-1) + 1$$

Last characters do not match: $x_i \neq y_j$

- Either x_i or y_j or both are <u>not</u> in LCS.

- x_i is not in LCS
  - we may ignore x_i + continue matching remainder $X_{i-1} = \langle x_1 \dots x_{i-1} \rangle$ with Y_j
  - $\Rightarrow$ $lcs(i-1, j)$



- y_j is not in LCS
  - (symmetrical) ignore y_j + continue matching remainder Y_{j-1} with X_i
  - $\Rightarrow$ $lcs(i, j-1)$

- Both x_i + y_j not in LCS
  - This will be handled by above cases.

- But which?

DP Credo: Don't be smart.
Try 'em all. Take the best.

$$\Rightarrow \text{if } (x_i \neq y_j)\ lcs(i,j) = \max \begin{cases} lcs(i-1, j) \\ lcs(i, j-1) \end{cases}$$

Final DP Formulation:

$$lcs(i,j) = \begin{cases} 0 & \text{if } \min(i,j) = 0 \\ 1 + lcs(i-1, j-1) & \text{if } x_i = y_j \quad (i,j>0) \\ \max \begin{cases} lcs(i-1,j) \\ lcs(i,j-1) \end{cases} & \text{if } x_i \neq y_j \quad (i,j>0) \end{cases}$$

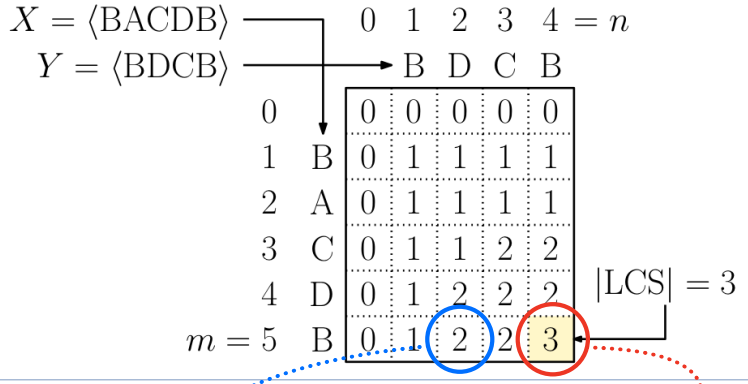
- Correctness follows from earlier derivation
- Recursive implementation will take exp. time
- Instead: Build table $lcs[0..m, 0..n]$ through
  - memoization (caching)
  - or - bottom-up

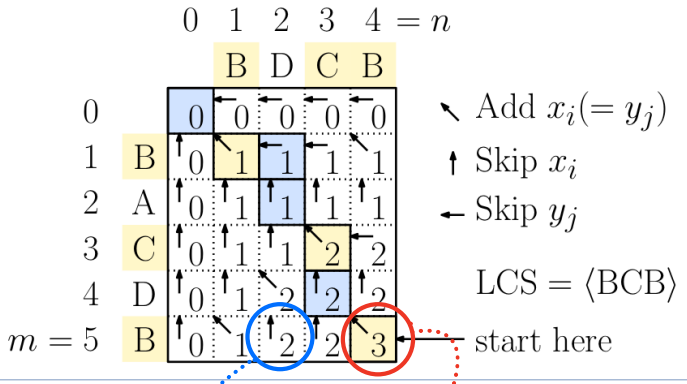
Memoized Implementation (+ Hooks)

- Build table lcs[i,j] recursively
- Add table H[0..m,0..n] to remind us of decisions made, so we can reconstruct LCS.
- Init: lcs[i,j] ← -1 (undefined)
- Final result: memo-lcs(m,n) $m = |X|, n = |Y|$

```
memo-lcs(i,j)                          // memoized LCS
    if (lcs[i,j] = -1)                 // undefined?
        if (i = 0 or j = 0)            // basis
            lcs[i,j] ← 0
        else if (x_i = y_j)            // match?
            lcs[i,j] = 1 + memo-lcs(i-1,j-1)
            H[i,j] = '↖'
        else // x_i ≠ y_j              // don't match
            skipX ← memo-lcs(i-1,j)    // lcs if skip x_i
            skipY ← memo-lcs(i,j-1)    // lcs if skip y_j
            if (skipX ≥ skipY)         // better to skip x_i
                lcs[i,j] ← skipX ; H[i,j] ← '↑'
            else                       // better to skip y_j
                lcs[i,j] ← skipY ; H[i,j] ← '←'
    return lcs[i,j]                    // final lcs value
```

Running time: $O(n \cdot m)$





$x_5 = y_4 = 'B'$

$\Rightarrow lcs[5,4] = lcs[4,3] + 1 = 3$

$\langle BCB \rangle = \langle BC \rangle + 'B'$

Add $x_5 = y_4 = 'B'$ to LCS

Cont. with $H[4,3]$

No change to LCS: $\langle BD \rangle$

Cont. with $H[4,2]$

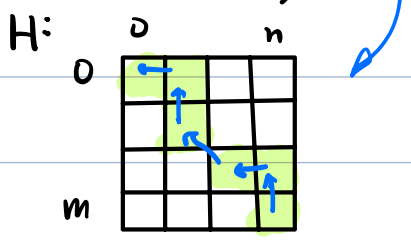
$x_5 = 'B' \neq y_2 = 'D'$

$\Rightarrow lcs[5,2] = \max(lcs[4,2], lcs[5,1]) = 2$

$\langle BD \rangle = \max(\langle BD \rangle, \langle B \rangle)$

Extracting the LCS:

- We use the **H matrix**
- Start at **H[m,n]** & trace back to **H[0,0]**



- **Entries:** H[i,j]
  - '↖': **Add $x_i = y_j$** to LCS, continue with **H[i-1, j-1]**
  - '↑': **Skip x_i**. Continue with **H[i-1, j]**
  - '←': **Skip y_j**. Continue with **H[i, j-1]**

- Note that changes to i+j mimic recursive structure

```
get-lcs-sequence()          // get the LCS sequence
    LCS ← ∅                 // initialize
    i ← m; j ← n            // start at bottom-right
    while (i ≠ ∅ or j ≠ 0)  // end at top-left
        switch (H[i,j])
            '↖' : prepend x_i to LCS   // match x_i = y_j
                  i--; j--
            '↑' : i--                  // skip x_i
            '←' : j--                  // skip y_j
    return LCS
```

(see figure above for example)

Running time: $O(n+m)$ - Each iteration decrements either i or j (or both)

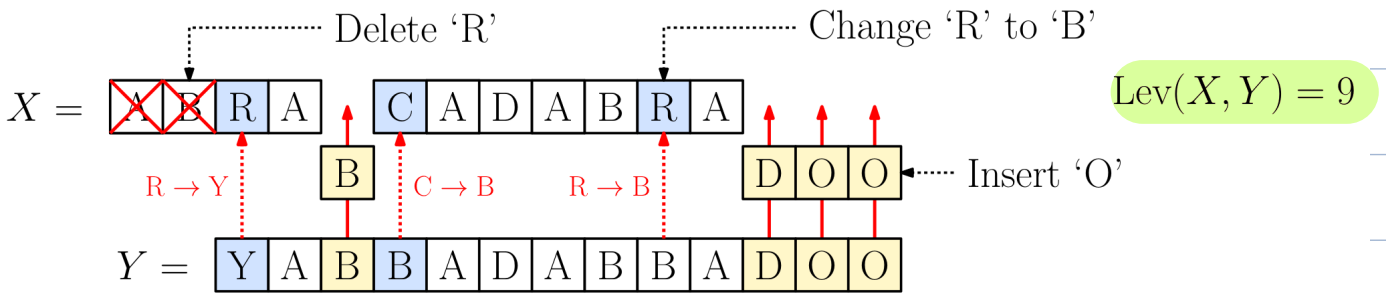
Bottom-up implementation (see pdf for details)

- fill row by row + col by col: $i \leftarrow 0..m$, $j \leftarrow 0..n$
- Also $O(n \cdot m)$

Edit Distance:

- Widely used in genomics
- Given $X = \langle x_1, \ldots x_m \rangle$ & $Y = \langle y_1, \ldots y_n \rangle$ how many edit ops are needed to convert X into Y, where edit ops:
  - insert a char of Y into X
  - delete a char of X
  - change a char of X to match a char of Y
- This defines a metric on strings called the Levenshtein distance (Vladimir Levenshtein)

Example: Change $\langle$ABRACADABRA$\rangle \rightarrow \langle$YABBADABBADOO$\rangle$



- We'll present the recursive DP formulation.
  - Implementable (memoized or bottom-up) in $O(m \cdot n)$ time

- Structurally similar to LCS
- Cute trick - inserting into X is like deleting from Y

Definition: Given $X = \langle x_1 \dots x_m \rangle$ & $Y = \langle y_1 \dots y_n \rangle$

for $0 \le i \le m$ & $0 \le j \le n$

$Lev(i,j)$ = Levenshtein distance between $X_i = \langle x_1 \dots x_i \rangle$ & $Y_j = \langle y_1 \dots y_j \rangle$

Final goal - $Lev(m,n)$

Basis:

$i = 0$ - No chars in X. Need j inserts from Y

$\Rightarrow Lev(0,j) = j$

$j = 0$ - No chars in Y. Need i deletes from X

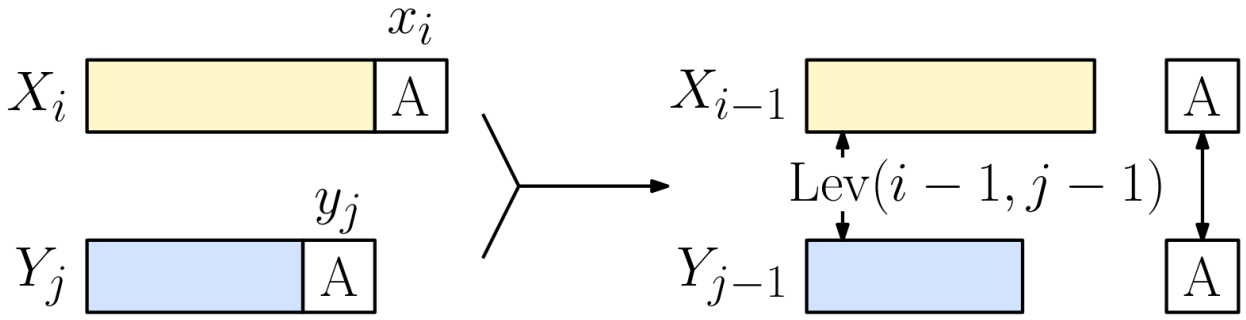
$\Rightarrow Lev(i,0) = i$

- otherwise if $\min(i,j) > 0$

Last characters match: $x_i = y_j$

- We should go ahead & match them & continue with remainders X_{i-1} & Y_{j-1}

$\Rightarrow$ if $(x_i = y_j)$ $Lev(i,j) = Lev(i-1, j-1)$



Last characters do not match: $x_i \neq y_j$

- Need to do something, but what?
  1. Insert y_j at end of X_i
  2. Delete x_i
  3. Change x_i into y_j

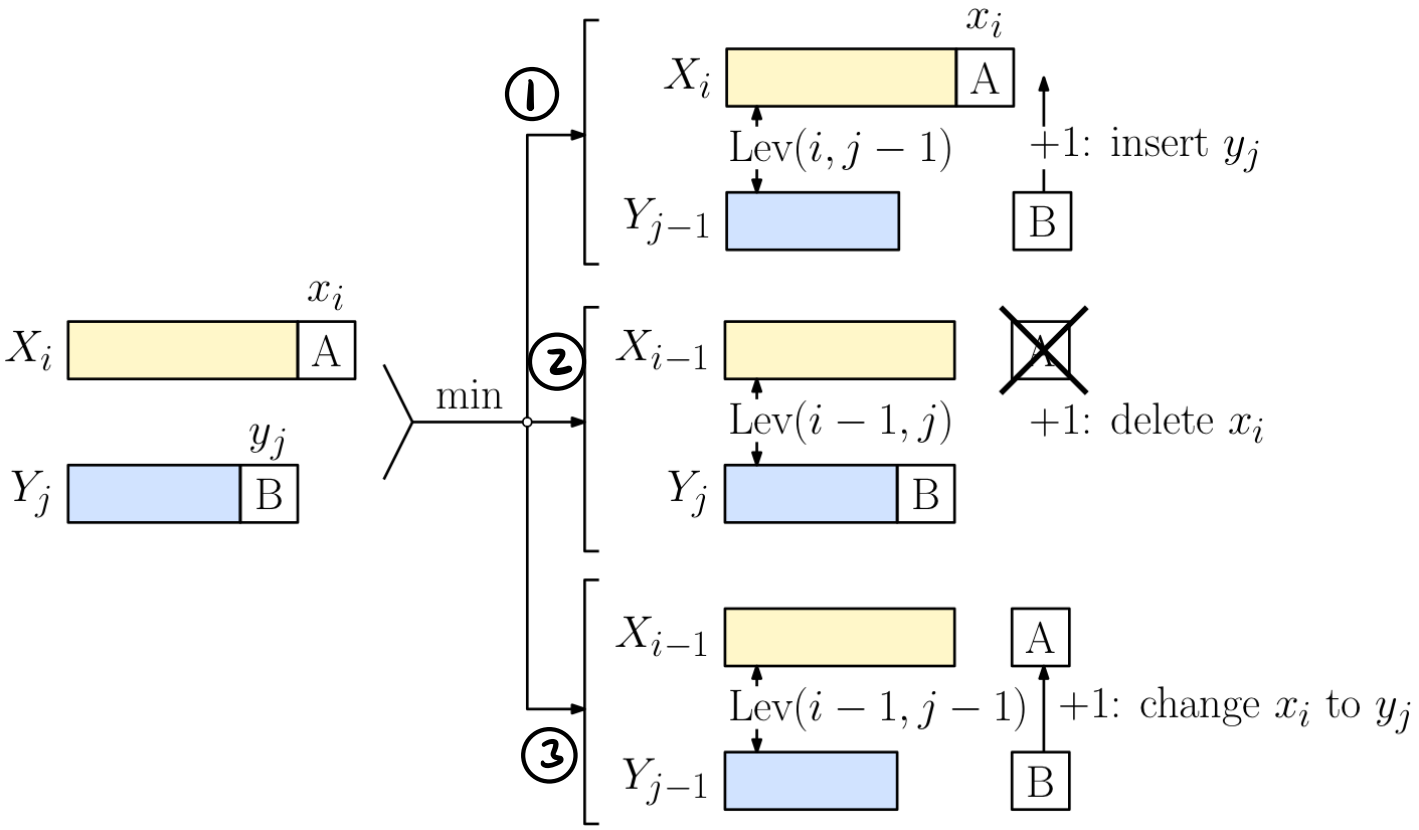
- In any of these cases, distance goes up by +1

If ①, we are done with y_j
+ continue with remainder Y_{j-1}
$\Rightarrow 1 + \text{Lev}(i, j-1)$

If ②, we are done with x_i
+ continue with remainder X_{i-1}
$\Rightarrow 1 + \text{Lev}(i-1, j)$

If ③, we are done with both x_i + y_j
Continue with remainders X_{i-1}, Y_{j-1}
$\Rightarrow 1 + \text{Lev}(i-1, j-1)$

3 options:



Which option?

- Remember the DP Credo - Try all
  Take best (min)
- Final DP formulation:

$$Lev(i,j) = \begin{cases} j & \text{(insert all } Y_j\text{)} & \text{if } i=0 \\ i & \text{(delete all } X_i\text{)} & \text{if } j=0 \\ Lev(i-1, j-1) & \text{(match)} & \text{if } i,j>0 + x_i = y_j \\ 1 + \min \begin{cases} Lev(i, j-1) \\ Lev(i-1, j) \\ Lev(i-1, j-1) \end{cases} & \begin{cases} \text{insert } y_j \\ \text{delete } x_i \\ \text{change } x_i \to y_j \end{cases} & \text{if } i,j>0 + x_i \neq y_j \end{cases}$$

Leave implementation as exercise.

- $O(n \cdot m)$ like LCS
- Can extract edits in $O(m+n)$ time

Summary:

DP Algorithms for

- Longest Common Subsequence (LCS)
- Edit (Levenshtein) distance

- Both run in $O(n \cdot m)$ time (quadratic)

↳ Can we do better?

LCS - Yes - Near linear time

Levenshtein - In practice - yes

In theory - no

↓

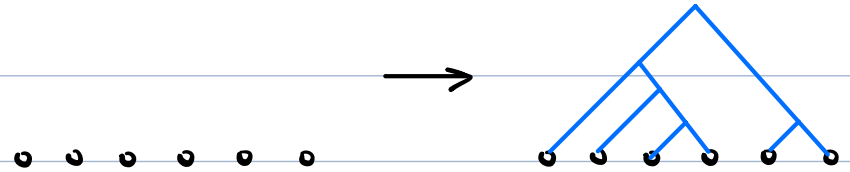
Lower bound of $O(n^{2-\varepsilon})$ for any $\varepsilon > 0$ under the Strong Exponential Time Hypothesis (SETH)

CMSC 451 - Algorithm Design
Lecture 10 - DP: Chain Matrix Multiplication

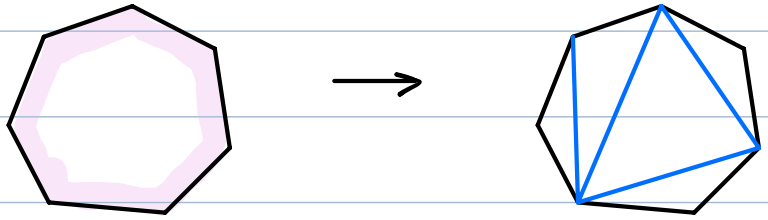
Dynamic programming for tree structures -
- Many optimization problems produce a **tree structure** as output

Optimal:

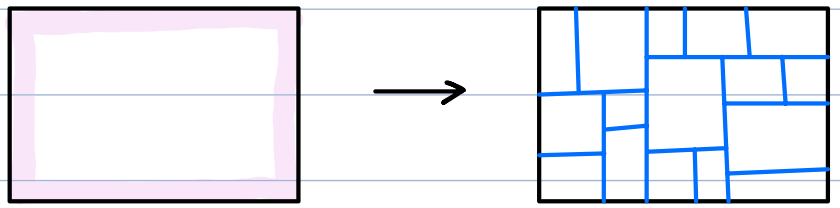
Binary search tree



Polygon triangulation



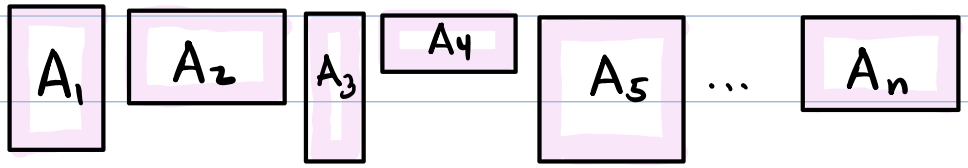
Quadtree or k-d tree



Chain Matrix Mutiplication -
A simple example of **tree-based DP** solutions.

Problem: Given a **sequence of matrices** (of various sizes), what is the **best evaluation order**?

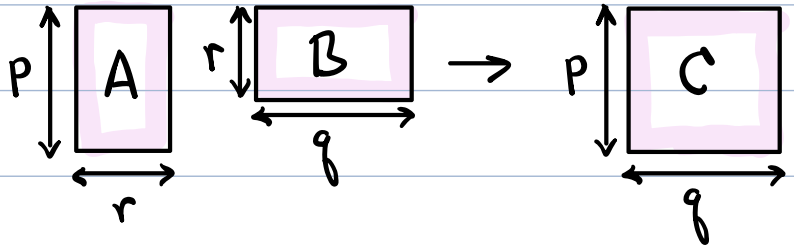
$A_1 \cdot A_2 \cdot \dots \cdot A_n$



Facts:

- Matrix mult is **associative**, but **not commutative**
  - ⇒ Can **parenthesize** as you like, but **cannot change order**

- To multiply: $A \cdot B \to C$
  - A is $p \times r$ (p rows, r columns)
  - B is $r \times q$
  - C is $p \times q$

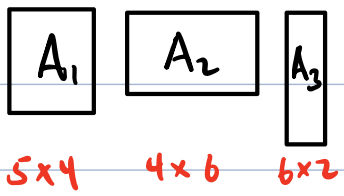


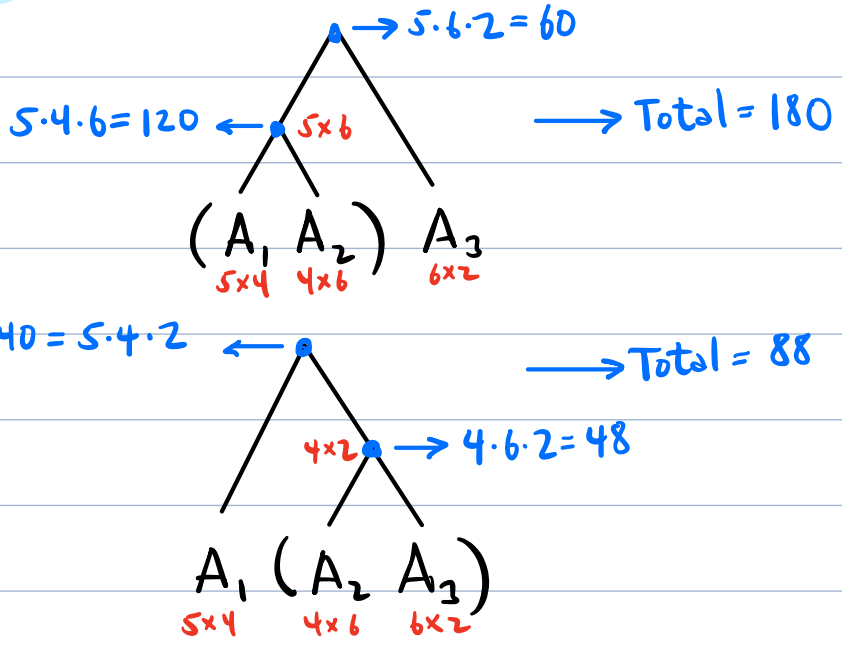
$$(p \times r)(r \times q) \to (p \times q)$$

for $1 \le i \le p$, $1 \le j \le q$, $c_{ij} = \sum_{k=1}^{r} a_{ik} \cdot b_{kj}$

- Time to multiply $\sim p \cdot q \cdot r$ (simplified)

Order matters:



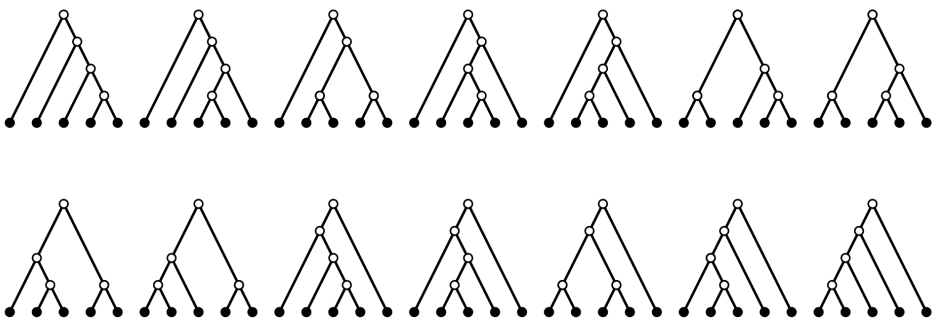


Chain Matrix Mult Problem:

Given a seq. of matrices $A_1, \ldots, A_n$ presented by their dimensions $\bar{p} = \langle p_0, p_1, \ldots, p_n \rangle$ where A_i is $p_{i-1} \times p_i$, determine the order of multiplications (binary tree) that minimizes num. of ops.

Brute force Solution?

- Enumerate all binary trees with n leaves
- Exponentially many!
- n = 5:



- Number of possible parenthesizings:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n \geq 2 \end{cases}$$

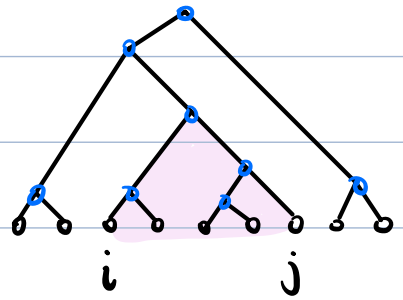
Solution is related to famous combinatorial function, Catalan Numbers. $P(n) = C(n-1)$

$$C(n) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Aside: Named for Eugène Catalan (1800's) but originally discovered in 1730's by Mongolian mathematician/astronomer Ming Antu.

DP Solution to Chain Matrix Mult:

- How to form subproblems for partioning problems?



- For $1 \leq i \leq j \leq n$, let
  - $A(i,j) = A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$
  - $M(i,j)$ = min. num. of ops to compute $A(i,j)$
- Goal: $M(1,n)$

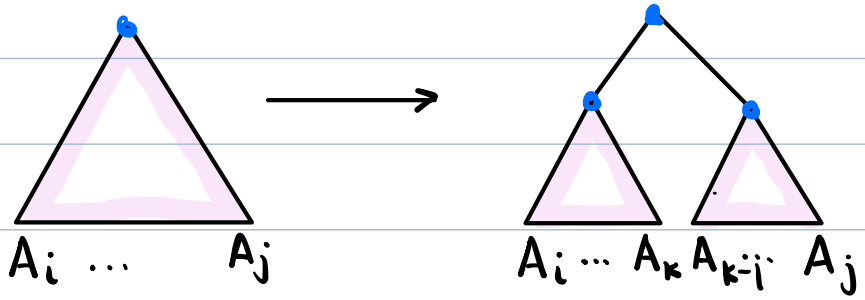
Deriving the recursive formulation:

- $A(i,i) = A_i$ — trivial (no ops. needed)
- Assume $i < j$

$$A_i \cdot A_{i+1} \cdots A_j = A(i,j)$$

$p_{i-1} \times p_i \quad p_i \times p_{i+1} \cdots p_{j-1} \times p_j \qquad p_{i-1} \times p_j$

- Where's the top split?



$$(A_i \cdots A_j) = (A_i \cdots A_k)(A_{k+1} \cdots A_j) \quad \text{for } i \le k \le j-1$$

$$A(i,j) = A(i,k) \cdot A(k+1,j)$$

Issues:

1. Where to split? (value of k)
2. Cost to compute $A(i,k) \cdot A(k+1,j)$?
3. How many ops for last multiplication?

Answers:

① DP Credo - Try all k. Take the best

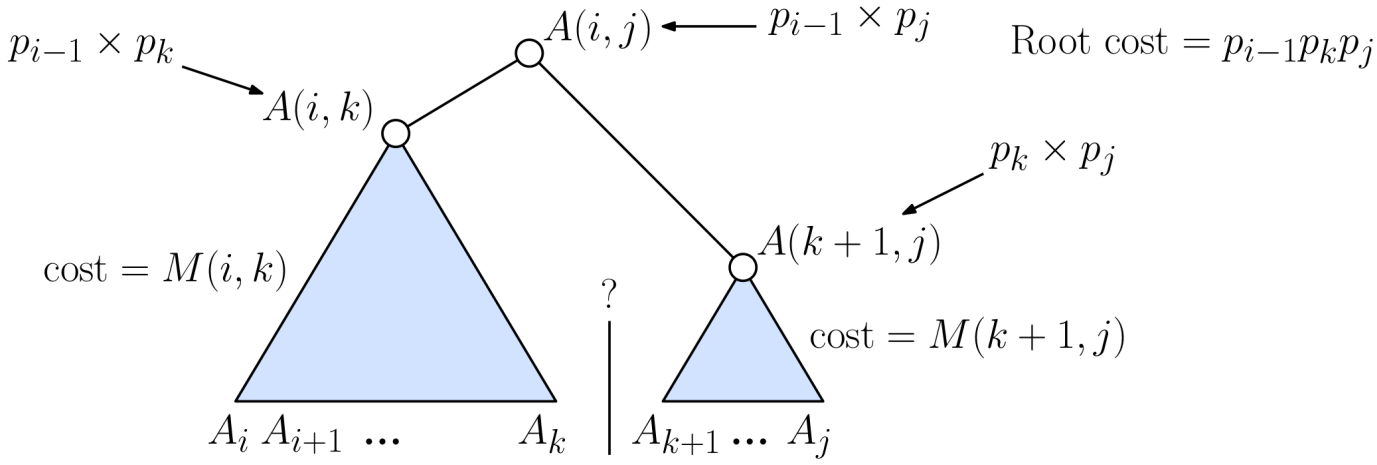
② Princip. of Optimality - Best possible

$M(i,k) + M(k+1,j)$

③ $A(i,k)$ is $p_{i-1} \times p_k$

$A(k+1,j)$ is $p_k \times p_j$

$A(i,k) \cdot A(k+1,j)$ takes $p_{i-1} \cdot p_k \cdot p_j$ ops



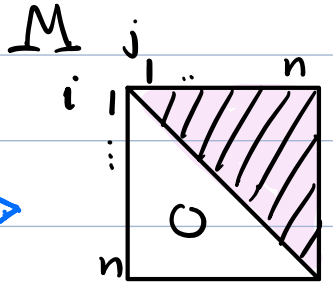
Recursive DP formulation: for $1 \le i \le j \le n$

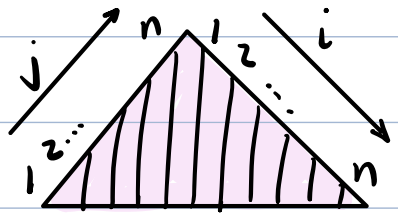
$$M(i,j) = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k \le j-1} \{ M(i,k) + M(k+1,j) + p_{i-1} \cdot p_k \cdot p_j \} & \text{if } i<j \end{cases}$$

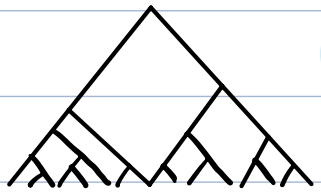
Final answer is $M(1,n)$

Memoized Implementation:

- Array M[i,j], $1 \le i \le j \le n$
  - Upper triangular
  - Init: M[i,j] = -1 (undef.)
- In our pictures we'll rotate it:





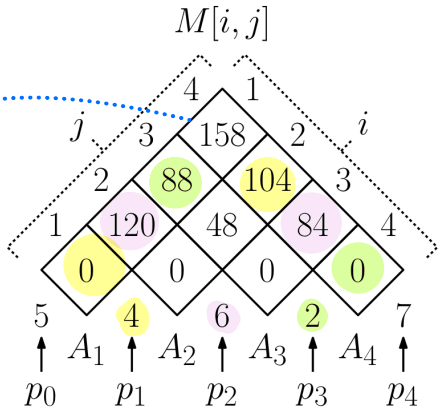


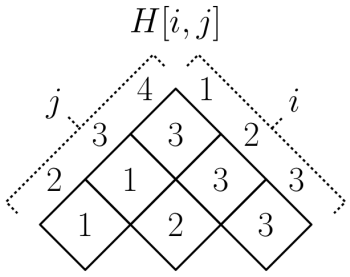
Looks more "treelike"

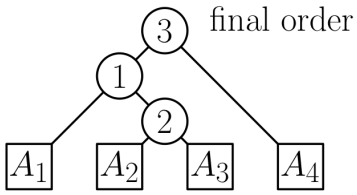
- Helpers: H[i,j] stores optimal split index k

```
memo-cmm(i, j)          // Memoized CMM
  if (M[i,j] = -1)      // undefined?
    if (i = j) M[i,j] ← 0   // basis
    else
      minCost ← +∞
      for (k ← i to j-1)    // try all splits
        cost ← (memo-cmm(i, k) + memo-cmm(k+1, j)
                + p[i-1] * p[k] * p[j])   // cost
        if (cost < minCost)   // new best cost?
          minCost ← cost      // ... save it
          H[i,j] ← k          // ... save split index
      M[i,j] ← minCost        // save final cost
  return M[i,j]
```

Example:







M[1,4] = min:

(k=1) $M[1,1] + M[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 104 + 140 = 244$

(k=2) $M[1,2] + M[3,4] + p_0 \cdot p_2 \cdot p_4 = 120 + 84 + 210 = 414$

(k=3) $M[1,3] + M[4,4] + p_0 \cdot p_3 \cdot p_4 = 88 + 0 + 70 = 158$

(k=3) ← best split

158 ← min

$M[1,4] \leftarrow 158$

$H[1,4] \leftarrow 3$

Correctness:

- Follows from correctness of DP formulation

Running Time:

- $O(n^2)$ table entries
- Each takes $O(j-i+1) \leq O(n)$ time to compute
- Total: $O(n^3)$ → $\frac{n^3}{6}$

Extracting the Final Sequence:

- So far we only compute the opt. cost
- To extract final sequence - use hooks
- Recall - $H[i,j]$ stores best split for $A_i \cdots A_j$
- $A(i,j) = A_i A_{i+1} \cdots A_j$

$$= (A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

where $k = H[i,j]$

- Continue splitting until $i = j$, $A(i,i) = A_i$

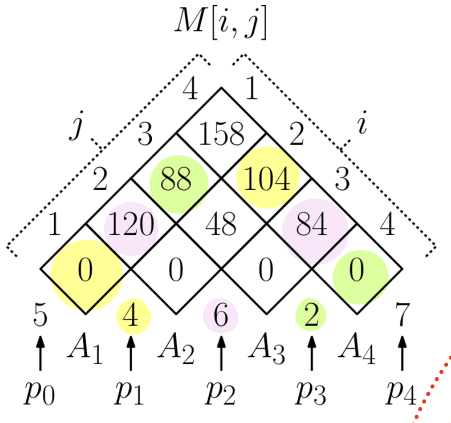
```
do-mult(i, j)        // multiply Ai ... Aj optimally
    if (i = j) return Ai   // basis - one matrix
    else
        k ← H[i,j]              // opt split index
        X ← do-mult(i, k)       // X ← Ai ... Ak
        Y ← do-mult(k+1, j)     // Y ← Ak+1 .. Aj
        return X·Y              // final product
```

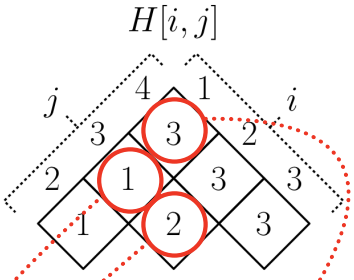
Initial call: do-mult(1, n)

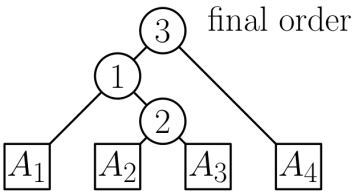
Running time:

- $O(n)$ to extract opt. evaluation tree
- $O(M[1,n])$ to perform multiplications

Example:







$A_1 .. A_4 =$

H[1,4]=3 → $(A_1 .. A_3)(A_4)$

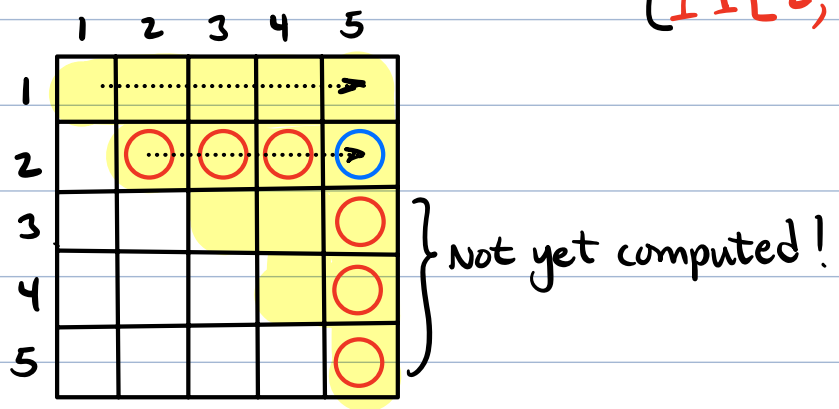
H[1,3]=1 → $((A_1)(A_2 .. A_3))\ (A_4)$

H[2,3]=2 → $((A_1)((A_2)(A_3)))\ (A_4)$

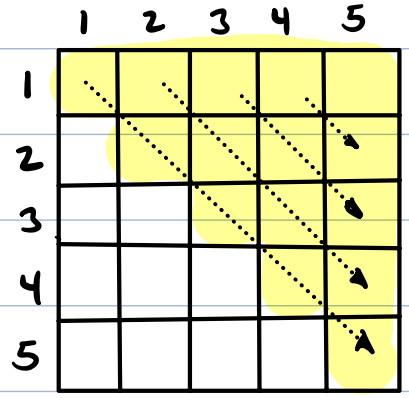
Bottom-Up Implementation:

- Normally this is an easy extension.
  Just fill table row by row.
- Doesn't work!

