CCUT - PSU

CS 163 Data Structures

Lecture 1

- Intro
- Graph basics
- Graph traversal

Fang Song 宋方

Portland State University

Intro - my group

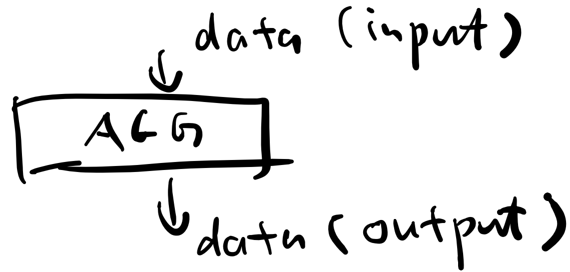
Quantum Computing → CRYPTOGRAPHY

→ Computational Complexity

fang.song@pdx.edu

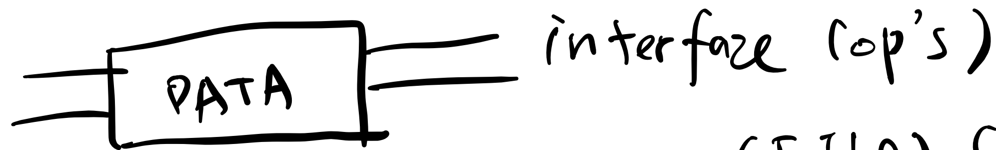
Intro - this course

→ Algorithms :



→ Data structures : organize data so that it can be accessed quickly & usefully

• ADT (Abstract Data Type)



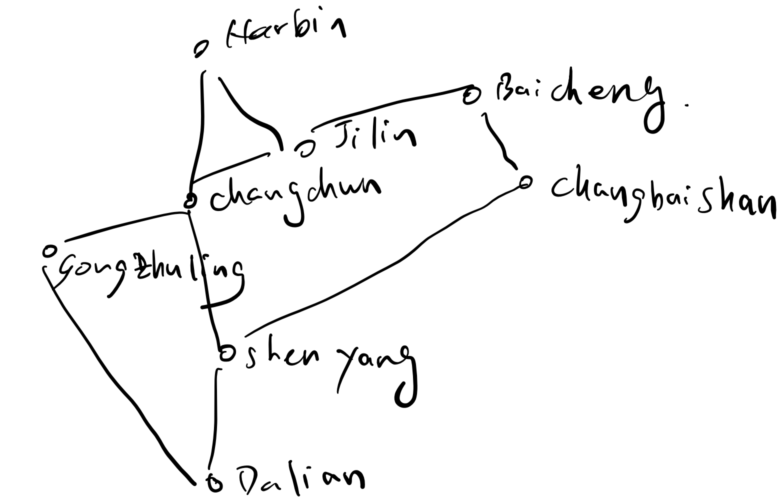
→ so far : L.L, (linear) List, (binary) tree, stack (FILO), queue (FIFO),

Intro - guiding questions

Graphs:

Map of Dongbei: A city: node o

Rail/road: (line) edge



Q1: Can I go from A → B by train? (Connectivity)

Q2: what's the "fastest" option?

→ SHORTEST PATH PROBLEM

→ LONGEST PATH PROBLEM

→ MINIMUM Spanning Tree (MST)

→ Tavelsalesperson Problem (TSP)

Intro - guiding questions

→ Graphs: core concepts, algorithms, D. S. comp. complexity

(design techniques)

→ Flavor switch: . More on ideas: conceptual, abstract.

. less on low-level. implementations

☆: "Diversity" of CS!

Scratch

CCUT - PSU

CS 163 Data Structures

Lecture 2

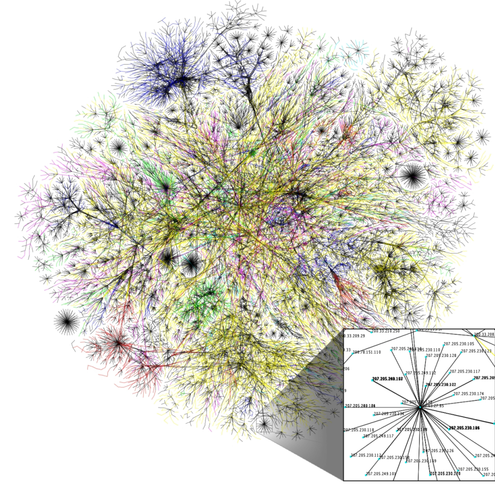
- Graph basics
- Graph traversal

Fang Song 宋方

Portland State University

Graphs

- **A graph is a set of vertices that are pairwise connected by edges.**
  - Representing relationships between pairs of objects.
  - Two categories: directed vs. undirected.



- **Why care about graphs?**
  - Graphs are a very useful abstraction.
  - Graphs have numerous applications.
  - A lot of graph algorithms exist (and more under way).

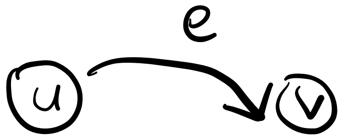
Versatile abstraction

Application	Vertices	Edges
Traffic	Intersections	Roads
Social network	People	Friendship
Game	Board position	Legal move
Financial	Stock/currency	Transactions
Programs	Procedures	Procedure call
Precedence constraints	Courses	Prerequisite

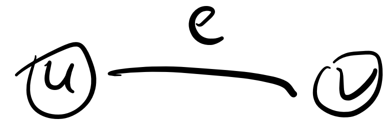
Defining graphs

- An undirected graph $G = (V, E)$ consists of
  - V: a finite set. (Vertex/node set)
  - $E \subseteq \{(u, v) : u, v \in V\}$. (Edge set)
  - NB. Self loop (u, u) not allowed.

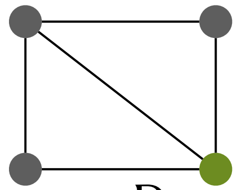
- A directed graph $G = (V, E)$ consists of
  - V: a finite set. (Vertex/node set)
  - $E \subseteq \{u \to v : u, v \in V\}$. (Edge set)
  - The set of edges need NOT be symmetric.



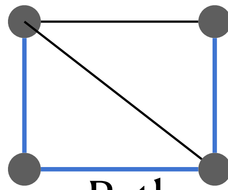
Graph terminology



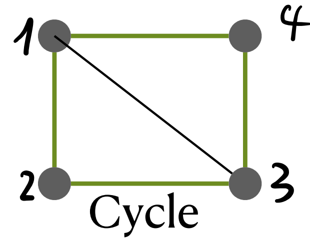
- If $e = (u, v)$ is an edge in a graph, then v is called adjacent to u. (a.k.a neighbors)
- Edge e is said to be incident to u and v .
- Degree of a vertex $d(u)$: the number of edges incident to the vertex u.



Degree $d(u) = 3$



Path



Cycle

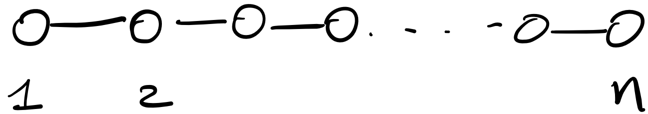
{1, 2, 3, 4, 1}

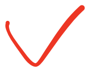
- A path is a sequence of vertices that are connected by edges.
  - $\{v_1, \ldots, v_k\}$, s.t. $(v_i, v_{i+1}) \in E$ for all $i = 1,\ldots,k-1$.
- A cycle is a path whose first and last vertices are the same.
- Two vertices are connected if and only if (iff.) there is a path between them.

Pop quiz

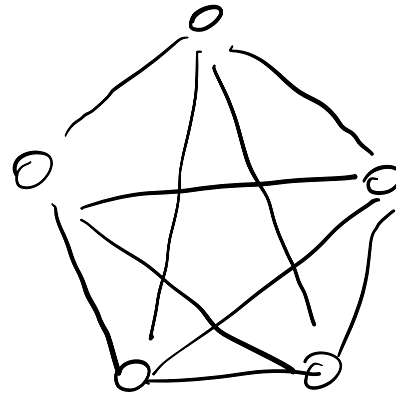
- Consider an undirected graph with n vertices and no parallel edges. Assume that the graph is connected, meaning "in one piece." What are the minimum and maximum numbers of edges, respectively, that the graph could have?

min #: $(n-1)$





max #: $n \cdot (n-1)/2$

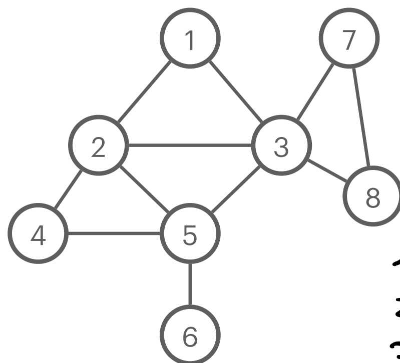


Complete graph

Graph representation 1: adjacency matrix

邻接 矩阵

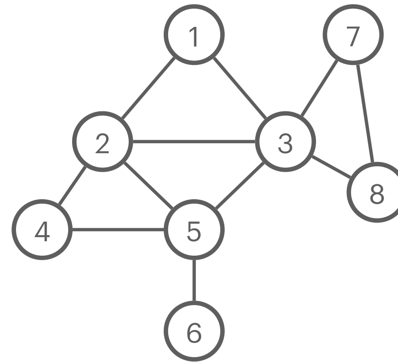
- **Given:** $G = (V, E)$, $|V| = n$, $|E| = m$.
- **Adjacency matrix** A: $n \times n$, $A_{uv} = 1$ **iff.** $(u, v) \in E$ **is an edge.**



- **Basic properties**
  - Lookup an edge: (4, 6)? $O(1)$
  - List all neighbors: scan row i: $O(n)$
  - Space: $O(n^2)$

Graph representation 1: adjacency matrix

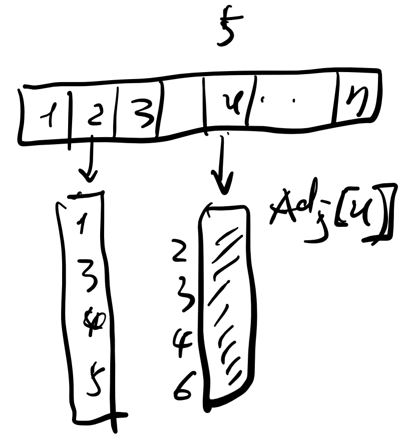
- **Given:** $G = (V, E), |V| = n, |E| = m$.
- **Adjacency matrix** A: $n \times n$, $A_{uv} = 1$ **iff.** $(u, v) \in E$ **is an edge.**



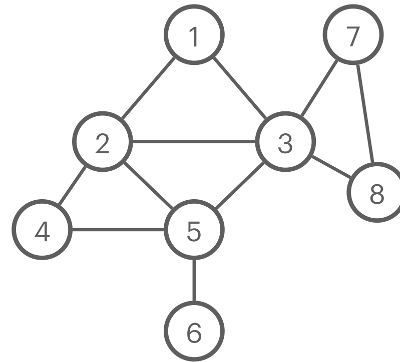
- **Basic properties**
  - Lookup an edge: $\Theta(1)$.
  - List all neighbors: $\Theta(n)$.
  - Symmetric for undirected graphs.
  - Space: $\Theta(n^2)$, good for dense graphs.

Graph representation 1: adjacency list

- **Given:** $G = (V, E), |V| = n, |E| = m$.
- **Adjacency list** Adj: $\forall u \in V, Adj[u] = \{v : v$ adjacent to $u\}$.

