Questions of the day

- Graph distance is the length of the shortest path between vertices

- How do we compute graph distance?
- Do we need to look at all paths and find the shortest?
- What if there are a lot of paths?
- What if the edges are different lengths?

Graph traversals and weighted graph algorithms

William Hendrix

Lecture 11

Outline

- Review
  - Graph algorithm analysis
  - More graph terminology
  - BFS
  - DFS
- Traversal complexity
- Traversal trees
- Applications of BFS/DFS
- Dijkstra's Algorithm
- Prim's Algorithm
- Kruskal's Algorithm

Graph algorithm analysis review

- May include graph features
  - n: # vertices
  - m: # edges
  - deg(v): degree of v
- Possibly consider graph operations separately

Operation	Adjacency matrix	Adjacency list	Edge list
Graph(n)	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$
AddEdge(u, v)	$\Theta(1)$	$\Theta(1)$*	$\Theta(m)$
RemoveEdge(u, v)	$\Theta(1)$	$\Theta(1)$*	$\Theta(m)$
IsAdjacent(u, v)	$\Theta(1)$	$\Theta(1)$*	$\Theta(\lg(m))$
GetNeighbors(v)	$\Theta(n)$	$\Theta(\deg v)$	$\Theta(\mathrm{m})$
Space:	$\Theta(n^2)$	$\Theta(m+n)$	$\Theta(m)$

- Handshaking Lemma: $\sum_{v \in V} \deg v = 2m = \Theta(m)$
  - All neighbors of all vertices = all edges

Graph algorithm exercise

$\binom{n}{3} \in O(n^3)$

```
Input: G = (V, E): graph with n vertices and m edges
Input: n, m: order and size of G
Output: number of triangles in G; that is, sets of 3
        vertices that are all adjacent
Algorithm: Triangles
tri = 0
for u = 1 to n do
    for v ∈ N(u) do
        for w = 1 to n do
            if w is adjacent to u and v then
                tri = tri + 1
            end
        end
    end
end
return tri/6
```

Operation	Adj. matrix	Opt. adj. list
IsAdjacent(u, v)	$\Theta(1)$	$\Theta(1)$*
GetNeighbors(v)	$\Theta(n)$	$\Theta(\deg(v))$*

GN: $\Theta(n)$

$\Theta(\deg u)$

n iters

deg u iters

n iters

$\Theta(n)$

$\Theta(1)$

$\Theta(\deg u)$

$\Theta(1)$

$\sum_{u \in V} n \deg u = n \deg v_1 + n \deg v_2 + \dots = n(\deg v_1 + \deg v_2 + \dots) = \Theta(nm)$

1. What is the worst case time complexity for Triangles when using an adjacency matrix?
2. What is the expected case time complexity for Triangles when using an adjacency list with hash tables? $\Theta(nm)$
3. How does this algorithm compare with the naïve algorithm of checking all sets of 3 vertices individually?

Graph algorithm exercise

Input: $G = (V, E)$: graph with n vertices and m edges
Input: n, m: order and size of G
Output: number of *triangles* in G; that is, sets of 3 vertices that are all adjacent

1 **Algorithm:** Triangles
2 $tri = 0$
3 **for** $u = 1$ to n **do**
4   **for** $v \in N(u)$ **do**
5   **for** $w = 1$ to n **do**
6   **if** w is adjacent to u and v **then**
7                 $tri = tri + 1$
8   **end**
9   **end**
10   **end**
11 **end**
12 **return** $tri/6$

Adj. matrix

$\Theta(1)$
n iters
$\deg(u)$ iters, $\Theta(n)$
n iters
$\Theta(1)$
$\Theta(1)$

Lines 5–9: $\Theta(n)$

Lines 4–10: $\Theta(n \deg(u))$

Lines 3–11:
$$\sum_{u=1}^{n} \Theta(n \deg(u)) = \Theta(nm)$$

$\Theta(1)$

Total time: $\Theta(nm)$

Adj. list

$\Theta(\deg(u))$ for GetNeighbors

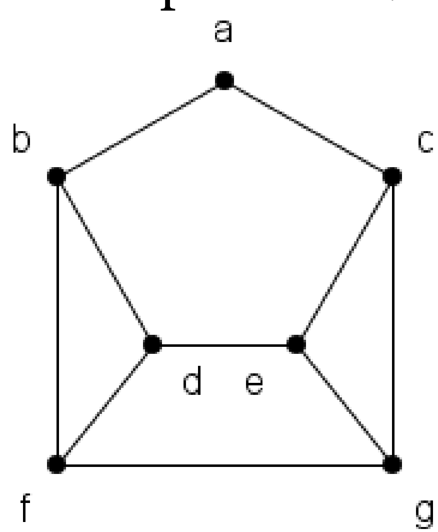
Everything else the same

Total time: $\Theta(nm)$

Graphs have $\binom{n}{3} = \Theta(n^3)$ sets of 3 vertices
This is strictly worse than $\Theta(nm)$

Graph traversal terminology

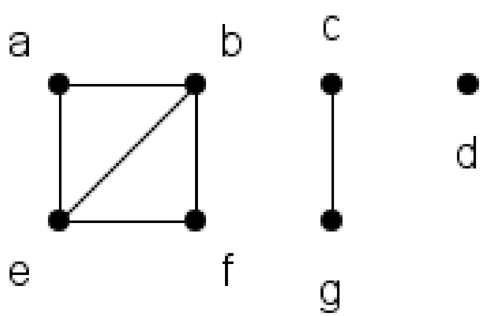
- **Walk:** sequence of vertices joined by edges
  - If directed, must obey directionality of edges
  - May include the same vertex or edge multiple times
  - **Length of a walk:** number of edges (vertices – 1)
    - If weighted, sum of edge weights
- **Path:** walk with no repeated vertices or edges
- **Circuit:** walk that begins and ends at the same vertex
  - A.k.a., "closed walk"
- **Cycle:** nontrivial path that begins and ends at the same vertex
- **Example**



Walk: b, d, e, g, f, g, c
Path: b, a, c, e, d, f, g
Circuit: f, d, e, c, a, b, d, e, g, f
Cycle: b, a, c, g, f, b

Graph connectivity terminology

- **Connected:** two vertices that have a path between them
  - Also, a graph in which all vertices are connected
  - **Theorem:** if there exists a (u, v)-walk, there exists a (u, v)-path
- **Component:** maximal set of connected vertices in a graph
  - Maximal = cannot be enlarged by adding more vertices
- **Distance:** length of the shortest path between two vertices
  - Only defined for connected vertices
  - Denoted d(u, v) or $d_G(u, v)$
  - Positive definite, symmetric (if undirected), triangle inequality
- **Example**



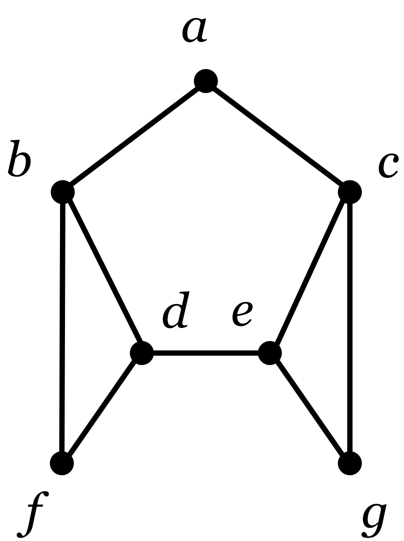
Connected: a and f (not adjacent!)
Components: {a, b, e, f}, {c, g}, {d}
Distance: d(a, f) = 2