E.g., Computing M[2,5] accesses { M[2,2] M[3,5] / M[2,3] M[4,5] / M[2,4] M[5,5] }



Trick: Compute entries **diagonal by diagonal**, working out from **main diagonal**.



Coding this is a bit tricky. (Think about it)

Recall: $1 \le i \le j \le n$ (upper triangular)

Main = 1^{st} diagonal: $[1,1]\ [2,2] \cdots [n,n] \Rightarrow j-i+1=1$

2^{nd} diagonal: $[1,2]\ [2,3] \cdots [n-1,n] \Rightarrow j-i+1=2$

3^{rd} diagonal: $[1,3]\ [2,4] \cdots [n-2,n] \Rightarrow j-i+1=3$

$\vdots$

n^{th} diagonal: $[1,n] \Rightarrow j-i+1=n$

Let $\ell = j-i+1$ = which diagonal

ℓ runs from 1 (main diag) to n (corner)

$i = 1, 2, \ldots, n-\ell+1$

$j = i+\ell-1$

$n-\ell+1$ chosen so that $j \le n$

```
bottom-up-cmm ()        // bottom-up CMM
    for (i ← 1 to n) M[i,i] = 0   // basis - main diag.
    for (ℓ ← 2 to n)              // ℓ = diagonal number
        for (i ← 1 to n-ℓ+1)
            j ← i+ℓ-1             // pick j so j-i+1 = ℓ
            minCost ← ∞
            for (k ← i to j-1)
                minCost ← min (minCost,
                    M[i,k] + M[k+1,j] + p_{i-1} p_k p_j )
            M[i,j] ← minCost
    return M[1,n]
```

same as before

Smaller j-i differences so entry is defined

Running Time:

$O(n^3)$ - 3 nested loops

- Each in range 1..n

Summary:

- $O(n^3)$ DP algorithm for chain matrix mult
- Explary of DP's that build hierarchies
- Example: Optimum binary search tree

CMSC 451 - Algorithm Design
Lecture 11 - All-pairs Shortest Paths + Floyd-Warshall

Earlier, we covered the single-source shortest path problem for edge-weighted digraphs.

Dijkstra - $O(m + n \log n)$ $n = |V|$ $m = |E|$

Bellman-Ford - $O(n \cdot m)$ [allows neg. edge weights]

Notation:

- The cost of a path is sum of edge weights
- The distance between two vertices u & v is min. cost of any path from u to v.

All-pairs shortest paths:

Given digraph $G = (V, E)$. Each edge $(u,v) \in E$ has weight $w(u,v)$, may be negative (but no neg. cost cycles), compute distance between all pairs of vertices $u, v \in V$.

We'll present Floyd-Warshall algorithm.
DP-based, $O(n^3)$ time.

Notes:

- Discovered independently by Robert Floyd + Stephen Warshall (mid 1960's).
- First discovered (but not credited) by Bernard Roy.
- General DP structure works for many reachability problems:
  - Transitive closure of a binay relation
  - Convert finite state automaton to regular expression.
  - Compute max capacity paths

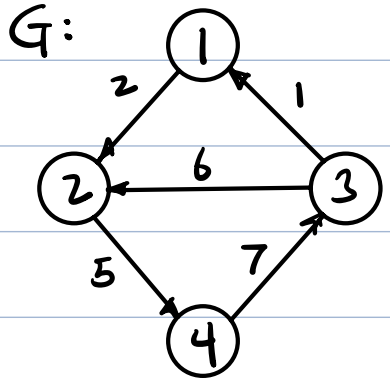
Input/Output Representation:

Input: (Augmented) adjacency matrix:

$V = \{1, 2, \ldots, n\}$

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j, (i,j) \in E \\ +\infty & \text{if } i \neq j, (i,j) \notin E \end{cases}$$

G:



w:

	1	2	3	4
1	0	2	∞	∞
2	∞	0	∞	5
3	1	6	0	∞
4	∞	∞	7	0

Output: Distance matrix

d_{ij} = distance from i to j
(∞ if no path)

d:

	1	2	3	4
1	0	2	14	7
2	13	0	12	5
3	1	3	0	8
4	8	10	7	0

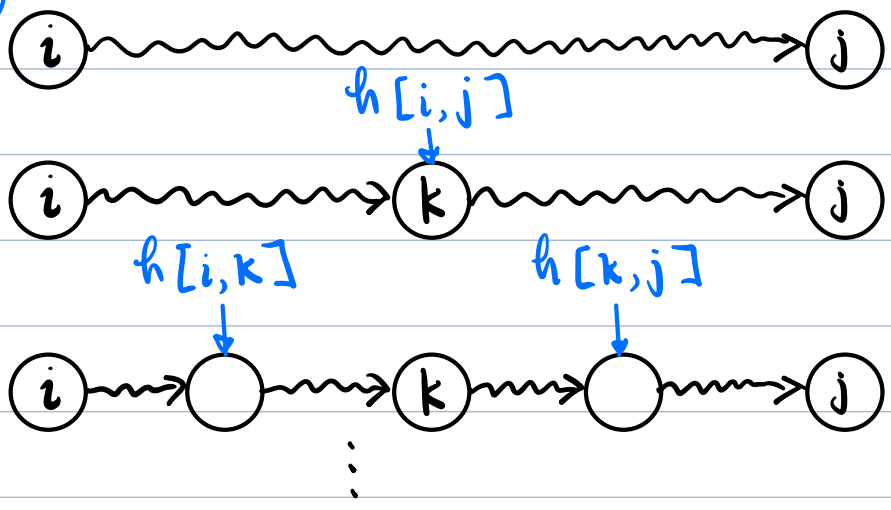
→ Space = $O(n^2)$

Paths? $O(n^2)$ paths ⇒ $O(n^3)$ storage?

Clever trick - Reduces storage to $O(n^2)$
- "Hook" matrix $H = h[1..n, 1..n]$
- For each $i, j \in V$

$$h[i,j] = \begin{cases} \emptyset & \text{if shortest path is direct (edge } (i,j)) \\ k & k = \text{any vertex along the shortest path} \end{cases}$$

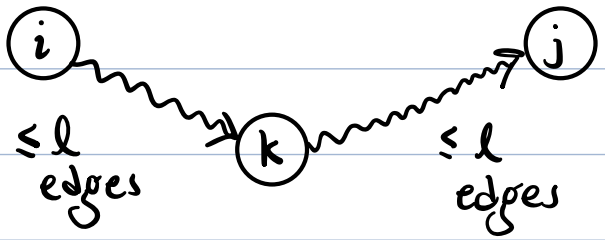
Fill in the path recursively



Floyd-Warshall Algorithm:

- How to reduce shortest paths to smaller subproblems?
- Obvious? (But not best!)
  - Restrict path length
    e.g. $d_{ij}^{(\ell)}$ = shortest path from i to j using $\leq \ell$ edges
  - Build by doubling:

$$d_{ij}^{(2\ell)} = \min_k \left(d_{ik}^{(\ell)} + d_{kj}^{(\ell)} \right)$$



  - This leads to a slower algorithm.

- Floyd-Warshall insight:
  - Don't restrict length, restrict which vertices you can go through.

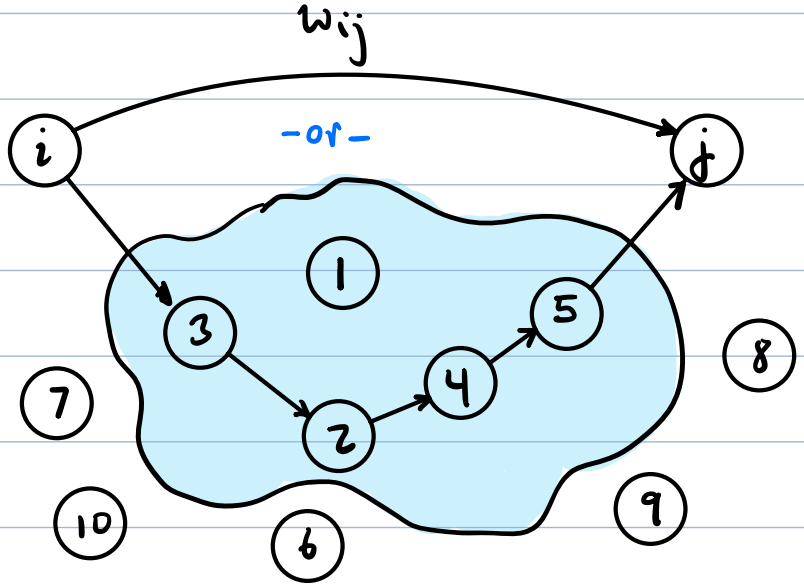
  - Given a path $\langle v_1, v_2, \ldots, v_{\ell-1}, v_\ell \rangle$
    $v_2 \ldots v_{\ell-1}$ are the intermediate vertices.

- for $1 \le i,j \le n$ & $0 \le k \le n$, define:

$d_{ij}^{(k)}$ = cost of shortest path from i to j with intermediate vertices from $\{1,\dots,k\}$.

Eg. $k=5$

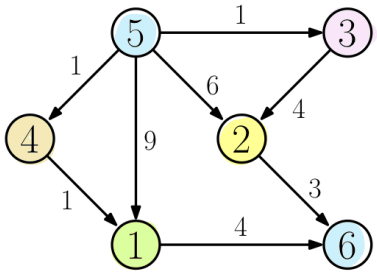
$d_{ij}^{(5)}$:



Approach - $d_{ij}^{(0)} = w_{ij}$ (no intermediates $\Rightarrow$ edge)

for $k = 1$ to n

compute $d_{ij}^{(k)}$ for all $1 \le i,j \le n$.



$d_{5,6}^{(0)} = \infty$ (no path)

$d_{5,6}^{(1)} = 13 \ \langle 5,1,6 \rangle$

$d_{5,6}^{(2)} = 9 \ \ \langle 5,2,6 \rangle$

$d_{5,6}^{(3)} = 8 \ \ \langle 5,3,2,6 \rangle$

$d_{5,6}^{(4)} = 6 \ \ \langle 5,4,1,6 \rangle$

$d_{5,6}^{(5)} = d_{5,6}^{(6)} = 6$ (no change)

DP Formulation:

Basis: $d_{ij}^{(0)} = w_{ij}$ (no intermediates ⇒ edge weight)

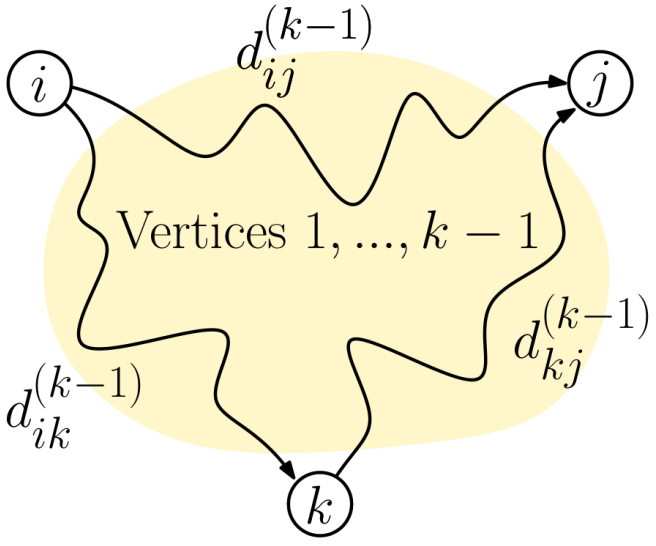
$k \geq 1$: Cases:

- Shortest path does not pass through k:
  - ⇒ only intermediates are $\{1, \ldots, k-1\}$
  - ⇒ $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$

- Shortest path passes through k:
  - Since no neg. cost cycles, visits k once
  - Path goes $i \rightsquigarrow k$ then $k \rightsquigarrow j$
  - These paths use intermediates $\{1, \ldots, k-1\}$
  - ⇒ $d_{ij}^{(k)} \leftarrow d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

Principle of optimality holds

k may be equal to i or j. Check that this is still correct



DP Credo - Try all options + take best

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min \left\{ \begin{matrix} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{matrix} \right\} & \text{if } k \geq 1 \end{cases}$$

Bottom-Up Implementation

- Build matrix $d[1..n, 1..n]$
- Matrix of "hooks" to save decisions

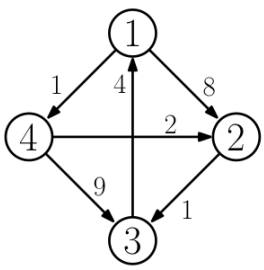
```
floyd-warshall (w[1..n, 1..n])
    for (1 ≤ i,j ≤ n)                    // init d^(0)
        d[i,j] ← w[i,j]
        h[i,j] ← ∅
    for (k = 1 to n)                     // compute d^(k)
        for (1 ≤ i,j ≤ n)
            newCost ← d[i,k] + d[k,j]
            if (newCost < d[i,j])        // better to go
                d[i,j] ← newCost         // through k
                h[i,j] ← k               // update hook
    return d + h matrices                // return dists + hooks
```

Running time: $O(n^3)$

Huh?

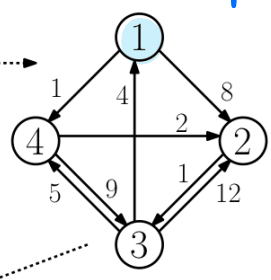
- DP formulation had three parameters: i, j, k but algorithm does not use k.
- We only use k + k-1, so we really only need two copies of d_{ij} matrix
  - current + previous
- In fact, we can show that only one matrix suffices (see pdf notes for details)

Example:



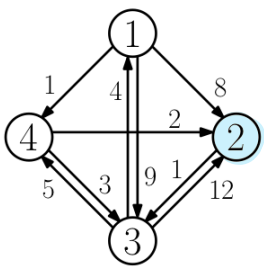
$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

(w_{ij})

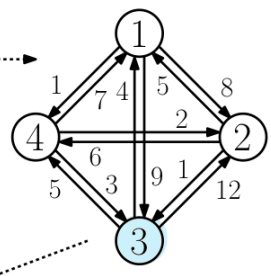


$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

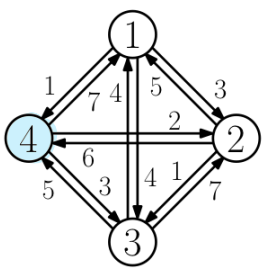
(updates, hooks)



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

$$d = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \qquad H = \begin{bmatrix} \emptyset & 4 & 4 & \emptyset \\ 3 & \emptyset & \emptyset & 3 \\ \emptyset & 4 & \emptyset & 1 \\ 3 & \emptyset & 2 & \emptyset \end{bmatrix}$$

Extracting the Shortest Path:

- As mentioned above, we use $h[i,j]$ to recursively extract path
- If $d[i,j] = \infty \Rightarrow$ no path
  otherwise get-path(i,j) yields edges on shortest path.

```
get-path(i, j)      // path from i to j
    if (h[i,j] = ∅)     // direct path
        return (i,j)    // path is single edge
    else
        mid ← h[i,j]    // i ⇝ mid ⇝ j
        return get-path(i, mid) ⊕ get-path(mid, j)
                          ↑ concatenate
```

Summary:

- Floyd-Warshall all-pairs shortest paths
- $O(n^3)$ time / $O(n^2)$ space
- General structure can be applied to many similar path-related problems.

CMSC 451 - Algorithm Design
Lecture 12 - Network Flow - Basic Concepts

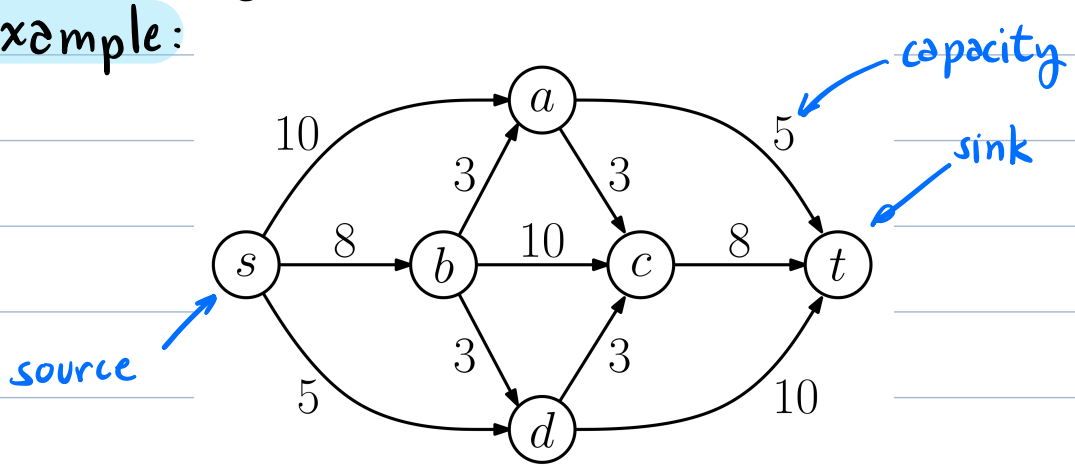
Network Flow is a classical problem, which emerged from the field of **operations research**, a branch of **applied math**.

A **flow network** (or **s-t network**) is a directed graph $G=(V,E)$, where each edge $(u,v) \in E$ has an associated **capacity** $c(u,v) \geq 0$ + there are two **special vertices**:

source (s) + **sink** (t)

(No edges enter s, no edges leave t)

Example:



Max-Flow: Thinking of edge (u,v) as a **pipe** that can carry **$c(u,v)$ units of flow**, **how much flow** can we **push** from **s to t**?

What do mean by flow?

- A flow is a function f mapping each edge to a real number $f(u,v) \geq 0$
- Satisfies:
  - Capacity constraint:

$$\forall (u,v) \in E,\ f(u,v) \leq c(u,v)$$

(flow cannot exceed capacity)

  - Flow conservation (or balance):

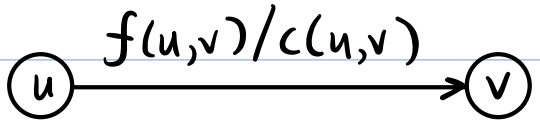
$$\forall v \in V \setminus \{s,t\}$$

$$\sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$$

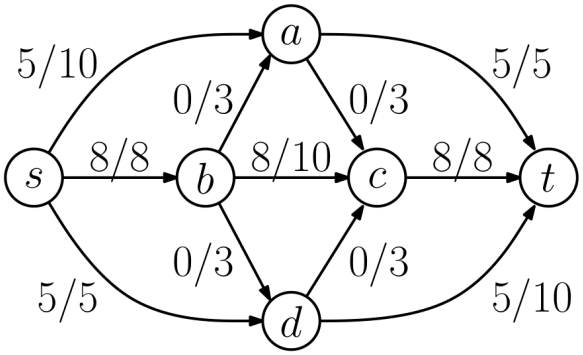
$f^{in}(v)$ $f^{out}(v)$

(flow in = flow out, except at source + sink)

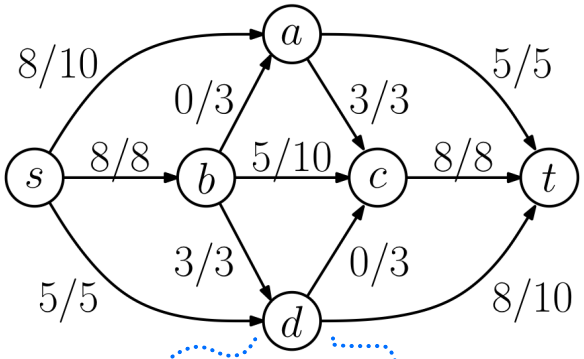
2 Examples:



Flow 1: f_1



Flow 2: f_2



$f^{in}(d) = 5+3$ $f^{out}(d) = 0+8$

Want to maximize **total flow value**, defined

$|f| = f^{out}(s) = f^{in}(t)$

Flow conservation implies flow out of s = flow into t

$|f_1| = 5+8+5 = 18$

$|f_2| = 8+8+5 = 21$ (This the max flow)

Max-Flow Problem: Given a **flow network**, compute the **flow** of **max total value**

This is a **heavily studied** problem with a **long history**: $n = |V|$, $m = |E|$, C = sum of capacities

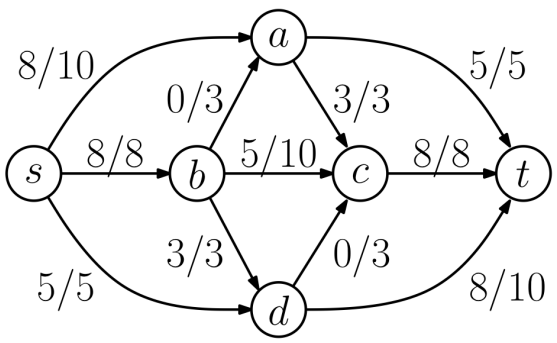
- **Ford-Fulkerson** (1956) - $O((n+m)C)$
- **Dinitz** (1970) - $O(n^2 \cdot m)$
- **Edmonds-Karp** (1972) - $O(n \cdot m^2)$
- **Gabow** (1985) - $O(nm \log C)$
- **Goldberg-Tarjan** (1986) - $O(nm \log n^2/m)$

Path-Based View: An equivalent way to view a flow is as a collection of paths from s to t (like wires)

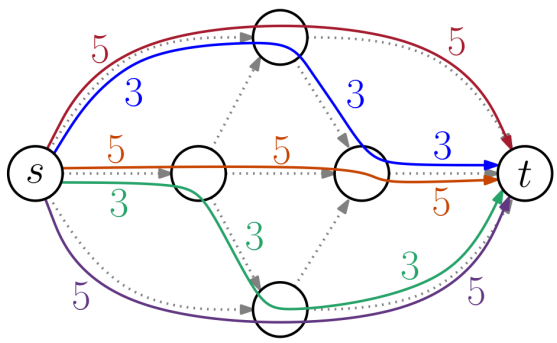
- Each s-t path is assigned a weight
- Sum of weights cannot exceed edge capacity

Example:

Edge-based:



Path-based



$|f| = 8+8+5 = 21$

$|f| = 5+3+5+3+5 = 21$

Just a different perspectives on the same math. concept

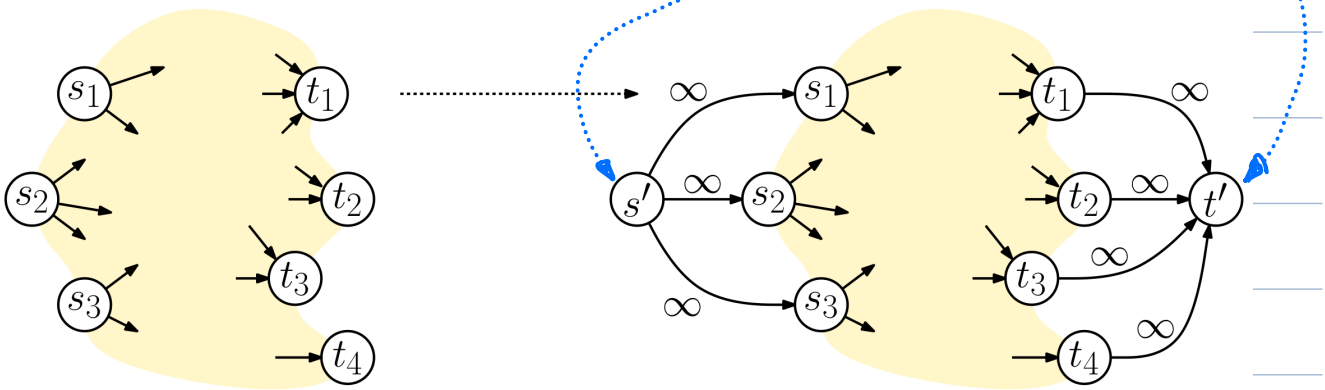
- some algorithms are more edge based
- some are more path based

Proof-Exercise

Claim: An s-t network has an edge-based flow of value x iff it has a path-based flow of value x.

Why just one source/sink?

- Can easily simulate multiple sources/sinks.
- Add "super source" and/or "super sink"
  + connect to others.



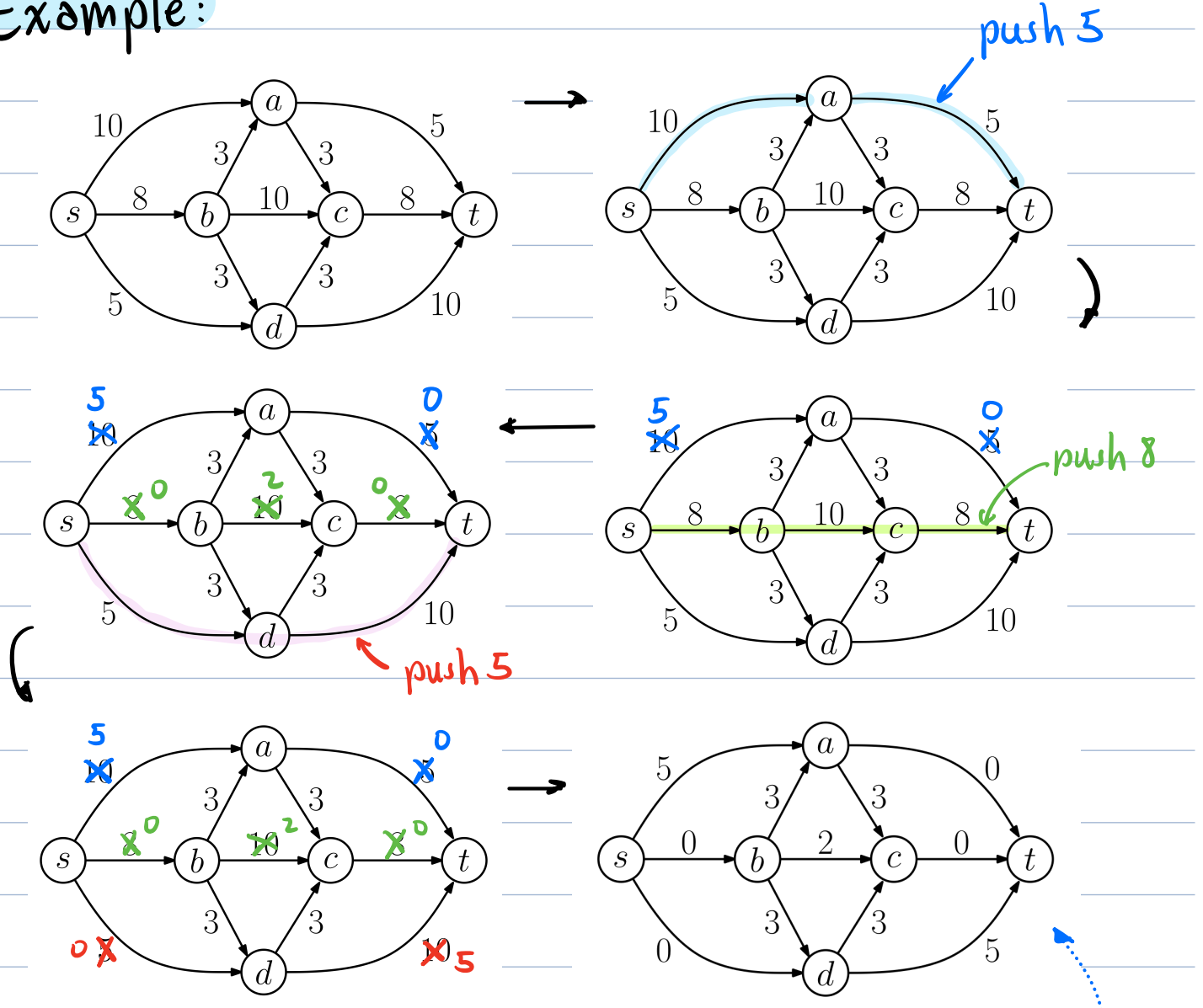
- Can you also specify linkages?
  (e.g. all flow from s_i must go to t_i)
  - No - called mult-commodity flow
    - NP-hard!

Max-Flow - Why greedy fails

Simple greedy strategy:

- find any path from s to t
  of strictly positive capacities
- push as much flow as you can
  along this path
- reduce (remaining) capacities
  on edges of path

Example:



Total flow = 5 + 8 + 5 = 18

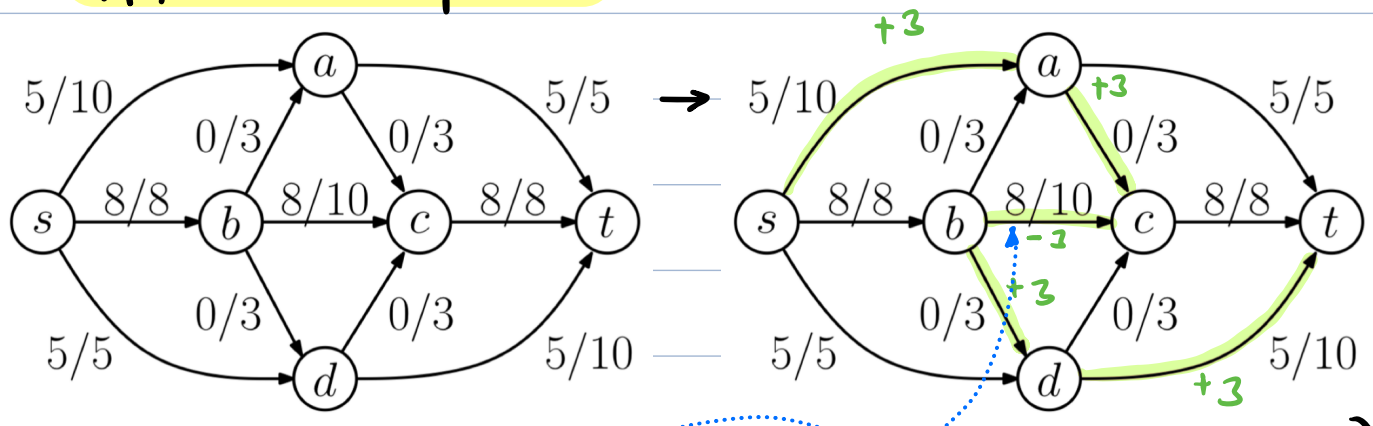
Not optimal!