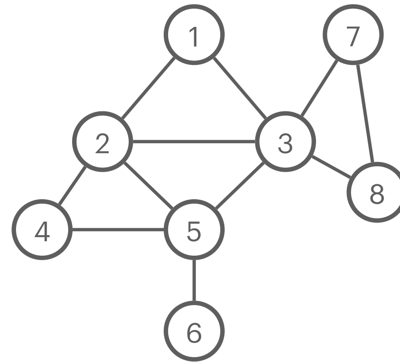


- **Basic properties**
  - Lookup an edge:
  - Space:



Graph representation 1: adjacency list

- **Given:** $G = (V, E), |V| = n, |E| = m$.
- **Adjacency list** Adj: $\forall u \in V, Adj[u] = \{v : v$ **adjacent to** $u\}$.



- **Basic properties**
  - Lookup an edge: $\Theta(\text{degree}(u))$.
  - Space: $\Theta(m + n)$, good for sparse graphs.

Trees

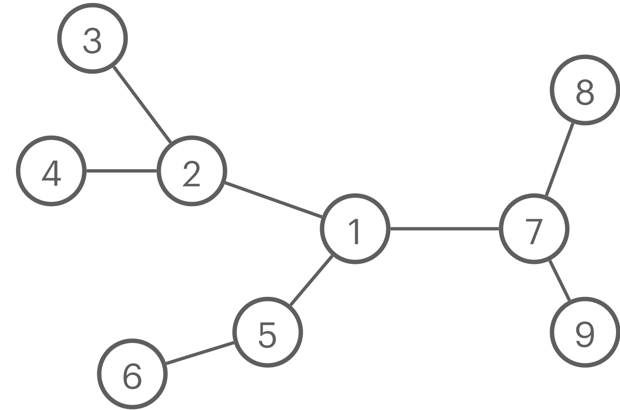
- A graph is connected if every pair of vertices u & v are connected.
- A tree is an undirected graph that is connected and does not contain a cycle.

- Theorem. Let G be an undirected graph on n nodes.
  Any two of the following statements imply the third.

  a. G is connected.

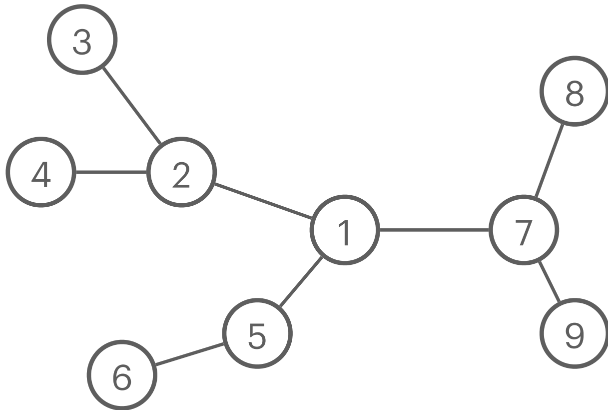
  b. G does not contain a cycle.

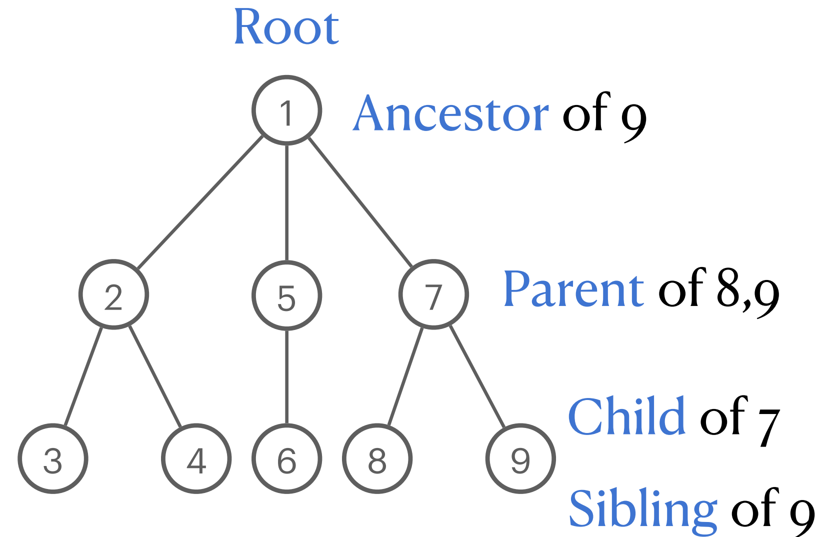
  c. G has $n-1$ edges.



Rooted trees

- **Given a tree, choose a root node r, and orient each edge away from r.**
  - Models hierarchical structure.





Exploring a graph

Given: vertices $s, t \in V$.

Goal: decide if there is a path from s to t.

Exploring a graph

Given: vertices $s, t \in V$.

Goal: decide if there is a path from s to t.

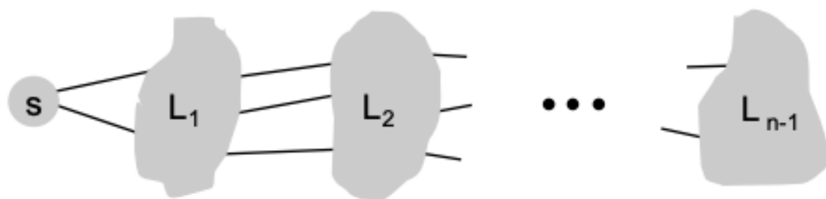
Generic traverse(s):
// if $\exists s \sim t$ path if and only if it is marked as "explored."
1. mark s as explored, all others as unexplored
2. While there is an edge $(v, w) \in E$ with v explored and w unexplored do
   Choose some such edge (v, w) //unspecified
   Mark w as explored

- **Breadth-first search (BFS)**
  - Explore children in order of distance to start node.
- **Depth-first search (DFS)**
  - Recursively explore node's children before exploring siblings.

Breadth-first search (BFS)

Breadth-first search

- **Intuition. Explore outward from s in all possible directions.**
  - Adding nodes one layer at a time.

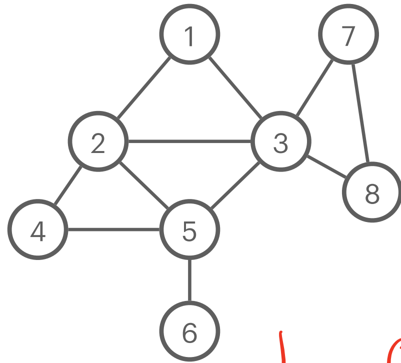


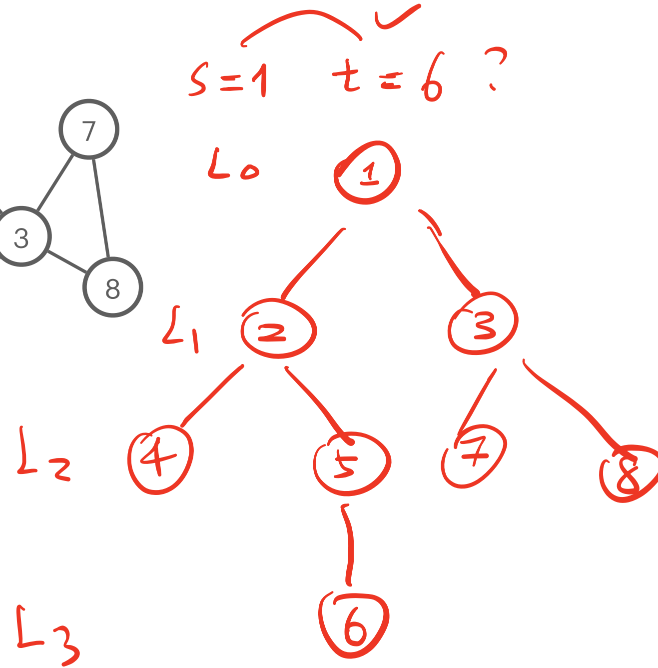
- $L_0 = \{s\}$
- $L_1 = \{\text{neighbors of } L_0\}$
- $L_2 = \{\text{neighbors of } L_1 \text{ not in } L_0 \ \& \ L_1\}$

Analogy: wave front of a ripple

Understanding BFS

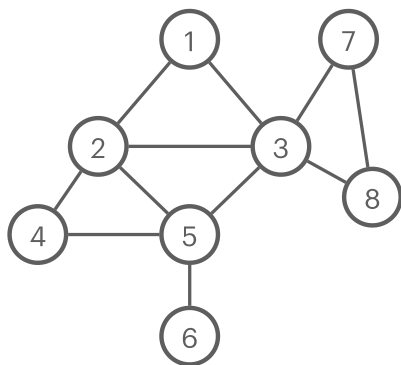
- BFS demo.

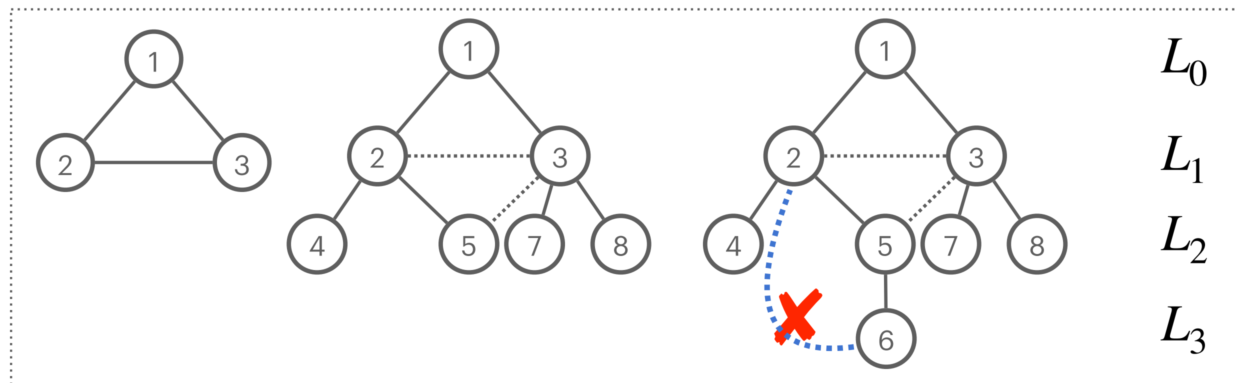




Understanding BFS

- BFS demo.



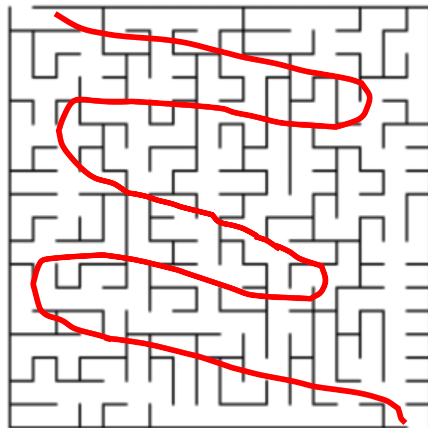


- Running time: linear $O(|V| + |E|)$ [more to come]
- For each i, L_i consists of all nodes at distance exactly i from s.
- There is a path from s to t iff. t appears in some layer.
- Let T be a BFS tree of $G = (V, E)$, and (u, v) an edge of G. Then the levels of u and v differ by at most 1.

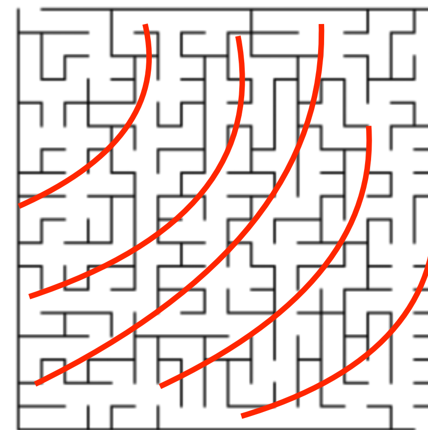
Depth-first search (BFS)

Depth-first search

- Intuition. Childen prior to siblings.