Graph traversals

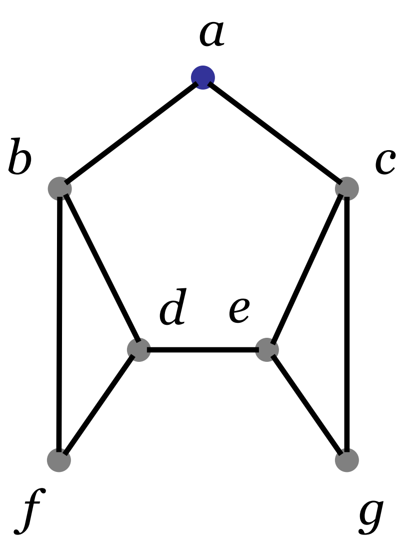
- Traversal: iterating through graph in connected order
  - Basis for many graph algorithms
  - Start at a specific vertex, iterate through all connected vertices
- Two main strategies: breadth-first and depth-first traversal
  - Both mark vertices in order as they go
- **Breadth-first search (BFS)**
  - Mark all vertices as undiscovered
  - Add v_o to queue
  - Mark v_o as discovered
  - While queue is not empty:
    - Dequeue vertex v
    - Enqueue children of v that are undiscovered
    - Mark v as explored
    - Mark children as discovered
  - If any vertex still undiscovered, choose one and restart

BFS example

- In what order would BFS process the vertices of the graph below when starting at vertex a?



BFS example

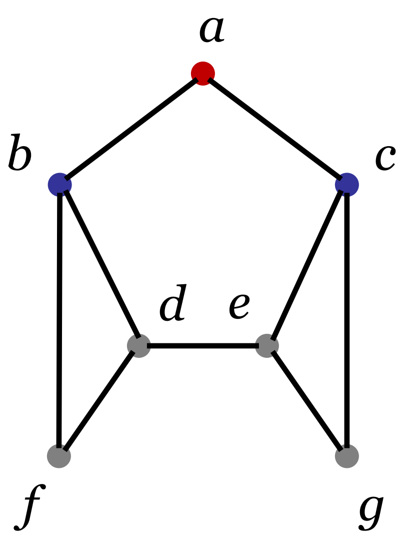


Undiscovered
Discovered
Explored

Queue: a

Traversal order:

BFS example

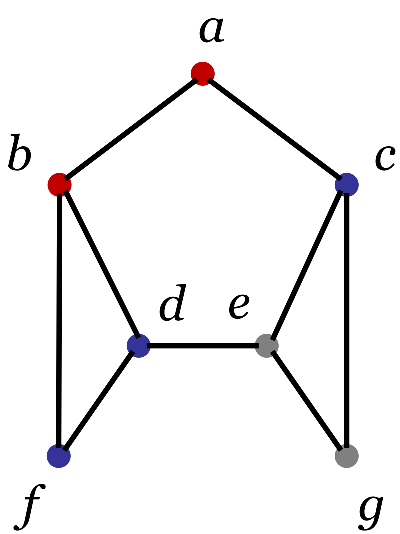


Undiscovered
Discovered
Explored

Queue: *b*, *c*

Traversal order: *a*

BFS example

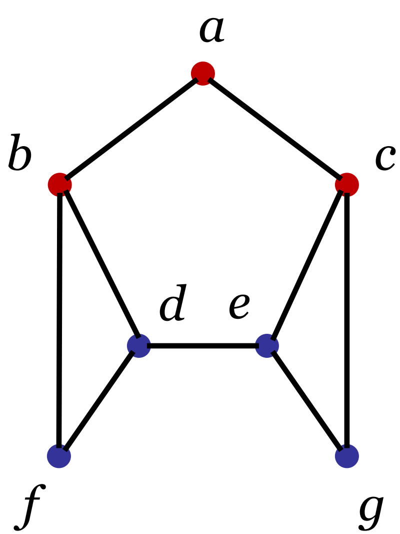


Undiscovered
Discovered
Explored

Queue: c, d, f

Traversal order: a, b

BFS example

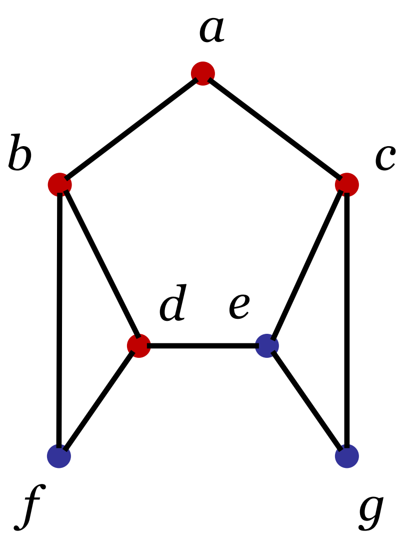


Undiscovered
Discovered
Explored

Queue: *d, f, e, g*

Traversal order: *a, b, c*

BFS example

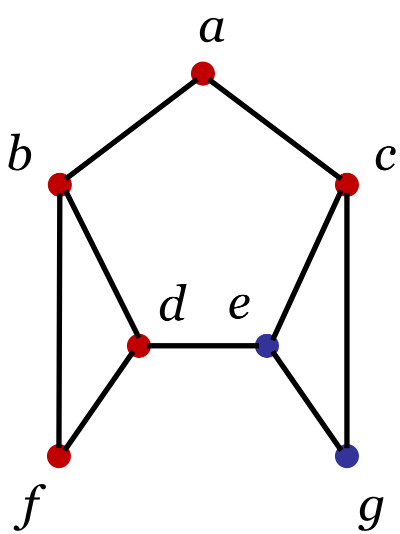


Undiscovered
Discovered
Explored

Queue: *f, e, g*

Traversal order: *a, b, c, d*

BFS example

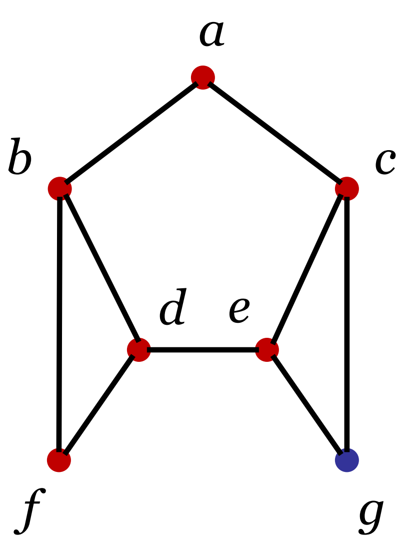


Undiscovered
Discovered
Explored

Queue: *e, g*

Traversal order: *a, b, c, d, f*

BFS example

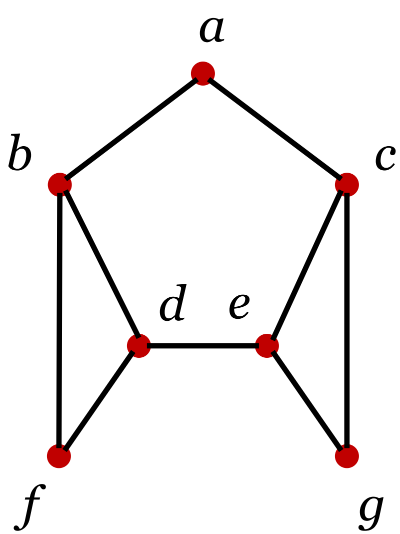


Undiscovered
Discovered
Explored

Queue: *g*

Traversal order: *a, b, c, d, f, e*

BFS example



Undiscovered
Discovered
Explored

Queue:

Traversal order: *a, b, c, d, f, e, g*

Depth-first search

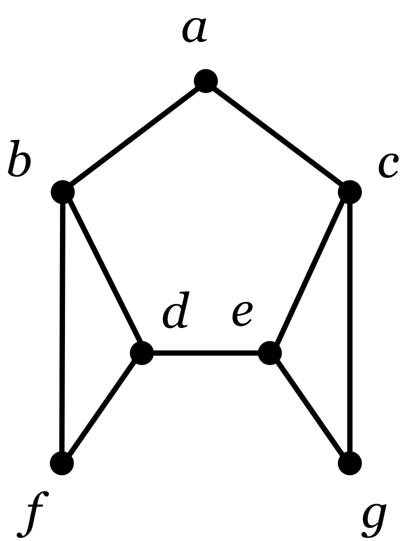
- Recursive algorithm
  - Simpler code than BFS
  - Implicitly stack-based

- DFS:
  - Call DFS-Recursive(v_0)
  - If any vertex still undiscovered, choose one and restart
- DFS-Recursive(v):
  - Mark v as discovered
  - For c in N(v):
    - If c is undiscovered
      - Call DFS-Recursive(c)
  - Mark v as explored

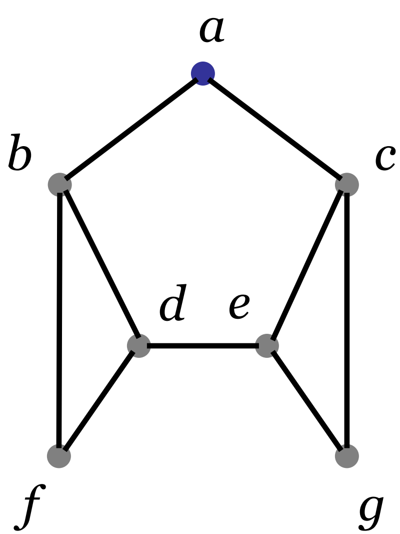
- **Observation:** vertices are discovered iff they are in stack/queue

DFS example

- In what order would DFS process the vertices of the graph below when starting at vertex a?



DFS example

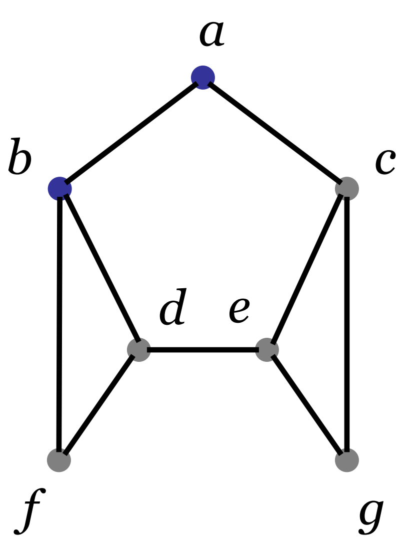


Undiscovered
Discovered
Explored

Call stack: a

Traversal order:

DFS example

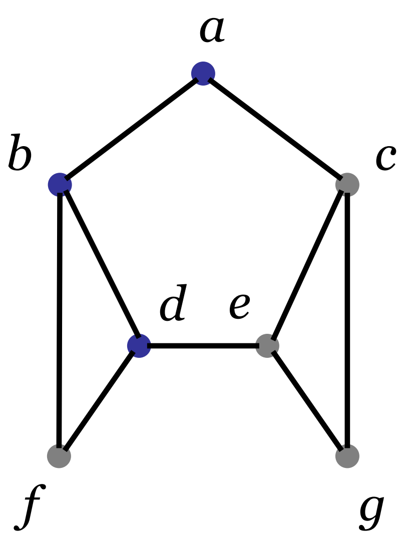


Undiscovered
Discovered
Explored

Call stack: a, b

Traversal order: a

DFS example

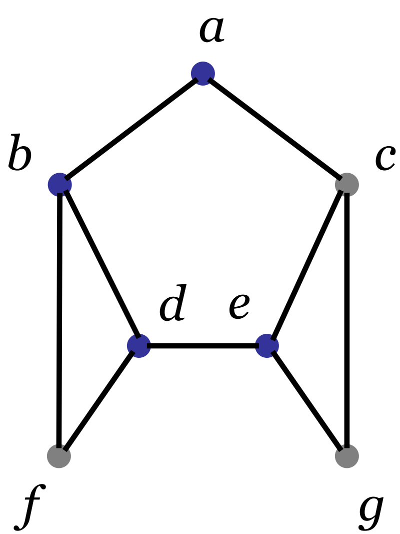


Undiscovered
Discovered
Explored

Call stack: *a, b, d*

Traversal order: *a, b*

DFS example

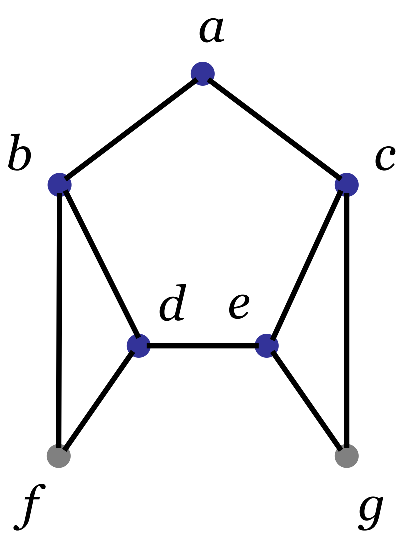


Undiscovered
Discovered
Explored

Call stack: *a, b, d, e*

Traversal order: *a, b, d*

DFS example

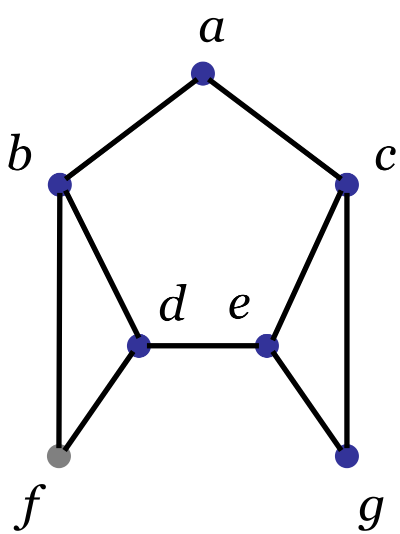


Undiscovered
Discovered
Explored

Call stack: *a, b, d, e, c*

Traversal order: *a, b, d, e*

DFS example

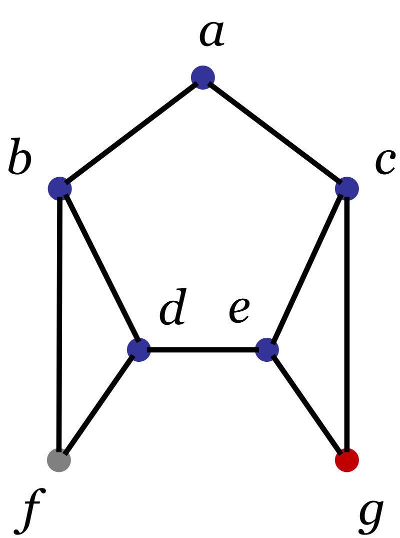


Undiscovered
Discovered
Explored

Call stack: a, b, d, e, c, g

Traversal order: a, b, d, e, c

DFS example

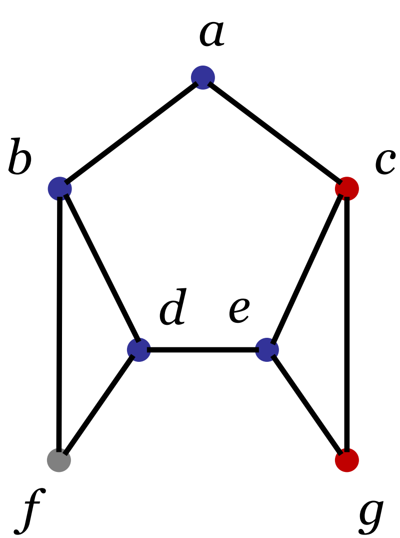


Undiscovered
Discovered
Explored

Call stack: *a, b, d, e, c*

Traversal order: *a, b, d, e, c, g*

DFS example

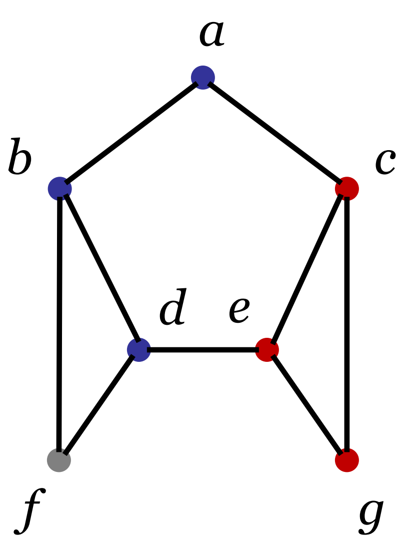


Undiscovered
Discovered
Explored

Call stack: *a, b, d, e*

Traversal order: *a, b, d, e, c, g*

DFS example

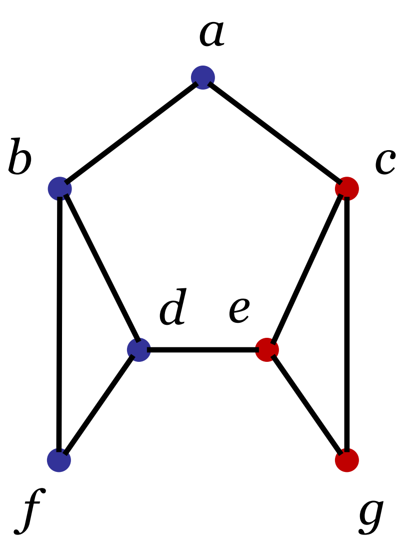


Undiscovered
Discovered
Explored

Call stack: a, b, d

Traversal order: a, b, d, e, c, g

DFS example

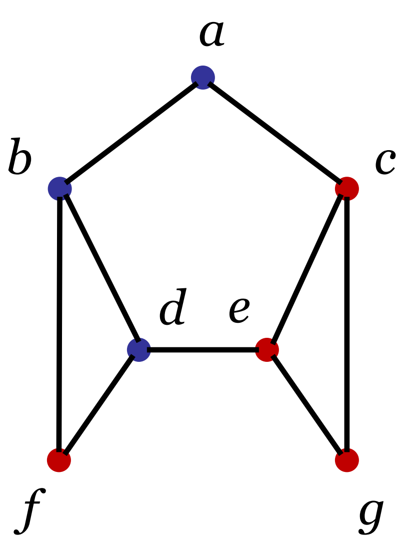


Undiscovered
Discovered
Explored

Call stack: *a, b, d, f*

Traversal order: *a, b, d, e, c, g, f*

DFS example

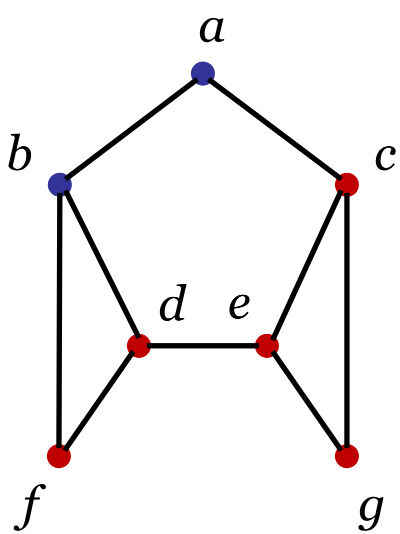


Undiscovered
Discovered
Explored

Call stack: *a, b, d*

Traversal order: *a, b, d, e, c, g, f*

DFS example



Undiscovered
Discovered
Explored

Call stack: a, b

Traversal order: a, b, d, e, c, g, f

Traversal complexity example

- What is the complexity of BFS with an adjacency list? Adjacency matrix?
  - How many times do we find the neighbors of a vertex?
  - How many total times do we increment count?

Total: $\Theta(n+m)$

$\Theta(n^2)$

Input: $G = (V, E)$: input graph with n vertices and m edges
Input: m, n: size and order of G
Mark all vertices of G as undiscovered $\Theta(n)$
Let q be a new queue
$count = 1$
$\Theta(1)$
repeat
  Add $V[count]$ to q $\Theta(1)$
  while $q \neq \emptyset$ **do**
    Get next vertex from q $\Theta(1)$
    `/* Process this vertex */`
    Mark this vertex as explored $\Theta(1)$
    Add all undiscovered neighbors of this vertex to q and mark as discovered $\Theta(\deg v)$
  end
  Increase $count$ until $V[count]$ is undiscovered
until $count > n$
return

$\Theta(n^2)$

$\Theta(n)$

$\sum_{v \in V} \deg v : \Theta(m)$

n iters $\sim n$
$\Rightarrow \Theta(n)$

Traversal complexity example

- What is the complexity of BFS with an adjacency list? Adjacency matrix?
  - How many times do we find the neighbors of a vertex?
  - How many total times do we increment count?

```
Input: G = (V, E): input graph with n vertices
       and m edges
Input: m, n: size and order of G
Mark all vertices of G as undiscovered
Let q be a new queue
count = 1
repeat
    Add V[count] to q
    while q ≠ ∅ do
        Get next vertex from q
        /* Process this vertex                */
        Mark this vertex as explored
        Add all undiscovered neighbors of this
         vertex to q and mark as discovered
    end
    Increase count until V[count] is undiscovered
until count > n
return
```

Adjacency list:

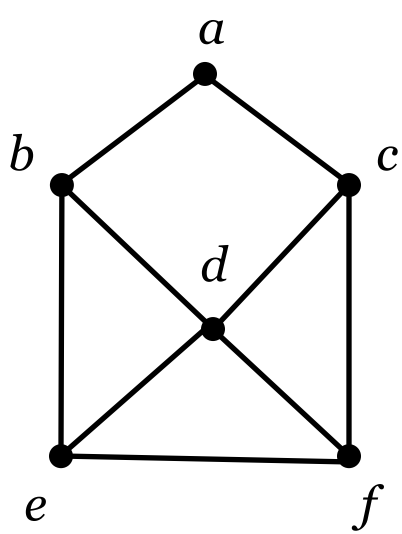
- Mark all vertices: $\Theta(n)$
- Loop iterates once per vertex: $\Theta(n)$
- Call getNeighbors on all vertices once: $\Theta(m)$
- *count* incremented n times: $\Theta(n)$
- ***Total:*** $\Theta(n+m)$

Adjacency matrix:

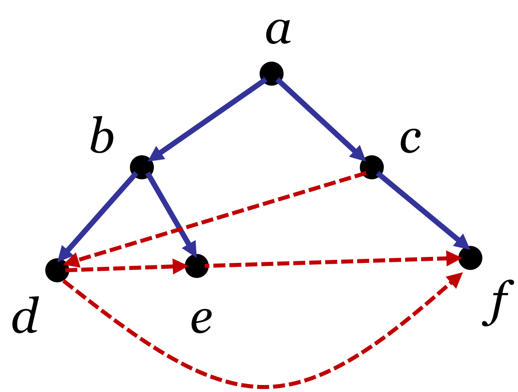
- Mark all vertices: $\Theta(n)$
- Loop iterates once per vertex: $\Theta(n)$
- Call getNeighbors on all vertices once: $\Theta(n^2)$
- *count* incremented n times: $\Theta(n)$
- ***Total:*** $\Theta(n^2)$

BFS and DFS traversal trees

- Traversal tree
  – Set of edges traversed by BFS or DFS,
  – Augmented with other edges of graph

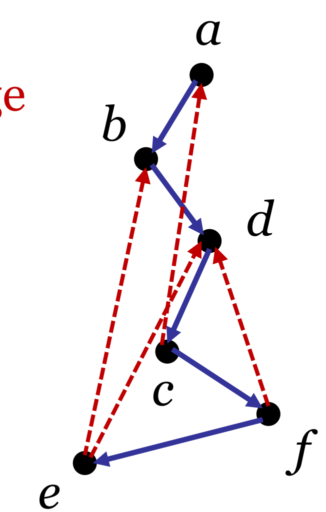


BFS



Tree edge
Non-tree edge

DFS

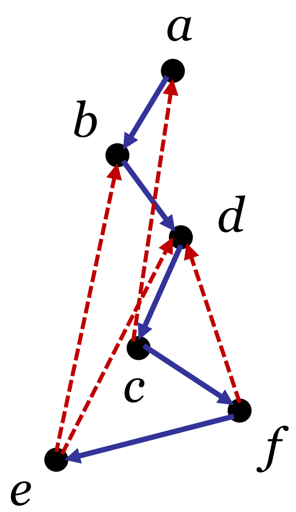


- Non-tree edges are siblings, cousins, or nieces/nephews
  – “Cross-edges”

- Non-tree edges are ancestors
  – “Back-edges”

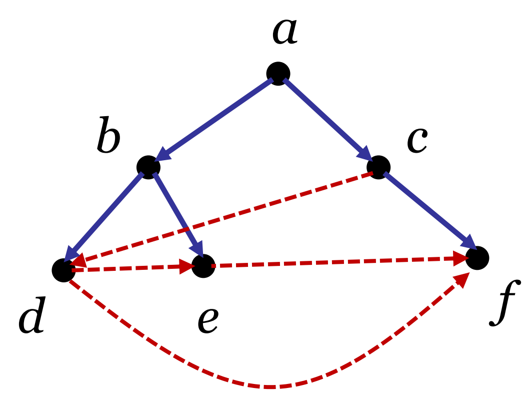
Traversal tree structure

DFS



Tree edge
Non-tree edge

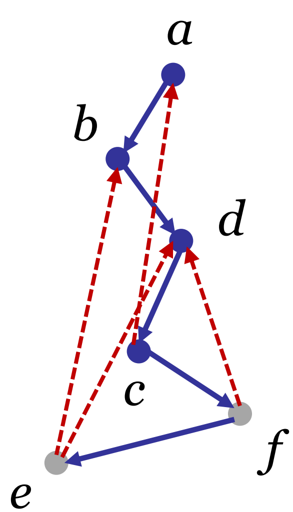
BFS



- **Observation 1:** non-tree edges connect to discovered vertices
  - Tree edges are undiscovered
- **Observation 2:** vertices are discovered iff they are in stack/queue

Traversal tree structure

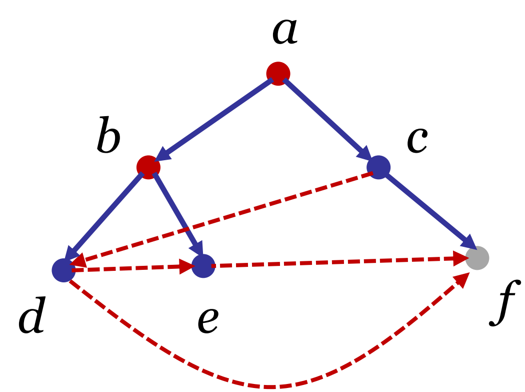
DFS



Tree edge
Non-tree edge

undiscovered
discovered
explored

BFS



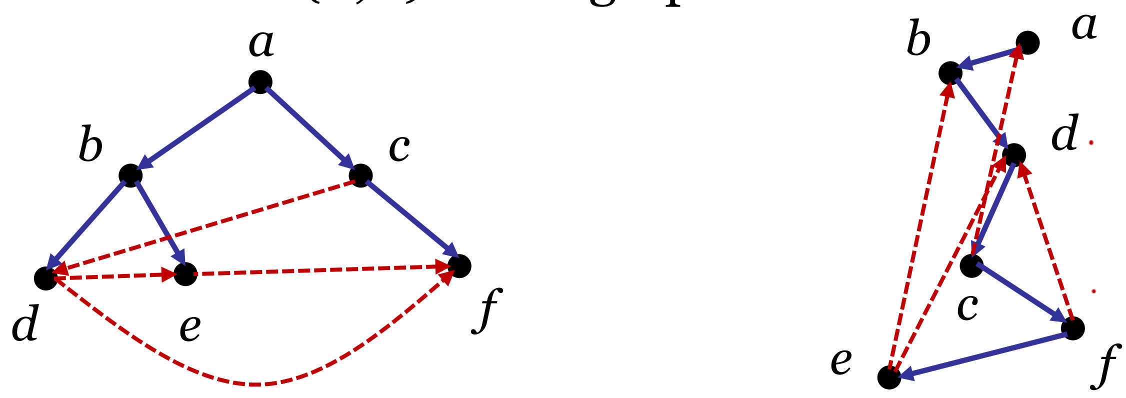
- **Observation 1:** non-tree edges connect to discovered vertices
  - Tree edges are undiscovered
- **Observation 2:** vertices are discovered iff they are in stack/queue
  - Stack: all vertices back to root
  - Queue: remaining vertices on same level + vertices on next level
- DFS: non-tree-edges are ancestors (“back edges”)
- BFS: non-tree-edges are on same or next level (“cross edges”)

Graph traversal example

- *Recall:* length of the *shortest* path between two vertices
  - In undirected graph, fewest number of edges

1. Should we use BFS or DFS? Why?

- *Hint:* what is *d(a, e)* in the graph below?



2. Prove that *d(a, v)* equals the level of *v* in a BFS traversal tree starting at *a*, for every vertex *v*.

Graph traversal example

1. BFS, because *d(a, v)* equals the level of *v* in the BFS traversal tree starting at *a*

2. *Proof.* We use induction on n to prove that $d(a,v) = n$ if and only if v is on level n of the BFS traversal tree starting at a.

   (*Base case*) When v is on level 0 of the tree, $v = a$, and $d(a,a) = 0$. When $d(a,v) = 0$, $v = a$, so v must be on level 0 of the tree.

   (*Inductive step*) Suppose that $d(a,u) = k$ for every vertex u on level k of the tree and that every vertex with distance k to a is on level k, and suppose that vertex v is on level $k+1$ of the tree. Since v is on level $k+1$ of the tree, its parent p must be on level k. By the inductive hypothesis, $d(a,p) = k$, so there must be some path $P : a, v_1, v_2, \ldots, v_{k_1}, p$ of length k from a to p. The sequence $Q : a, v_1, v_2, \ldots, v_{k-1}, p, v$ must be a path from a to v of length $k+1$, so $d(a,v) \leq k+1$. Moreover, there can't be any shorter path, as v cannot have any neighbor higher than level k in the tree.