But no more pos.-capacity s-t paths!

How to fix greedy?

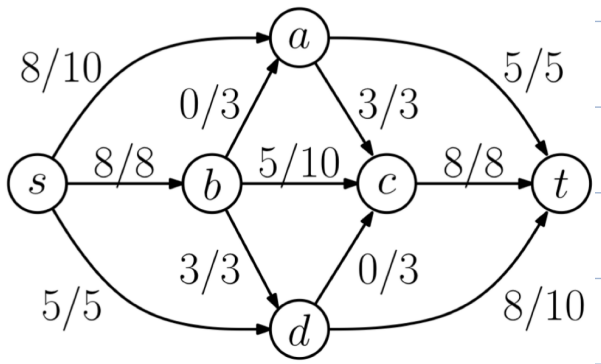
- We need to consider s-t paths that both:
  - add new flow where capacity exists
  - reduce flow where flow exists

$|f| = 18$ (not optimal)



Edge (b,c) carried flow of 8. We can reduce this by -3

Conserves flow balance



$|f| = 21$ optimal!

How to formalize augmenting/reducing paths?

Residual Network - Given s-t network G + flow f, define G_f to be s-t network

- same vertices as G
- Forward edges (can add more flow)

  for $(u,v) \in E$ s.t. $f(u,v) < c(u,v)$

  add edge (u,v) with capacity:

  $c_f(u,v) = c(u,v) - f(u,v)$

Intuition: can add $c_f(u,v)$ more flow

- **Backward edges** (can reduce existing flow)

for $(u,v) \in E$ s.t. $f(u,v) > 0$

add reverse edge (v,u) with capacity:

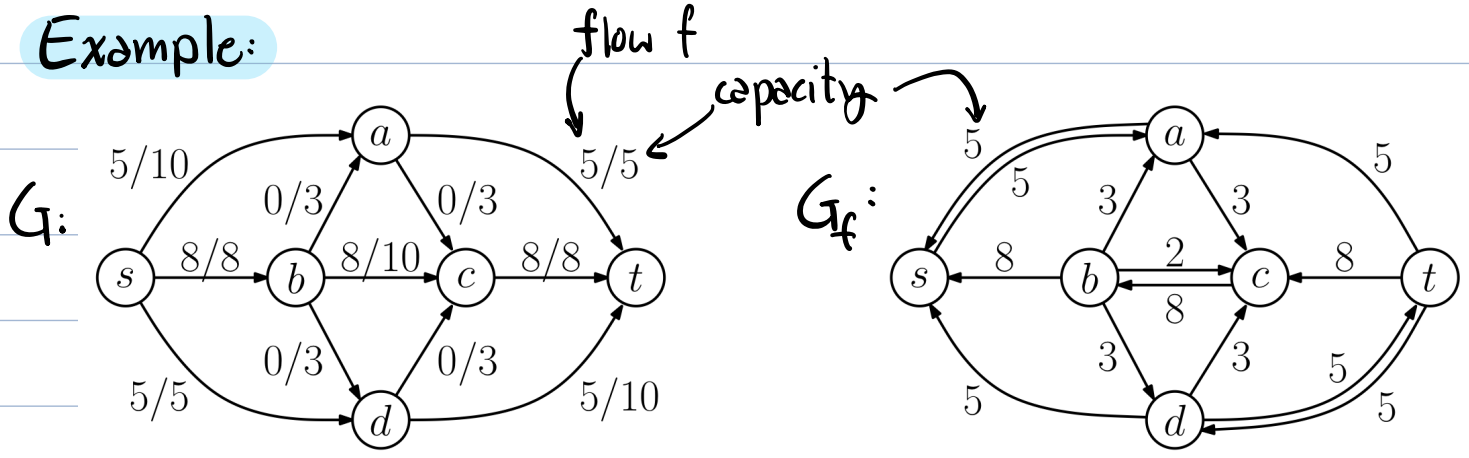
$c_f(v,u) = f(u,v)$

Intuition: can turn off $c_f(u,v)$ flow

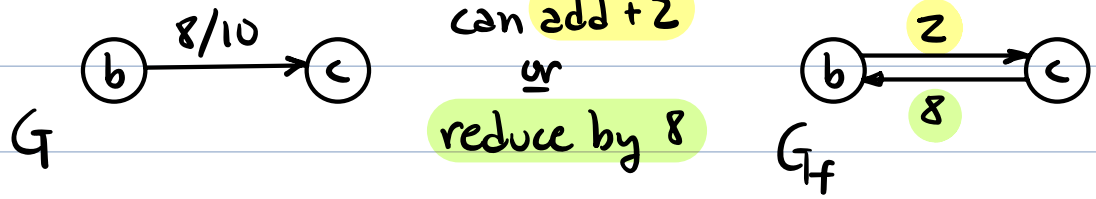
Example:

flow f

capacity

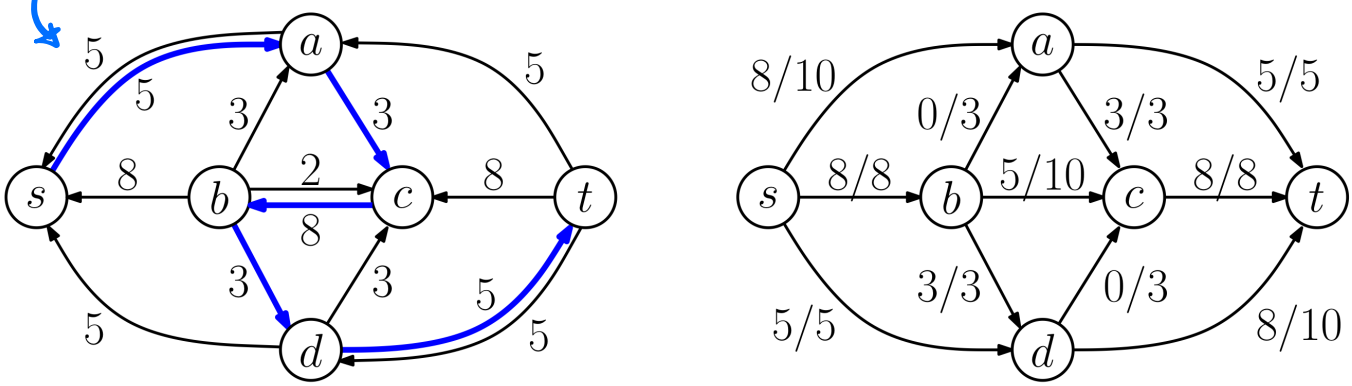


for example:



In G_f, we can push +3 flow on this path.

$|f| = 21$ (optimal)



Questions:

(1) If G_f has a flow, can we use it to increase G's flow?

(2) If we repeat this process, will it lead to the optimal flow (or get stuck, like greedy)?

Both answers are "yes"!

(1) Straightforward.

(2) Not so easy (Min-Cut/Max-Flow Thm) — Prove this later

Claim: Given network G + flow f, if f' is a flow in G_f, then $f + f'$ is a flow in G.

Proof sketch:

Capacity constraint: if (u,v) is forward edge

f valid for $G \Rightarrow 0 \leq f(u,v) \leq c(u,v)$

f' valid for $G_f \Rightarrow 0 \leq f'(u,v) \leq c_f(u,v)$

def. of $c_f \Rightarrow$ $c_f(u,v) = c(u,v) - f(u,v)$

$$\Rightarrow 0 \leq f(u,v) + f'(u,v) \leq f(u,v) + (c(u,v) - f(u,v))$$
$$= c(u,v) \checkmark$$

(Leave flow conservation + backward edges as exercise.)

Ford-Fulkerson Algorithm:

- Init flow: $f = 0$
- Find an s-t path in residual graph, G_f (called an augmenting path)
- $f' \leftarrow$ max. flow on this path (min capacity edge)
- Update residual graph based on new flow: $f \leftarrow f + f'$

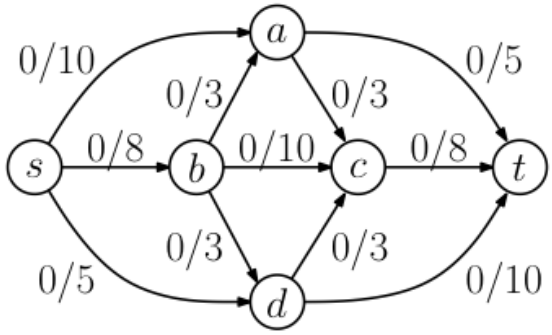
```
ford-fulkerson(G, s, t)
    f ← 0                          // set f(u,v) ← 0, ∀(u,v) ∈ E
    while (true)
        G_f ← compute resid. graph for f   // O(n+m)
        if (G_f has no s-t path)   // DFS → O(n+m)
            return f               // f is max flow
        π ← any path from s to t in G_f
        c ← min capacity of any edge on π
        increase f by adding +c to edges of π
```

Running time:

- Each iteration of the while loop can be done in $O(n+m)$ time (DFS)
- We'll discuss number of iterations in next lecture.

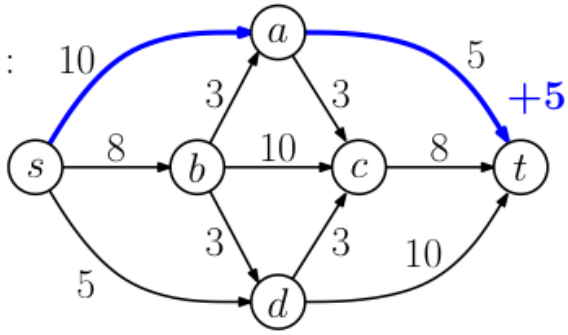
Example:

f_0 :

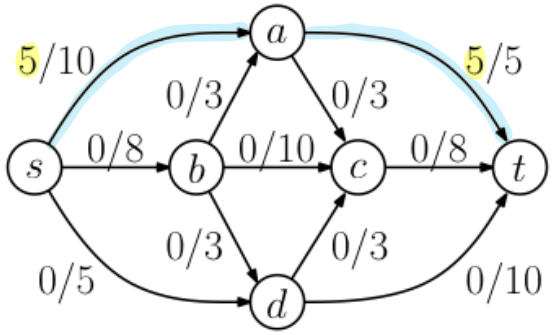


$|f_0| = 0$

G_{f_0} :

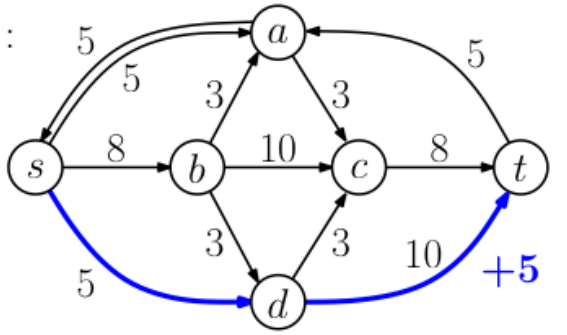


f_1 :

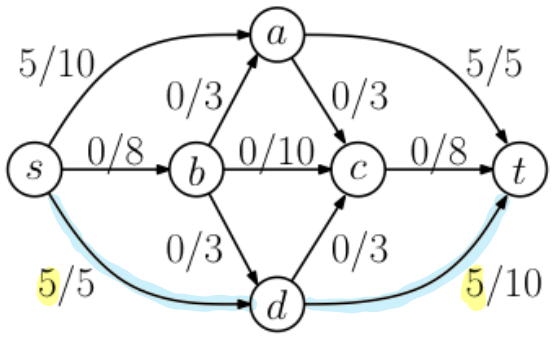


$|f_1| = 5$

G_{f_1} :

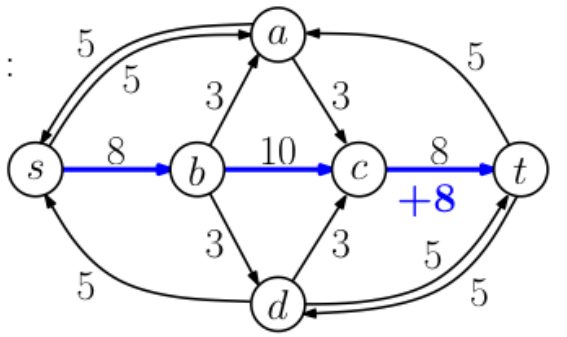


f_2 :

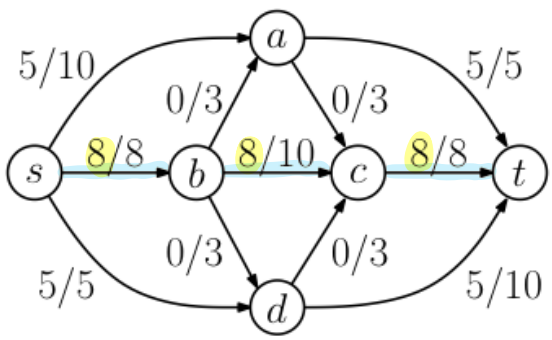


$|f_2| = 10$

G_{f_2} :

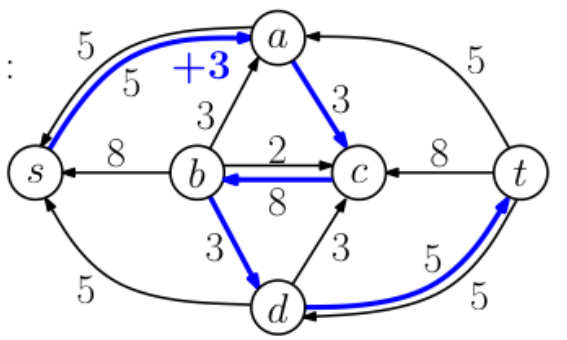


f_3 :



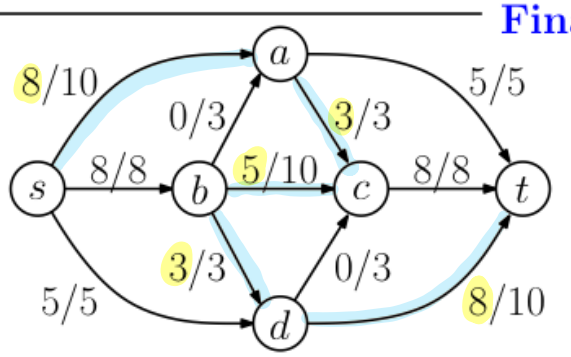
$|f_3| = 18$

G_{f_3} :



Final flow

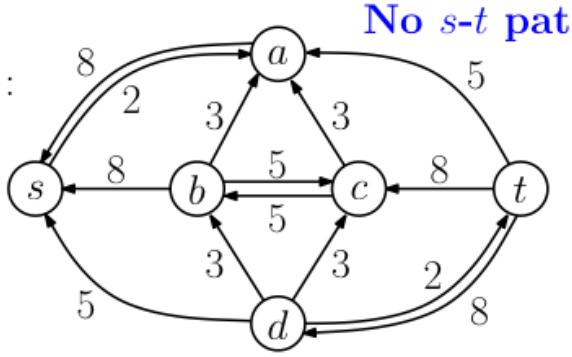
f_4 :



$|f_4| = 21$

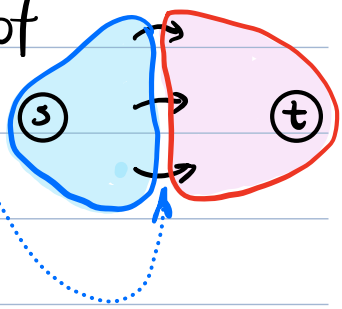
No s-t path

G_{f_4} :



Correctness:

- Easy to see that F-F produces a valid flow.
- On termination - Is it optimal?
- To prove this we need to introduce a related concept - cut
- Intuitively - Flow cannot be increased because it saturates all edges of a bottleneck.
- Removing these bottleneck edges cuts the network in two



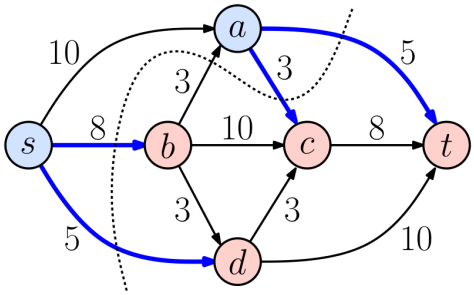
Definition: Given an s-t network, a cut is a partition of the vertex set X, Y such that $s \in X$ & $t \in Y$.

The capacity of cut (X, Y) is the sum of capacities from X to Y,

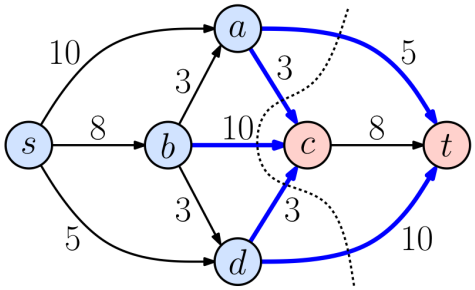
$$c(X,Y) = \sum_{x \in X} \sum_{y \in Y} c(x,y)$$

if $(x,y) \notin E$, $c(x,y) = 0$

$X = \{s, a\}$ $Y = \{b, c, d, t\}$
$c(X,Y) = 5 + 3 + 8 + 5 = 21$



$X = \{s, a, b, d\}$ $Y = \{c, t\}$
$c(X,Y) = 5 + 3 + 10 + 3 + 10 = 31$

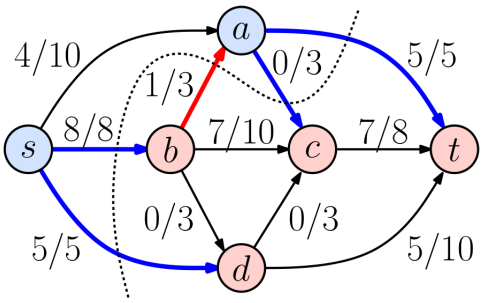


Definition: Given G, a flow f, + a cut (X, Y) define the net flow across the cut to be sum of $X \to Y$ flows minus the $Y \to X$ flows.

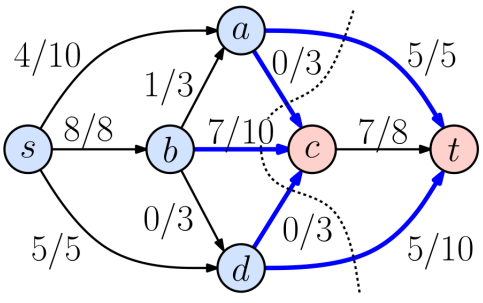
$$f(X,Y) = \sum_{(x,y) \in X \times Y} f(x,y) - \sum_{(y,x) \in Y \times X} f(y,x)$$

Example: Two cuts on the same flow, $|f| = 17$

$X = \{s, a\} \quad Y = \{b, c, d, t\}$
$f(X,Y) = (5 + 0 + 8 + 5) - 1 = 17$



$X = \{s, a, b, d\} \quad Y = \{c, t\}$
$f(X,Y) = 5 + 0 + 7 + 0 + 5 = 17$

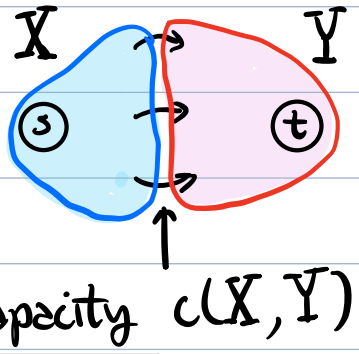


Intuitively - The flow across any cut = $|f|$ (by flow conservation)

Lemma: Given any network G, any flow f, and any cut (X, Y), $f(X,Y) = |f|$.

(Proof given in pdf lecture notes)

Given any cut (X, Y), all the flow must cross over the edges of the cut. Thus:



Lemma: Given any network G, any flow f, and any cut (X, Y):

$$|f| \leq c(X, Y)$$

This holds for every cut, so it holds for the minimum-capacity cut.

To prove that the F-F algorithm is optimal, it suffices to prove that the F-F flow equals the min-capacity cut.

This is a consequence of the following famous theorem

Max-Flow/Min-Cut Theorem:

The following are equivalent:

(i) f is a max flow for G

(ii) Residual network G_f has no s-t path

(iii) $|f| = c(X, Y)$ for some cut (X, Y) of G

Proof:

(i) ⇒ (ii) [by contradiction]

If G_f had an s-t path, we could push flow on this path, increasing flow value. ✓

(ii) ⇒ (iii)

- Let X = vertices reachable from s in G_f
- Let Y = $V \setminus X$ (all remaining vertices)
- Since no s-t path, $t \in Y$.

⇒ (X, Y) is a cut

⇒ for each $(x, y) \in X \times Y$

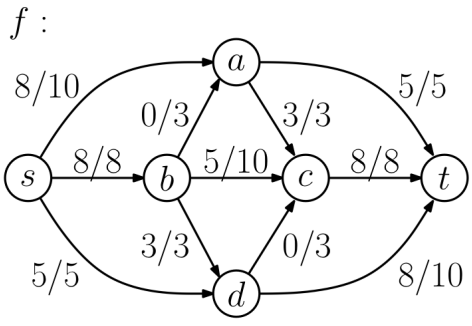
$$f(x, y) = c(x, y)$$

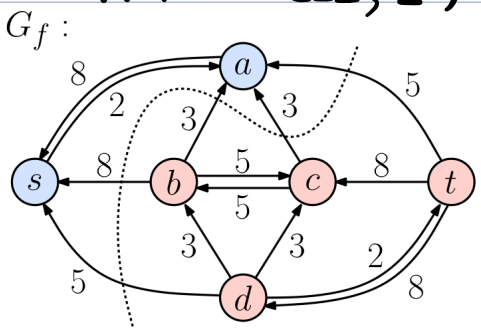
⇒ $$f(X, Y) = c(X, Y)$$

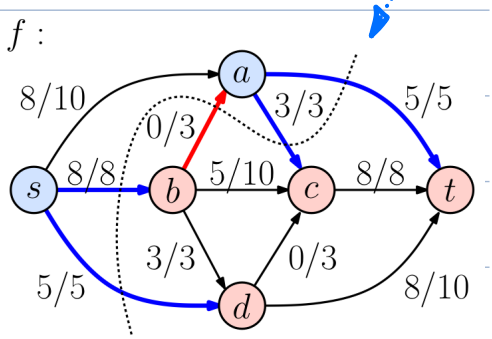
Edges crossing cut are saturated

- By previous lemma, $f(X, Y) = |f|$

⇒ $$|f| = c(X, Y)$$ ✓







(iii) ⇒ (i)

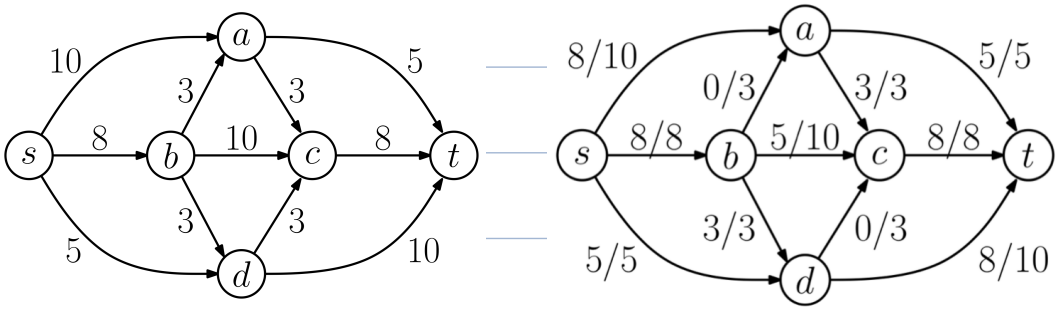
- By previous lemma $|f| \leq c(X, Y)$ for all flows f & all cuts (X, Y)
- If equality is attained for any flow and any cut, this flow must be maximum. ✓

Summary:

- Flow networks + flows
- Max-flow problem
  - Greedy fails
- Residual network + augmenting paths
- Ford-Fulkerson Algorithm
  (find augmenting path, update residual)
- Cuts + capacities
- Max-Flow/Min-Cut Theorem
  (Max flow ≡ Min Cut)
- Whew!

CMSC 451 - Algorithm Design
Lecture 13 - Network Flow - Algorithms

Previous lecture - s-t Networks - source, sink, capacities
- Flows - Capacity + Conservation
- Max-Flow Problem



- Residual Network + Augmenting paths
- Ford-Fulkerson Algorithm
- Cuts
- Max-Flow/Min-Cut Theorem

This Lecture:
- Analysis of Ford-Fulkerson
- Efficient algorithms
  - Scaling
  - Edmonds-Karp
- Maximum bipartite matching

How Fast is Ford-Fulkerson (F-F)?

Observe:

If capacities are integers, F-F augmentations are integer valued. $\Rightarrow$

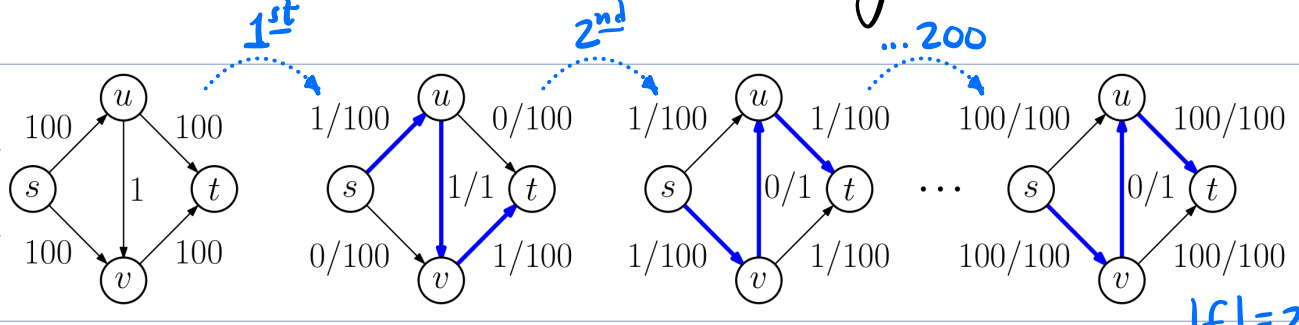
> **Lemma:** Given an s-t network with integer-valued capacities, the max flow value will be an integer

F-F running time:

- Each iteration takes $O(n+m)$ time
- Assuming G is connected $O(n+m) = O(m)$
- Running time is $O(m \cdot (\text{num. of iterations}))$
- How many iterations?

F-F does not specify which augmenting path

- What if we are unlucky?



Number of iterations can be as large as $|f|$

- This is bad! (replace 100 with 1,000,000,000)

Generally - Let C be any upper bound on $|f|$

- F-F running time is $O(m \cdot C)$

Let's explore more efficient algorithms -

Scaling Algorithm (Gabow, 1980's)

- Augment high capacity paths first
- Can compute max capacity s-t path in $O(m \log n)$ time
- Faster to compute a close-to-max capacity path in $O(m)$ time

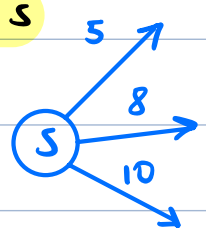
Close-to-max capacity?

- Assume capacities are all integers
- $C \leftarrow$ any upper bound on max flow value
- Eg.

$C \leftarrow$ sum of capacities out of s

$\leftarrow \sum_{(s,v) \in E} c(s,v)$

Eg.



C 5+8+10 = 23

- $\Delta \leftarrow$ largest power of 2 $\leq C$

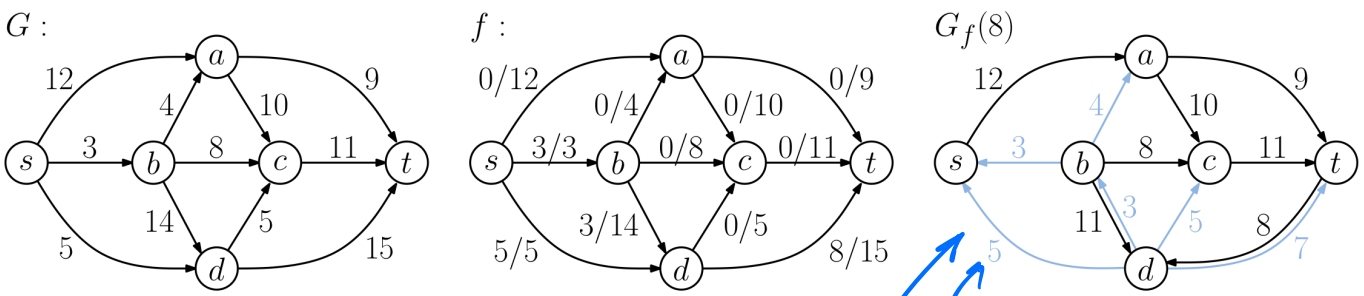
$\leftarrow 2^{\lfloor \log_2 C \rfloor}$

$\Delta \leftarrow 16$

- Initial flow: $f \leftarrow 0$ (0 flow on all edges)

- Given any flow f & Δ, define

$G_f(\Delta)$ = residual network G_f keeping only edges of capacity $\geq \Delta$



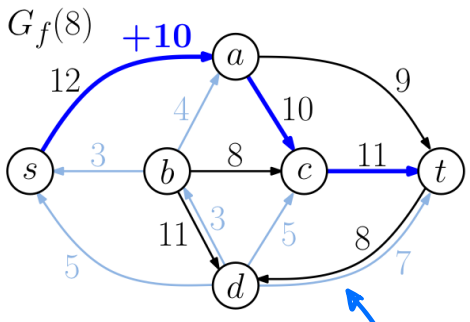
These edges are omitted since $< \Delta$

By computing augmenting paths in $G_f(\Delta)$, flow increases rapidly

- Run F-F on $G_f(\Delta)$
- When no aug. path exists $\Delta \leftarrow \Delta/2$
- Stop when $\Delta < 1$.