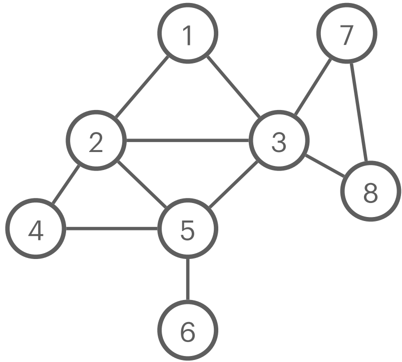


DFS: an impatient maze runner

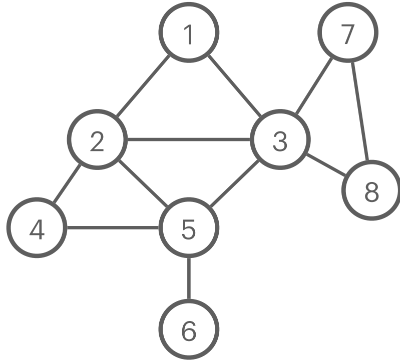


BFS: a patient maze runner

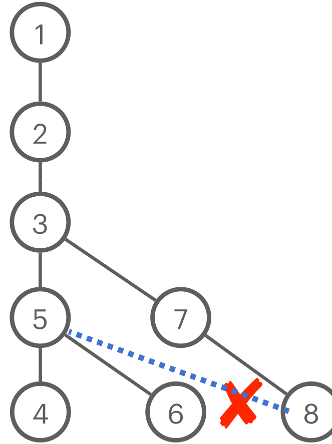
DFS in action



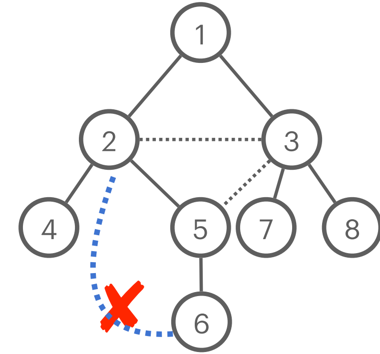
Understanding DFS



DFS Tree



Contrast with BFS Tree



- **Running time: linear $O(|V| + |E|)$ [more to come]**
- **Let T be a DFS tree of $G = (V, E)$, and let u & v be nodes in T.**
  - If (u, v) is an edge of G that is not an edge of T.
  - Then one of u or v is an ancestor of the other.

Implementing BFS/DFS

- **Generic traversal algorithm**

> 1. $R = \{s\}$
> 2. While there is an edge (u, v) where $u \in R$ and $v \notin R$, add v to R.

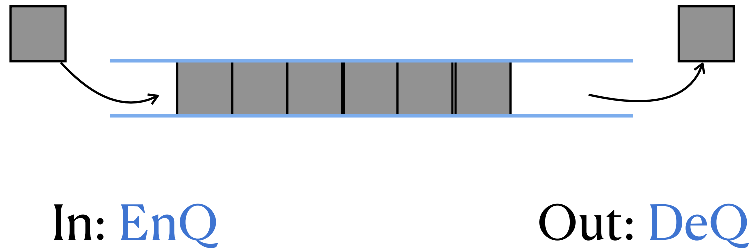
To implement it, need to choose

- **Graph representation**
- **Data structure to track …**
  - Vertices already explored.
  - Edges to be followed next.

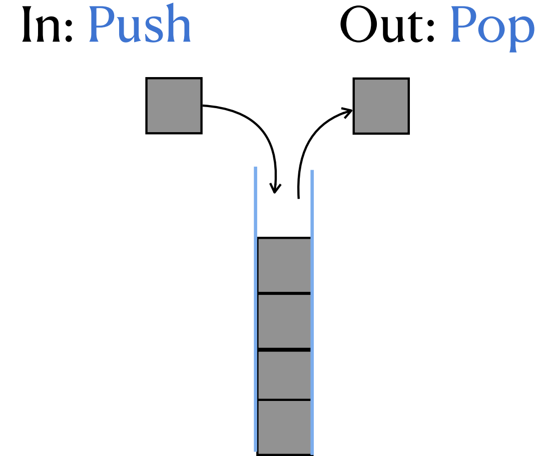
> These choices affect the order of traversal

Review: queue & stack

1. Queue: first-in first-out (FIFO)



2. stack: last-in first-out (FIFO)

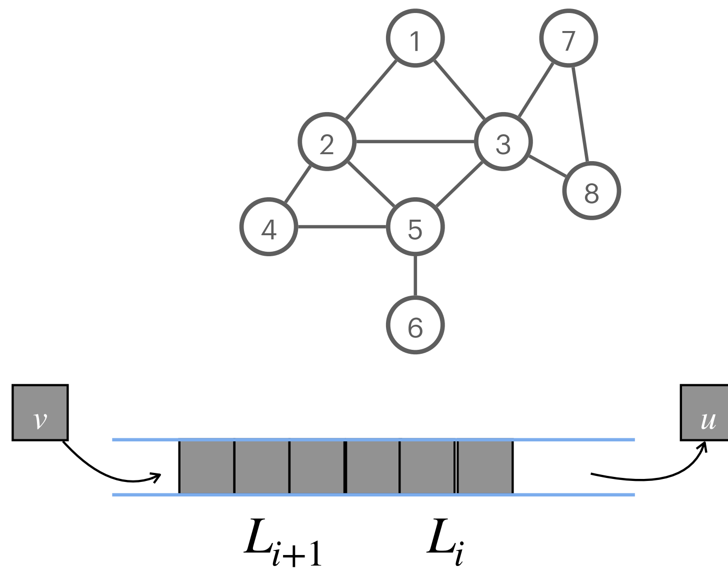


BFS implementation

Input: $G = (V, E)$ by adjacency list Adj. Start node s.

Goal: BFS tree T (rooted at s).

```
BFS(s):
// Discoverd[1,...,n] array of bits (explored or not),
initialized to all zeros.
// Queue Q ← ∅
1.  Set Discovered[s] = 1
2.  EnQ(s) // add s to Q
3.  While Q not empty  DeQ(u)
       For  each (u,v) incident to u
          If Discovered[v]=0 then
             Set Discovered[v]=1
             Add edge (u,v) to T
             EnQ(v)
```



BFS running time

```
BFS(s):
// Discoverd[1,...,n] array of bits (explored or not),
initialized to all zeros.
// Queue Q ← ∅
1.  Set Discovered[s] = 1
2.  EnQ(s) // add s to Q
3.  While Q not empty  DeQ(u)
        For  each (u,v) incident to u
           If Discovered[v]=0 then
               Set Discovered[v]=1
               Add edge (u,v) to T
               EnQ(v)
```

$O(1)$, run once for all

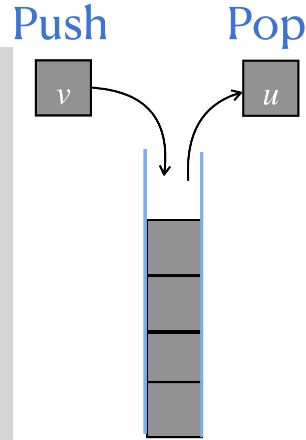
$O(1)$, run once per vertex

$O(1)$, run $\leq$ twice per edge

Theorem. BFS takes $O(m + n)$ time (linear in input size).

DFS implementation

```
BFS(s):
// Discoverd[1,...,n] array of bits (explored or not),
initialized to all zeros.
// Stack S ← ∅
1.  Set Discovered[s] = 1
2.  Push(s) // add s to S
3.  While S not empty  Pop(u)
        If Discovered[v]=0 then
        Set Discovered[u]=1
        For each (u, v) incident to u
          Push(v)
```



```
BFS(s):
...
3.  While Q not empty  DeQ(u)
      For  each (u,v) incident to u
        If Discovered[v]=0 then
          Set Discovered[v]=1
          Add edge (u,v) to T
          EnQ(v)
```

Theorem. DFS takes $O(m+n)$ time (linear in input size).

- **Exercise. How to build DFS tree T along the way?**

Scratch

CCUT - PSU

CS 163 Data Structures

Lecture 3

- Graph traversal
- Shortest Path

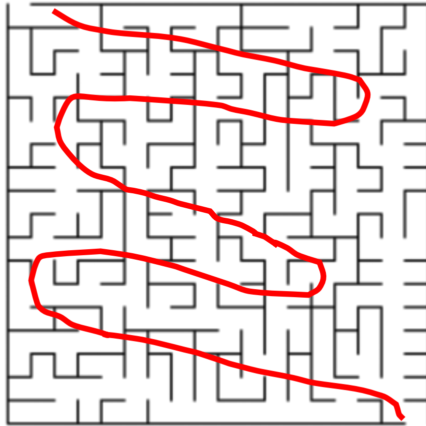
Fang Song 宋方

Portland State University

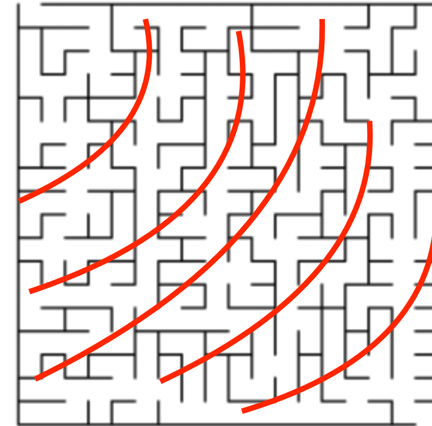
Depth-first search (DFS)

Depth-first search

- Intuition. Childen prior to siblings.

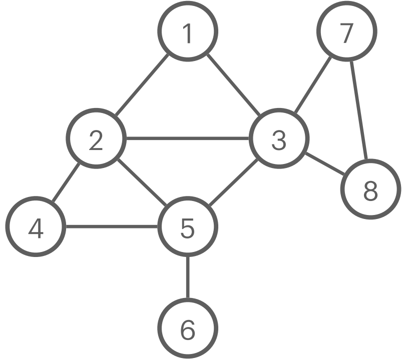


DFS: an impatient maze runner

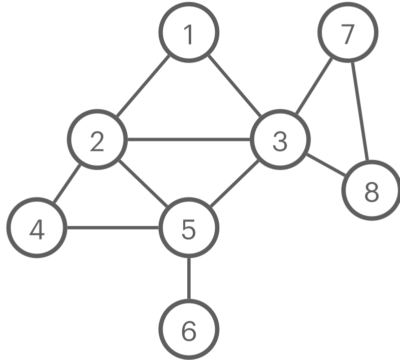


BFS: a patient maze runner

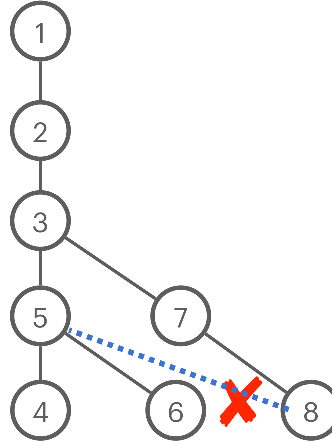
DFS in action



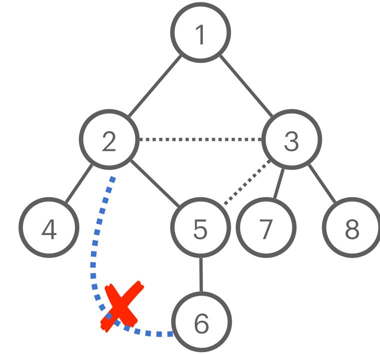
Understanding DFS



DFS Tree



Contrast with BFS Tree



- **Running time: linear $O(|V| + |E|)$ [more to come]**
- **Let T be a DFS tree of $G = (V, E)$, and let u & v be nodes in T.**
  - If (u, v) is an edge of G that is not an edge of T.
  - Then one of u or v is an ancestor of the other.