   Conversely, if a $d(a,v) = k+1$, then there is some path $P : a, v_1, v_2, \ldots, v_k, v$ of length $k+1$ from a to v. The second-to-last vertex on this path, v_k, must be distance k to a, so it is on level k. Since v is adjacent to a vertex on level k, it must be on levels $k-1$ to $k+1$, but it can't be on level $k-1$ or k because these levels have distance $k-1$ and k to a, respectively. Thus, v must be on level $k+1$ of the BST traversal tree. $\square$

Traversal exercise

Example: 

- Design an algorithm to label each vertex with a *component ID* such that all vertices in the same connected component have the same ID.
  - What is the runtime of your algorithm when using an adjacency list?

BFS:
Initialize ID and component
component[v_0] = ID
Mark v_0 discovered
Enqueue v_0
While queue not empty:
    Dequeue vertex v
    Mark v explored
    component[v] = ID
    Mark children discovered
    Enqueue children
If any vertex still undiscovered:
    ID++
    Choose one and repeat

DFS:
ID = 0
Let component be an array-based map
While some vertices undiscovered:
    v = next undiscovered vertex
    ID++
    DFS-Rec(v)

DFS-Rec(v):
Mark v as discovered
component[v] = ID
For undiscovered neighbors u:
    DFS-Rec(u)

Traversal exercise

- Design an algorithm to label each vertex with a *component ID* such that all vertices in the same connected component have the same ID.
  - What is the runtime of your algorithm when using an adjacency list?

- **ComponentDFS:**

  components = 0

  While vertices still undiscovered

    components = components + 1

    DFS-Rec(v) //label all vertices with ID components

  Return components

- Can also be done w/ BFS
  - Increment components every time you "repeat" (last line)
- Complexity: $\Theta(n + m)$

BFS vs. DFS

Both BFS and DFS:

- Traverse one component at a time
- Take $\Theta(n+m)$ time
- Traversal trees have special properties
  - Sometimes critical, sometimes unimportant

- DFS is somewhat easier to implement
- Usage depends on application

BFS and DFS application examples

Both:

- Component detection
- Cycle detection

BFS:

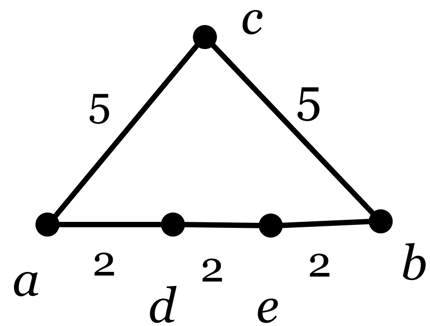
- Unweighted distance
- Radius and center of unweighted graph
- Betweenness centrality
  - Ranks vertices by “importance”
  - Counts shortest paths that pass through a vertex

DFS:

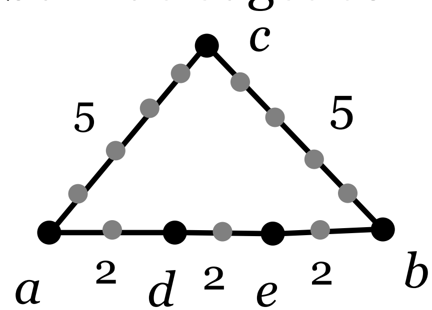
- Cut edge or cut vertex detection
- Sorting (on BST)
- Topological sorting
  - Order vertices of Directed Acyclic Graph (DAG) so all edges point to right
- *Backtracking*

Weighted graph distance

- Edge weights may represent distance or strength of connection
  - E.g., road network or chemical interactions
- Distance: sum of edge weights
- BFS doesn't necessarily find shortest path:



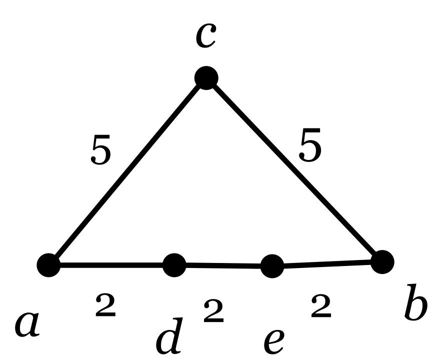
  - Finds path with fewest edges
- Naïve strategy: subdivide edges so that all edges have same length



- Finds shortest path
- Complexity depends on edge weights and GCD

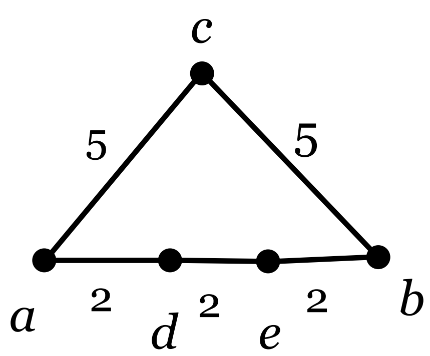
A better solution

- Don't actually add "interstitial" vertices
- Process vertices in same order as naïve strategy
  - Store when we would reach each vertex



A better solution

- Don't actually add "interstitial" vertices
- Process vertices in same order as naïve strategy
  - Store when we would reach each vertex



- t=0: *a*
  - Reach *c* at 5, *d* at 2
- t=2: *d*
  - Reach *e* at 4
- t=4: *e*
  - Reach *b* at 6
- t=5: c
  - Reach *b* at 10 (worse than projected)
- t=6: *b*

Dijkstra's algorithm

- Pseudocode

Input: $G = (V, E)$: graph in question
Input: a, b: vertices to measure distance between
Input: n, m: order and size of G
Label all vertices as unreachable except a
$v = a$, $D = 0$
while $v \neq b$ **do**
    for $u \in N(v)$ **do**
        Label u with distance $D + d(v, u)$ unless u has a smaller label
    end
    Choose reachable vertex v with min distance D
end
return D

  - To return the path, add a back-link every time you label u
  - Traverse back-links from b to a

- Greedy algorithm
  - Always selects vertex with min distance

Dijkstra's analysis

Input: $G = (V, E)$: graph in question
Input: a, b: vertices to measure distance between
Input: n, m: order and size of G

Label all vertices as unreachable except a
$v = a$, $D = 0$
while $v \neq b$ **do**
    for $u \in N(v)$ **do**
        Label u with distance $D + d(v, u)$ unless u has a smaller label
    end
    Choose reachable vertex v with min distance D
end
return D

Dijkstra's analysis

Complexity	
	Input: $G = (V, E)$: graph in question
	Input: a, b: vertices to measure distance between
	Input: n, m: order and size of G
$\Theta(n)$	Label all vertices as unreachable except a
$\Theta(1)$	$v = a$, $D = 0$
	while $v \neq b$ **do**
$\Theta(\deg(v))$	**for** $u \in N(v)$ **do**
$\Theta(1)^*$	Label u with distance $D + d(v, u)$ unless u has a smaller label
	end
	Choose reachable vertex v with min distance D
	end
$\Theta(1)$	**return** D