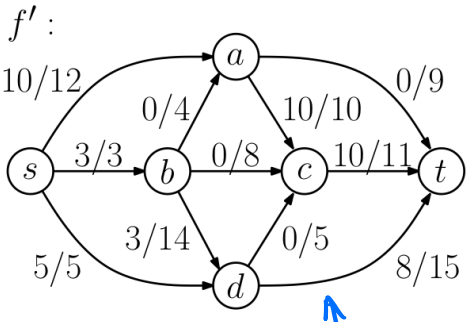
```
scaling-flow(G)   // scaling alg for network flow
    f ← 0                                   // init flow is zero
    C ← Σ_{(s,v)∈E} c(s,v)                  // capacity out of s
    Δ ← 2^⌊log2 C⌋                          // initial Δ
    while (Δ ≥ 1)                           // stop if Δ<1
        G_f(Δ) ← residual G_f, with         // heavy residual
                 capacities < Δ removed
        if (G_f(Δ) has an s-t path π)       // augment
            c ← min capacity on π
            f ← add c to edges of π
        else                                // no more augments
            Δ ← Δ/2                         // reduce Δ
    return f
```

Example: (see previous figure)

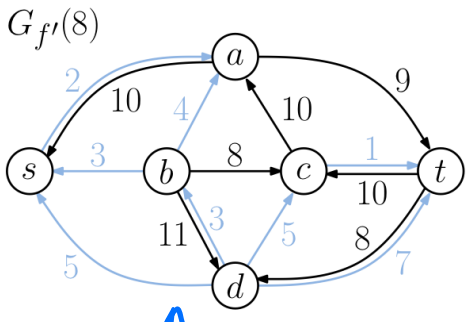


$\Delta = 8$

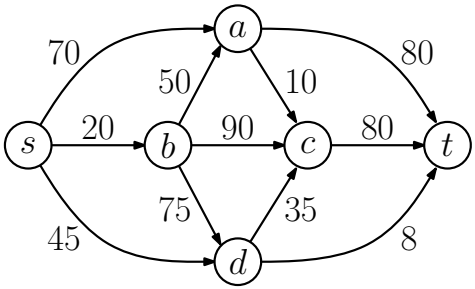
Find any path in $G_f(\Delta)$



Augment

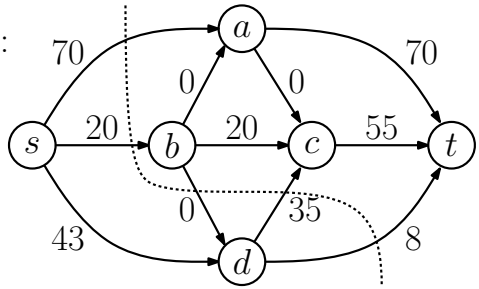


- Update $G_f(\Delta)$
- No s-t path so $\Delta \leftarrow \Delta/2 = 4$

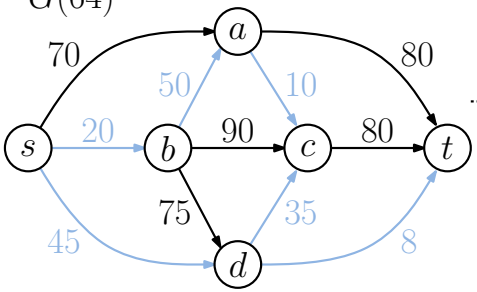


Scaling (Full example)

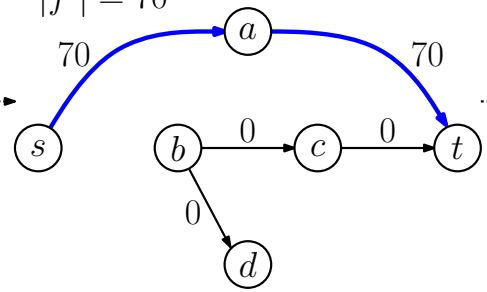
Flow after 4 stages:



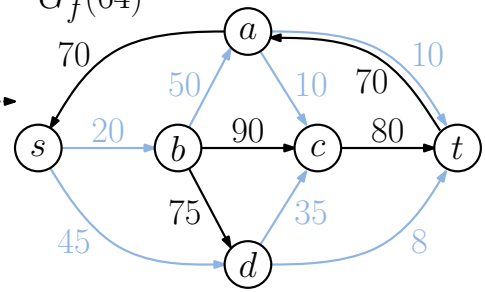
$G(64)$



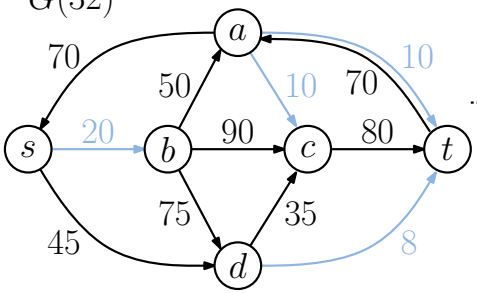
$|f'| = 70$



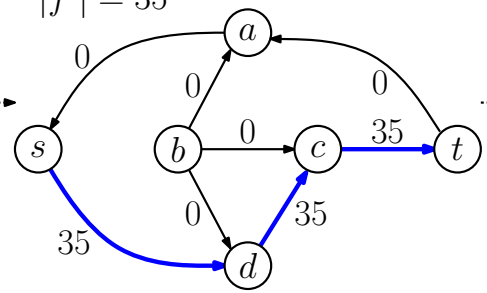
$G_f(64)$



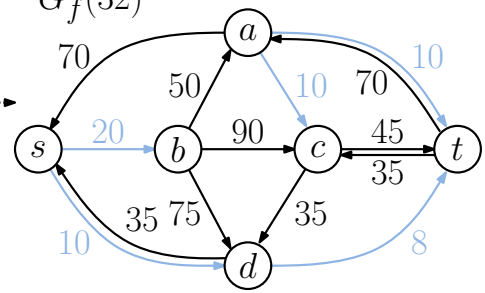
$G(32)$



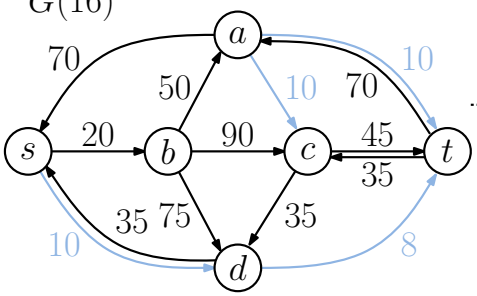
$|f'| = 35$



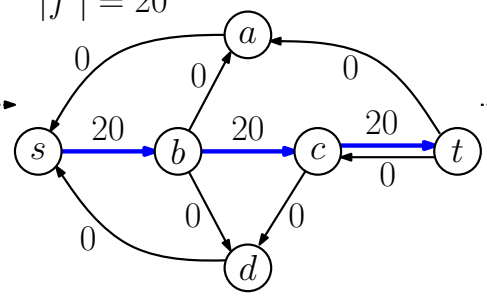
$G_f(32)$



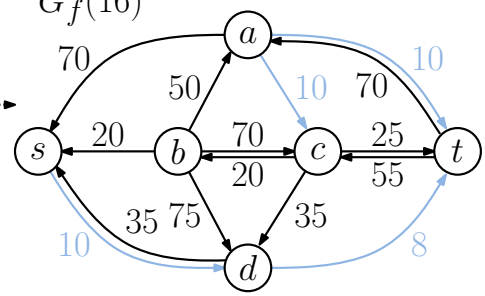
$G(16)$



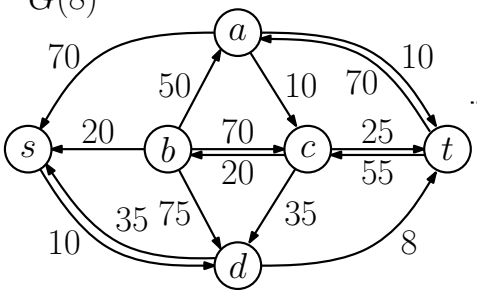
$|f'| = 20$



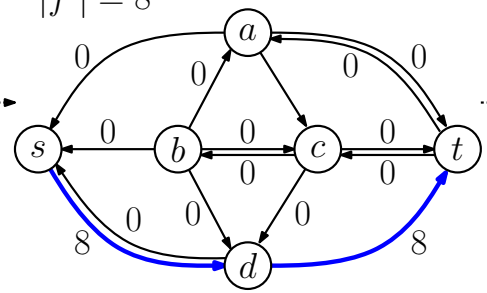
$G_f(16)$



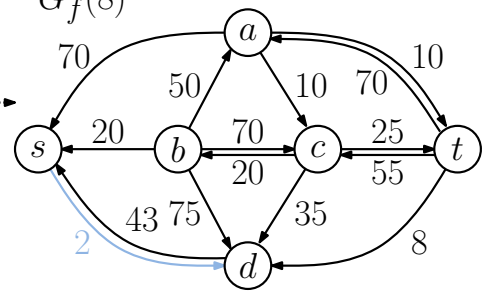
$G(8)$



$|f'| = 8$



$G_f(8)$



Correctness:

Same as F-F, just picking paths smarter.

Running time:

- Each time we augment in $G_f(\Delta)$ the flow increases by at least Δ
  - $\Rightarrow$ residual capacities on these edges decrease by at least Δ
  - $\Rightarrow$ After $O(m)$ augmentations, all residual capacities fall below Δ
  - $\Rightarrow$ No augmentation $\Rightarrow$ $\Delta \leftarrow \Delta/2$
  - Summary: $O(m)$ augmentations $\Rightarrow$ Δ halved

- After $O(\log C)$ halvings, $\Delta < 1$
- From F-F, each augmentation takes $O(m)$ time.

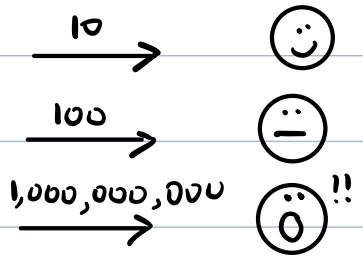
- Total time: $O(m \cdot m \cdot \log C)$

  - m: Time per augmentation
  - m: Num. augmentations until Δ is halved
  - $\log C$: Number of halvings

$= O(m^2 \log C)$

Is this really efficient?

- C = sum of capacities is an input parameter
- It could be arbitrarily large, independent of $n + m$



- "Efficient" ≡ Polynomial time
  - Running time polynomial in input size - $O(n+m)$, $O(n \cdot m^3)$, $O(n \log n)$
  - As opposed to exponential time
    - $O(2^n)$, $O(3^{n+m})$, $O(n!)$
- But $\log C$ is related to input size
  = Num. of bits needed to rep. capacities
- So $O(m \cdot \log C)$ is a polynomial in input size
  - if we are counting bits of input

To avoid confusion, we distinguish between:

Strongly Polynomial Time -
- Polynomial in number of words of input (ignoring no. of bits)
- E.g. $O(n \cdot m^2)$, $O(n \log n)$, $O(n^{5.12} + m)$

Weakly Polynomial Time -
Polynomial in number of bits of input
- E.g. $O(m^2 \log C)$ [but not $O(m \cdot C)$]

Is there a strongly polynomial alg. for max flow?

Edmonds-Karp Algorithm

- Discovered indep. by Dinitz (Dinic) + Edmonds + Karp in early 1970's.

- Just run Ford-Fulkerson, but select the augmenting path with the fewest edges.

- Converges in $O(n \cdot m)$ augmentations (See text for proof)

- Total time = $O(n \cdot m^2)$
  (Dinitz further reduced to $O(n^2 m)$.) (Better since $n \leq m \leq n^2$)

Faster still?

- Goldberg + Tarjan (1986) - $O(n \cdot m \log \frac{n^2}{m})$

- King, Rao, Tarjan (1994) - $O(n \cdot m \cdot \frac{\log n}{\log(m / n \log n)})$
- Orlin (2013) - $O(n \cdot m)$

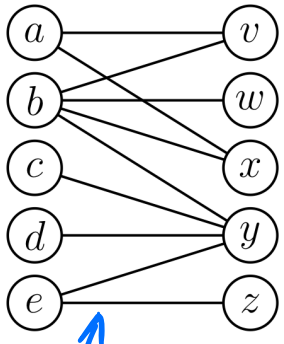
These are all quite complicated!

Applications of Network Flow:

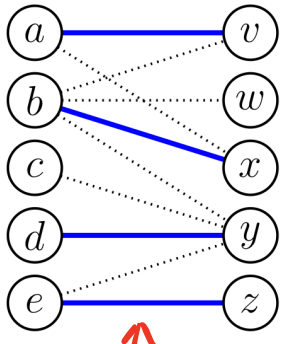
Maximum matching in bipartite graphs

- Given two sets of objects
  (e.g. males + females, students + grad schools
  customers + agents)
- ... and some compatibility relation:
  (e.g. female swipes right on male,
  student wants to attend grad school,
  agent has expertise to serve customer)
- ... pair up as many as possible (one to one)

Bipartite Graph

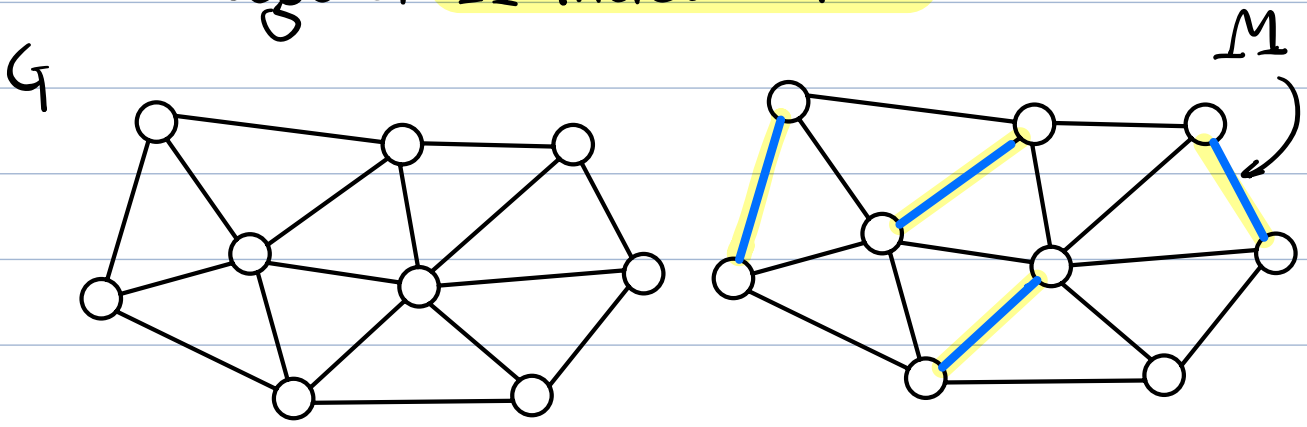


edge ≡ compatible



pairing up

Def: Given a graph $G = (V, E)$, a matching is a subset of edges $M \subseteq E$ such that for each $v \in V$, there is at most one edge of M incident to v.



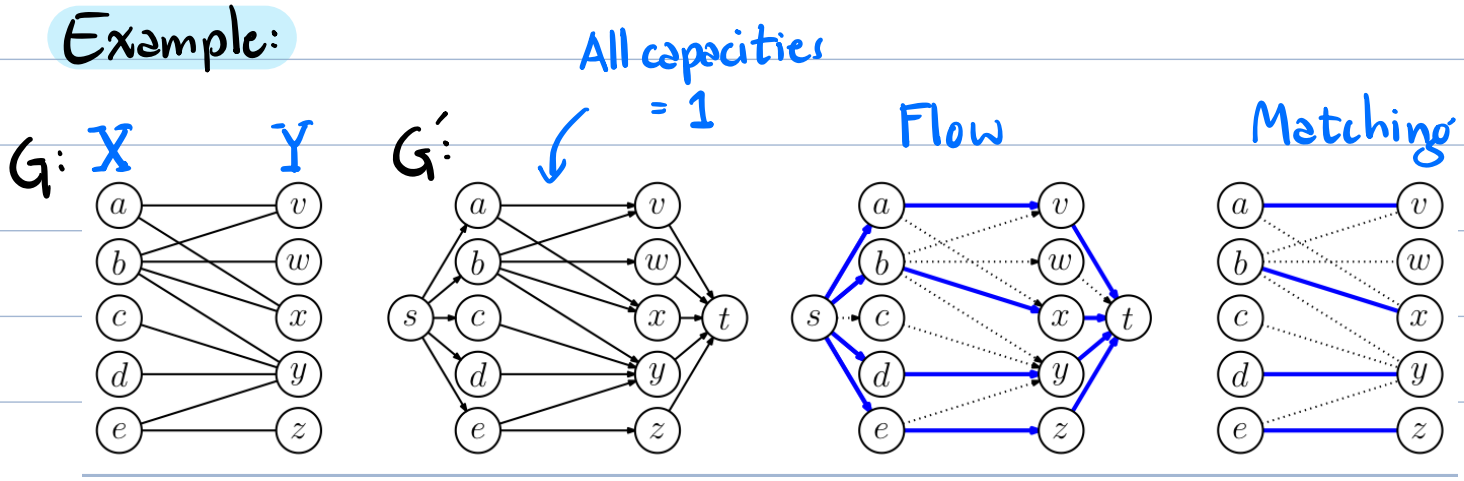
Recall: G is bipartite if $V = X \cup Y$ & all edges $\in X \times Y$

Problem: Given a bipartite graph G, compute the max. sized matching in G.

Claim: We can reduce max. bipartite matching to network flow.

- Create source s & add edges (s, u) $\forall u \in X$
- Create sink t & add edges (v, t) $\forall v \in Y$
- All capacities = 1

Example:



Claim: G has a matching of size k iff G' has a flow of value k

Proof: (Sketch)

- Because capacity-in + capacity-out = 1, cannot push more than one unit of flow through each vertex
- $\Rightarrow$ At most edge will carry flow (assuming integer flow values)

Summary:

- Ford-Fulkerson - Could be very slow
- Scaling Algorithm - $O(m^2 \log C)$ simple
- Strong / Weak Polynomial Time
- Application - Bipartite Max Matching

CMSC 451 - Algorithm Design
Lecture 14 - Network Flow - Circulations + Applications

Circulation with Demands:

Transportion:

- Network (where commodities flow)
  - Edge capacities
- Supply nodes - generate flow
- Demand nodes - consume flow

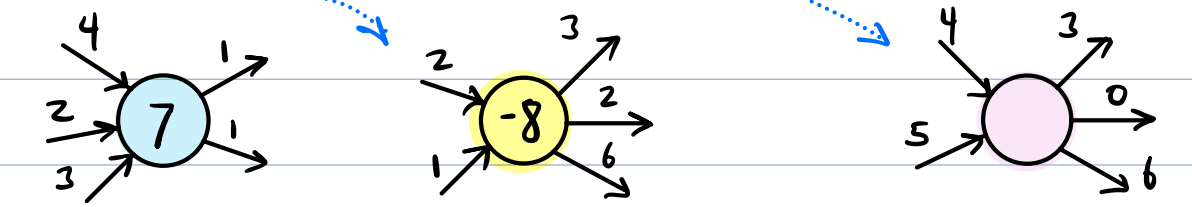
Decision problem:

- Can we manage all the flow from suppliers to consumers?

Supply + Demand:

Given digraph $G=(V,E)$

- Each node $v \in V$ has demand $d(v)$:
  - demand: $d(v) > 0$ (d_v net incoming)
  - supply: $d(v) < 0$ ($-d_v$ net outgoing)
  - balanced: $d(v) = 0$ (incoming = outgoing)



$f^{in} - f^{out} = 7$ $f^{in} - f^{out} = -8$ $f^{in} - f^{out} = 0$

Circulation: Given a directed graph $G=(V,E)$ with edge capacities $c(u,v)>0$ and vertex demands $d(v)$, a circulation is a function $f: E \to \mathbb{R}$ such that:

Capacity constraint: $0 \le f(u,v) \le c(u,v), \forall (u,v) \in E$

Supply/Demand constraint:

$$f^{in}(v) - f^{out}(v) = d(v), \forall v \in V$$

Recall:

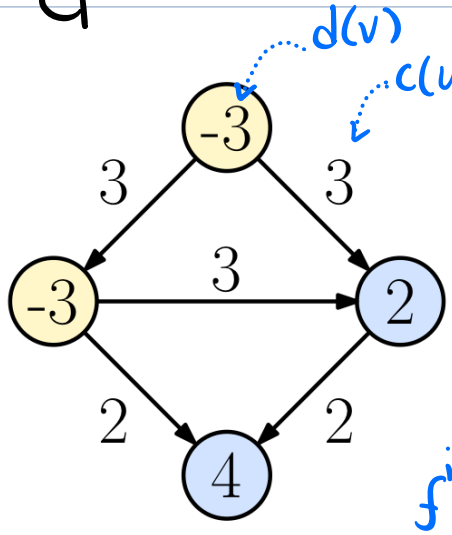
$$f^{in}(v) = \sum_{(u,v)\in E} f(u,v)$$

$$f^{out}(v) = \sum_{(v,w)\in E} f(v,w)$$

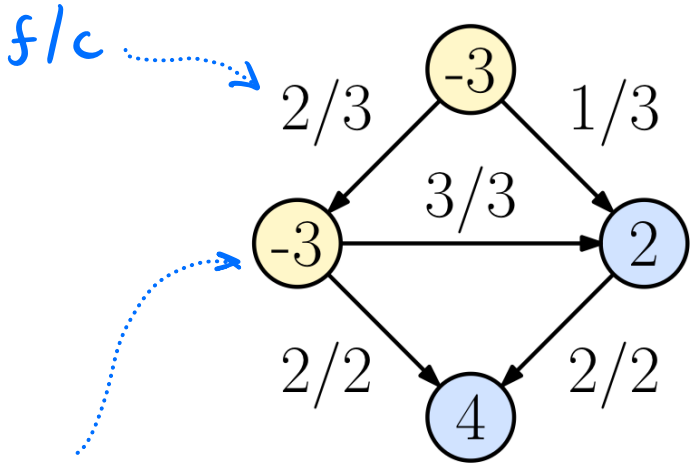
(-5) supply node

(4) demand node

G:



A circulation:



$f^{in} = 2$

$f^{out} = 3+2 = 5$

$f^{in} - f^{out} = 2-5 = -3 = d(v)$

Define: $S = \{v \mid d(v) < 0\}$ Supply nodes

$T = \{v \mid d(v) > 0\}$ Demand nodes

Clearly: $\sum_{v \in T} d(v) = -\sum_{v \in S} d(v)$

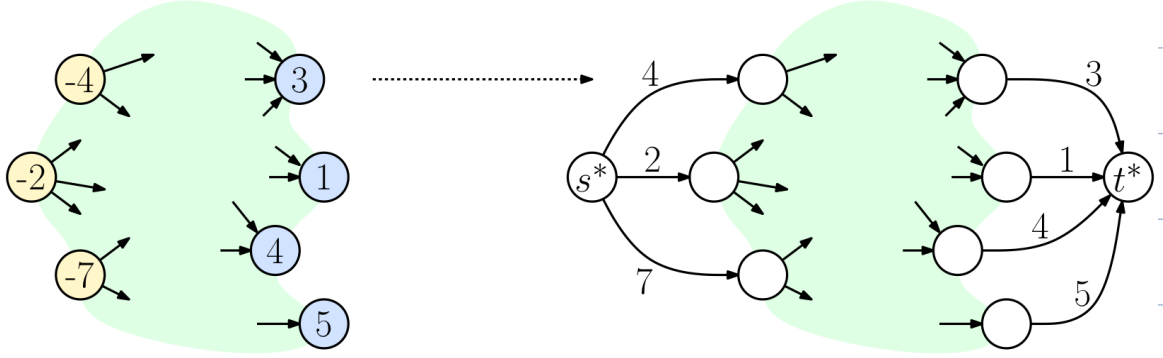
demand = supply

(or no feasible circulation exists)

Define: Total demand: $\sum_{v \in T} d(v)$ denoted $D(G)$

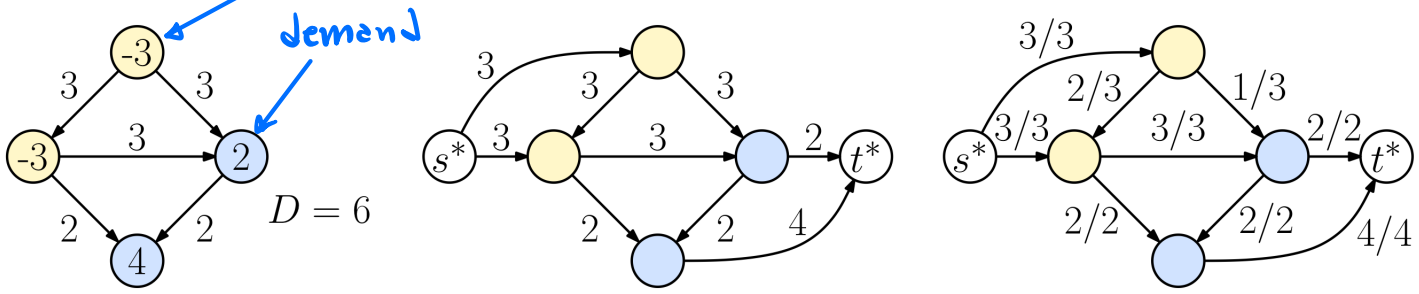
Solving Circulation by Network Flow

- Treat supply nodes as sources
- Treat demand nodes as sinks
- Add super-source & super-sink to connect



- Create network $G' = (V', E')$, $V' \leftarrow V$, $E' \leftarrow E$
- Add s^* & t^* to V'
- For $v \in V$: Add (s^*, v) capacity $= -d(v)$ if $v \in S$
  Add (v, t^*) capacity $= d(v)$ if $v \in T$

Example: supply, demand



Claim: G has a feasible circulation iff G′ has a flow of value $D(G)$

Recall:
$= \sum_{v \in T} d(v)$
$= -\sum_{v \in S} d(v)$

Proof:

($\Rightarrow$) Let f be a circulation in G.

Create flow f' by:
- saturate all edges out of s^*
- " " " into t^*
- all other edges same as f

This satisfies -
- capacity constraints
- supply-demand constraints

} Exercise

($\Leftarrow$) Let f' be flow of value D in G′

Obs: All edges out of s^* + into t^* must be saturated

$\Rightarrow$ All supply/demands satisfied

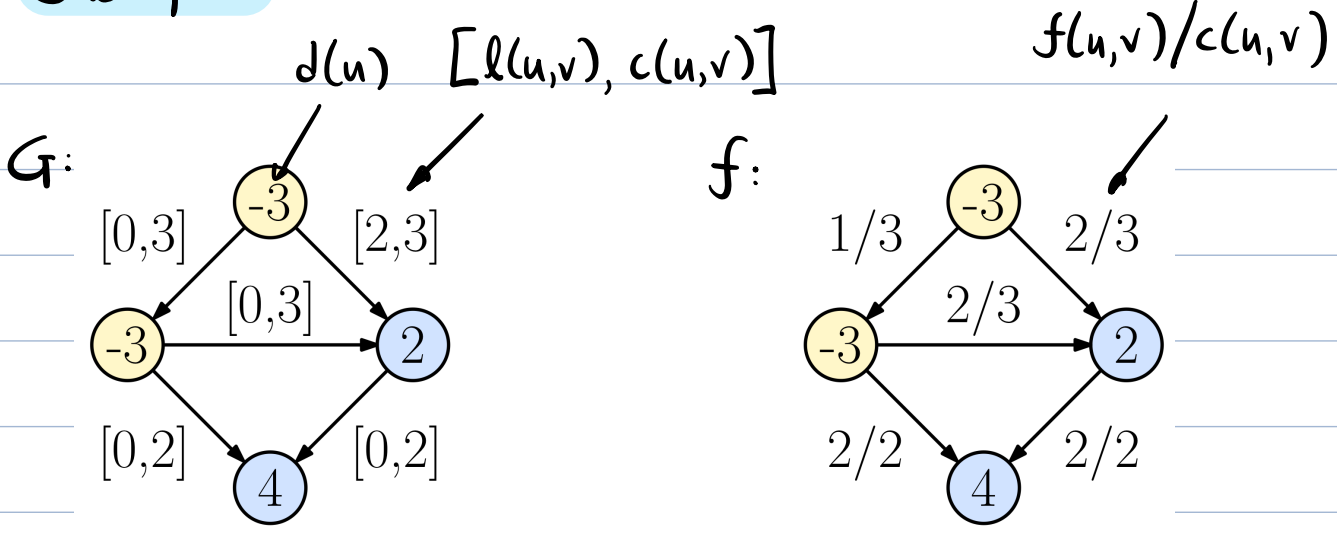
} Exercise

Circulations with Upper + Lower Capacity Bounds

- Sometimes, we want to apply both upper + lower bounds on flow amounts
- Lower bound ⇒ guarantees minmum utilization

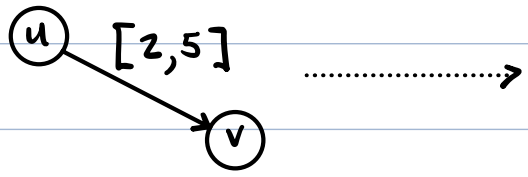
- For each edge $(u,v) \in E$, let:
  - $c(u,v)$ = upper bound on flow
  - $\ell(u,v)$ = lower bound on flow
- New conditions: A flow is a circulation if
  - $\ell(u,v) \leq f(u,v) \leq c(u,v)$, $\forall (u,v) \in E$
  - $f^{in}(v) - f^{out}(v) = d(v)$, $\forall v \in V$

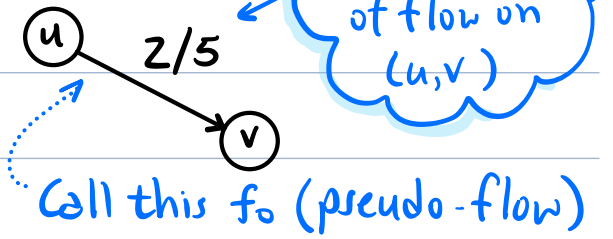
Example:



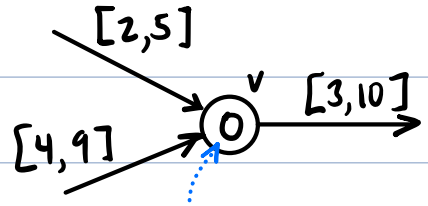
But how can we enforce these lower bounds?
- Can do this by "mucking" with demands!

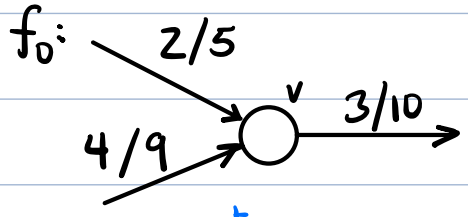
Step 1: Start by forcing flow of $\ell(u,v)$ on each edge (u,v)





But this forced flow won't generally satisfy vertex demands:



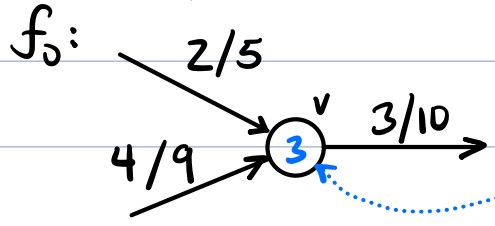


$f_0^{in}(v) - f_0^{out}(v) = (2+4) - 3$

$= 3 \neq d(v)$ ☹

Step 2: Determine excess flow through each vertex

$$\chi(v) = f_0^{in}(v) - f_0^{out}(v)$$



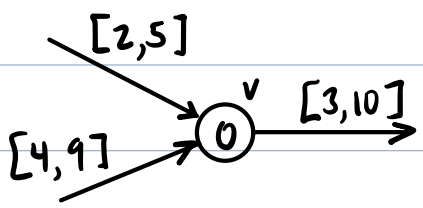
$f_0^{in}(v) - f_0^{out}(v) = (2+4) - 3$

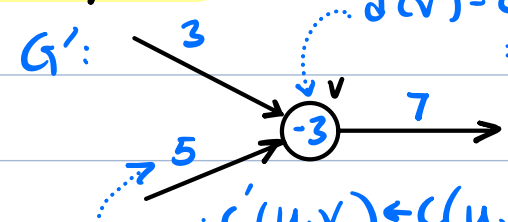
$= 3 = \chi(v)$

Step 3: Eliminate f_0 by reducing upper capacities + adjusting demands:

$$c'(u,v) \leftarrow c(u,v) - f_0(u,v) = c(u,v) - \ell(u,v)$$

$$d'(v) \leftarrow d(v) - \chi(v)$$





$d'(v) = d(v) - \chi(v)$

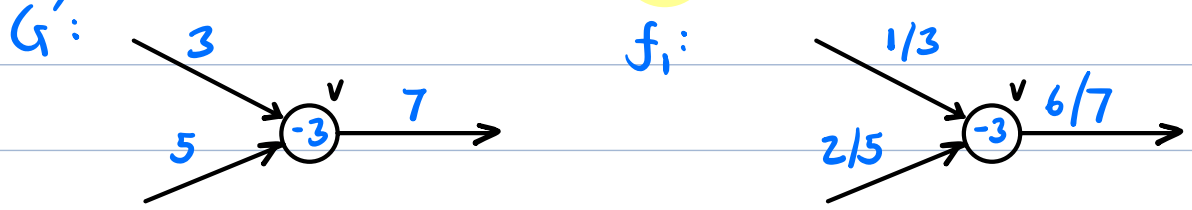
$= 0 - 3$

$c'(u,v) \leftarrow c(u,v) - \ell(u,v)$

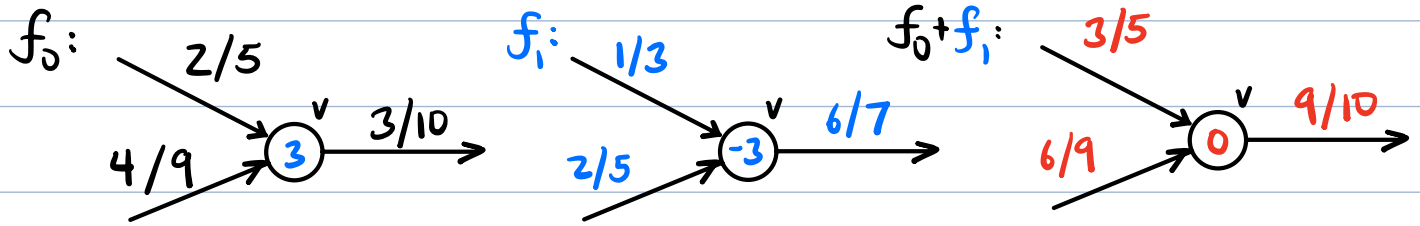
$= 9 - 4 = 5$

Observe: Lower constraints gone - just upper remain

Step 4: Apply standard circulation algorithm to this modified network G'
→ Let f_1 be result



Step 5: Add back the (eliminated) forced lower flow:
return $f_0 + f_1$



Summary: Given G with $[\ell, u]$ capacity constraints

- Create pseudo-flow $f_0(u,v) \leftarrow \ell(u,v)$
- Create network G' (same vertices + edges)
  $c'(u,v) \leftarrow c(u,v) - \ell(u,v)$
  $d'(u,v) \leftarrow d(v) - x(v)$, $x(v) = f_0^{in}(v) - f_0^{out}(v)$
- Compute (standard) circulation in $G' \rightarrow f_1$
- If G' has no circulation – report infeasible
  else return $f \leftarrow f_0 + f_1$

Correctness:

Lemma: G (with both lower + upper constraints) has a valid circulation iff G' (just upper constraints) has a valid circulation.