Implementing BFS/DFS

- **Generic traversal algorithm**

1. $R = \{s\}$
2. While there is an edge (u, v) where $u \in R$ and $v \notin R$, add v to R.

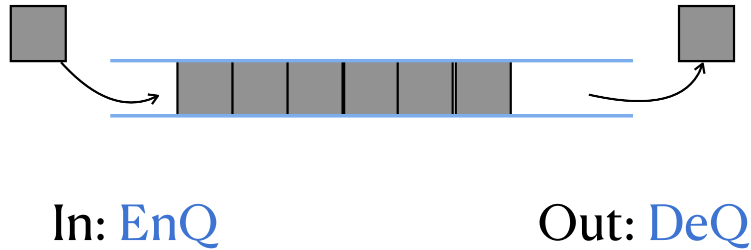
To implement it, need to choose

- **Graph representation**
- **Data structure to track …**
  - Vertices already explored.
  - Edges to be followed next.

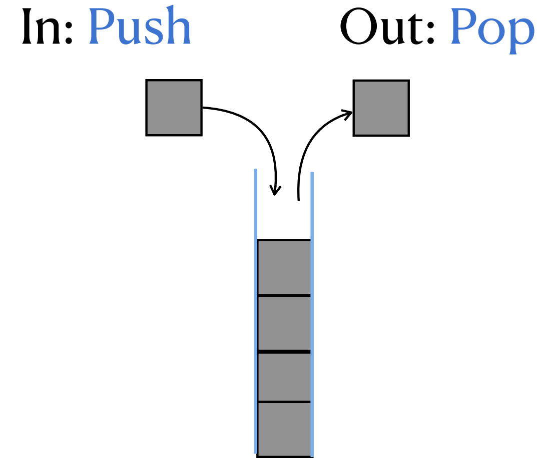
These choices affect the order of traversal

Review: queue & stack

1. Queue: first-in first-out (FIFO)



2. stack: last-in first-out (FIFO)

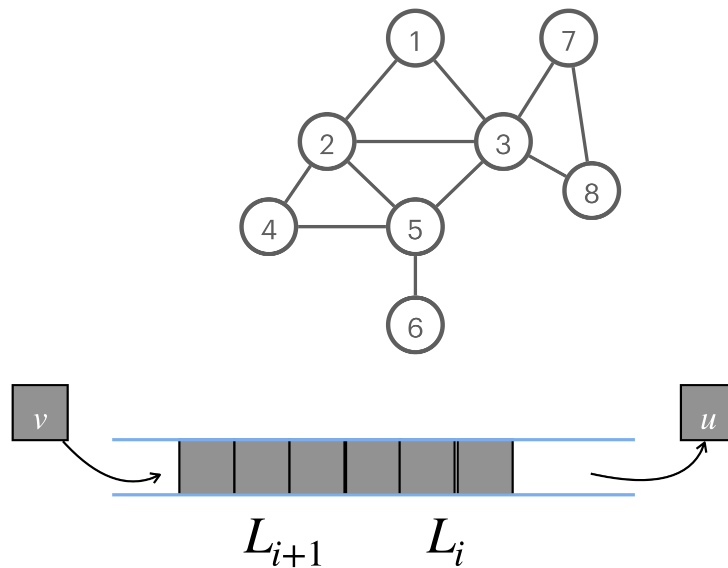


BFS implementation

Input: $G = (V, E)$ by adjacency list Adj. Start node s.

Goal: BFS tree T (rooted at s).

BFS(s):
// Discoverd[1,...,n] array of bits (explored or not), initialized to all zeros.
// Queue $Q \leftarrow \varnothing$
1. Set Discovered[s] = 1
2. EnQ(s) // add s to Q
3. While Q not empty DeQ(u)
    For each (u,v) incident to u
        If Discovered[v]=0 then
            Set Discovered[v]=1
            Add edge (u,v) to T
            EnQ(v)



BFS running time

```
BFS(s):
// Discoverd[1,...,n] array of bits (explored or not),
initialized to all zeros.
// Queue Q ← ∅
1.  Set Discovered[s] = 1
2.  EnQ(s) // add s to Q
3.  While Q not empty  DeQ(u)
        For  each (u,v) incident to u
           If Discovered[v]=0 then
               Set Discovered[v]=1
               Add edge (u,v) to T
               EnQ(v)
```

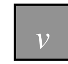
- Lines 1–2: $O(1)$, run once for all
- Line 3: $O(1)$, run once per vertex
- For loop body: $O(1)$, run $\leq$ twice per edge

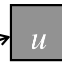
Theorem. BFS takes $O(m + n)$ time (linear in input size).

DFS implementation

D
BFS(s):
// Discoverd[1,...,n] array of bits (explored or not), initialized to all zeros.
// Stack $S \leftarrow \emptyset$
1. Set Discovered[s] = 1
2. Push(s) // add s to S
3. While S not empty Pop(u)
   If Discovered[v]=0 then
   Set Discovered[u]=1
   For each (u, v) incident to u
   Push(v)

Push Pop





BFS(s):
...
3. While Q not empty DeQ(u)
   For each (u,v) incident to u
   If Discovered[v]=0 then
   Set Discovered[v]=1
   Add edge (u,v) to T
   EnQ(v)

Theorem. DFS takes $O(m + n)$ time (linear in input size).

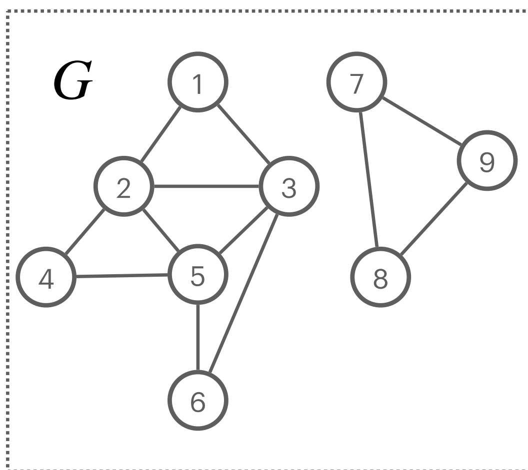
- **Exercise. How to build DFS tree T along the way?**

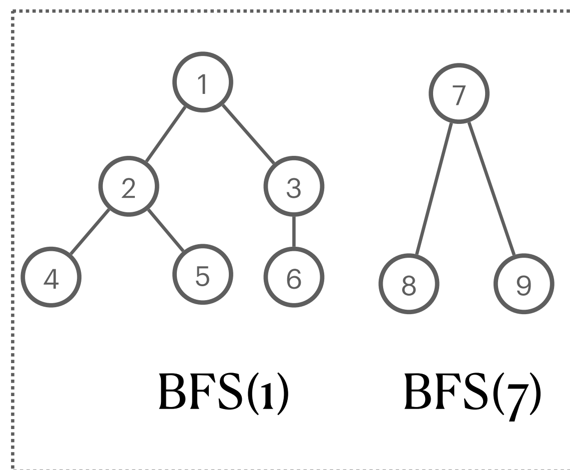
BFS/DFS applications

Connected components

- B/DFS tell more than s-t connectivity.

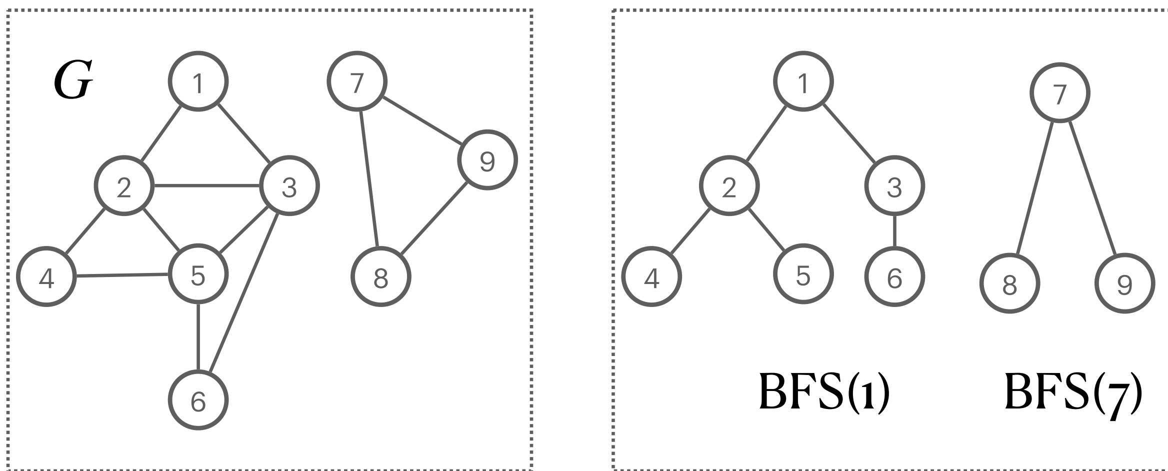
Connected component of G containing s: all nodes reachable from s.





- Claim. For any tow nodes s and t, their connected components are either identical or disjoint.

The set of all connected components



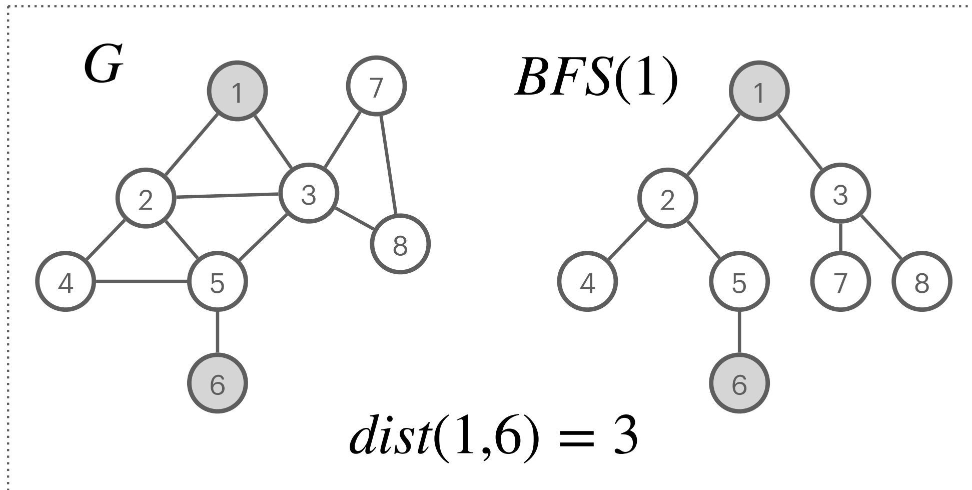
- **How to find all?**
- **How fast?**
- **Why care?**

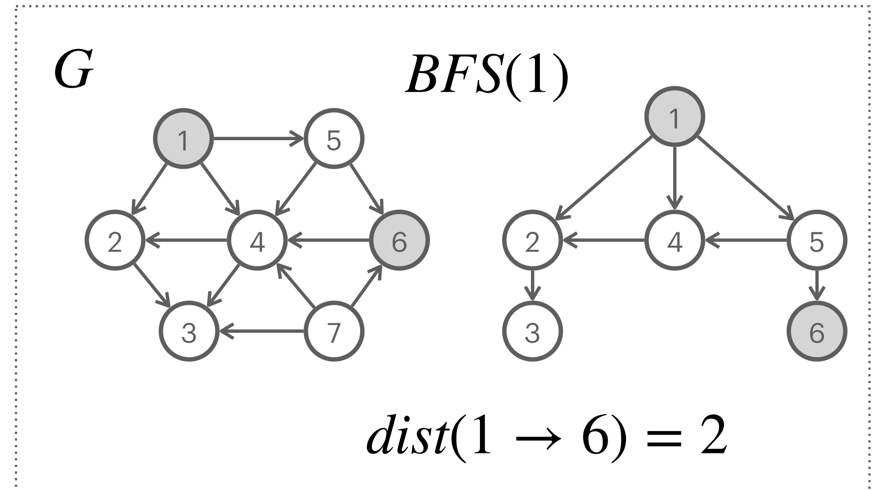
- Iterate over V, run B/DFS.
- $\sum_i n_i + m_i = O(m + n)$.
- Basic topology about G.