Dijkstra's analysis

Complexity	
	Input: $G = (V, E)$: graph in question
	Input: a, b: vertices to measure distance between
	Input: n, m: order and size of G
$\Theta(n)$	1 Label all vertices as unreachable except a
	2 $v = a$, $D = 0$
	3 **while** $v \neq b$ **do**
$\Theta(\deg(v))^*$	4 **for** $u \in N(v)$ **do**
	5 Label u with distance $D + d(v, u)$ unless u has a smaller label
	6 **end**
Depends	7 Choose reachable vertex v with min distance D
	8 **end**
$\Theta(1)$	9 **return** D

- Line 7: depends on implementation
  - Also affects line 5
- Unsorted array:
  - Min is $\Theta(n)$
- Heap:
  - Min is $\Theta(\lg n)$
  - Insert is $\Theta(\lg n)$
  - Decrease is $\Theta(\lg n)$

Dijkstra's analysis

<u>Complexity</u>	
	Input: $G = (V, E)$: graph in question
	Input: a, b: vertices to measure distance between
	Input: n, m: order and size of G
$\Theta(n)$	Label all vertices as unreachable except a
	$v = a$, $D = 0$
	while $v \neq b$ **do**
$\Theta(\deg(v) \lg n)$	**for** $u \in N(v)$ **do**
	Label u with distance $D + d(v, u)$ unless u has a smaller label
	end
$\Theta(\lg n)$	Choose reachable vertex v with min distance D
	end
$\Theta(1)$	**return** D

Dijkstra's analysis

Complexity	
	Input: $G = (V, E)$: graph in question
	Input: a, b: vertices to measure distance between
	Input: n, m: order and size of G
$\Theta(n)$	Label all vertices as unreachable except a
	$v = a$, $D = 0$
$O(n)$ iters	**while** $v \neq b$ **do**
$\Theta(\deg(v) \lg n)$	**for** $u \in N(v)$ **do**
	Label u with distance $D + d(v, u)$ unless u has a smaller label
	end
	Choose reachable vertex v with min distance D
	end
$\Theta(1)$	**return** D

Dijkstra's analysis

Complexity

Input: $G = (V, E)$: graph in question
Input: a, b: vertices to measure distance between
Input: n, m: order and size of G

$\Theta(n)$

Label all vertices as unreachable except a
$v = a$, $D = 0$

$O\left(\sum_{v \in V} \deg(v) \lg n\right) = O(m \lg n)$

while $v \neq b$ **do**
    for $u \in N(v)$ **do**
        Label u with distance $D + d(v, u)$ unless u has a smaller label
    end
    Choose reachable vertex v with min distance D
end

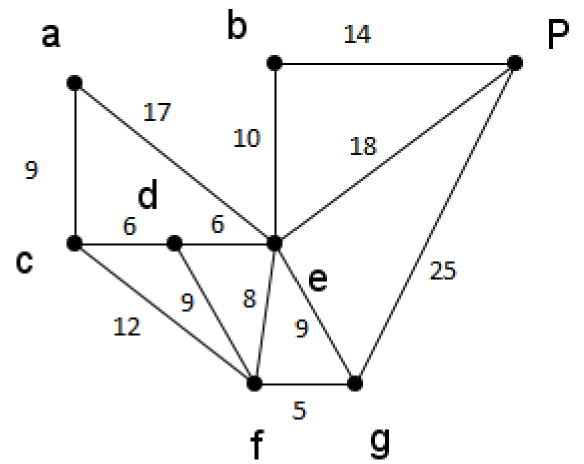
$\Theta(1)$

return D

- Time complexity: $O(n + m \lg n)$
- Space complexity
  - Dominated by graph
  - $\Theta(n + m)$
- Same complexity to return *(a, b)*-path
  - Add parent links when you update labels
  - Follow links backwards from goal
- Same complexity to find distance to every other vertex

Minimum spanning tree

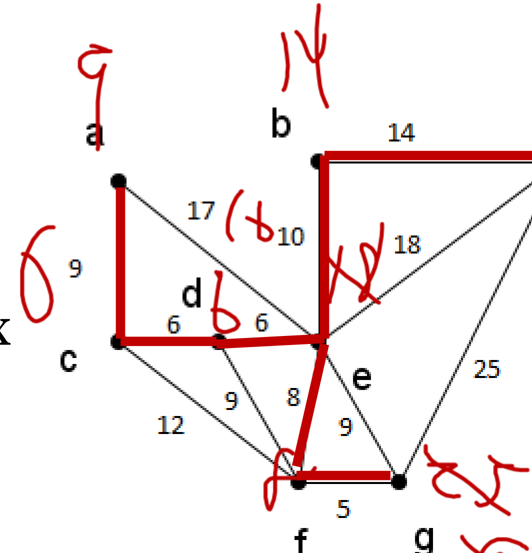
- Consider the following scenario:
  - Imagine you are a creating a new power company
  - Need to connect customers to your power plant
  - Need to minimize costs
- **Example**
  - Connect *P* to *a*-*g*:

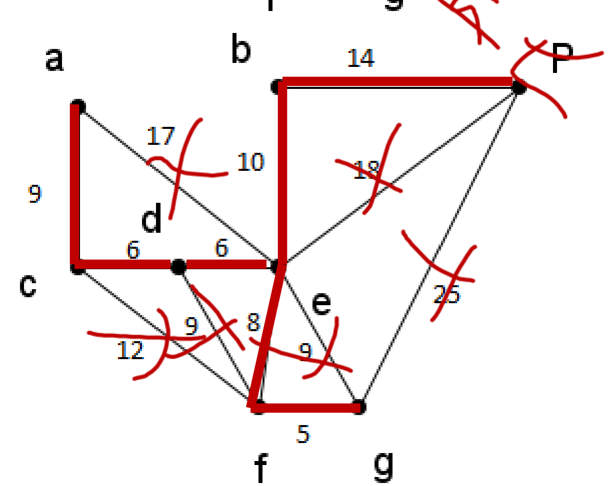


- Subgraph of a connected weighted graph that:
  - Connects all vertices of the graph (*spans*)
  - Has no cycles (*tree*)
  - Has the lowest edge weight sum among all spanning trees

Computing the MST

- Two main algorithms
  - Both greedy
- Prim's algorithm
  - Starts at one vertex
  - Adds edge to nearest unconnected vertex
  - Repeat until spanning
- Kruskal's algorithm
  - Sort edges
  - Adds min edge between disconnected vertices
  - Repeat until connected





Prim's analysis

Input: $G = (V, E)$: graph in which to find MST
Input: n, m: order and size of G
Output: Minimum spanning tree of G

$edge = \text{Array}(n)$
Let *heap* be a heap where every vertex has value ∞
Choose v_0 and decrease its value in *heap* to 0
for $i = 1$ to n **do**
    $v = heap.\text{DeleteMin}()$
    If $v \neq v_0$, add $(v, edge[v])$ to MST
    for $u \in N(v)$ **do**
        if $u \in heap$ and $d(u, v) < \text{value}(u)$ **then**
            Decrease value of u to $d(u, v)$ in *heap*
            $\text{edge}[u] = v$
        end
    end
end
return MST

Prim's analysis

Complexity	
	Input: $G = (V, E)$: graph in which to find MST
	Input: n, m: order and size of G
	Output: Minimum spanning tree of G
$\Theta(1)$	$edge = \text{Array}(n)$
$\Theta(n)$	Let *heap* be a heap where every vertex has value ∞
$\Theta(\lg n)$	Choose v_0 and decrease its value in *heap* to 0
	for $i = 1$ to n **do**
	$v = heap.\text{DeleteMin}()$
	If $v \neq v_0$, add $(v, edge[v])$ to MST
	for $u \in N(v)$ **do**
$\Theta(1)$	**if** $u \in heap$ and $d(u, v) < \text{value}(u)$ **then**
$\Theta(\lg n)$	Decrease value of u to $d(u, v)$ in *heap*
$\Theta(1)$	$\text{edge}[u] = v$
	end
	end
	end
$\Theta(1)$	**return** MST