See Kleinberg + Tardos for proof

- Intuitively: f_0 handles lower bounds
  d' handles imbalance in demands
  c' limits additional flow above $\ell(u,v)$ so it doesn't exceed $c(u,v)$
  $f_0 + f_1$ combines the two flows

Application - Survey Design

Setup:

- A company sells k products

- Survey n customers - "How satisfied..."
- Rules:
  1. Only ask about products customer purchased
  2. Customer i: Ask about $\geq c_i^-$ products and $\leq c_i^+$ products
  3. Product j: Survey $\geq p_j^-$ customers and $\leq p_j^+$ customers

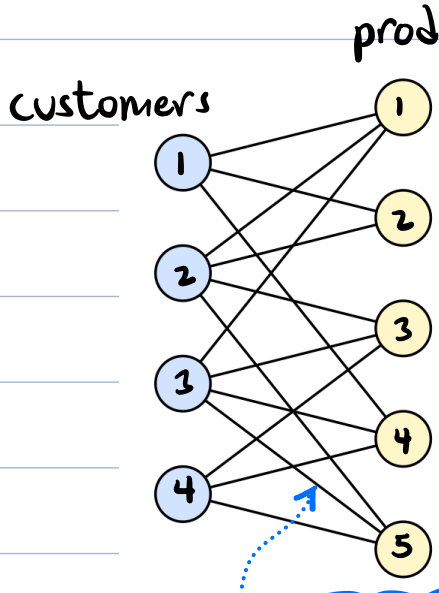
Q: Given $C = \{[c_1^-, c_1^+], \dots, [c_n^-, c_n^+]\}$

$P = \{[p_1^-, p_1^+], \dots, [p_k^-, p_k^+]\}$

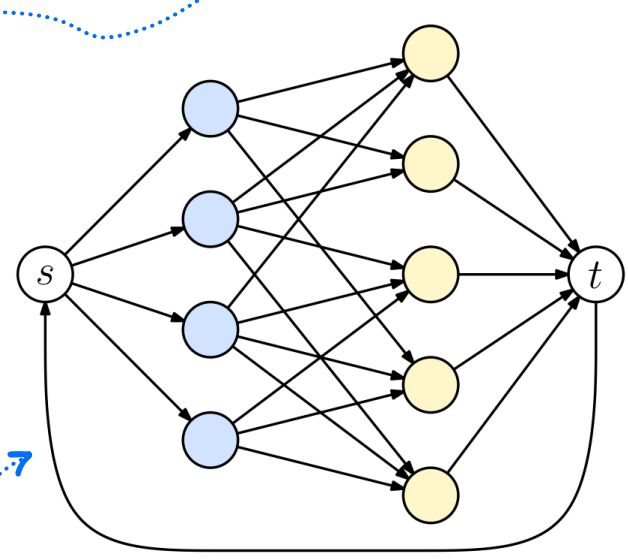
+ purchase information

does there exist a survey satisfying all rules?

Purchase information - Modeled as bipartite graph



customer 3 purchased product 5



- Direct edges from cust. to prod.
- Add vertex s + edges to customers
- Add vertex t + edges from products
- Add edge (t, s)

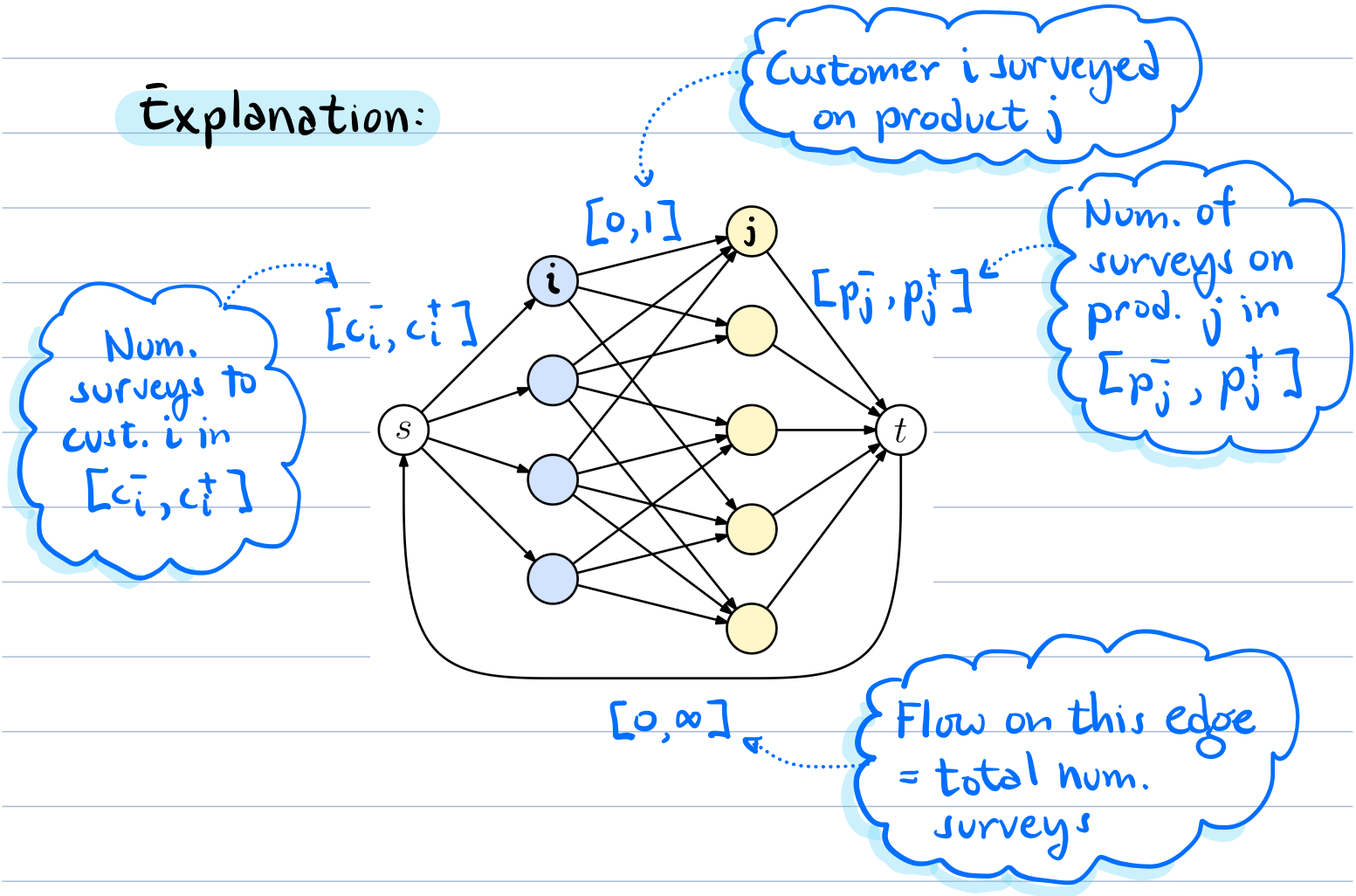
All demands = 0

(flow conservation)

Capacities:

- Customer edges (s, i): $[c_i^-, c_j^+]$
- Product edges (j, t): $[p_j^-, p_j^+]$
- Customer/product edges: $[0, 1]$
- (t, s): $[0, \infty]$

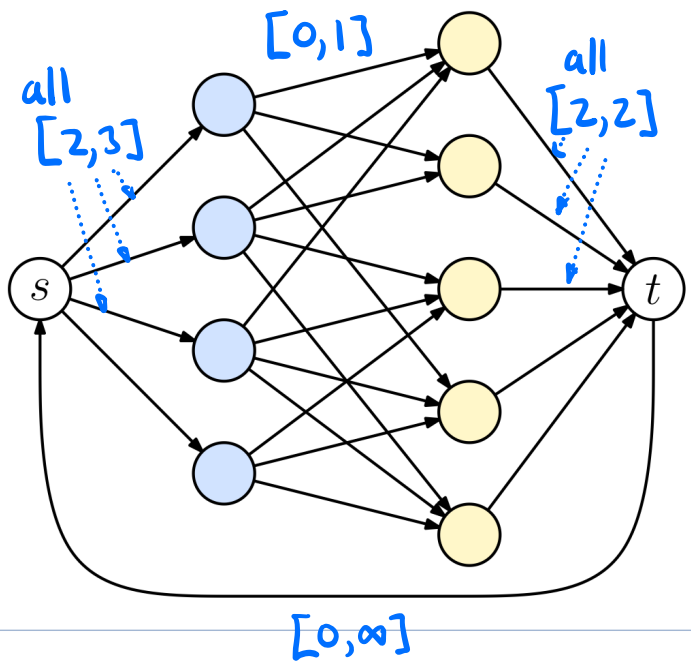
Explanation:



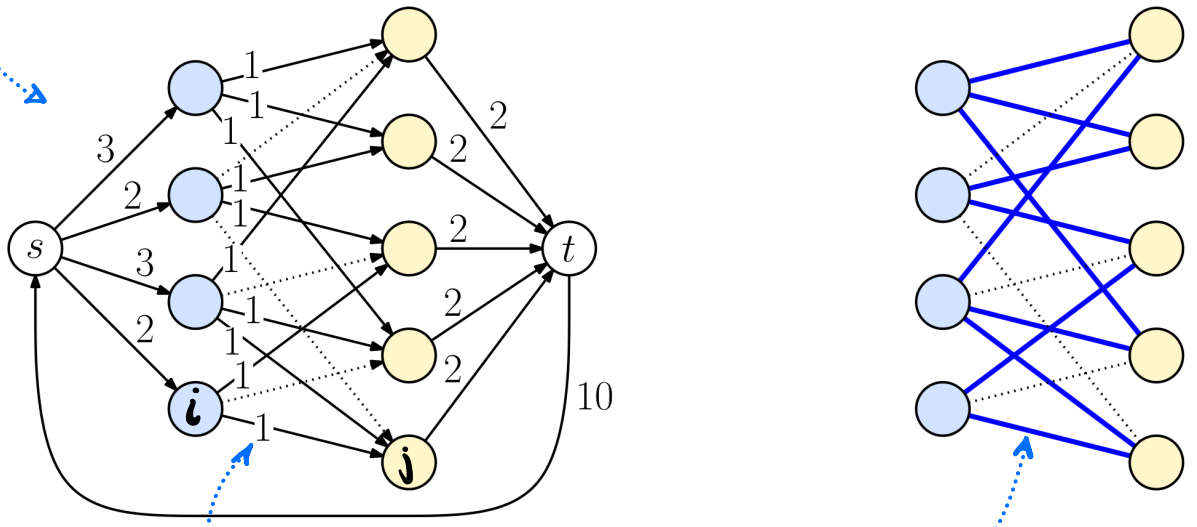
Example:

- For simplicity, set $[c_i^-, c_i^+] = [2,3]$, for all i

  $[p_j^-, p_j^+] = [2,2]$, for all j



Compute Circulation - f (integer valued)



if f(i,j) = 1 survey customer i about product j

Is this a faithful implementation of survey rules?

Claim: There exists a valid circulation in G iff there is a valid survey design

Proof:

(⇒) Suppose G has a valid circulation f.

- Survey: Ask cust. i about prod j if f(i,j) = 1
- We'll show this survey is valid:
  - ① Only survey customers about purchases
    - ⇐ Create edge (i,j) if i purchases j

② Num. of surveys for cust i in $[c_i^-, c_i^+]$

⇐ Since $f(s,i) \in [c_i^-, c_i^+]$

+ flow conservation $(d(i) = 0)$

③ Num. of surveys for prod. j in $[p_j^-, p_j^+]$

⇐ Since $f(j,t) \in [p_j^-, p_j^+]$

+ flow conservation $(d(j) = 0)$

⇒ This is a valid survey ✓

(⇐) Suppose there is a valid survey.

Construct flow in G:

- $f(i,j) = 1$ if cust. i surveyed on prod j
- $f(s,i)$ = total surveys for cust i
- $f(j,t)$ = total surveys for prod j
- $f(t,s)$ = total num. of surveys

We'll show this is a valid circulation:

① Cust. i surveyed about prod. j only if purchased ⇒ edge $(i,j) \in G$

② Cust. i surveyed about $[c_i^-, c_i^+]$ products ⇒ $f(s,i)$ satisfies capacities

③ Prod. j involved in $[p_j^-, p_j^+]$ surveys ⇒ $f(j,t)$ satisfies capacities

Finally flows-in = flows-out so all demands (zero) are met ⇒ valid circulation ✓

Ugh! 

- These proofs are lengthy + rather tedious.
- Important to understand structure
  - Similer to NP-complete reductions

Summary:

- Circulations - vertex demand/supply
  - Reduction to max flow
- Circulations with lower + upper edge capacities
  - Reduction standard circulations
- Application - Survey Design
  - Reduction to circulations w. lower/upper capacities

CMSC 451 - Algorithm Design
Lecture 15 - NP-Completeness: Basic Definitions

So far...

- How to design and analyze efficient algorithms for combinatorial problems
- What do we mean by efficient?

- Polynomial time - $O(n^c)$, where c = constant (worst case)
- Exponential time - $\Omega(c^n)$, $c > 1$

Hard Problems:

Near end of 1960's:

- Poly-time algorithms for many problems
- Many problems defied poly-time solutions
  - Best solutions based on brute-force
    $\Rightarrow$ Exponential time

- Researchers sought the crucial property that distinguishes between them

- Spoiler alert - They failed!

Enter Stephen Cook (1971)

- For many combinatorial problems, the best solution has the following form:
  - guess the answer + verify it is correct

- NP-Complete - The "hardest problems" in NP

> If you can solve any of these efficiently - you can solve every problem in NP efficiently.

NO WAY!

- Boolean satisfiability (SAT) is NP-complete

Enter Richard Karp (1972)

- Showed that 21 well-known problems are NP-Complete - This is major!

  - Travelling salesman
  - Hamiltonian cycle
  - Clique
  - Independent set
  - Graph coloring
  - ⋮

- Is P = NP? $1M prize for the answer

Decision Problems + Language Recognition

- It will simplify the theory to assume all problems have same output - yes or no → Decision problem

- Optimization: Find min. spanning tree for G
- Decision: Given $G + z \geq 0$, does G have a spanning tree of weight $\leq z$?

Language Recognition

- For historical reasons (automata theory predates algorithm theory) NP problems are expressed as language recognition

- Assume we have function serialize(I) - Encodes input I (e.g., graph, set, number) as a string

Decision problems	→	Language recognition
Does G have an MST of weight $\leq z$?	→	MST = $\{(G, z) \mid$ G's MST has weight $\leq z\}$

Complexity Class P:

Decision Problems

Def: P is the set of languages such that membership can be determined in polynomial time.

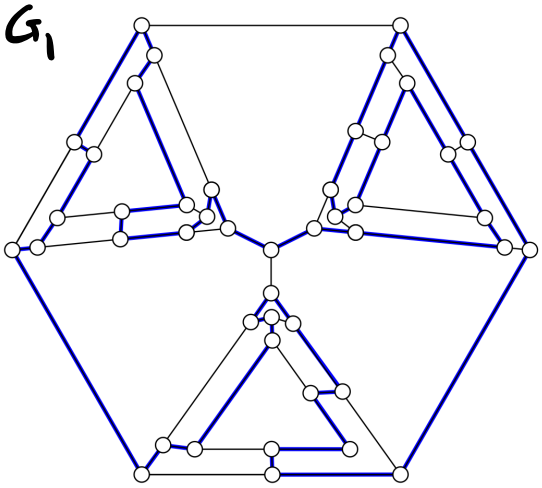
Eg. Can solve MSTs efficiently, so MST $\in$ P

Are there problems (languages) not in P?

Hamiltonian Cycle: Given a graph G, a Hamiltonian cycle is a simple cycle that visits each vertex of G (exactly once)

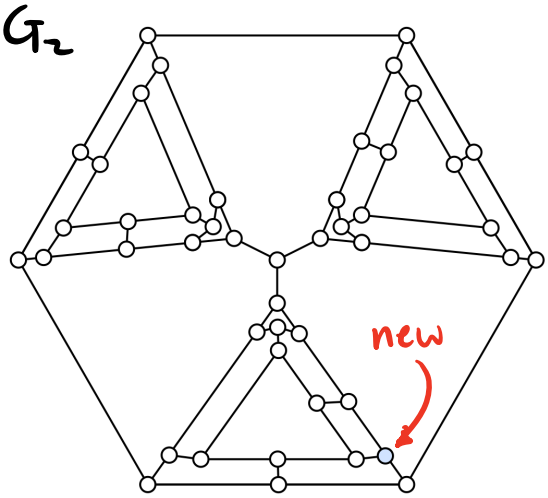
$HC = \{ G \mid G \text{ has a Ham. cycle} \}$

G_1



$G_1 \in HC$

G_2



$G_2 \notin HC$? (don't know)

Is $HC \in P$? (Not known to science)

HC has a nice property - verifiability

- If $G \in HC$, it is easy to verify this
  (Show me the cycle, and I'll check)
- However, if $G \notin HC$, how to verify ???
  (No idea)

Complexity Class NP:

> **Def:** NP is the set of languages such that membership can be verified in poly time.

What do you mean by "verified"?

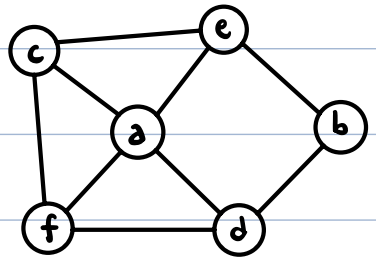
Given a language L (set of strings)

- Given string x, want to know whether $x \in L$.
- A certificate is a "helper string" y:
  - if $x \in L$, y can be used to prove this in polynomial time
  - if $x \notin L$, then we don't care

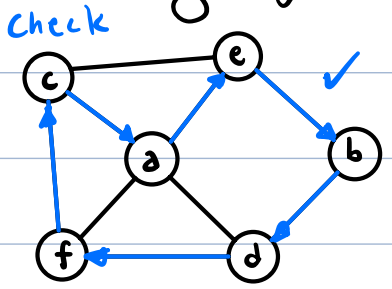
HC = { G | G has a Hamil. Cycle }

Certificate = List of vertices forming cycle

x:



y: ⟨c, a, e, b, d, f⟩



This is weirdly assymetric!

- if $x \in L$ - need to verify
- if $x \notin L$ - don't care

That is just how it is defined.

Are their (natural) languages that are not poly-time verifiable?

UHC = { G | G has a unique Ham. cycle }

If $x \in$ UHC, you can show one Ham. cycle, but how do you prove there no other?

How to show P ≠ NP? Strategy:

- Find the hardest problem in NP
- Show that this problem is not in P
- Such a problem will be called NP-complete

Before defining NP-Completeness, we need to discuss reductions.

Polynomial-Time Reductions:

Motivation: How do we show that problem is hard to solve?

Consider:

H - a well-known (very likely) hard problem - Many experts tried + failed

U - your problem, which you suspect may be hard (but who trusts you?)

Want to show:

$H \notin P \Rightarrow U \notin P$

$H \notin P$: what everyone believes

$U \notin P$: what you want to prove

Let's prove contrapositive!

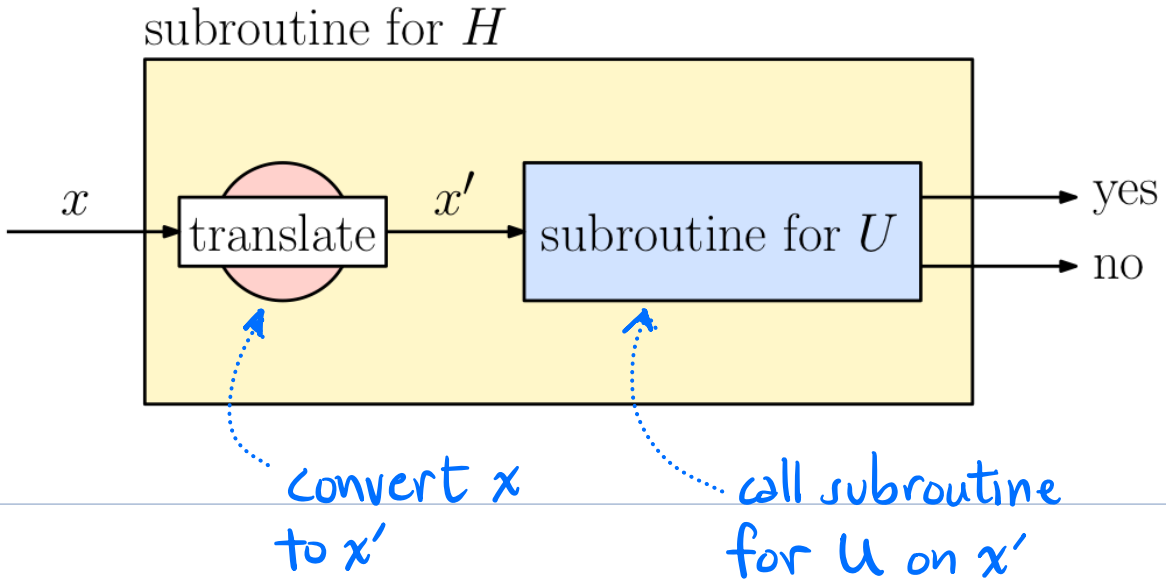
$U \in P \Rightarrow H \in P$

$U \in P$: If I could solve my problem

$H \in P$: Then I could solve a famous hard problem ← NO WAY!

How? Use a subroutine for my problem U (as a black box) to solve H

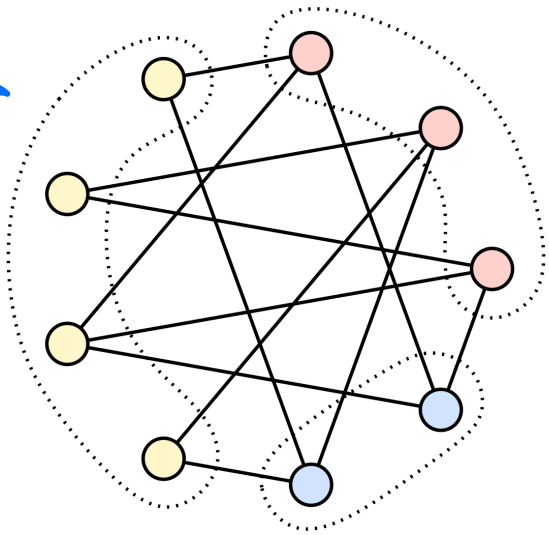
Pictorially: Is $x \in H$?



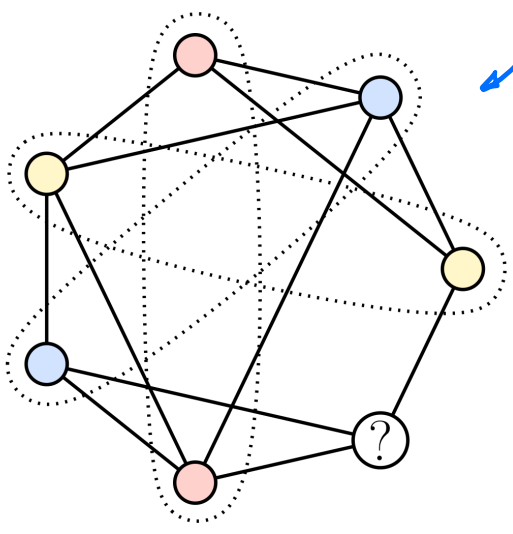
Example: Let's try this on two actual problems.

Hard: 3-Coloring (3COL) - Given a graph G can its vertices be labeled with one of three colors, so that no two adjacent vertices have same color.

3-Colorable



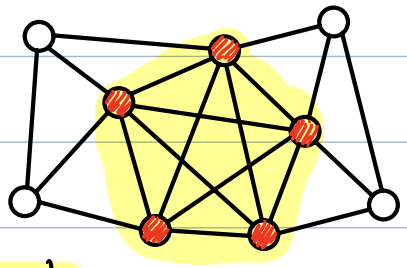
Not 3 colorable (Need at least 4)



No know poly-time algorithm - even for planar graphs

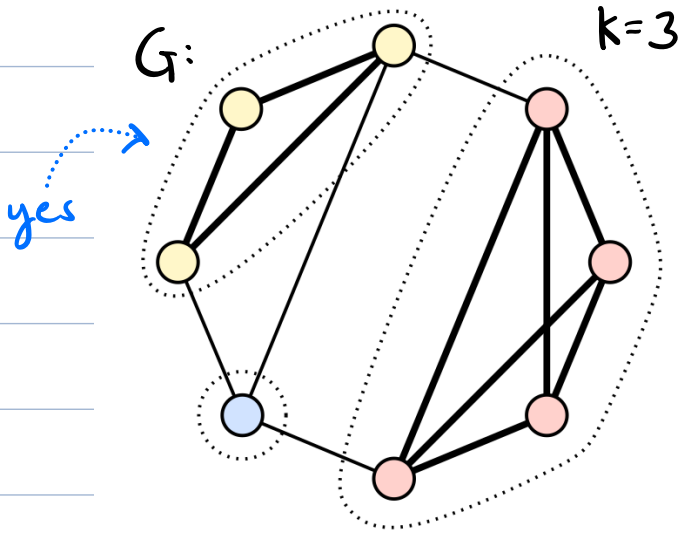
Your Problem (which you want to show is hard)

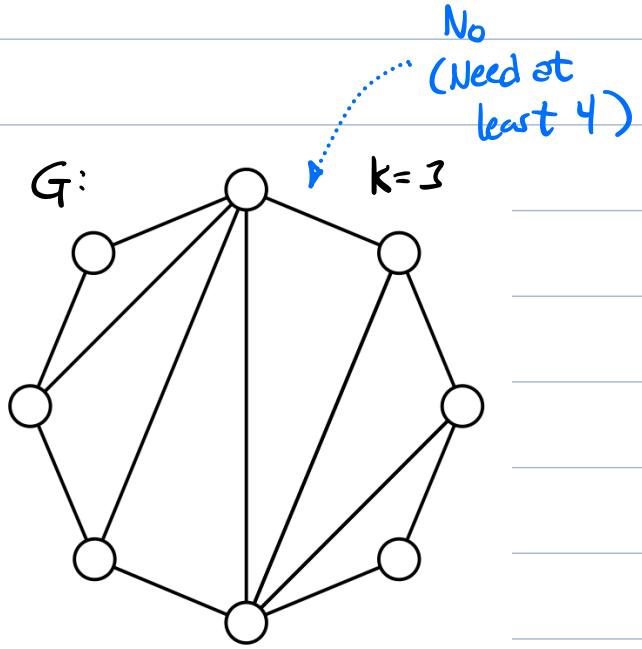
Def: A clique is a completely connected subgraph



Clique Cover (CCov): Given a graph $G=(V,E)$ and integer k, can we partition vertices into k sets $V_1, V_2, \ldots, V_k$ such that each V_i is a clique in G.

Example:





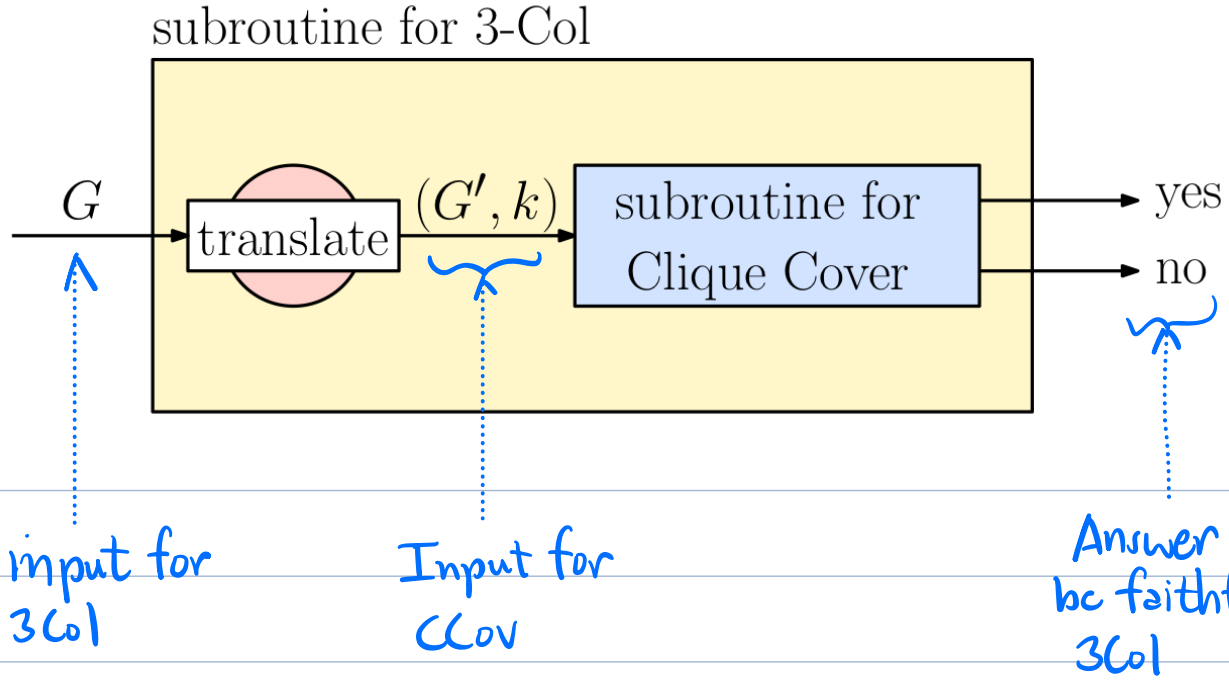
Want to show:

$$\text{CCOV} \in \text{P} \Rightarrow \text{3COL} \in \text{P}$$

If we could solve our problem efficiently — then — We could solve 3Col efficiently (No way!)

How? Suppose we had an efficient subroutine for CCov

Show we could use it to solve 3Col



How to translate?

- Both involve partioning vertices
  - 3Col - 3 color classes
  - CCov - k cliques

  } set k=3?
- Adjacency condition
  - 3Col - If u≠v have same color, $(u,v) \notin E$
  - CCov - If u≠v in same clique, $(u,v) \in E$

  } complement G!

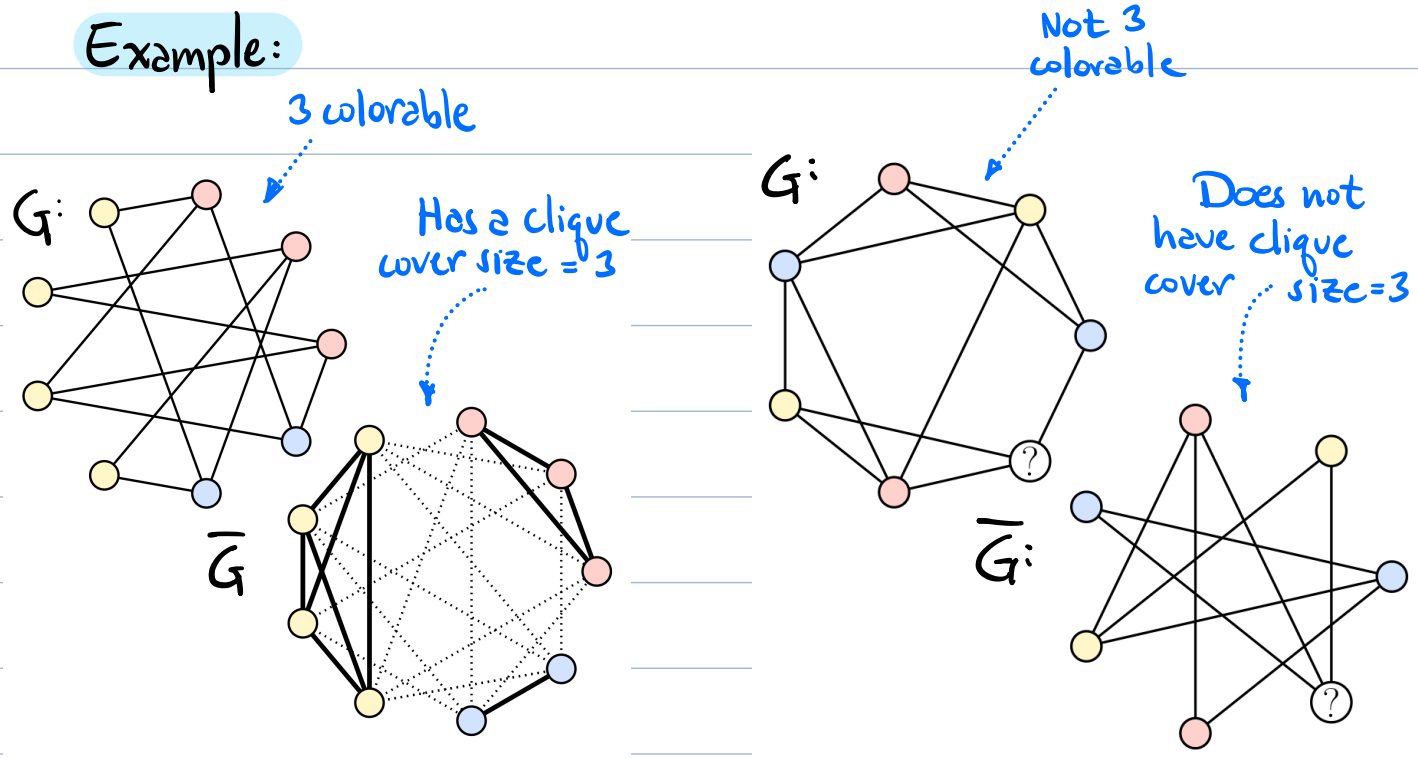
Reducing 3Col to CCov:

- Given G for 3Col:
  - Set $k \leftarrow 3$
  - Compute $\bar{G}$ (complement edge set)
- Run CCov on $(\bar{G}, k)$ + return whatever it returns

Is this a **faithful translation**?