Shortest path in a graph

Input: Graph G, nodes s and t .

Output: $dist(s, t)$.





Shortest Path Problem

Shortest path in a weighted graph

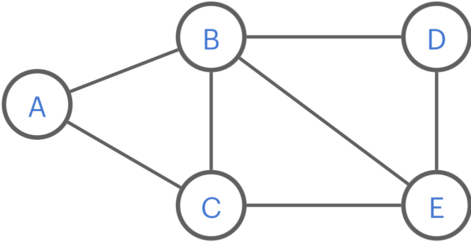
- **Weighted graphs**
  - Every edge has a length ℓ_e.
  - Length of a path $\ell(P) = \sum_{e \in P} \ell_e$.
  - Distance $dist(s, t) = \min_{P: u \rightsquigarrow v} \ell(P)$.

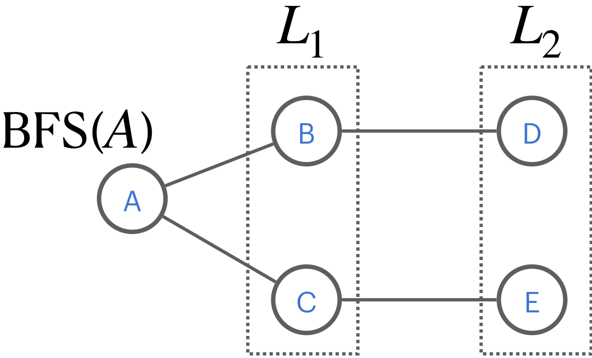
  (handwritten: u crossed out → s, v crossed out → t)

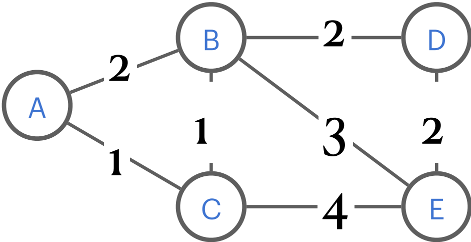
- **Length function:** $\ell : E \to \mathbb{Z}$
  - $\ell(u, v) = \infty$ if not an edge
  - Model time, distance, cost ...
  - Can be negative: fund transfer, heat in chemistry reaction ... [Later]

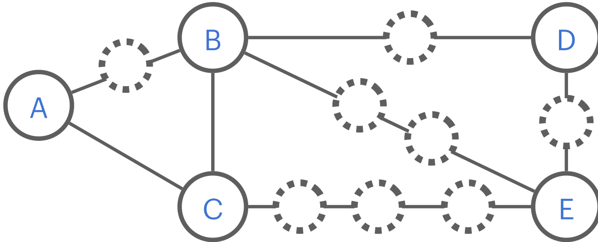
- $\forall e \in E, \ell(e) = 1$: **BFS solves it.**
- **How to solve weighted case?**

Reducing to BFS



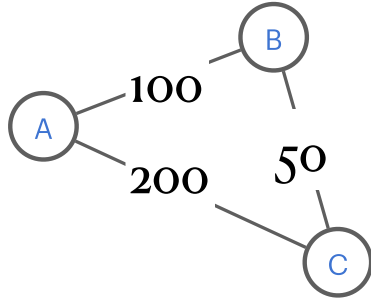


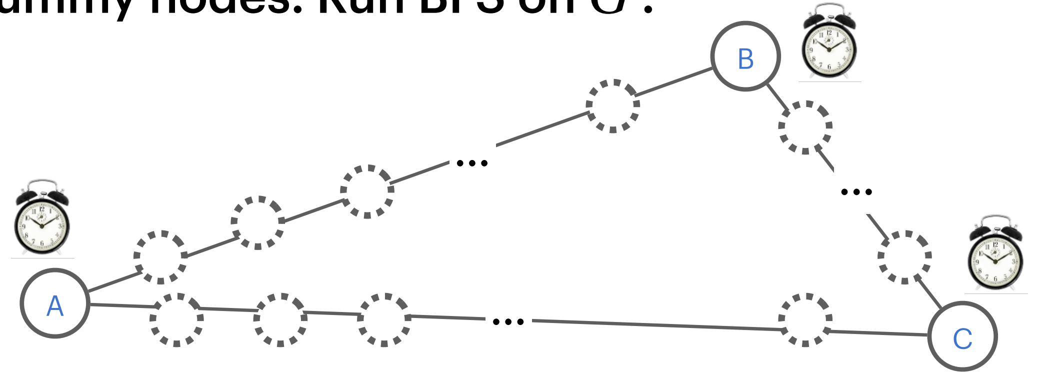




An alarm-clock algorithm

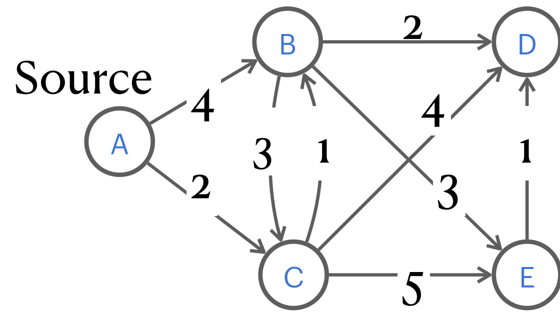
- **Idea.** convert G to G' by inserting dummy nodes. Run BFS on G'.





AlarmSP(G, s):
// set alarm clock for s at time 0
Repeat until no more alarms
// Suppose next alarm goes off at T for node u
    $dist(s, u) \leftarrow T$
    For each neighbor v of u
        If no alarm for v, set one for time $T + \ell(u, v)$
        Else If current alarm larger, reset it for time $T + \ell(u, v)$

Demo



Scratch

CCUT - PSU

CS 163 Data Structures

Lecture 4

- Dijkstra's Shortest Path Algorithm
- Minimum Spanning Tree

Fang Song 宋方

Portland State University

Shortest Path Problem

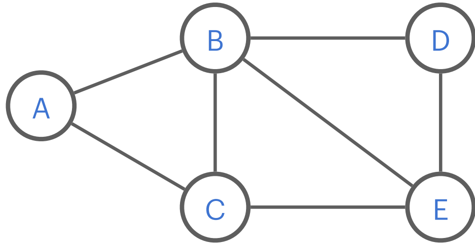
Shortest path in a weighted graph

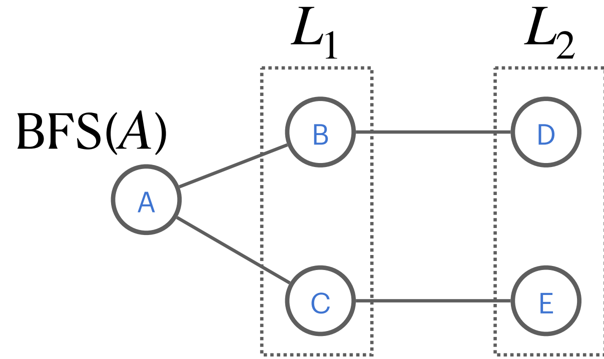
- **Weighted graphs**
  - Every edge has a length ℓ_e.
  - Length of a path $\ell(P) = \sum_{e \in P} \ell_e$.
  - Distance $dist(s,t) = \min_{P: u \rightsquigarrow v} \ell(P)$.

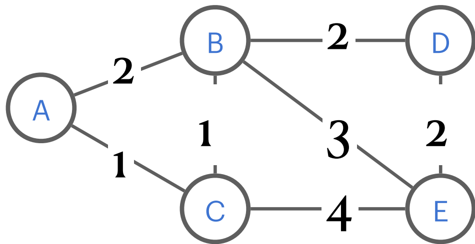
- **Length function:** $\ell : E \to \mathbb{Z}$
  - $\ell(u,v) = \infty$ if not an edge
  - Model time, distance, cost ...
  - Can be negative: fund transfer, heat in chemistry reaction ... [Later]

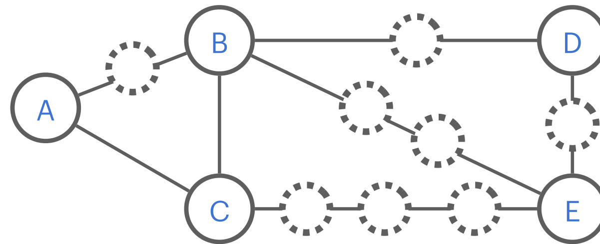
- $\forall e \in E, \ell(e) = 1$: **BFS solves it.**
- **How to solve weighted case?**

Reducing to BFS



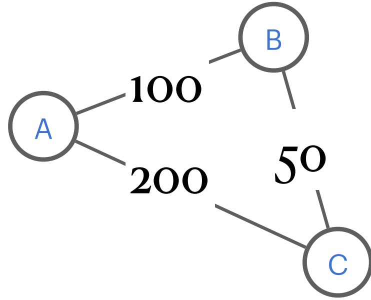


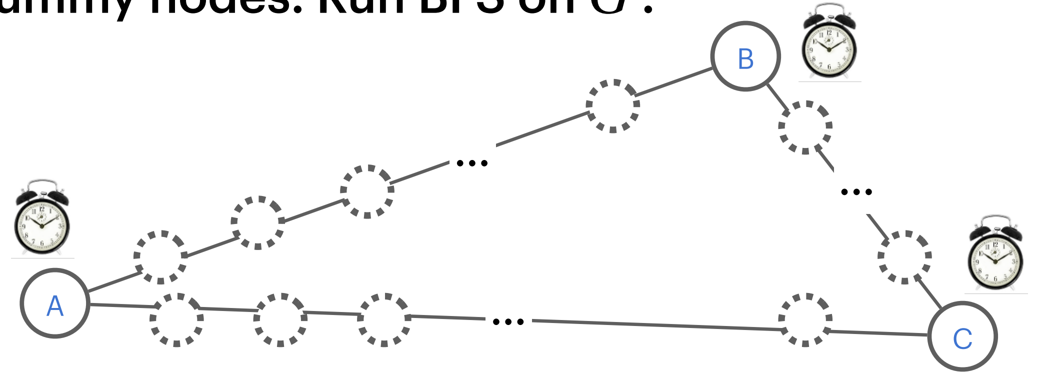




An alarm-clock algorithm

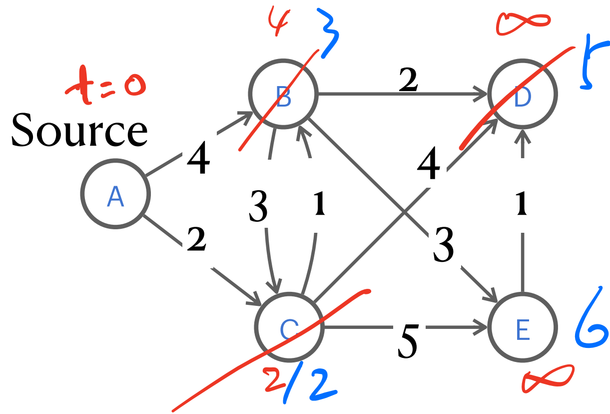
- **Idea.** convert G to G' by inserting dummy nodes. Run BFS on G'.