Prim's analysis

<u>Complexity</u>	
	Input: $G = (V, E)$: graph in which to find MST
	Input: n, m: order and size of G
	Output: Minimum spanning tree of G
$\Theta(1)$	$edge = \text{Array}(n)$
$\Theta(n)$	Let *heap* be a heap where every vertex has value ∞
$\Theta(\lg n)$	Choose v_0 and decrease its value in *heap* to 0
	for $i = 1$ to n **do**
	$v = heap.\text{DeleteMin}()$
	If $v \neq v_0$, add $(v, edge[v])$ to MST
$\Theta(\deg(v))$ iters	**for** $u \in N(v)$ **do**
$\Theta(\lg n)$	**if** $u \in heap$ and $d(u, v) < \text{value}(u)$ **then**
	Decrease value of u to $d(u, v)$ in *heap*
	$\text{edge}[u] = v$
	end
	end
	end
$\Theta(1)$	**return** MST

Prim's analysis

Complexity	
	Input: $G = (V, E)$: graph in which to find MST
	Input: n, m: order and size of G
	Output: Minimum spanning tree of G
$\Theta(1)$	$edge = \text{Array}(n)$
$\Theta(n)$	Let *heap* be a heap where every vertex has value ∞
$\Theta(\lg n)$	Choose v_0 and decrease its value in *heap* to 0
$\Theta(n)$ iters	**for** $i = 1$ to n **do**
$\Theta(\lg n)$	$v = heap.\text{DeleteMin}()$
$\Theta(1)$	If $v \neq v_0$, add $(v, edge[v])$ to MST
$\Theta(\deg(v) \lg n)$	**for** $u \in N(v)$ **do**
	if $u \in heap$ and $d(u, v) < \text{value}(u)$ **then**
	Decrease value of u to $d(u, v)$ in *heap*
	$\text{edge}[u] = v$
	end
	end
	end
$\Theta(1)$	**return** MST

Prim's analysis

Complexity	
	Input: $G = (V, E)$: graph in which to find MST
	Input: n, m: order and size of G
	Output: Minimum spanning tree of G
$\Theta(1)$	$\mathit{edge} = \text{Array}(n)$
$\Theta(n)$	Let *heap* be a heap where every vertex has value ∞
$\Theta(\lg n)$	Choose v_0 and decrease its value in *heap* to 0
$\Theta(m \lg n)$	**for** $i = 1$ to n **do**
	$v = \mathit{heap}.\text{DeleteMin}()$
	If $v \neq v_0$, add $(v, \mathit{edge}[v])$ to MST
	for $u \in N(v)$ **do**
	if $u \in \mathit{heap}$ and $d(u, v) < \text{value}(u)$ **then**
	Decrease value of u to $d(u, v)$ in *heap*
	$\text{edge}[u] = v$
	end
	end
	end
$\Theta(1)$	**return** MST

Total: $\Theta(n + m \lg n)$

Kruskal's analysis

Input: $G = (V, E)$: graph in which to find MST
Input: n, m: order and size of G
Output: Minimum Spanning Tree of G

```
E' = sort(E)
uf = UnionFind(n)
for i = 1 to m do
    (u, v) = E'[i]
    if uf.Find(u) ≠ uf.Find(v) then
        Add (u, v) to MST
        uf.Union(u, v)
    end
end
return MST
```

Kruskal's analysis

Complexity	
	Input: $G = (V, E)$: graph in which to find MST
	Input: n, m: order and size of G
	Output: Minimum Spanning Tree of G
$\Theta(m \lg m)$	$E' = \text{sort}(E)$
$\Theta(n)$	uf =UnionFind(n)
	for $i = 1$ to m **do**
$\Theta(1)$	$(u, v) = E'[i]$
$\Theta(\alpha(n))$	**if** uf.Find(u) $\neq$ uf.Find(v) **then**
$\Theta(1)$	Add (u, v) to MST
$\Theta(\alpha(n))$	uf.Union(u, v)
	end
	end
$\Theta(1)$	**return** MST

Kruskal's analysis

Complexity	
	Input: $G = (V, E)$: graph in which to find MST
	Input: n, m: order and size of G
	Output: Minimum Spanning Tree of G
$\Theta(m \lg m)$	$E' = \text{sort}(E)$
$\Theta(n)$	$uf = \text{UnionFind}(n)$
$\Theta(m)$	**for** $i = 1$ to m **do**
$\Theta(\alpha(n))$	$\quad (u, v) = E'[i]$
	$\quad$ **if** $uf.\text{Find}(u) \neq uf.\text{Find}(v)$ **then**
	$\quad\quad$ Add (u, v) to MST
	$\quad\quad uf.\text{Union}(u, v)$
	$\quad$ **end**
	end
$\Theta(1)$	**return** MST

Kruskal's analysis

Input: $G = (V, E)$: graph in which to find MST
Input: n, m: order and size of G
Output: Minimum Spanning Tree of G

Complexity	
$\Theta(m \lg m)$	$E' = \text{sort}(E)$
$\Theta(n)$	$uf = \text{UnionFind}(n)$

$\Theta(m\alpha(n))$	**for** $i = 1$ to m **do**
	    $(u, v) = E'[i]$
	    if $uf.\text{Find}(u) \neq uf.\text{Find}(v)$ **then**
	        Add (u, v) to MST
	        $uf.\text{Union}(u, v)$
	    end
	end

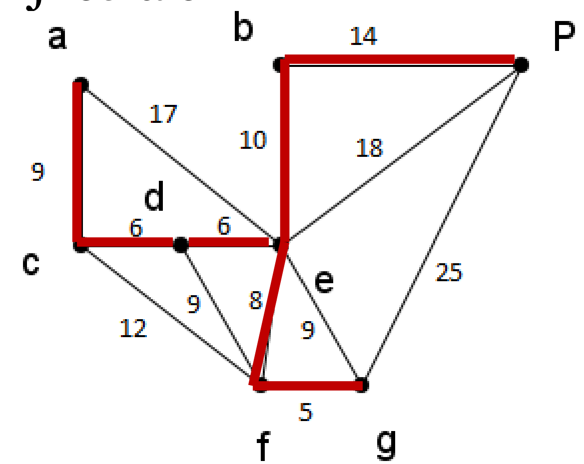
$\Theta(1)$	**return** MST

Total complexity: $\Theta(n + m \lg m)$

MST analysis

- Prim's vs. Kruskal's
  - Prim's: $\Theta(n + m \lg n)$
  - Kruskal's: $\Theta(n + m \lg m)$
  - Asymptotically equivalent
    - Prim's typically faster for dense graphs
    - Similar time for sparse graphs
  - Linear space for both

- Note: Prim's algorithm very similar to Dijkstra's
- BUT: MST ≠ min distance



- MST Applications
  - Connecting spatial points
  - Travelling Salesman Problem
  - Clustering

Coming up

- Weighted graph algorithms
- Backtracking

- **Recommended readings:** Sections 13.1-13.3, 14.1-14.2, and 15.1-15.3
- *Practice problems:* R-13.1, R-13.2, R-13.5, R-13.7, C-13.4, R-14.2, R-14.3, C-14.3, C-14.4, A-14.6, R-15.9, C-15.1, C-15.4, A-15.6