> **Claim:** $G=(V,E)$ is 3-colorable iff $\bar{G}=(V,\bar{E})$ has a clique cover of size 3.

Example:



A **picture** is **not a proof**.

Proof:

($\Rightarrow$) If G is 3-colorable, let V_1, V_2, V_3 be partition of vertices into color classes.

If u & v in same color class, then $(u,v) \notin E$.

This implies that $(u,v) \in \bar{E}$ ← complement

$\Rightarrow$ All vertices of V_i (for $i=1,2,3$) are adjacent in $\bar{G}$

$\Rightarrow$ V_i is a clique in $\bar{G}$ (for $i=1,2,3$)

$\Rightarrow$ $\bar{G}$ has a clique cover of size 3. ✓

($\Leftarrow$) If $\bar{G}$ has a clique cover V_1, V_2, V_3 (complement → $\bar{\ }$)

then if u,v in same set V_i, then $(u,v) \in \bar{E}$

$\Rightarrow$ $(u,v) \notin E$

$\Rightarrow$ All vertices of V_i (for $i=1,2,3$) are non-adjacent in G

$\Rightarrow$ V_1, V_2, V_3 define a 3-coloring in G

$\Rightarrow$ G is 3-colorable ✓

Finally, note that our reduction runs in polynomial time (complement G)

- It did not try to color G
- It does not even know whether G can be 3-colored

Polynomial Reducibility:

Def: Language L_1 is **polynomial-time reducible** to language L_2 (denoted $L_1 \leq_p L_2$) if there is a **poly-time function f**, s.t.

$$\forall x \quad x \in L_1 \text{ iff } f(x) \in L_2$$

We showed $3Col \leq_p CCov$ (where $f(G) \to (\bar{G}, 3)$)

If $L_1 \leq_p L_2$ + L_2 is solvable in poly-time, so is L_1

also, if L_1 is not solvable in poly-time, neither is L_2

Summary:

Lemma: Given languages L_1, L_2, L_3

(i) $L_1 \leq_p L_2$ + $L_2 \in P$ $\Rightarrow L_1 \in P$

(ii) $L_1 \leq_p L_2$ + $L_1 \notin P$ $\Rightarrow L_2 \notin P$

(iii) $L_1 \leq_p L_2$ + $L_2 \leq_p L_3$ $\Rightarrow L_1 \leq_p L_3$

We now can define **NP-completeness.**

Intuitively- The "**hardest**" problems in NP.

Recall:

NP: Set of languages **verifiable in poly-time**

Def: A language L is **NP-hard** if

$$\forall\, L' \in NP,\ L' \leq_P L$$

Def: A language L is **NP-complete** if

(1) $L \in NP$ ⟵ verifiable

(2) L is NP-hard ⟵ but not easy!

This condition seems impossible, since it involves all languages in NP - an infinite set!

Next lecture: We'll prove there is an NP-complete problem

Summary:

- P - polynomial time
- NP - poly time verifiable - certificate
- Poly-time reducibility "$\leq_P$"
- NP-hardness
- NP-completeness

CMSC 451 - Algorithm Design
Lecture 16 - NP-Completeness: 3SAT + Independent Set

Recap:

P: Set of languages (decision probs.) solvable in poly-time

NP: Set of languages verifiable in poly-time

$\leq_p$: $L_1 \leq_p L_2$ means there is a poly-time function f s.t.

$\forall x \quad x \in L_1$ iff $f(x) \in L_2$

NP-hard: L is NP-hard if $\forall L' \in NP, \; L' \leq_p L$

NP-complete: L is NP-complete if

(1) $L \in NP$

(2) L is NP-hard

If any NP-complete problem is solvable in poly-time, they all are ($P = NP$)

If any problem in NP is not solvable in poly-time, then no NP-complete problem is solvable in poly-time ($P \neq NP$)

Boolean Satisfiability + Cook's Theorem:

- Prove problems NP-complete through reductions
- Need one initial NP-complete problem to start

Boolean Satisfiability (SAT)

Boolean formula:

variables: x, y, z or $x_1, x_2, \ldots$

→ Each can be assigned true or false

operations:

- $\wedge$ – "and" $x \wedge y$ - true if both true
- $\vee$ – "or" $x \vee y$ - true if either/both true
- $\neg$ – "not" $\neg x$ - reverses true ↔ false

often written: $\bar{x} \equiv \neg x$

Examples:

$$F_1(x,y,z) = (x \wedge (y \vee \bar{z})) \wedge ((\bar{y} \wedge \bar{z}) \vee \bar{x})$$

$x = T$
$y = z = F$

$$F_2(x,y) = (\bar{z} \vee x) \wedge (z \vee y) \wedge (\bar{x} \wedge \bar{y})$$

Def: A formula is satisfiable if it is possible to assign values to variables so it evaluates to true

Not satisfiable

Cook's Theorem: SAT is NP-Complete

The proof is long + technical, but here is a high-level intuitive overview

SAT ∈ NP: Given a formula $F(x_1, \ldots, x_n)$

certificate is assignment to variables

$x_1 \leftarrow T$, $x_2 \leftarrow F$, $x_3 \leftarrow T$, ...

To verify, plug these into formula

evaluate + check that result = true

SAT is NP-hard:

- Given any language $L \in NP$,
  want to show $L \leq_p SAT$ ($x \in L$ iff $f(x) \in SAT$)
- For any instance x for L we know
  - Can verify that $x \in L$
  - Involves a certificate + verification program
  - Encode certificate as a bit string

    e.g. $C = 0101101\ldots$
  - Express bits as variables

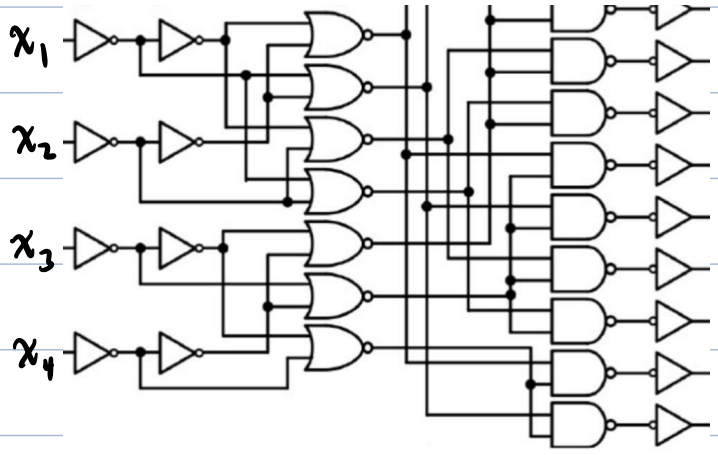
    x_1 = 1st bit of C (0 = F, 1 = T)

    x_2 = 2nd bit of C

    ⋮

- Verification is a poly-time program.

- Program can be expressed as a circuit of polynomial size



- Circuit can be expressed as boolean formula $f(x)$ of polynomial size.

- This formula faithfully simulates the verification program:

$$x \in L \text{ iff } f(x) \in SAT$$

$\therefore \forall L \in NP \quad L \leq_p SAT \Rightarrow$ SAT is NP-hard!

□

Cook proved a stronger result -

Satisfiability is NP-complete, even if we restrict the format of the formulas to 3CNF.

Notation:

literal: A variable or its complement, x_i or $\bar{x}_i$

3-conjunctive normal form (3CNF):

A formula that is the conjunction ("$\wedge$") of clauses, each of which is the disjunction ("$\vee$") of 3 literals.

Example:

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge \dots$$

joined by "ands"

clause = "or" of 3 literals

Def: 3SAT - Satisfiability of formulas expressed in 3CNF form.

Theorem (Cook). 3SAT is NP-complete

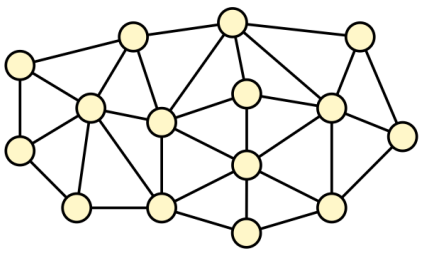
Reductions are simpler

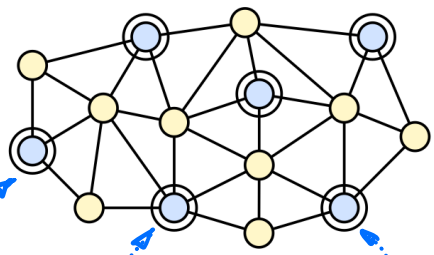
Given that 3SAT is NP-Complete, let's try another.

Independent Set (IS): Given a graph $G=(V,E)$ + integer k, does G contain a subset of vertices $V' \subseteq V$ of size k so that no two are adjacent?

Example:

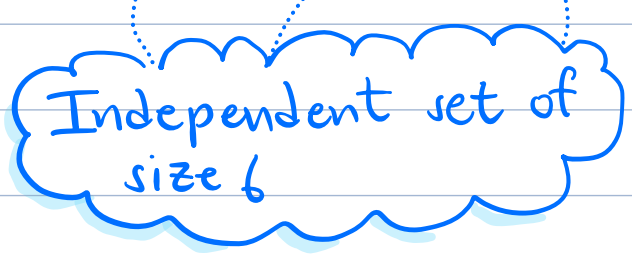
G:





$(G,6) \in IS$

$(G,7) \notin IS$



Claim: IS is NP-Complete

Need to show: (1) IS $\in$ NP (verifiable)

(2) IS is NP-hard

(1) IS $\in$ NP:

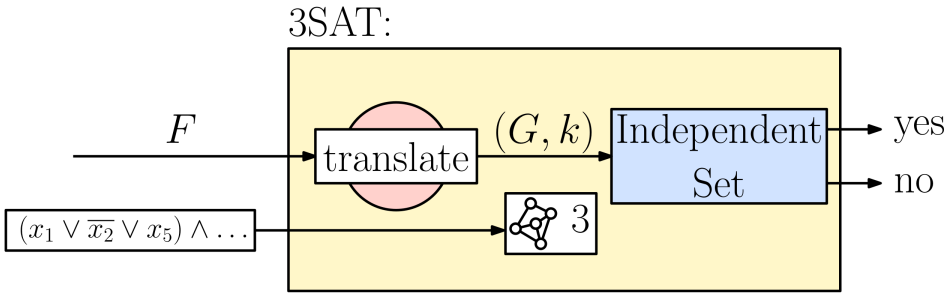
Given instance (G,k), the certificate consists of a set of k vertices of G. We consider each pair u,v from this set & check that they are not adjacent. If so, we accept the certificate & otherwise we reject it.

(2) IS is NP-hard:

We'll show that: $3SAT \leq_p IS$

There exists poly-time function f that maps a 3CNF formula F to a pair (G,k) s.t.

$F \in 3SAT$ iff $(G,k) \in IS$



How? These problems seem unrelated!

Let's see what they share in common:

Selection:

3SAT: Which variables are true
≡ which literals are true

IS: Which vertices are in indep. set

literals map to vertices?

Requirements:

3SAT: Logical "and" of clauses, each is "or" of 3 literals
⇒ at least one true literal per clause

IS: V' must contain k vertices

set k = number of clauses

Restrictions:

3SAT: x_i + $\bar{x}_i$ cannot both be true

IS: If $(u,v) \in E$, u + v cannot both be in indep. set

Create edge between x_i — $\bar{x}_i$

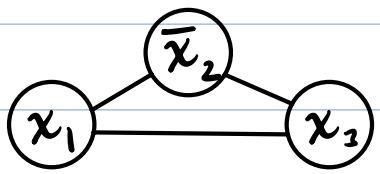
Plan:

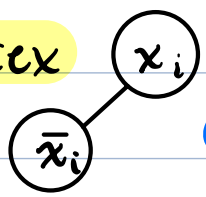
- For each clause, create a vertex for each literal. (Clusters of 3 vertices)

$(x_1 \vee \bar{x}_2 \vee x_3) \longrightarrow$

- Connect vertices with each cluster (Forces us to select only one as the "satisfying" literal)

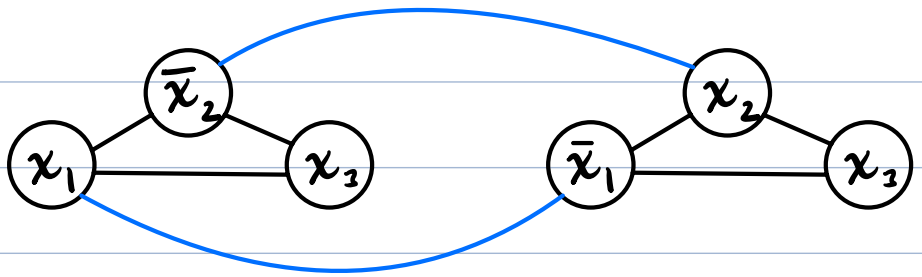
$(x_1 \vee \bar{x}_2 \vee x_3) \longrightarrow$



- Connect each vertex 

with its complements (conflict links)
(Forces us to choose one or other to be true)

$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$



Reduction: Given formula F in 3CNF:

$k \leftarrow$ number of clauses

for each clause $(x_a \vee x_b \vee x_c)$:

- create vertices x_a x_b x_c
- create edges $(x_a, x_b), (x_b, x_c), (x_c, x_a)$

for each variable x_i:

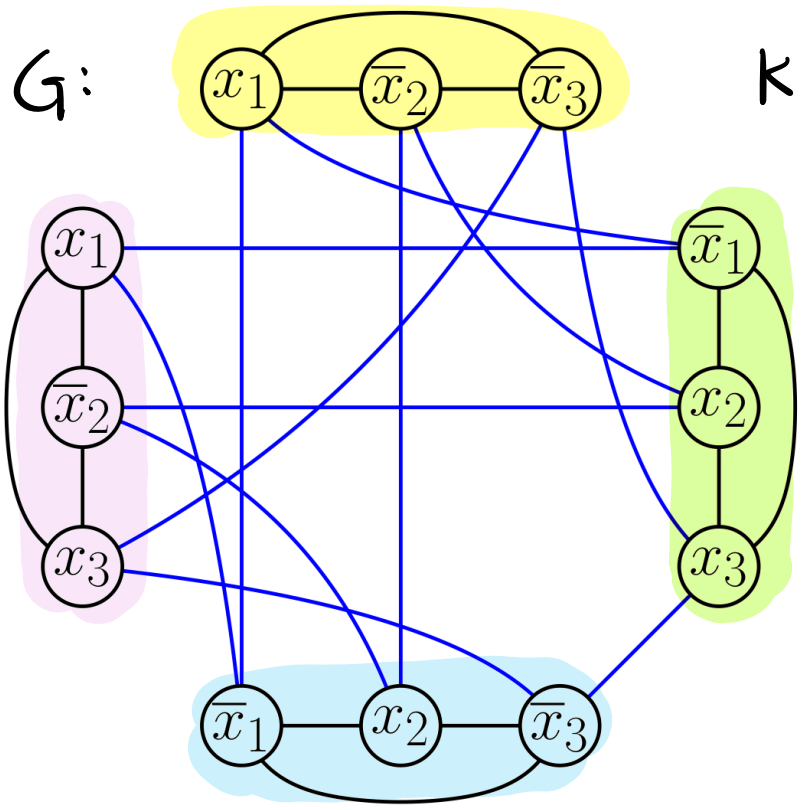
- create edges between all copies of x_i and $\bar{x}_i$

return (G, k)

Example: $F =$

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$

G:



$k = 4$

This can computed in poly-time in size of formula.

(Exercise)

Why does it work? Some intuition...

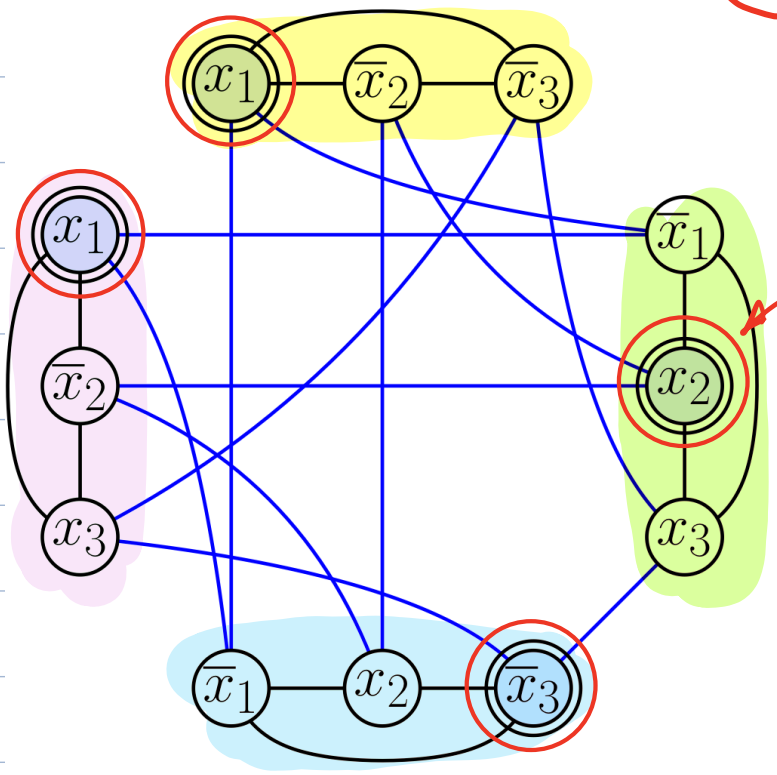
$\Rightarrow$ If F is satisfiable, there must be at least one true literal per clause $\longrightarrow$ add it to indep. set

$\Leftarrow$ If G has indep. set of size k, must be one vertex from each cluster. Set this literal true. This assignment satisfies F.

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$

T T T T

$x_1 = T$
$x_2 = T$
$x_3 = F$



There may be choices on which vertex to use

Circled vertices form an indep. set size $k = 4$

Formal proof:

Claim: F is satisfiable iff G has ind. set size k

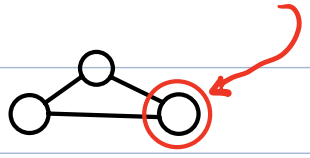
($\Rightarrow$) If F is satisfiable then (because F in CNF form) each clause of F must have at least one true literal.

- Select one true literal from each clause and add to V'
- There are k clauses $\Rightarrow |V'| = k$
- Both x_i and $\bar{x}_i$ cannot be true, so no edge between vertices of V'

$\Rightarrow$ V' is ind. set of size k ✓

($\Leftarrow$) If G has ind. set V' of size k

- Because clusters are connected, can take at most one from each cluster.



- Because k = no. of clauses = no. of clusters must take exactly one from each cluster.

- Because x_i — $\bar{x}_i$ are adjacent, both cannot be in V'.
  - if $x_i \in V' \rightarrow$ set $x_i \leftarrow T$
  - $\bar{x}_i \in V' \rightarrow$ set $x_i \leftarrow F$
  - if neither $\rightarrow$ set $x_i \leftarrow$ either T or F
- This assignment satisfies every clause of F

$\Rightarrow$ F has a satifying assignment ✓

□

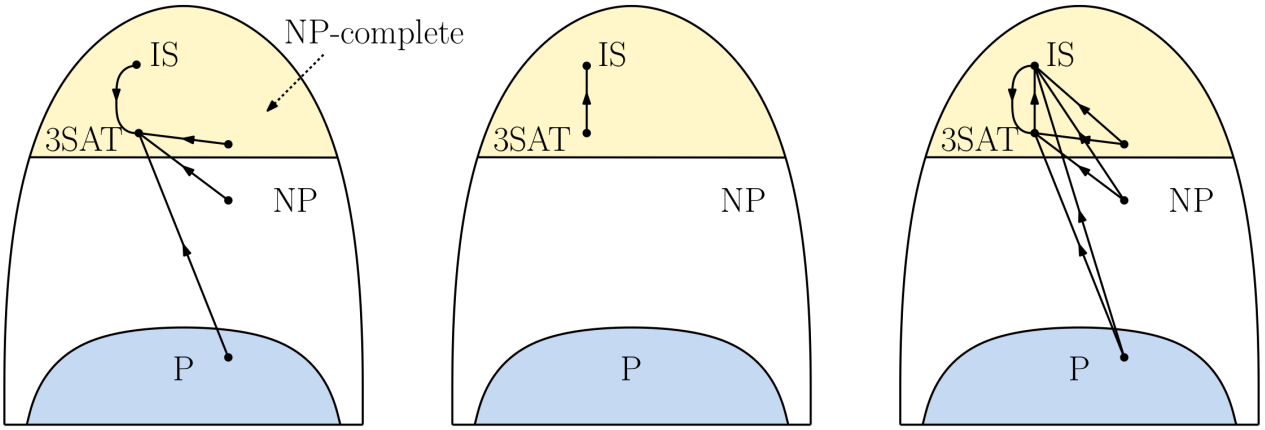
Done! Showed IS is NP-complete.

Notes:

- Hardest part is the reduction
- Reduce known NP-hard to our problem
- Reduction merely translates common elements.
- It is agnostic (doesn't care) which variables are true, which vertices are in ind. set or even if an ind. set exists!

Summary:

- 3SAT is NP-complete
- IS is NP-complete ($3SAT \leq_p IS$)



CMSC 451 - Algorithm Design
Lecture 17 - NP-Completeness: Clique, Vert. Cov. & Dom. Set

This lecture - 3 easy reductions

Recap: To show L is NP-complete:

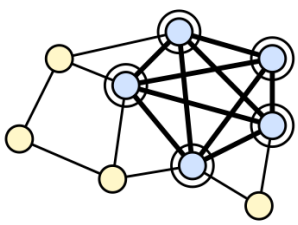
(i) $L \in NP$ - it is verifiable

(ii) L is NP-hard - suffices to show $L' \leq_p L$ for some known NP-hard problem L'

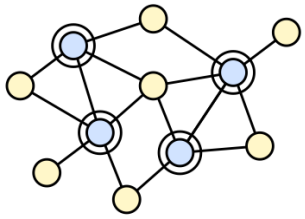
Three problems:

Clique (CLIQUE): Given a graph $G=(V,E)$ & integer k, is there a subset $V' \subseteq V$ of size k s.t. all pairs of vertices in V' are adjacent.

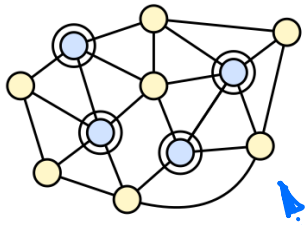
Clique of size 5



Vertex cover size 4



Dominating set size 4



Vertex Cover (VC): Given graph $G=(V,E)$ & integer k, is there a subset $V' \subseteq V$ of size k s.t. every edge of G is incident to some vert. of V'

Dominating Set (DS): Given graph $G=(V,E)$ + integer k, is there a subset $V' \subseteq V$ of size k s.t. every vertex in G is in V' or adjacent to a vertex in V'

Applications:

Clique - Analysis of tightly coupled groups in social networks

Vertex Cover + Dominating Set - Used in applications in facility location (placing service facilities close to clients)

Clique / IS / VC: Closely related.

Recall: $\bar{G}$ = complement graph $(u,v) \in \bar{G}$ iff $(u,v) \notin G$

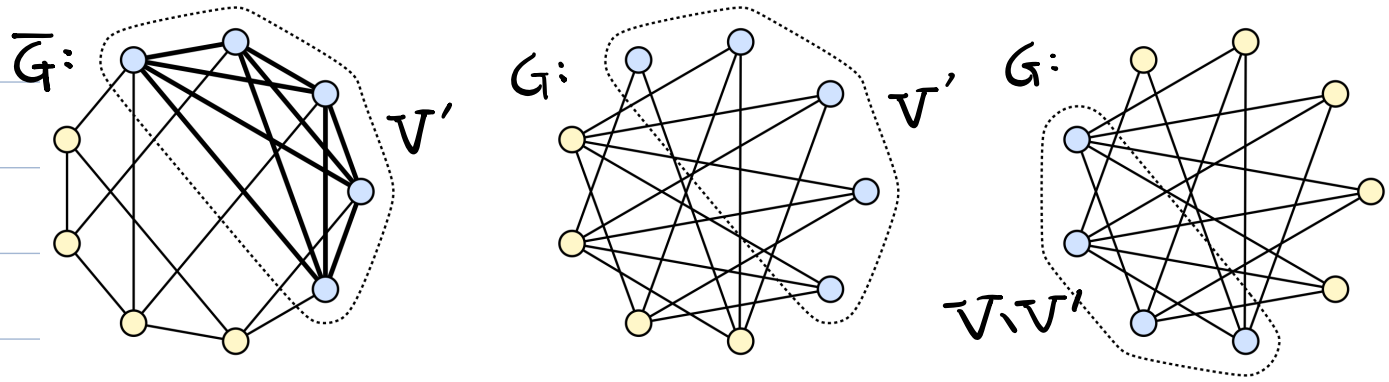
Lemma: Given a graph $G=(V,E)$ with $n=|V|$. Let $V' \subseteq V$ + $k=|V'|$. Then the following are equivalent:

(i) V' is a clique of size k in $\bar{G}$

(ii) V' is an indep. set of size k in G

(iii) $V \setminus V'$ is a vertex cover of size $n-k$ in G

V' is clique in $\bar{G}$ | V' is indep set in G | $V \setminus V'$ is VC in G



Proof: (sketch)

(i) $\Leftrightarrow$ (ii): All adjacent in $\bar{G}$

$\Leftrightarrow$ none adjacent in G

(ii) $\Leftrightarrow$ (iii): No edges among V'

$\Leftrightarrow$ all edges touch $V \setminus V'$

Theorem: CLIQUE is NP-complete

Proof: (i) CLIQUE $\in$ NP

Given input $(G=(V,E),k)$

certificate = list of k vertices $\rightarrow V'$

verification:

- for each $u, v \in V'$
  - check that $(u,v) \in E$
- if yes for all - accept
  - else - reject

(ii) $IS \leq_p$ CLIQUE

- Given instance (G, k) for IS
  our translation function outputs
  $(\bar{G}, k)$ for CLIQUE.
- Can complement graph in $O(n^2)$ time
- Correctness:
  G has indep. set of size k
  iff
  $\bar{G}$ has CLIQUE of size k
  (by prior lemma (i)$\Leftrightarrow$(ii)) □

Theorem: VC is NP-complete

Proof: (i) VC $\in$ NP

Given input $(G=(V,E), k)$
certificate = list of k vertices $\rightarrow V'$
verification:
- for each edge $(u,v) \in E$
  check that $u \in V'$ or $v \in V'$
- if yes for all - accept
  else - reject

(ii) $IS \leq_p VC$

- Given instance (G, k) for IS
  our translation function outputs
  $(G, n-k)$ for VC $(n = |V|)$
- Can compute in $O(n)$ time
- Correctness:
  G has indep. set of size k
  iff
  G has vertex cover of size n-k
  (by prior lemma (ii) $\Leftrightarrow$ (iii)) $\square$

Theorem: DS is NP-complete

- This is trickier!
  - Looks similar to VC (if V' is a vertex cover then it's a dom. set, right?)
  - But, translation must be faithful
    (if V' is a dom. set it may not be VC)

- Goal: Poly-time function $f: (G, k) \rightarrow (G', k')$ s.t.

G has VC of size k

iff

G' has DS of size k'

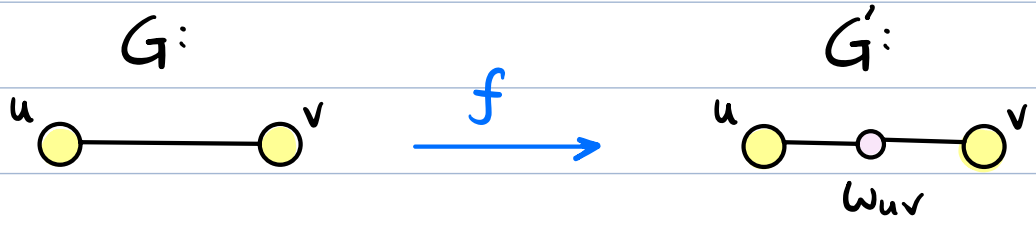
- What are corresponding elements?

VC: Every edge is incident to V'

DS: Every vertex is adjacent (or in) V'

⇒ Need to make edges behave like vertices

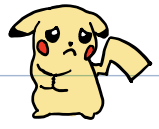
Idea 1: Add a vertex within each edge:



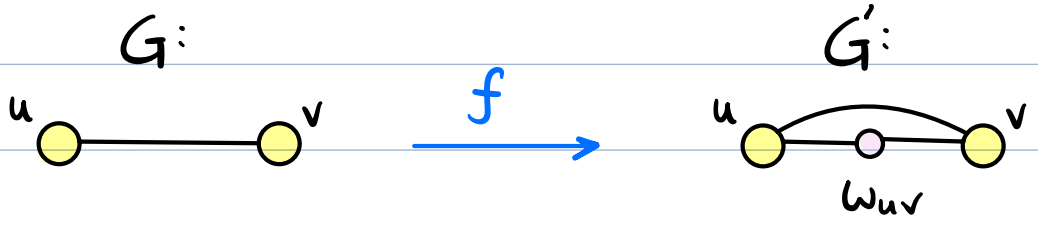
Putting u or v in VC covers (u, v)

Putting u or v in DS dominates w_{uv}

→ But if u in DS, who dominates v?



Idea 2: Add a vertex within each edge + keep the old edge.



Putting u or v in VC covers (u,v)

Putting u/v in DS dominates w_{uv} + v/u

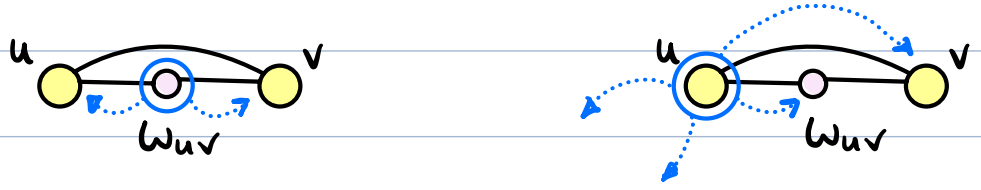
This is almost perfect. Two issues:

① Don't want DS to use the mid-edge vertices. How to enforce this?

② Who dominates isolated vertices? (no incident edges)

Fixes:

① Not a problem. No advantage in using w_{uv}, since u or v dominates at least as many vertices.