AlarmSP(G, s):
// set alarm clock for s at time 0
Repeat until no more alarms
// Suppose next alarm goes off at T for node u
    $dist(s, u) \leftarrow T$
    For each neighbor v of u
        If no alarm for v, set one for time $T + \ell(u, v)$
        Else If current alarm larger, reset it for time $T + \ell(u, v)$

Demo



$t=2$ Alarm C sets off.

B: Alarm update $4 \to 2+1=3$

D: Alarm $\infty \to 2+4=6$

E: $\infty \to 2+5=7$

$t=3$ Alarm B sets off.

D: update $6 \to 3+2=5$

E: Update $7 \to 3+3=6$

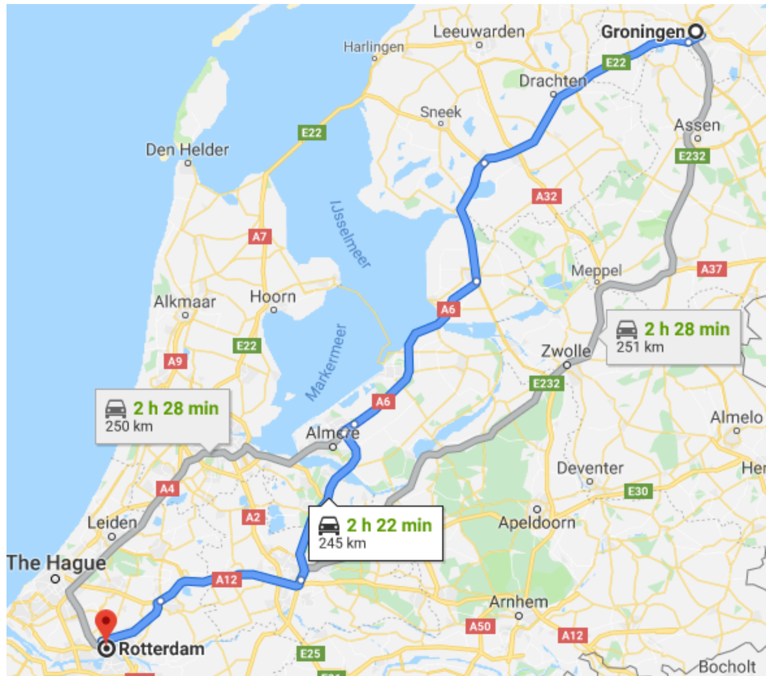
$t=5$ Alarm D sets off.

E: no update.

$t=6$ E sets off.

Edsger W. Dijkstra

- Pioneer in graph algorithms, distributed computing, concurrent computing, programming ...



"What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes.
One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. "

https://cacm.acm.org/magazines/2010/8/96632-an-interview-with-edsger-w-dijkstra/fulltext

Dijkstra's algorithm: priority queue for alarms

- **PriorityQueue Q: set of elements w. associated key values (alarm)**
  - Change-key(x). change key value of an element.
  - Delete-min. Return the element with smallest key, and remove it.
  - Can be done in $O(\log n)$ time (by a heap).

```
Dijkstra(G, s):
// initialize d(s) = 0, others d(u) = ∞
1.  Make Q from V using d( · ) as key value
2.  While Q not empty
        u ← Delete-min(Q)
        // pick node with shortest distance to s
        For all edges (u, v) ∈ E
            If d(v) > d(u) + ℓ(u, v)
                d(v) ← d(u) + ℓ(u, v) and Change-key(v)
```

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

Further improvement possible by Fibonacci heap

NB. BFS uses ordinary Queue.
Dijkstra = BFS w/ priority queue

Dijkstra's algorithm: priority queue for alarms

- **PriorityQueue Q: set of elements w. associated key values (alarm)**
  - Change-key(x). change key value of an element.
  - Delete-min. Return the element with smallest key, and remove it.
  - Can be done in $O(\log n)$ time (by a heap).

```
Dijkstra(G, s):
// initialize d(s) = 0, others d(u) = ∞
1. Make Q from V using d( · ) as key value
2. While Q not empty                                ] O(n log n)
       u ← Delete-min(Q)
       // pick node with shortest distance to s
       For all edges (u, v) ∈ E                     ] O(m log n)
          If d(v) > d(u) + ℓ(u, v)
             d(v) ← d(u) + ℓ(u, v) and Change-key(v)
```

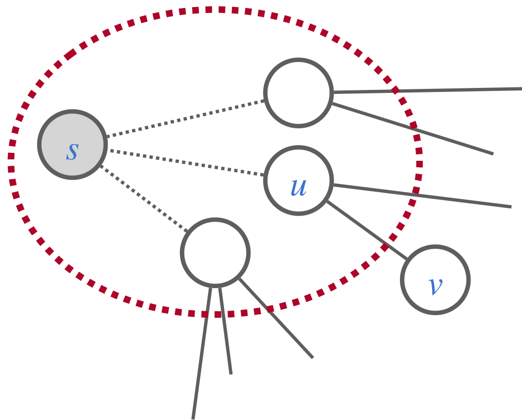
Dijkstra: $O((m+n)\log n)$

Further improvement possible by Fibonacci heap

NB. BFS uses ordinary Queue.
Dijkstra = BFS w/ priority queue

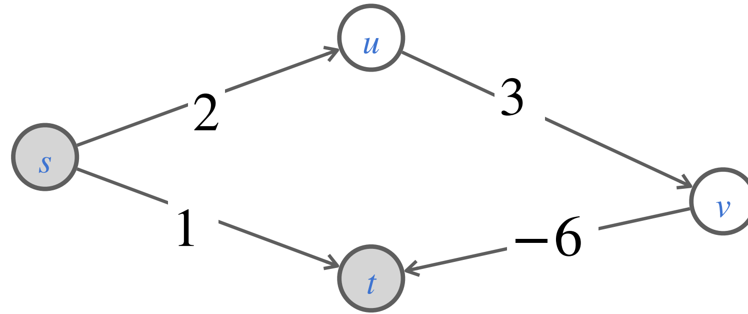
Reflection on Dijkstra: greedy stays ahead

- **Known region R:** in which the shortest distance to s is known.
- **Growing R:** adding v that has the shortest distance to s.
- **How to identify v:** the one that minimizes $d(u) + \ell(u, v)$
  - Shortest path to some u in known region, followed by a single edge (u, v).



Dijkstra(G, s):
// initialize $d(s) = 0, d(u) = \infty, R = \varnothing$
1. While $R \neq V$
   pick $v \notin R$ w. smallest $d(v)$ // by priority q
   add v to R
   For all edges $(v, w) \in E$
     If $d(v) > d(u) + \ell(u, v)$
       $d(v) \leftarrow d(u) + \ell(u, v)$

How it fails on negative lengths



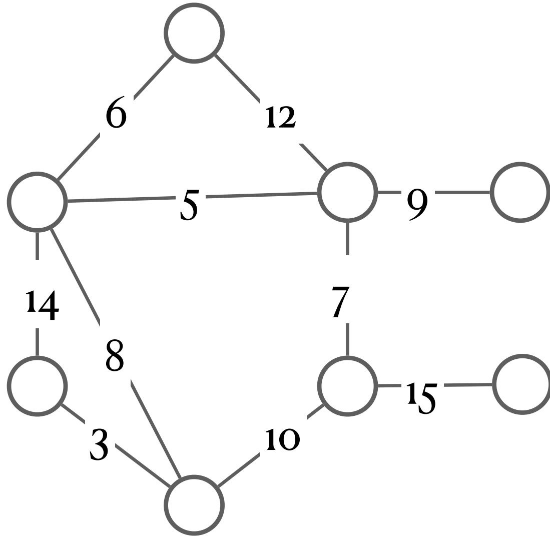
Jumping to a short one too early!

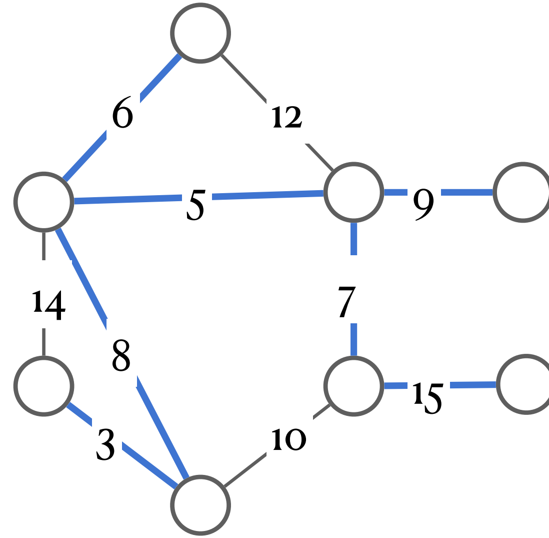
Minimum Spanning Tree

Minimum Spanning Tree (MST)

- **Input.** A connected undirected graph $G = (V, E)$
  - Weight function $w : E \to R$
  - Assuming all weights are distinct
- **Output.** A spanning tree T of minimum weight.
  - Spanning tree: a tree that connects all nodes.
  - $w(T) := \sum_{(u,v)\in T} w(u, v)$
- **Applications.**
  - Cluster, Real-time face verification
  - Network design (communication, electrical, computer, road).

Example of MST





Pop quiz

- **Which of the following are true for all spanning trees?**

  A. Contains $m - 1$ edges (n-1)

  B. The removal of any edge disconnects it.

  C. The addition of any edge creates a cycle.

  D. All of the above.

Caley's theorem

The complete graph on n nodes has n^{n-2} spanning trees

[⇒brute-force search forbidden]

Brainstorming

Greedy strategies for computing an MST?

Idea 1: → Start w/ a node u
→ add node v s.t. $w(u,v)$ smallest among $w(u,t)$

Idea 2: Remove edges w/ largest weight.

Idea 3: add edges w/ smallest weight.

Greedy algorithms for MST

- **Kruskal. Start with $T = \emptyset$. Inset edges in ascending order of weights.**
  - Unless: it creates a cycle
- **Reverse-Delete. Start with $T = E$. Remove edges in descending order of weights.**
  - Unless: it disconnects T
- **Prim. Start with some node s. (Grow a tree T from s outward)**
  - Add v to T such that $w(u, v)$ cheapest and $u \in T$.

Sounds familiar? (Dijkstra)

In this extremely lucky case, all of them work!

Proving correctness is non-trivial (stay tuned till senior year).

Implementing Prim

- **Maintain $V - T$ as a priority queue [as in Dijkstra's]**
  - key(x): weight of the least-weight edge connecting to a node in T.

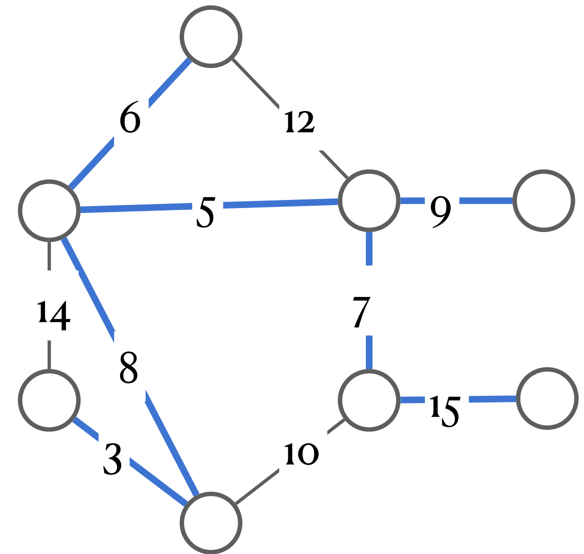
Prim$(G, \{w_e\})$:
1. Make Q from V
   // $key(s) = 0$, otherwise $key(v) = \infty$
2. While Q not empty
   - $u \leftarrow$ Delete-min(Q)
   - For all edges $(u, v) \in E$
   - // consider all neighbors of u
     - If $v \in Q$ and $w(u, v) < key[v]$
       - $key[v] \leftarrow w(u, v)$ and Change-key(v)
       - $parent(v) \leftarrow u$
3. Return $T \leftarrow \{v \,.\, parent(v)\}$

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

Same as Dijstra's

Implementing Kruskal

- **Using a new data structure: disjoint-set (aka Union-Find)**
  - `Make-set(x)`: create a singleton set containing x.
  - `Find-set(x)`: return the name of the unique set containing x.
  - `Union-set(x,y)`: merge the sets containing x, y.