② Isolated vertices are trivial to handle:

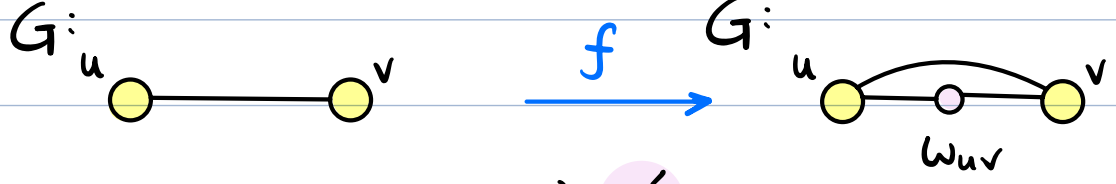
VC – They never help

DS – They must be included

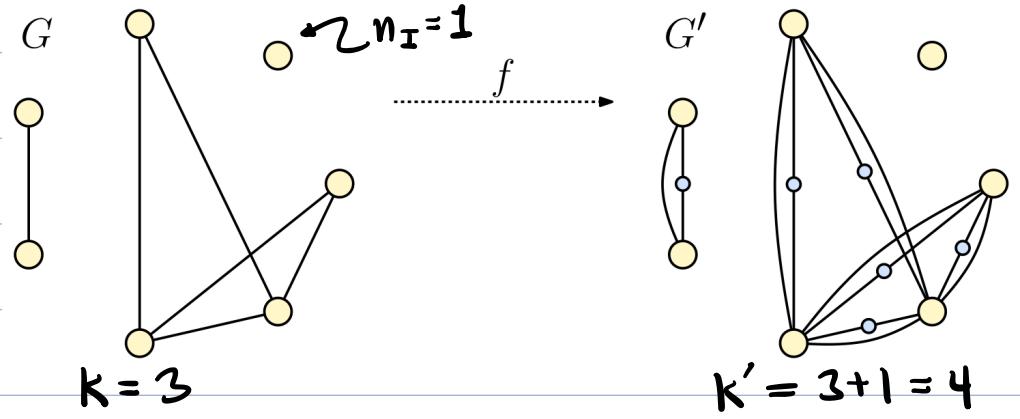
Let n_I denote no. of isolated vertices in G. Set $k' \leftarrow k + n_I$

Final reduction function f:

- Given (G, k) for VC, $G = (V, E)$
- for each edge $(u, v) \in E$
  - create vertex w_{uv}
  - add new edges (u, w_{uv}) + (v, w_{uv})



- Call resulting graph G'
- let n_I = no. isolated vertices in G
- set $k' \leftarrow k + n_I$
- return (G', k')



Lemma: G has a vertex cover of size k iff G' has a dom. set of size k'

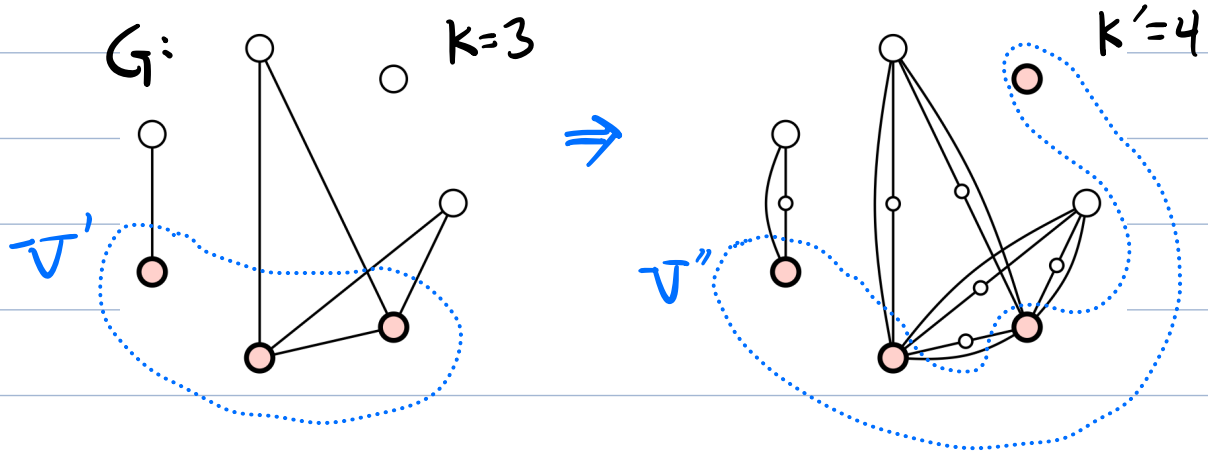
Proof:

($\Rightarrow$) Let V' be a VC in G of size k

- Let:

$$V'' = V' \cup \{\text{all isolated vertices}\}$$

- Clearly, $|V''| = k + n_I = k'$



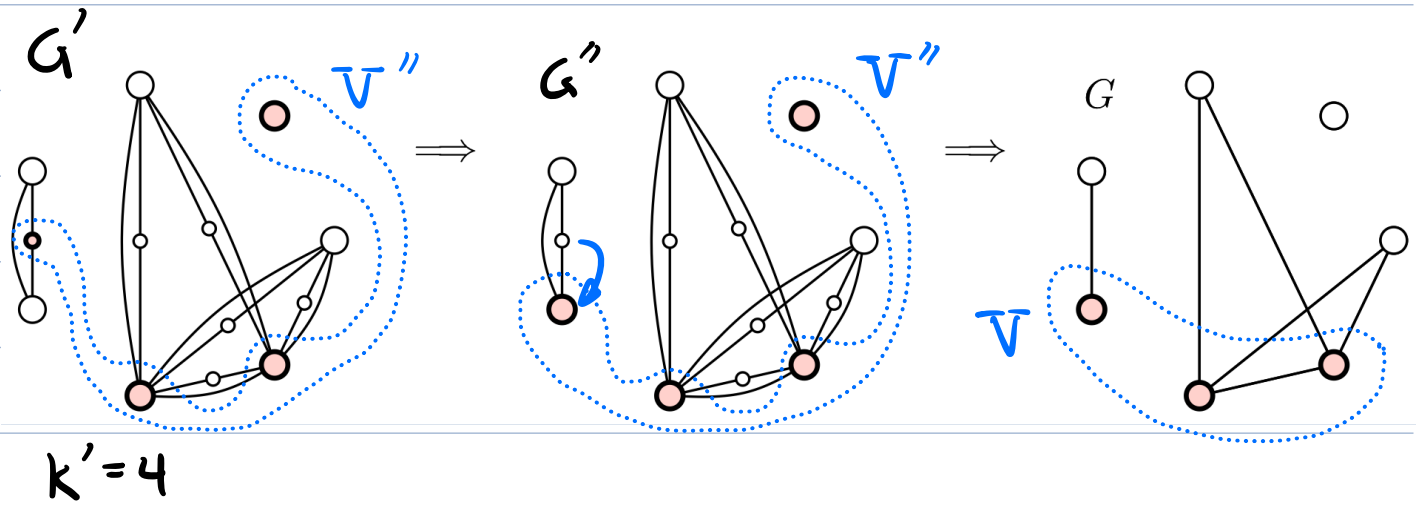
- Since V' is VC, $\forall (u,v) \in E$, either $u \in V'$ or $v \in V'$

- In either case we dominate u, v, w_{uv}

$\Rightarrow$ V'' is dom. set of size k' in G' ✓

(⇐) Let V'' be a D.S. in G' of size k'

- If V'' contains any mid-edge vertices w_{uv}, replace with either u or v.
- Still a dom. set (as observed earlier)



- Let $V \leftarrow V''$ with isolated vertices removed
  (Isolated vertices must be in dom. set)

- $|V| = k' - n_I = k$

- Since V'' dominated all vertices
  $\Rightarrow V''$ dominates all mid-edge vertices
  $\Rightarrow V$ covers all edges

$\Rightarrow V$ is a vertex cover in G of size k ✓

□

Theorem: DS is NP-complete

Proof: (i) $DS \in NP$

Given input $(G=(V,E),k)$

certificate = list of k vertices $\rightarrow V'$

verification:

- for each $v \in V$
  - check that v or some neigh. in V'
- if yes for all - accept
  - else - reject

(ii) $VC \leq_p DS$

Follows from previous lemma.

□

Summary: Three new NP-Complete problems

- CLIQUE
- Vertex cover (VC)
- Dominating Set (DS)

CMSC 451 - Algorithm Design
Lecture 18 - Approximation: VC, TSP, & Bin Packing

Coping with NP-Completeness:

What to do if your problem is NP-hard?

- Brute-force search: Only for small inputs
- Heuristics: (e.g. greedy) Fast but no guarantees (unless you prove them)
- General search: Often the best choice
  - Branch & bound
  - Metropolis-Hastings
  - Simulated annealing
  - Genetic / Evolutionary
  - Local search
  - ...
- Approximations:
  - Guaranteed performance
  - Specialized for particular problem

How close to optimal?

Approximation ratio: Ratio of heuristic value to optimal = ρ (Greek rho)

Common approx. results: (Best to worst)

- Can be approximated arbitrarily well
  - User gives an allowed error bound $\varepsilon > 0$ and approx. ratio is $\leq 1+\varepsilon$
  - Running time poly. in n but grows with ε (e.g. $\mathcal{O}((1/\varepsilon)^2 \cdot n^3) \to \infty$ as $\varepsilon \to 0$)
  - Called Polynomial Time Approximation Scheme (PTAS - pronounced "P-tas")
  - Examples: Subset sum, Euclidean TSP
- Constant approx. ratio
  - Examples:
    - k-center (Gonzalez): $\rho = 2$
    - Vertex cover: $\rho = 2$
    - TSP: $\rho = 2$ (or better $\rho = 1.5$)
    - Bin packing: $\rho = 2$
- Logarithmic:
  - Example:
    - Greedy set cover: $\rho = \ln n$
    - Essentially best possible (unless P = NP)

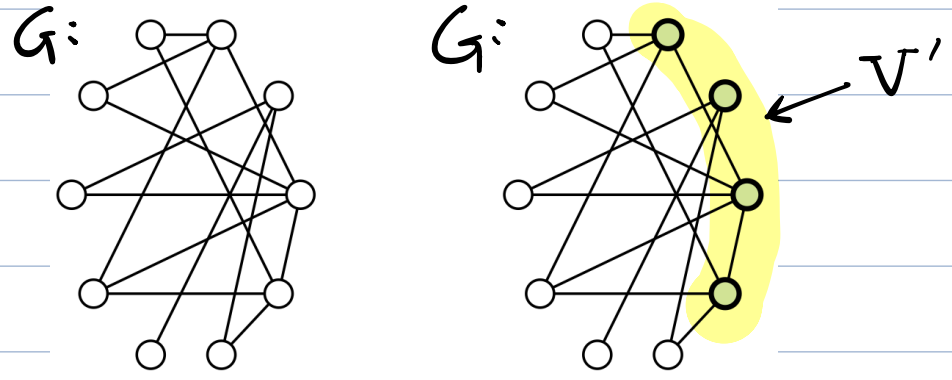
- Inapproximable: No poly. time algorithm achieves an approx. unless P = NP.

Vertex Cover Approximation:

- Present a simple approx. alg with $\rho = 2$

Vertex Cover (optimization)

Given a graph $G = (V, E)$ compute the smallest subset $V' \subseteq V$ s.t. every edge is incident to some vertex in V'.



First Attempt: Greedy Heuristic

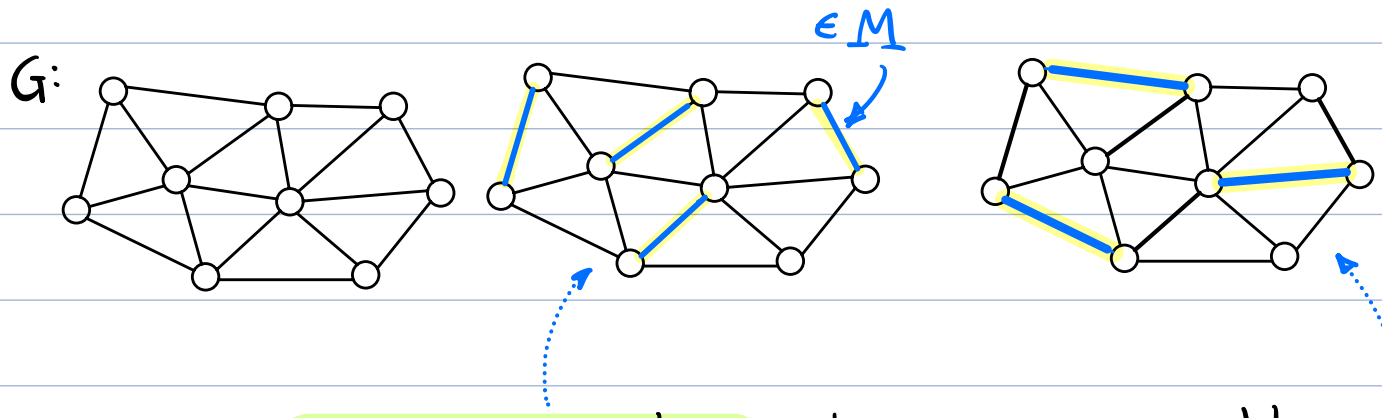
- Repeatedly select vertex u of highest degree + add to V'
- Remove u + all its incident edges
- Update vertex degrees

- Equivalent to greedy set cover on $(X, \mathcal{S})$
  - X = edge set = E
  - $\mathcal{S} = \forall u \in V, \mathcal{S}_u = \{e \mid e \text{ incident to } u\}$

Approx. ratio = $\ln m$ $\quad m = |E|$

Second attempt: Based on maximal matching

- Recall: A matching in G is a subset of edges $M \subseteq E$ s.t. each vertex incident to at most one edge of M



- Maximum matching: largest size possible
- Maximal matching: cannot add more edges

- Maximum matchings are nontrivial to compute (Edmond's blossom algorithm)
- Maximal matchings are very easy

Key observation: M = any maximal matching
V^* = optimal vertex cover

Claim:

(i) $|V^*| \geq |M|$

(ii) $|V^*| \leq 2 \cdot |M|$

Proof:

(i) For any $(u,v) \in M$, either u or v must be in V^* (BTW, This holds for any matching and any vertex cover.)

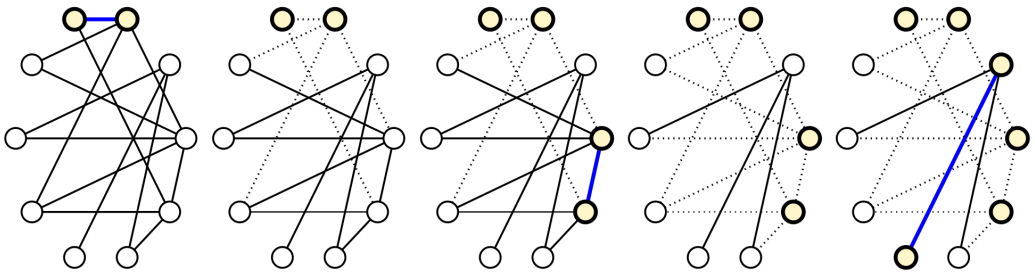
(ii) Since M is maximal, for every $(u,v) \in E$, either u or v is incident to an edge of M (otherwise, we could add (u,v) to M).

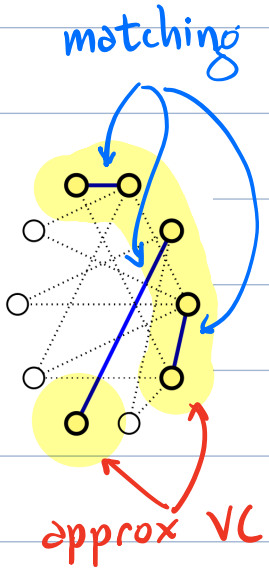
$\Rightarrow$ Taking all endpts of M yields a vertex cover, of size $2 \cdot |M|$. V^* can't be larger. □

Approach: Compute any maximal matching M
+ for each $(u,v) \in M$, add both u+v to V'.

```
approx-VC(G)
    V' ← ∅
    while (G has an edge, (u,v))   // take an edge
        V' ← V' ∪ {u,v}            // add it endpts
        delete u + v and incid. edges
```

Example:





Claim: approx-VC achieves approx. ratio of 2

Proof: $\bar{V}'$ is a vertex cover.

$|\bar{V}'| = 2 \cdot |M|$, where M is maximal match.

By Lemma(i) $|M| \leq |\bar{V}^*|$

$\Rightarrow |V'| = 2|M| \leq 2|V^*|$

Reductions + Approximations:

- Since VC has a 2-factor approximation
  don't all NP problems (via reduction)
  have a 2-factor approximation?

- No! Reductions don't necessarily preserve approx. ratios.

- Case study: $IS \leq_p VC$ (we showed:)

G has vertex cover of size k
iff
G has indep. set of size n-k

- Let k^* = opt. VC size.

$\Rightarrow$ We can compute a vert. cover of size $\leq 2k^*$

$\Rightarrow$ can compute an indep. set of size $\geq n-2k^*$

- Approx. ratio for IS:

$$\frac{\text{opt}}{\text{approx}} \leq \frac{n-k^*}{n-2k^*}$$ ← Could be huge!

E.g. $n = 1{,}001$, $k^* = 500$, $n-k^* = 501$

$n-2\cdot k^* = 1$

$$\frac{n-k^*}{n-2\cdot k^*} = \frac{501}{1} = 501 \text{ !}$$

Traveling Salesman Problem (Metric case)

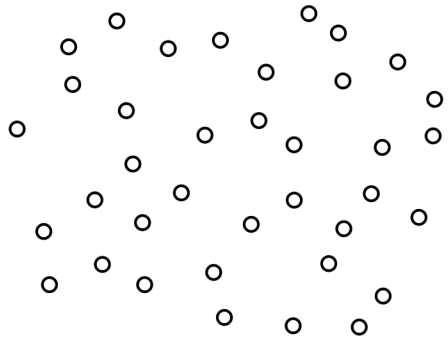
Recall: A (discrete) metric space is a finite point set P + distance function $d(u,v)$ $\forall\ u,v \in P$, where:

- $d(u,v) \geq 0$ + $d(u,v) = 0$ iff $u = v$
- $d(u,v) = d(v,u)$
- $d(u,w) \leq d(u,v) + d(v,w)$ (Triangle inequality)

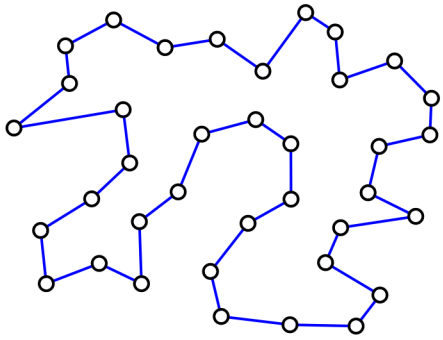
- Most natural dist. functions are metrics:
  - Euclidean dist
  - Shortest path dist. in graphs
  - Edit dist. (Levenshtein dist.)
- Representable as:
  - $n \times n$ distance matrix (symmetric)
  - complete weighted graph

(Metric) Traveling Salesman Problem (TSP):
Given a set P of n points in a metric space, compute a simple cycle that visits all vertices of minimum weight.

Example: P:



TSP(P):



Given a set T of edges, let wt(T) = sum of distances.

Obs: Given any TSP tour, remove an edge. A spanning tree remains ⇒

$$wt(MST(P)) \leq wt(TSP(P))$$

Idea: If we can compute a valid tour H of weight $\leq c \cdot wt(MST(P))$ then:

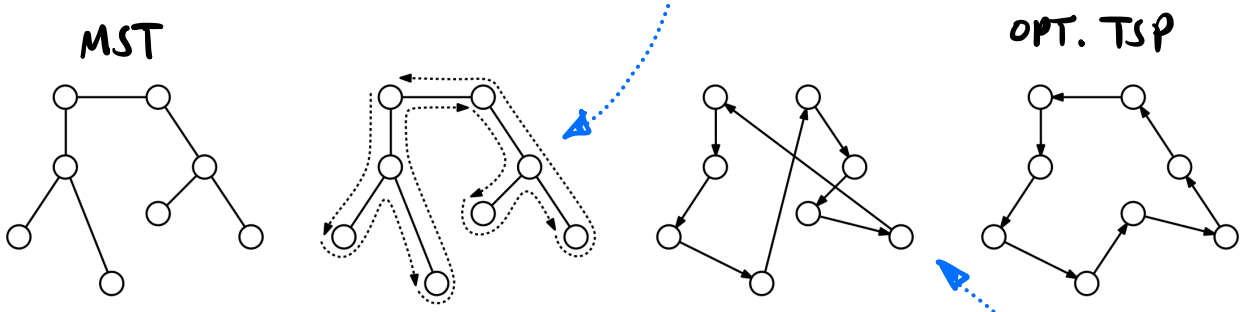
$$wt(H) \leq c \cdot wt(MST(P)) \leq c \cdot wt(TSP(P))$$

⇒ H is a factor-c approximation!

How to convert MST into a cycle?

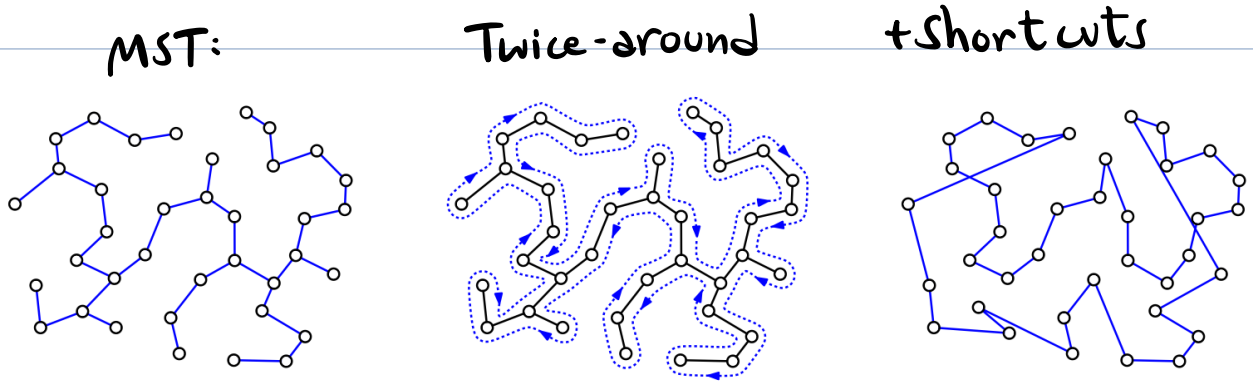
The Twice-Around Tour:

- Start at any vertex of MST
- Hold your hand against the MST "wall" walk around the MST



- This cycle hits each edge twice, but repeats points, not allowed.
- Shortcut the path by removing repeats.
- By triangle inequality - shortcutting can only reduce lengths.

Example:



Since graph is complete, can compute MST in $O(n^2 \log n)$ time. Twice-around + shortcuts add $O(n)$ time.

Lemma: The Twice-Around alg. has an approx. ratio of 2.

Proof: Let H = twice-around tour +
let H^* = opt TSP tour +
let T = MST edges

Earlier:

$$wt(T) \leq wt(H^*)$$

Since "pure" twice around hits each edge twice + shortcutting only decreases weights:

$$wt(H) \leq 2 \cdot wt(T)$$

$\Rightarrow$

$$wt(H) \leq 2 \cdot wt(T)$$
$$\leq 2 \cdot wt(H^*) \Rightarrow \frac{wt(H)}{wt(H^*)} \leq 2$$

□

Christofides Algorithm:

- A $\frac{3}{2}$-factor approx to metric TSP
- By Nicos Christofides (1976)

Obs: Wastage in twice-around comes from visiting each MST edge twice.

- Can we avoid this repetition?

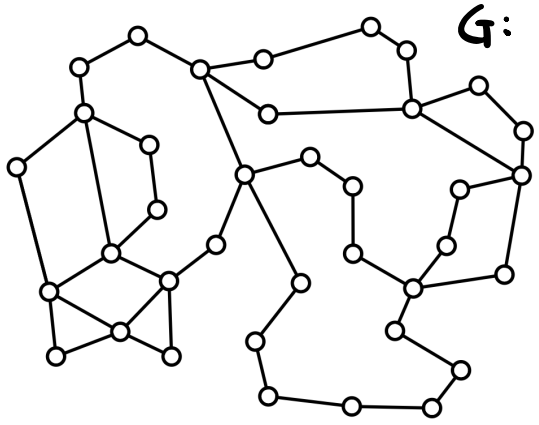
Eulerian Circuit: A cycle in a graph that visits every edge exactly once

- The following is well known:

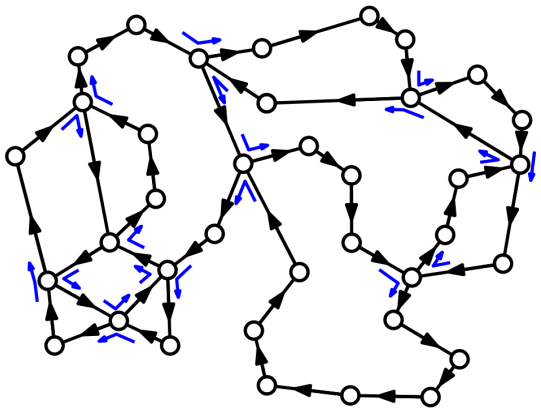
> **Lemma:** A graph $G = (V, E)$ has an Eulerian circuit iff every vertex has even degree. Can be computed in $O(n+m)$ time.

Example:

G:



Eulerian Circuit

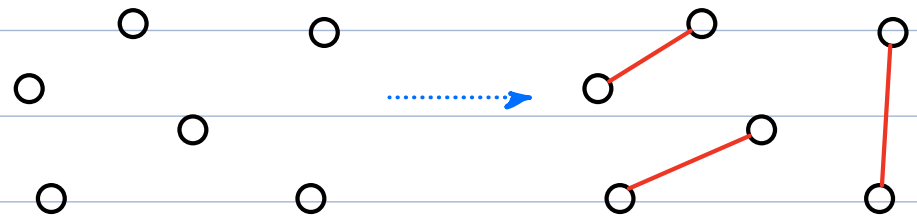


How to exploit this?

- MST definitely has vertices of odd degree (leaves have degree 1)

Christofides idea:

- Extract all vertices of MST of odd degree
- Fact: Num. of vertices of odd degree is even (Handshaking lemma)
- Compute a matching on these vertices



- Add these edges to MST
  ⇒ now all vertices have even degree!
- Use Eulian circuit rather than twice around

Background:

Matching: Set of edges, each vertex incident to ≤ 1

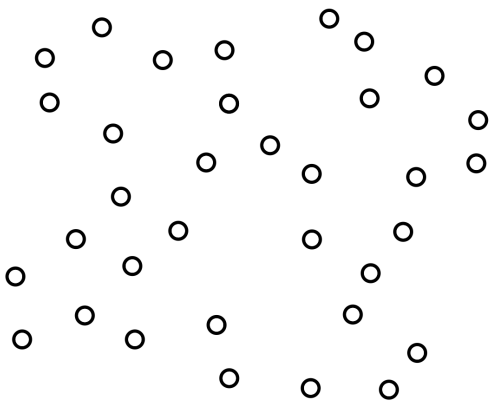
Perfect mething: Every vertex incident to 1 edge

Min-weight perfect matching (MWM(P))

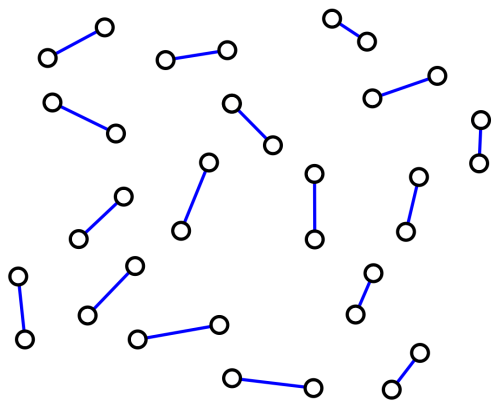
- perfect matching of min weight
- Computable in $\mathcal{O}(|V|^2 \cdot |E|)$

Assume $|P|$ even

Example: P:



MWM(P)



Christofides Algorithm: Given P in metric space.

(a) Compute edges T of $MST(P)$

(b) Let $P_1 \leftarrow$ pts of P with odd deg. in $MST(P)$

(c) Compute edges M of $MWM(P_1)$

(d) Form graph $G = (P, T \cup M)$ [G has even deg.]

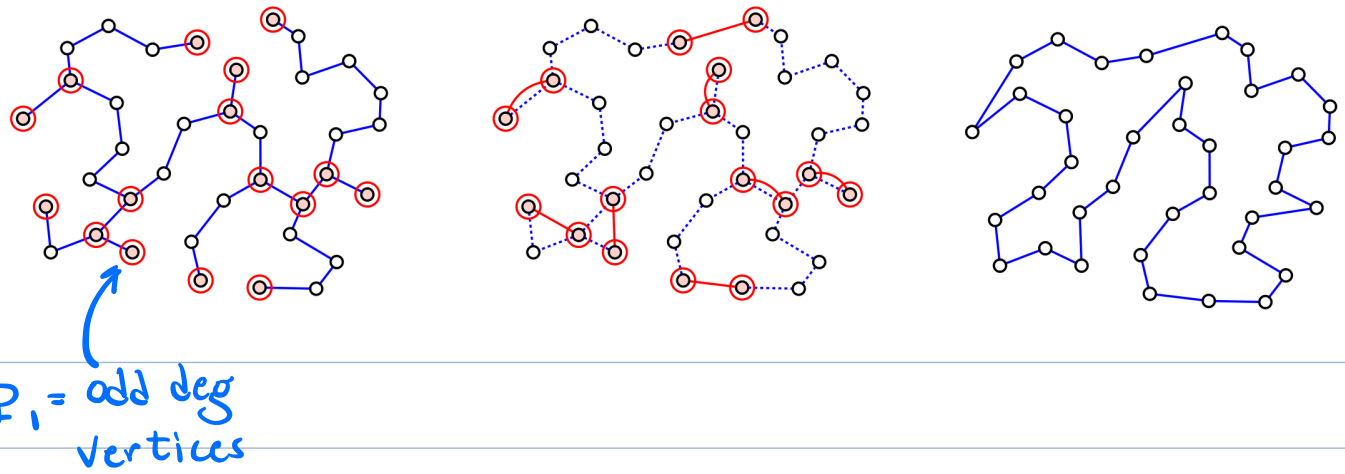
(e) $C' \leftarrow$ Eulerian circuit in G

$C \leftarrow$ apply short-cutting to C' [$wt(C) \leq wt(C')$]

(f) return C

Example:

MST(P) | MWM(P_1) | Eulerian circuit + shortcuts



P_1 = odd deg vertices

- Using an Eulerian circuit is more cost efficient, but how much weight does MWM(P_1) add?

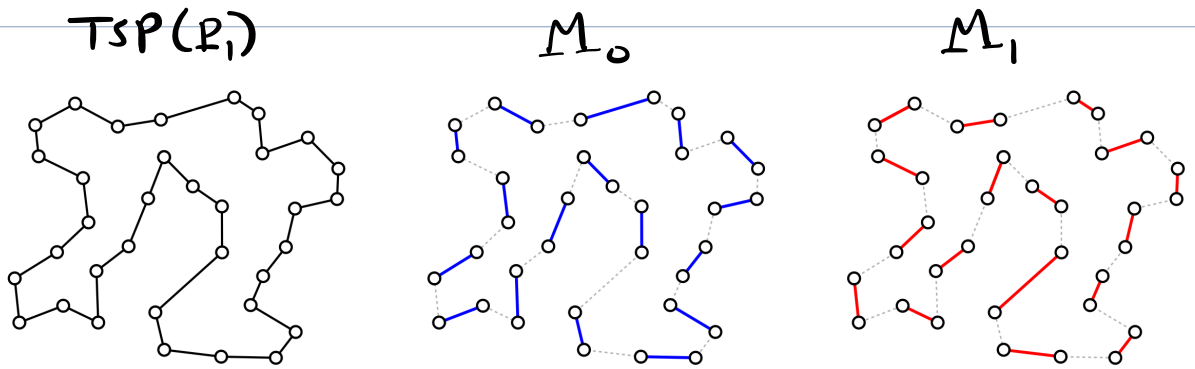
Lemma: Given point set P, let P_1 be any subset of even cardinality. Then:

$$wt(MWM(P_1)) \leq \frac{wt(TSP(P))}{2}$$

Proof: Removing pts. only decreases TSP:

$$wt(TSP(P_1)) \leq wt(TSP(P))$$

- Split $TSP(P_1)$ into alternating groups M_0 & M_1



$\Rightarrow wt(TSP(P_1)) = wt(M_0) + wt(M_1)$

- Both M_0 + M_1 are perfect matchings (but not nec. minimum weight)

$\Rightarrow wt(MWM(P_1)) \leq \min(wt(M_0), wt(M_1))$

- Min of two numbers $\leq$ average

$\Rightarrow \min(wt(M_0), wt(M_1)) \leq \frac{wt(M_0) + wt(M_1)}{2}$

- Combining:

$$wt(MWM(P_1)) \leq \min(wt(M_0), wt(M_1)) \leq \frac{wt(M_0) + wt(M_1)}{2}$$

$$= wt(TSP(P_1))/2 \leq \frac{wt(TSP(P))}{2}$$

□

Lemma: Christofides alg. has approx ratio $3/2$

Proof: Let: H^* = opt TSP tour on P

$T = MST(P)$

$M = MWM(P_1)$

As shown earlier: $wt(T) \leq wt(H^*)$

By previous lemma:

$$wt(M) \leq \frac{1}{2} wt(H^*)$$

Let C' be Euler circuit in $T \cup M$

* C = shortcut(C')

We have:

$$wt(C) \leq wt(C') = wt(T) + wt(M)$$
$$\leq wt(H^*) + \frac{1}{2} wt(H^*)$$
$$\leq \frac{3}{2} wt(H^*)$$

$$\Rightarrow \frac{wt(C)}{wt(H^*)} \leq 3/2$$

□

Summary –

- Approximation ratios
- Vertex cover ($\rho = 2$)
- Metric TSP
  - Twice-around ($\rho = 2$)
  - Christofides ($\rho = 3/2$)

CMSC 451 - Algorithm Design
Lecture 19 - Subset Sum Approximation

Subset Sum (SS):

- Given a set of positive integers $S = \{x_1, \ldots, x_n\}$ + target value t, what is largest sum $\leq t$?
- Objective:

$$\max_{S' \subseteq S} \left\{ \sum_{x \in S'} x \;:\; \sum_{x \in S'} x \leq t \right\}$$

Example: $S = \{3, 6, 9, 12, 15, 23, 32\}$ $t = 34$

Answer: $S' = \{6, 12, 15\}$ $6+12+15 = \underline{33} \leq t$ (can't make 34)

Application:

- A simplified version of 0-1 knapsack and various packing problems.
- SS is NP-complete (won't prove this) (see optional material at end)

This lecture:

Approximation algorithm

PTAS - ε-error ← User specifies ε

- Returns answer within $(1-\varepsilon) \cdot opt$
- Running time - Roughly $\tilde{O}(n^2/\varepsilon)$

Polynomial-Time Approximation to Subset Sum

- We'll present a poly-time algorithm for SS which given ε, where $0 < \varepsilon < 1$, returns a subset $S' \subseteq S$ whose sum z, satisfies

$$(1-\varepsilon)\, z^* \leq z \leq z^* \leq t$$

where z^* is the optimum sum.