Warning on greedy algorithms

Correctness

Greedy algorithms are tempting but rarely works.

Proceed with care!

Scratch

Lecture 5

- MST cont'd
- Revisit Shortest Path
- Longest path and NP-hardness

Fang Song 宋方
Portland State University

MST cont’d

Greedy algorithms for MST

- **Kruskal.** Start with $T = \varnothing$. Inset edges in ascending order of weights.
  - Unless: it creates a cycle
- **Reverse-Delete.** Start with $T = E$. Remove edges in descending order of weights.
  - Unless: it disconnects T
- **Prim.** Start with some node s. (Grow a tree T from s outward)
  - Add v to T such that $w(u, v)$ cheapest and $u \in T$. Sounds familiar? (Dijkstra)

In this extremely lucky case, all of them work!

Proving correctness is non-trivial (stay tuned till senior year).

Implementing Prim

- **Maintain $V - T$ as a priority queue [as in Dijkstra's]**
  - key(x): weight of the least-weight edge connecting to a node in T.

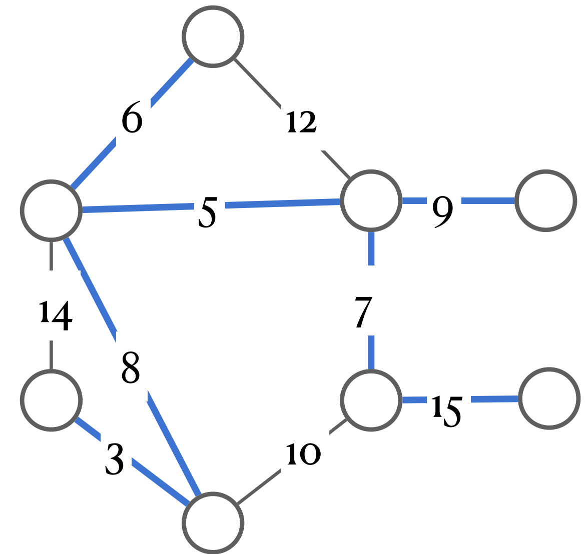
```
Prim(G, {w_e}):
1.  Make Q from V
    // key(s) = 0, otherwise key(v) = ∞
2.  While Q not empty
        u ← Delete-min(Q)
        For all edges (u, v) ∈ E
        // consider all neighbors of u
            If v ∈ Q and w(u, v) < key[v]
                key[v] ← w(u, v) and Change-key(v)
                parent(v) ← u
3.  Return T ← {v . parent(v)}
```

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

Same as Dijkstra's

Implementing Kruskal

- **Using a new data structure: disjoint-set (aka Union-Find)**
  - `Make-set(x)`: create a singleton set containing x.
  - `Find-set(x)`: return the name of the unique set containing x.
  - `Union-set(x,y)`: merge the sets containing x, y.



Warning on greedy algorithms

Correctness

Greedy algorithms are tempting but rarely works.

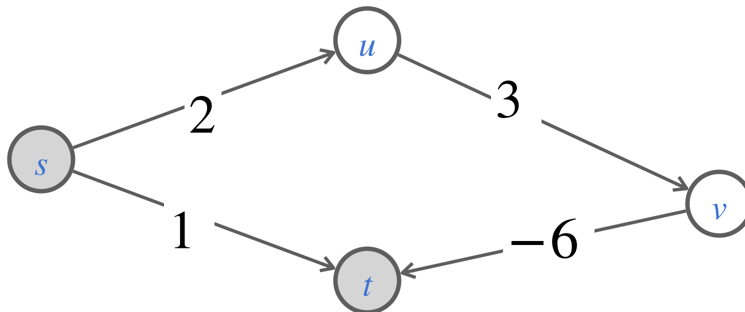
Proceed with care!

Shortest path revisited

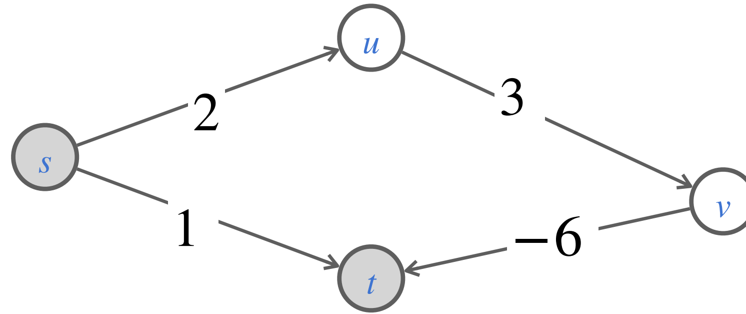
Shortest path in a weighted graph

- **Weighted graphs**
  - Every edge has a length ℓ_e.
  - Length of a path $\ell(P) = \sum_{e \in P} \ell_e$.
  - Distance $dist(s, t) = \min_{P: u \rightsquigarrow v} \ell(P)$.
- **Length function:** $\ell : E \to \mathbb{Z}$
  - $\ell(u, v) = \infty$ if not an edge
  - Model time, distance, cost ...
  - Can be negative: fund transfer, heat in chemistry reaction ...
- **Dijkstra fails on negative lengths**

Jumping to a short one too early!

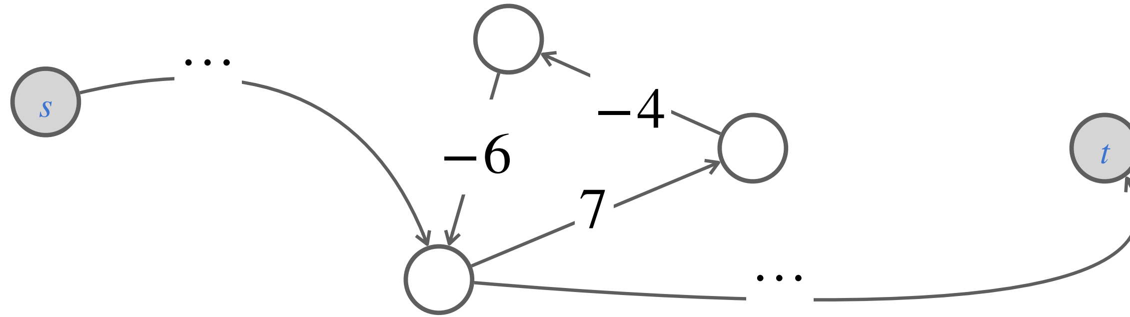


Dijkstra fails on negative lengths



Jumping to a short one too early!

A technical issue: negative length cycles



- **Observation**
  - If some $s \rightsquigarrow t$ path contains a negative length cycle, there does not exist a shortest $s \rightsquigarrow t$ path.
  - Otherwise there exists a simple (i.e., no repetition node) path $\leq n - 1$ edges.

- **For simplicity, assuming G has no NegativeLengthCycle**
  - Can be detected with little overhead.

Dynamic programming

Indispensable technique for optimization problems.

- Many soln's, each has a value.
- Find a solution with optimal (min or max) value

- Break up a problem into a series of overlapping subproblems.
- There is an ordering on the subproblems, and a relation showing how to solve a subproblem given answers to "smaller" subproblems (i.e., those appear earlier in the ordering).

DP1: develop a recurrence

- **Simplification: look at the length of a shortest path.**
- **1.a Specification. What problems to solve?**
  - Definition. $OPT(i, v) :=$ length of shortest $v \rightsquigarrow t$ path P using $\leq i$ edges.
  - Goal. Find $OPT(n-1, s)$.
- **1.b Recursion. Recurrence to solve a subproblem from smaller ones.**
  - Base. $OPT(i, v) = 0$ if $v = t$ or ∞ if $i = 0$.
  - How to compute $OPT(i, v)$ recursively?

DP1: develop a recurrence, cont'd

$OPT(i, v) :=$ length of shortest $v \rightsquigarrow t$ path P using $\leq i$ edges.

- **Case 1.** P uses at most $i - 1$ edges. $OPT(i, v) = OPT(i - 1, v)$
- **Case 2.** P uses exactly i edges.
  - If (v, w) is the first edge, then OPT uses (v, w) and then select best $w \rightsquigarrow t$ path using $\leq i - 1$ edges.
  - $OPT(i, v) = \min\limits_{v \to w \in E} \{OPT(i - 1, w) + \ell_{v \to w}\}$

$$
OPT(i, v) = \begin{cases} 0, & \text{if } v = t \\ \infty, & \text{if } i = 0 \\ \min\{OPT(i-1, v), \min_{v \to w \in E}\{OPT(i-1, w) + \ell_{v \to w}\}\}, & \text{otherwise} \end{cases}
$$

DP2: build up solutions

i \ V	t			s		v			v_n
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0								
	0								
	0								
i	0								
	0								
$n-1$	0								

- **Subproblems.** $O(n^2)$
- **Memoization data structure**
  - 2-D array $M[0,\ldots,n-1, v_1,\ldots,v_n]$.
- **Dependencies**
  - Each $OPT(i,v)$ depends on subproblems in the row above.
- **Evaluation order**
  - Row by row, arbitrary within a row.

$$OPT(i,v) = \begin{cases} 0, & \text{if } v = t \\ \infty, & \text{if } i = 0 \\ \min\{OPT(i-1,v), \min_{v\to w\in E}\{OPT(i-1,w) + \ell_{v\to w}\}\}, & \text{otherwise} \end{cases}$$

DP2: build up solutions, cont'd

V	t			s		v			v_n
i 0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0								
	0								
	0								
i	0								
	0								
$n-1$	0								

SPLen(G, s, t):
// $M[i, v]$ store subproblem values
// $M[0,t] = 0$, $M[0,v] = \infty$ otherwise.
1. For $i = 1,\ldots, n-1$ // row by row
2. For $v \in V$ // arbitrary order
   $M[i, v] \leftarrow M[i-1,v]$ // case 1
   For edge $v \rightarrow w \in E$ // case 2
   $M[i, v] \leftarrow \min\{M[i, v], M[i-1,w] + \ell_{vw}\}$
3. Return $M[n-1,s]$