Exact (Exponential time) Algorithm

- For $0 \leq i \leq n$, L_i = set of all possible sums of $\{x_1, \ldots, x_i\}$

E.g. $S = \{1, 4, 6, \ldots\}$

$L_0 = \langle 0 \rangle$

$x_1 = 1 \Rightarrow \quad L_1 = \langle 0, 1 \rangle$

$x_2 = 4 \Rightarrow \quad L_2 = \langle 0, 1, 4, 5 \rangle$

$x_3 = 6 \Rightarrow \quad L_3 = \langle 0, 1, 4, 5, 6, 7, 10, 11 \rangle$

- Observe: $|L_i| \leq 2^i$

- Final answer: Largest in L_n that is $\leq t$.

$$z^* = \max \{ z \in L_n \mid z \leq t \}$$

- Approach:
  - Compute L_i for $i = 0, 1, \ldots, n$

    Define: $L + x$ = add x to all $z \in L$

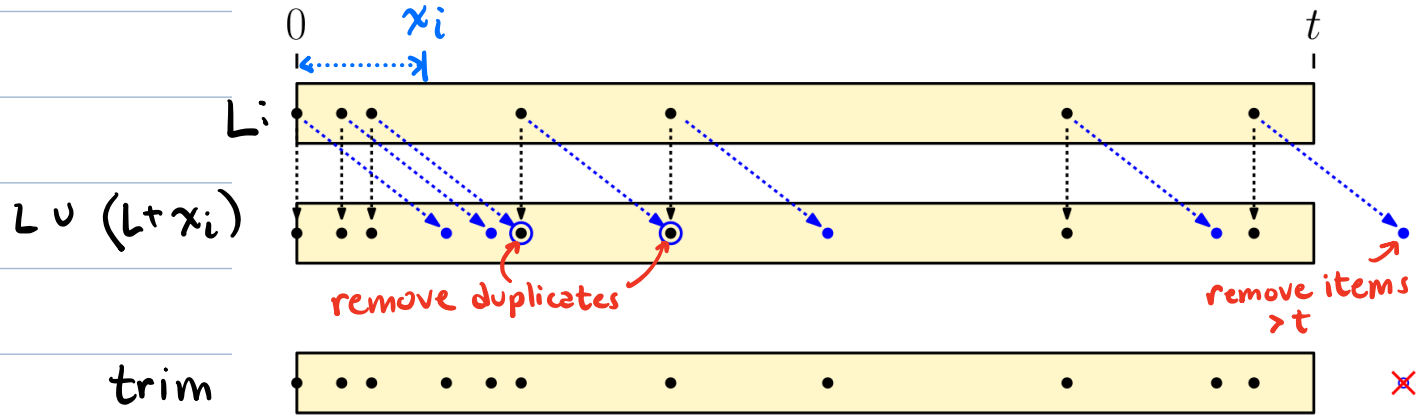
    E.g. $L = \langle 0, 3, 5 \rangle \rightarrow L + 4 = \langle 4, 7, 9 \rangle$

    $L_i \leftarrow L_{i-1} \cup (L_{i-1} + x_i)$ } merge$(L, L + x_i)$

  - Remove duplicates
  - Remove items $> t$

    } trim(L, t)

- Pictorially:

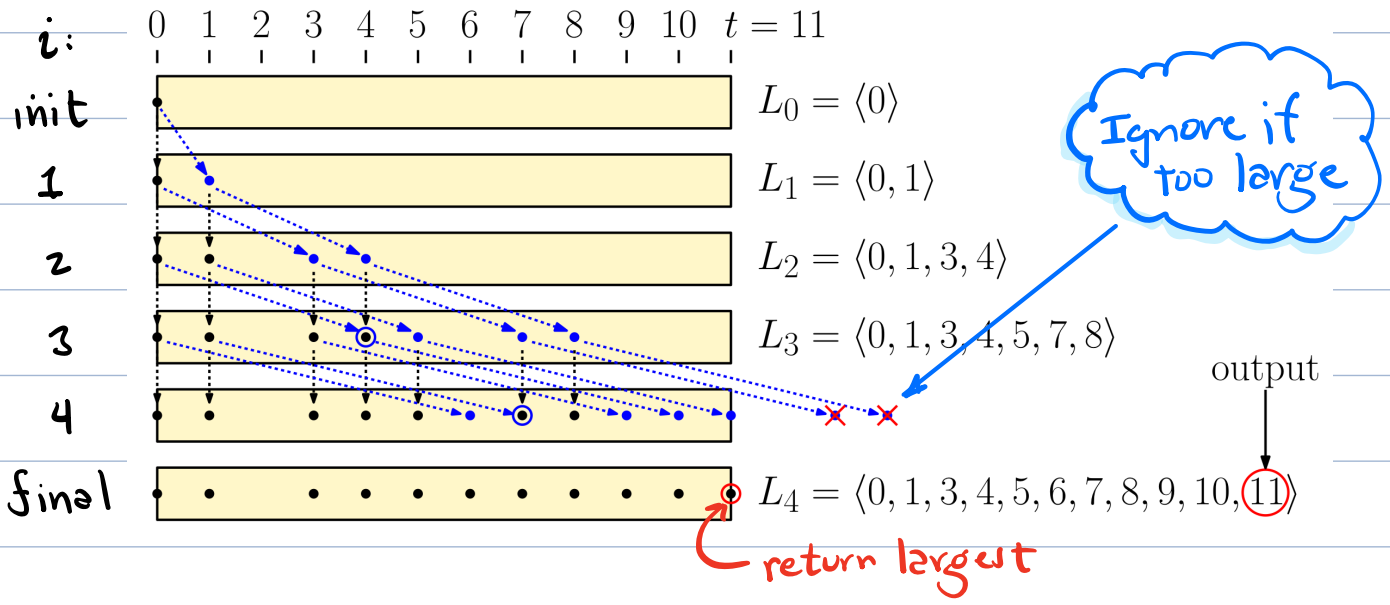



```
exact-subset-sum (x[1..n], t)      // exact subset sum
    L ← ⟨0⟩                        // basis
    for (i ← 1 to n)
        L ← merge(L, L + x[i])     // L ← L ∪ (L + x_i)
        L ← trim(L, t)             // rem. dups & > t
    return largest in L
```

Correctness – Easy

Running Time = $O(2^n)$ exponential!

Example:

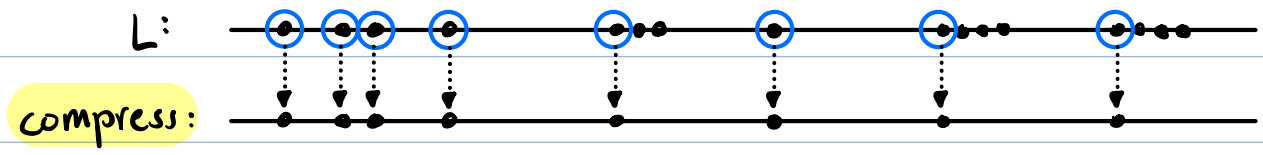


Approximation Algorithm

- Too many distinct sums (potentially 2^n)
- Idea: compress nearly equal sums

$L = \langle \ldots, 9851, 9852, 9860, 9950, 9952, 10{,}301, \ldots \rangle$

compress to 9851 compress to 9950



How large a difference can we tolerate?

prev = previous unpruned value

y = element under consideration

Cannot prune y if relative difference $> \delta$:

$$\frac{y - prev}{y} > \delta \iff y > \frac{prev}{1-\delta}$$

```
compress(L, δ, t)   // compress sorted list L
    L' ← ∅
    prev ← 0
    for each y ∈ L
        if (y > t) break          // ignore values > t
        if (y > prev/(1-δ))       // different enough?
            append y to L'        // add to L
            prev ← y
    return L'                     // final compressed list
```

Example: $L = \langle 10, 11, 12, 15, 20, 21, 22, 51, 53, 54, 56, 80 \rangle$

$\delta = 0.1 \quad t = 60$

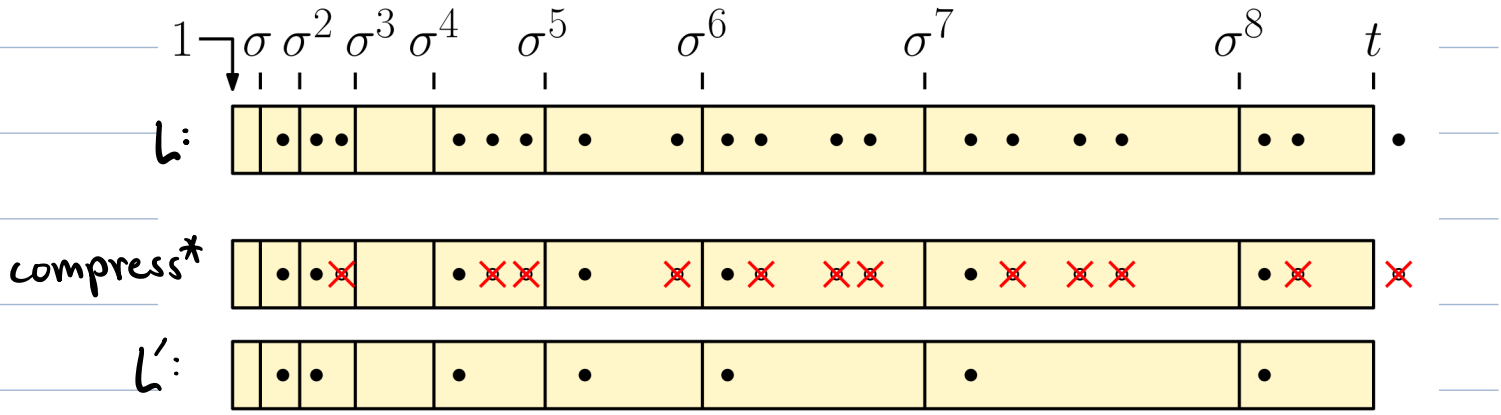
$L' = \langle 10, 12, 15, 20, 51 \rangle$ (>t)

return $\langle 10, 12, 15, 20, 51 \rangle$

"Bucket" Perspective:

- A bit easier to visualize in terms of buckets of same relative size
- Let $\sigma = 1/(1-\delta)$ $(\sigma > 1)$
- Breakpoints: $1, \sigma, \sigma^2, \sigma^3, \ldots, \sigma^i, \ldots$
- Bucket j: $[\sigma^{j-1}, \sigma^j]$

compress*(L, δ, t) - keep smallest in each bucket up to t



- Not exactly same as compress, but same properties. If y, y' in same bucket then $\sigma^{i-1} \le y < y' \le \sigma^i$

$$\frac{y' - y}{y'} = 1 - \frac{y}{y'} \le 1 - \frac{\sigma^{i-1}}{\sigma^i} = 1 - \frac{1}{\sigma} = \delta$$

$\sigma = 1/(1-\delta)$

$\Rightarrow 1 - \frac{1}{\sigma} = \delta$

$\Rightarrow$ in compress, y would prune y'

Value of compression – Number of surviving values is polynomial, not exponential!

Lemma: After compress(L, δ, t) where $\delta = \varepsilon/n$, number of nonzero items in L is $O((n \log t)/\varepsilon)$

Proof: Each consecutive pair of survivors is separated by factor of $\geq \sigma$, where $\sigma = 1/1-\delta$

- Let k = number of nonzero survivors
- $L_{final} = \langle 0, y_1, y_2, \ldots, y_k \rangle$ s.t. $y_i \geq \sigma \cdot y_{i-1}$

- $y_1 \geq 1 \Rightarrow y_i \geq \sigma^{i-1} \cdot y_1 \geq \sigma^{i-1}$

$\Rightarrow y_k \geq \sigma^{k-1}$ but $y_k \leq t$

$\Rightarrow \sigma^{k-1} \leq t$

$\Rightarrow (k-1) \ln \sigma \leq \ln t$

$\Rightarrow k \leq 1 + \ln t / \ln \sigma$

$= 1 + \ln t / \ln 1/(1-\delta)$

$= 1 + \ln t / (-\ln(1-\delta))$

Standard inequality: $-\ln(1-x) \geq x$

$\Rightarrow k \leq 1 + (\ln t)/\delta$

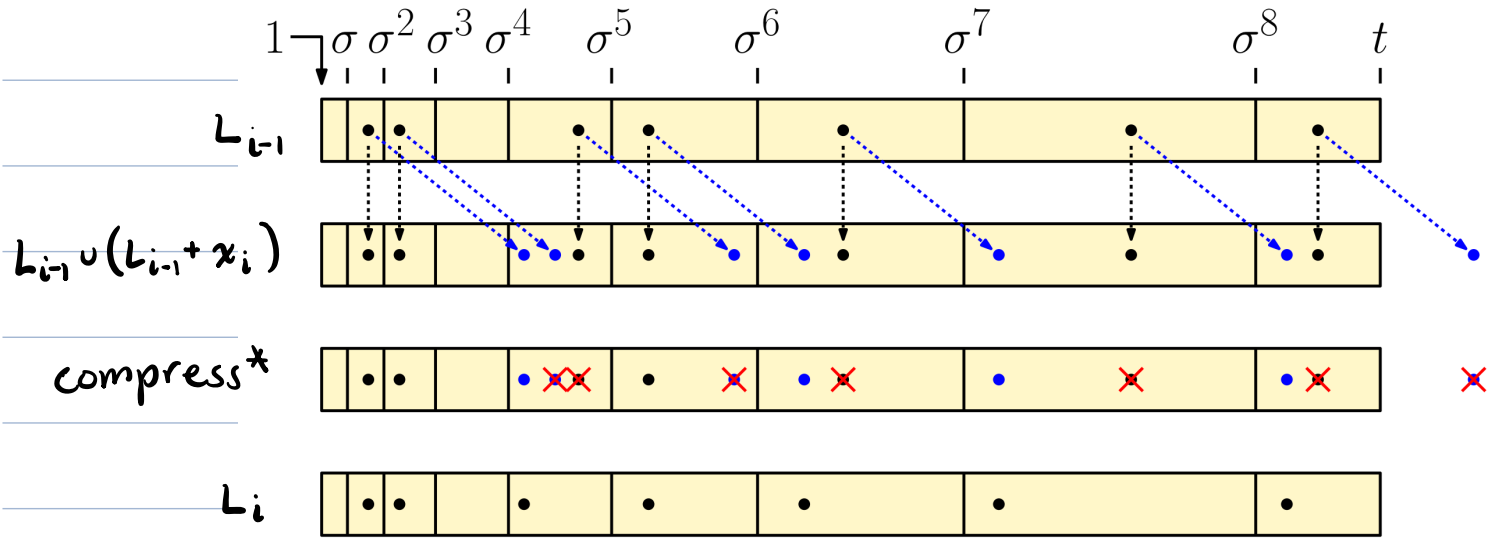
$\delta = \varepsilon/n \Rightarrow k \leq 1 + (n \cdot \ln t)/\varepsilon = O\left(\frac{n \cdot \log t}{\varepsilon}\right)$ □

Assuming $x_i \le t$, each number needs $O(\log t)$ bits

- Total input size = $O(n \log t)$
- List size $O((n \log t)/\varepsilon) = \frac{1}{\varepsilon} \cdot O(\text{input size})$
- Polynomial!

```
approx-subset-sum(x[1..n], t, ε)
    δ ← ε/n                          // pruning resolution
    L ← ⟨0⟩                          // basis case
    for (i ← 1 to n)
        L ← merge(L, L + x[i])       // add in x_i
        L ← compress(L, δ, t)        // + compress
    return largest item in L         // largest survivor
```

Example: (using compress* rather than compress)



Approximation Analysis:

Let Y^* = optimum subset sum value

Y_ε = algorithm output

→ Although we only output sum, there is a subset summing to Y_ε.

we know: $Y_\varepsilon \leq Y^* \leq t$

want to show $Y_\varepsilon \geq (1-\varepsilon) Y^*$

Intuition:

- Algorithm has n stages
- We allowed an error of $\delta = \varepsilon/n$ per stage
- Total error $\leq n \cdot \delta = \varepsilon$ ☺
- But these are multiplicative errors not additive errors ☹
- We'll show the intuition still works

Define:

L_i = algorithm's list after i^{th} iteration

L_i^* = exact list after i^{th} iteration (no compression)

Claim that for $y \in L_i^*$ there is "close" $z \in L_i$

Lemma: For $0 \le i \le n$, for all $y \in L_i^*$, there exists $z \in L_i$, s.t.

$$(1-\delta)^i y \le z \le y,$$

where $\delta = \varepsilon/n$.

z called y's representative

Proof: We'll just prove lower bound:

$$(1-\delta)^i y \le z$$

(Upper bound is an easy extension)

By induction on i. Basis: $L_0 = L_0^* = \langle 0 \rangle$ ✓

Let's assume lemma holds for $i-1$ & we'll show i

Every element of L_i^* arises in two ways:

$y \in L_{i-1}^*$

or $y + x_i$, where $y \in L_{i-1}^*$

We'll show both y & $y + x_i$ have representatives

By ind. hypothesis, $\exists\, z \in L_{i-1}$ s.t.

$$(1-\delta)^{i-1} y \le z \quad (a)$$

By adding x_i to both, we have

$$(1-\delta)^{i-1} y + x_i \leq z + x_i$$

$$\Rightarrow (1-\delta)^{i-1}(y+x_i) \leq z + x_i \quad (b) \quad (\text{since } 1-\delta < 1)$$

By compression, neither z nor $z+x_i$ in L_i^*

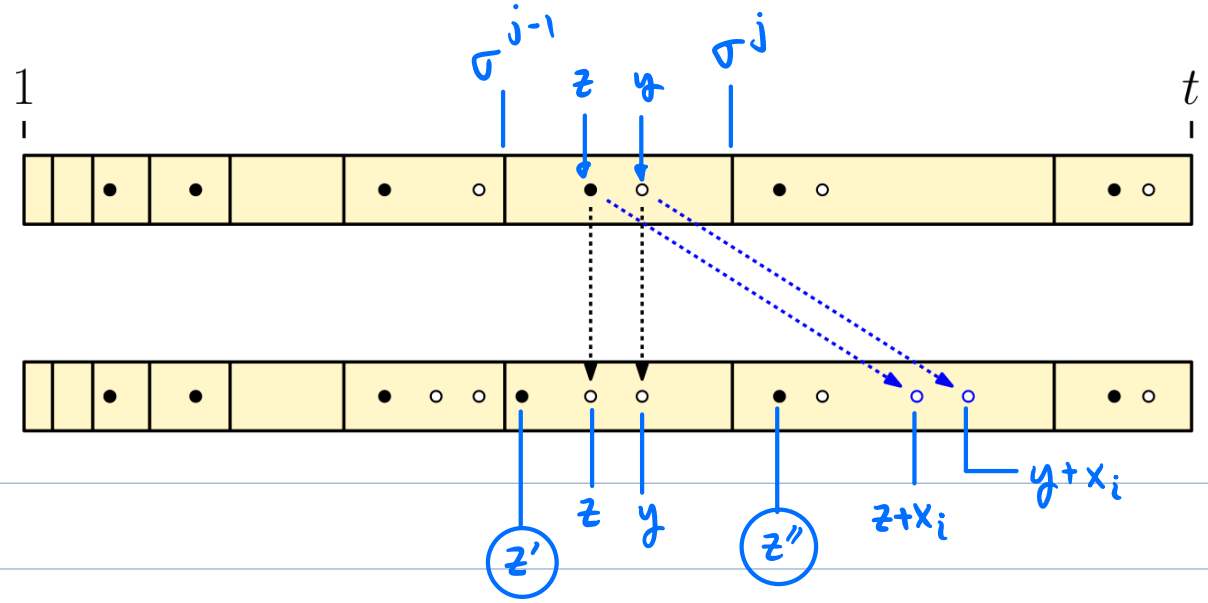
Let $z' + z''$ be elements that pruned z & $z+x_i$

By rules of compression: (c)

$$(1-\delta) z \leq z'$$

$$(1-\delta)(z+x_i) \leq z''$$

Illustration in compress*:



Combining (a), (b), (c):

$$(1-\delta)^{i-1}(1-\delta)y \le z' \Rightarrow (1-\delta)^{i} y \le z'$$

$$(1-\delta)^{i-1}(1-\delta)(y+x_i) \le z'' \Rightarrow (1-\delta)^{i}(y+x_i) \le z'$$

Since $z' + z'' \in L_i$, this completes proof □

Applying lemma with: $i = n$, $y = Y^*$ (opt s.s)
$z = Y$ (approx s.s.)

we have:

$$\left(1-\frac{\varepsilon}{n}\right)^n Y^* \le Y \le Y^*$$

Standard lemma: $\forall\, n > 0 \;\&\; \forall\, a \in \mathbb{R}$

$$(1+a) \le \left(1+\frac{a}{n}\right)^n$$

Set $a = -\varepsilon$

$$(1-\varepsilon) \le \left(1-\frac{\varepsilon}{n}\right)^n$$

$$\Rightarrow \quad (1-\varepsilon) Y^* \le Y \le Y^*$$

$\Rightarrow$ approx-subset-sum yields an ε-approximation ✓

Summary:

- Subset sum problem - SS
- SS is NP-complete (see below)
- PTAS for SS
  - User parameter ε
  - We return a sum Y s.t.
    $(1-\varepsilon)Y^* \leq Y \leq Y^*$
    where Y^* is optimum
  - Running time:
    - n phases
    - Each updates list of size
      $O((n \log t)/\varepsilon)$
    $\Rightarrow O((n^2 \log t)/\varepsilon)$

Weakly polynomial for fixed ε

SS is NP-Complete:

OPTIONAL (Not on quiz or final)

(i) SS ∈ NP (easy)

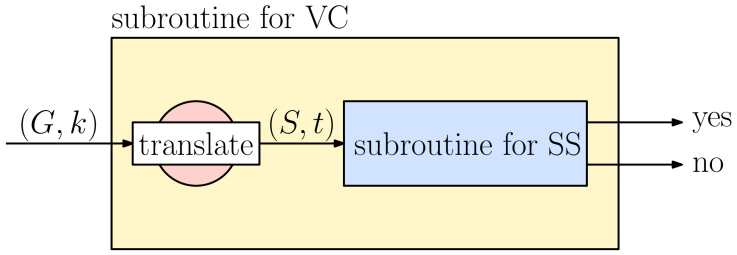
- certificate: indices of elements in S'
- verification: add up and check = t

(ii) SS is NP-hard

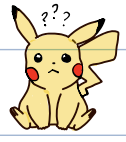
- We'll show VC $\leq_p$ SS
- Give poly-time function f:

$f(G, k) \longrightarrow (S, t)$, s.t.

G has vc of size k iff S has subset sums to t



Intuition:



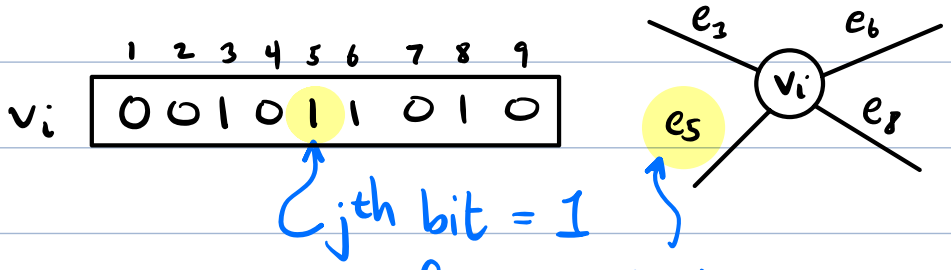
- Why is computing sums like building a cover?
- Numbers
  - ≡ bit strings
  - ≡ sets (bit vector)
- Addition
  - ≈ boolean-or
  - ≈ set union

First Attempt: Given graph $G = (V, E)$ and k, does G have a vertex cover of size k?

- Let: $n = |V|$ $m = |E|$

  $V = \{v_1, \ldots, v_n\}$ $E = \{e_1, \ldots, e_m\}$

- For each $v_i \in V$ create bit vector:

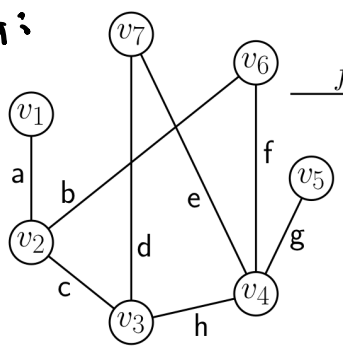


j^{th} bit = 1 if e_j incident to v_i

Example:

Vertex cover $\{v_2, v_3, v_4\}$

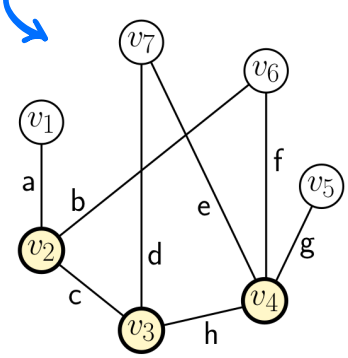
G:



f

	a	b	c	d	e	f	g	h
v_1	1	0	0	0	0	0	0	0
v_2	1	1	1	0	0	0	0	0
v_3	0	0	1	1	0	0	0	1
v_4	0	0	0	0	1	1	1	1
v_5	0	0	0	0	0	0	1	0
v_6	0	1	0	0	0	1	0	0
v_7	0	0	0	1	1	0	0	0
$t =$	1	1	1	1	1	1	1	1

$= v_2 \vee v_3 \vee v_4$



K=3

Every edge must be covered

⇒ logical-or has 1 in every position

Observe:

G has vertex cover of size k

iff

There are k bit vectors whose "or" is 1111...1

Not what we want:

① logical-or ≠ sum

② subset sum has no limit on number of items in sum

Second Attempt: (Correct)

- Addition has carries

$$\begin{array}{r} 1101 \\ \vee\ 0011 \\ \hline 1111 \end{array} \neq \begin{array}{r} 1101_2 \\ +\ 0011_2 \\ \hline 10000_2 \end{array}$$

Solution - Use larger base to avoid carries

$$\begin{array}{r} 1101 \\ \vee\ 0011 \\ \hline 1111 \end{array} \longleftrightarrow \begin{array}{r} 1101_4 \\ +\ 0011_4 \\ \hline 1112_4 \end{array}$$

Base 4 {0,1,2,3}

(closer)

- Each edge may covered once or twice

⇒ final sum has 1's + 2's: 121122_4

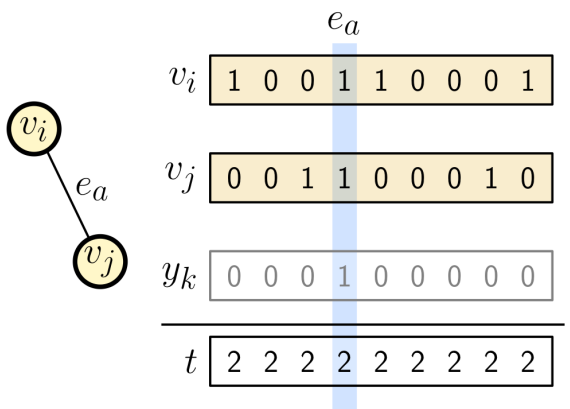
Solution: Create a set of **slack values** with **1 digit** in each. Use with **discretion** to make sum = $222\ldots2_4$

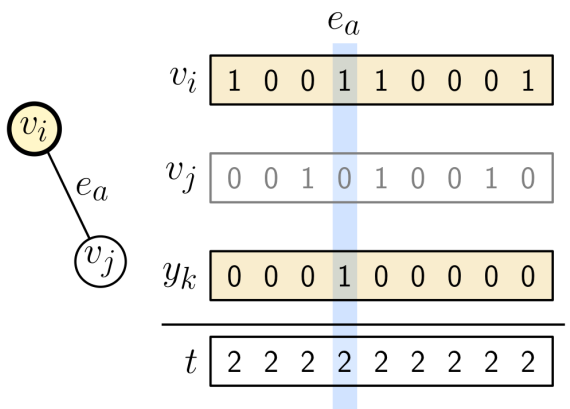
vertex sum: 121122_4

slacks:

y_1	1	0	0	0	0	0	← Select to bump 1-digits up to 2
y_2	0	1	0	0	0	0	
y_3	0	0	1	0	0	0	
y_4	0	0	0	1	0	0	
y_5	0	0	0	0	1	0	
	0	0	0	0	0	1	
t =	2	2	2	2	2	2	$_4$

Example:





- **How to control size k of vertex cover?**
  - Add **column of 1's**
  - Adjust **t** so **this column sums to k**

⇒ **k values selected** ⇒ **k vertices selected**

Here's the whole reduction: Given $G=(V,E)$, k

- Let $n=|V|$ + $m=|E|$
- Create n vertex values in base 4, $\{v_1,\dots,v_n\}$
  - For v_i: $m+1$ digits
    - Leading digit = 1
    - If v_i incident to edge e_j set j^{th} digit to 1 (else 0)

- Create m slack values in base 4, $\{y_1,\dots,y_m\}$
  - For y_j: $m+1$ digits
    - Leading digit = 0
    - Set j^{th} digit to 1 + all others = 0

- Set t to $m+1$ digits in base 4
  - Leading digit = k
  - Remaining m digits = 2

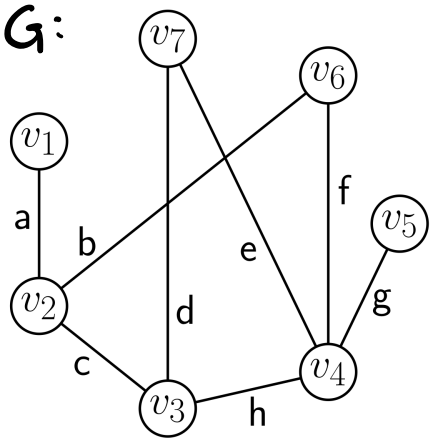
- Convert numbers from base 4 to base 10 (or whatever subset sum expects) $\rightarrow (S,t)$

Time - Total of $n+m$ numbers, each with $m+1$ digits

$\Rightarrow O(m(n+m))$ - polynomial

Example:

G:



$k = 3$

f

Vertex cover $= \{v_2, v_3, v_4\}$

		a	b	c	d	e	f	g	h
v_1	1	1	0	0	0	0	0	0	0
v_2	1	1	1	1	0	0	0	0	0
v_3	1	0	0	1	1	0	0	0	1
v_4	1	0	0	0	0	1	1	1	1
v_5	1	0	0	0	0	0	0	1	0
v_6	1	0	1	0	0	0	1	0	0
v_7	1	0	0	0	1	1	0	0	0
y_a	0	1	0	0	0	0	0	0	0
y_b	0	0	1	0	0	0	0	0	0
y_c	0	0	0	1	0	0	0	0	0
y_d	0	0	0	0	1	0	0	0	0
y_e	0	0	0	0	0	1	0	0	0
y_f	0	0	0	0	0	0	1	0	0
y_g	0	0	0	0	0	0	0	1	0
y_h	0	0	0	0	0	0	0	0	1
$t =$	3	2	2	2	2	2	2	2	2

vertex values

slack values

k

All numbers in base 4

		a	b	c	d	e	f	g	h
v_1	1	1	0	0	0	0	0	0	0
v_2	1	1	1	1	0	0	0	0	0
v_3	1	0	0	1	1	0	0	0	1
v_4	1	0	0	0	0	1	1	1	1
v_5	1	0	0	0	0	0	0	1	0
v_6	1	0	1	0	0	0	1	0	0
v_7	1	0	0	0	1	1	0	0	0
y_a	0	1	0	0	0	0	0	0	0
y_b	0	0	1	0	0	0	0	0	0
y_c	0	0	0	1	0	0	0	0	0
y_d	0	0	0	0	1	0	0	0	0
y_e	0	0	0	0	0	1	0	0	0
y_f	0	0	0	0	0	0	1	0	0
y_g	0	0	0	0	0	0	0	1	0
y_h	0	0	0	0	0	0	0	0	1
$t =$	3	2	2	2	2	2	2	2	2

take as needed to make 2's

Counts number of vertices in cover

Correctness:

Lemma: G has a vertex cover of size k iff S has a subset that sums to t.

Proof: (⇒) Suppose G has vertex cover V', $|V'| = k$

To construct subset sum S':

- for each $v_i \in V'$:
  - add vertex value v_i to S'
- for each edge $e_j \in E$:
  - if only one endpt in V', add y_j to S'

- Claim: S' sums to $t = [k\,2\,2\,2 \ldots 2_4]$
  - Since $|V'| = k$, k values contribute 1 to leading digit position
  - Every edge e_j is covered:
    - If once - take vertex + y_j = 2 ✓
    - If twice - take both vertices = 2 ✓

(⇐) Suppose S has subset S' sums to t.

- To match leading digit, there are k vertex values → V'
- The only way to get 222…2 is for each column to contribute one or two vertices to V'. Why?
  - slacks can only increase counts by +1
  - no carries possible because only two 1-digits per column.

□