Example

G

a, b, c, d, e, t

-3, -4, 6, -1, 4, -2, 2, 8, -3, 3

i \ V	t	A	B	C	D	E
0	0	∞	∞	∞	∞	∞
1	0					
2	0					
3	0					
4	0					
5	0					

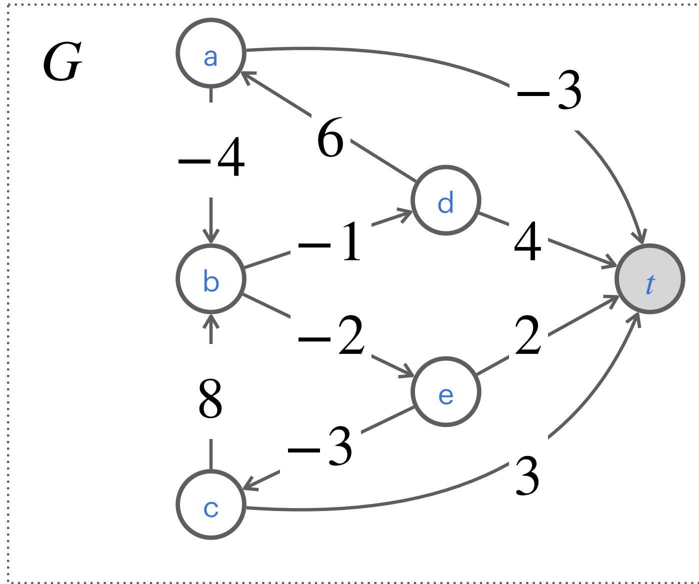
For $v \in V$ // arbitrary order

$M[i, v] \leftarrow M[i-1, v]$ // case 1

For edge $v \rightarrow w \in E$ // case 2

$M(i, j) \leftarrow \min\{M[i, v], M[i-1, w] + \ell_{vw}\}$

Example



i \ V	t	A	B	C	D	E
0	0	∞	∞	∞	∞	∞
1	0	-3		3	4	2
2	0	-3	0	3	3	0
3	0	-4	-2	3	3	0
4	0	-6	-3	3	2	0
5	0	-6	-2	3	0	0

For $v \in V$ // arbitrary order

  $M[i, v] \leftarrow M[i-1, v]$ // case 1

  For edge $v \to w \in E$ // case 2

    $M(i, j) \leftarrow \min\{M[i, v], M[i-1, w] + \ell_{vw}\}$

Single-source shortest path with negative weights

Year	Worst case	Discovered by
1955	$O(n^4)$	Shimbel
1956	$O(mn^2W)$	Ford
1958	$O(mn)$	Bellman, Moore
1983	$O(n^{3/4}m\log W)$	Gabow
1989	$O(mn^{1/2}\log(nW))$	Gabow-Tarjan
1993	$O(mn^{1/2}\log W)$	Goldberg
2005	$O(n^{2.38}W)$	Sankowsi, Yuster-Zwich
2016	$O(n^{10/7}\log W)$	Cohen-Madry-Sankowski-Vladu
20xx	???	you?

Weights between $[-W, W]$

Historic note on dynamic programming

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables.

- **Richard Bellman**
  - DP [1953] @RAND
  - B-Ford algorithm for general shortest path (stay tuned!)
  - Curse of dimensionality
  - …

- **Etymology**
  - Dynamic programming = planning over time
  - Secretary of Defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation

> "it's impossible to use dynamic in a pejorative sense"
> "something not even a Congressman could object to"
>
> Reference: Bellman, R. E. Eye of the Hurricane, An Autobiography.

Extension: all pairs shortest path

Input: Graph $G = (V, E)$, a real-valued length ℓ_e for each edge $e \in E$.

Output: $dist(u, v)$ for every $u, v \in V$.

- **For simplicity, assuming G has no NegativeLengthCycle**
  - Can be detected with little overhead.

- **Reducing to single-source shortest path**
  - How many invocations of Bellman-Ford? $O(n)$

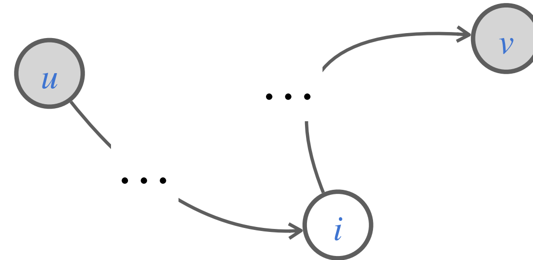
  $\Rightarrow$ Total time: $O(n \cdot m \cdot n) = O(n^4)$ if $m = O(n^2)$

Floyd-Warshall algorithm

- Key idea: restrict the identities of the nodes allowed in a solution

- 1.a Specification of subproblems [in B-F: using $\leq i$ edges]
  - $L_{i,u,v} :=$ length of shortest $u \rightsquigarrow v$ path P using only nodes in $\{1, \ldots, i\}$.

- 1.b Recursion. Recurrence to solve a subproblem from smaller ones.
  - Base.
  - $$L_{i,u,v} := \min \begin{Bmatrix} L_{i-1,u,v} & \text{(case 1)} \\ L_{i-1,u,i} + L_{i-1,i,v} & \text{(case 2)} \end{Bmatrix}$$



- Time: $O(n^3)$

Longest path & Travel salesperson

Longest path problem (LPP)

Input: Graph $G = (V, E)$, a real-valued length ℓ_e for each edge $e \in E$.

Output: longest path distance $ldist(s, t)$.

- Reducing to shortest path? $\ell_e \mapsto -\ell_e$

- Easy case: Directed Acyclic Graphs (DAGs)

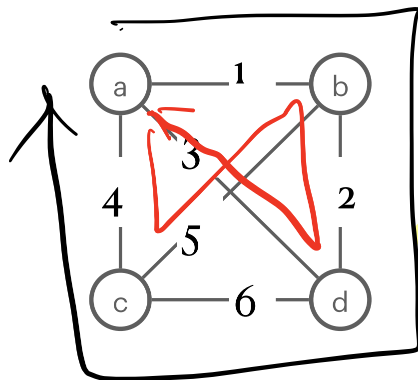
$\Rightarrow$ Application in critical path analysis: total time for completing a project with dependent milestones (e.g., completing college degree)

Travel salesperson problem (TSP)

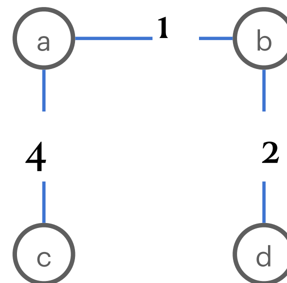
Input: A complete graph $G = (V, E)$, a real-valued cost ℓ_e for each $e \in E$.

Output: A tour T of G with the minimum $\sum_{e \in T} \ell_e$ of edge costs.

Tour: A cycle that spans all nodes



T	$\ell(T)$
abcd / adcb	1+5+6+3=15
abdc / acdb	1+2+6+4=13
acbd / adbc	4+5+2+3=14



- Contrast with MST: minimum cost spanning tree

Pop quiz

- Consider $G = (V, E)$ with $n \geq 3$ nodes (undirected, complete), how many distinct tours $T \subseteq E$ are there?

  A. 2^n

  B. $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$

  C. $\frac{1}{2}(n-1)!$

  D. n

Algorithms for LPP & TSP

Fact: as of this lecture (April 2026), there is no known fast (i.e., polynomial-time in input size) algorithm for LPP and TSP.

Speculation: No fast algorithm for LPP or TSP exist.

Both LPP and TSP are named NP-hard problems by complexity theorists.

A taste of complexity theory

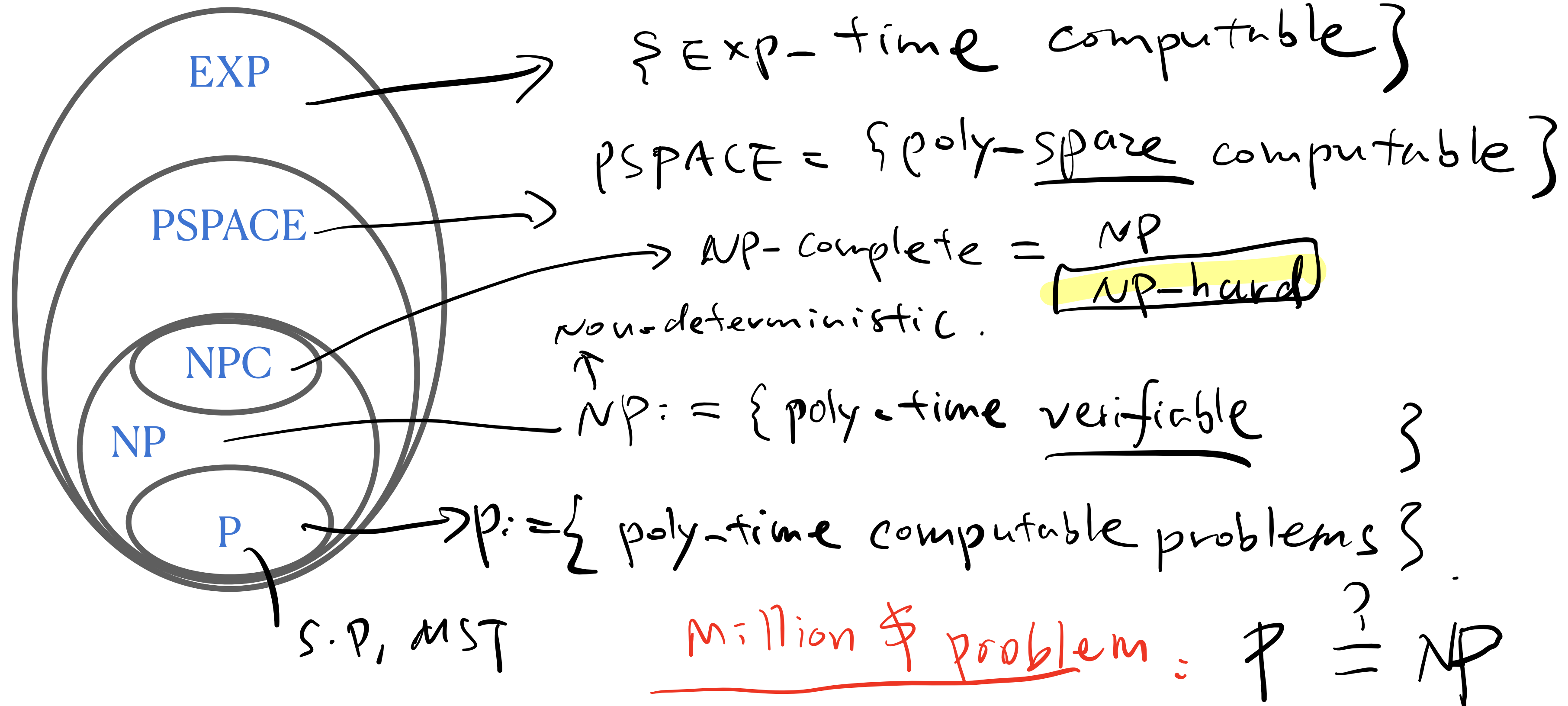
Easy vs. hard problems

Easy	Hard
Shortest path	Longest path
Minimum spanning tree	Travel salesperson
Sorting	3-coloring
...	...

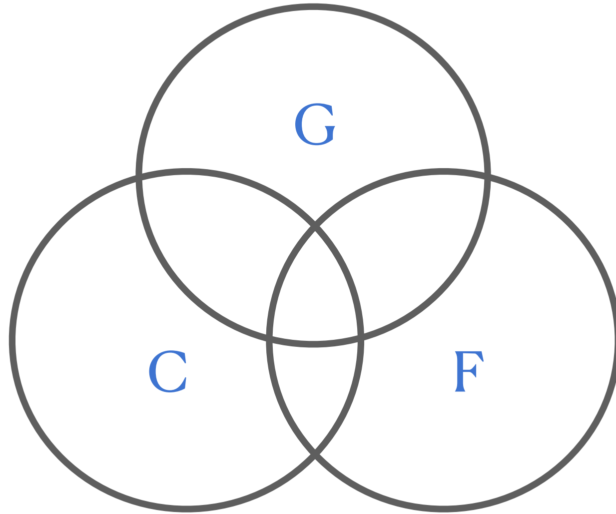
- **Goal: Classify problems according to the resources needed to solve them.**

Learn more at: https://complexityzoo.net/Complexity_Zoo

Complexity Zoo



Dealing with NP-hard problems



- General: no restrictions on inputs.
- Correctness: correct on every input.
- Fast: polynomial time on every input.

Current reality: pick two!

Scratch