Unit #9: Graphs

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

Unit Outline

- Topological Sort: Sorting vertices
- Graph ADT and Graph Representations
- Graph Terminology
- More Graph Algorithms
  - Shortest Path (Dijkstra's Algorithm)
  - Minimum Spanning Tree (Kruskal's Algorithm)

use:

DFS .. stack

BFS .. queue

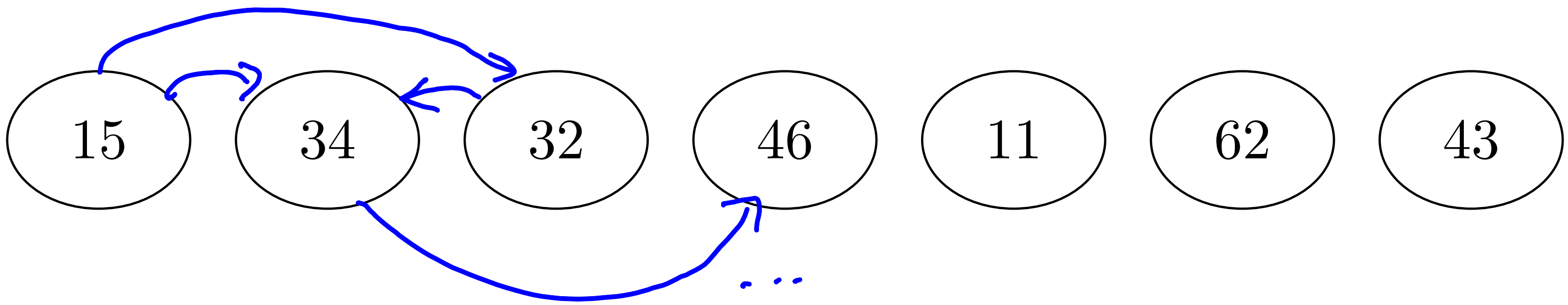
"smart FS" .. priority queue

Learning Goals

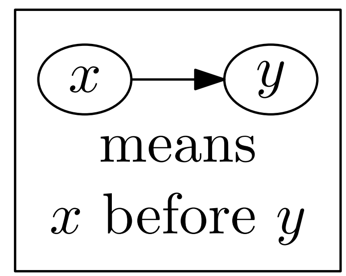
- Describe the properties and possible applications of various kinds of graphs (e.g., simple, complete), and the relationships among vertices, edges, and degrees.
- Prove basic theorems about simple graphs (e.g. handshaking theorem).
- Convert between adjacency matrices/lists and their corresponding graphs.
- Determine whether two graphs are isomorphic.
- Determine whether a given graph is a subgraph of another.
- Perform breadth-first and depth-first searches in graphs.
- Execute Dijkstra's shortest path and Kruskal's minimum spanning tree algorithms on a given graph.

Sorting Total Orders

Insertion sort:



comparison graph



total order

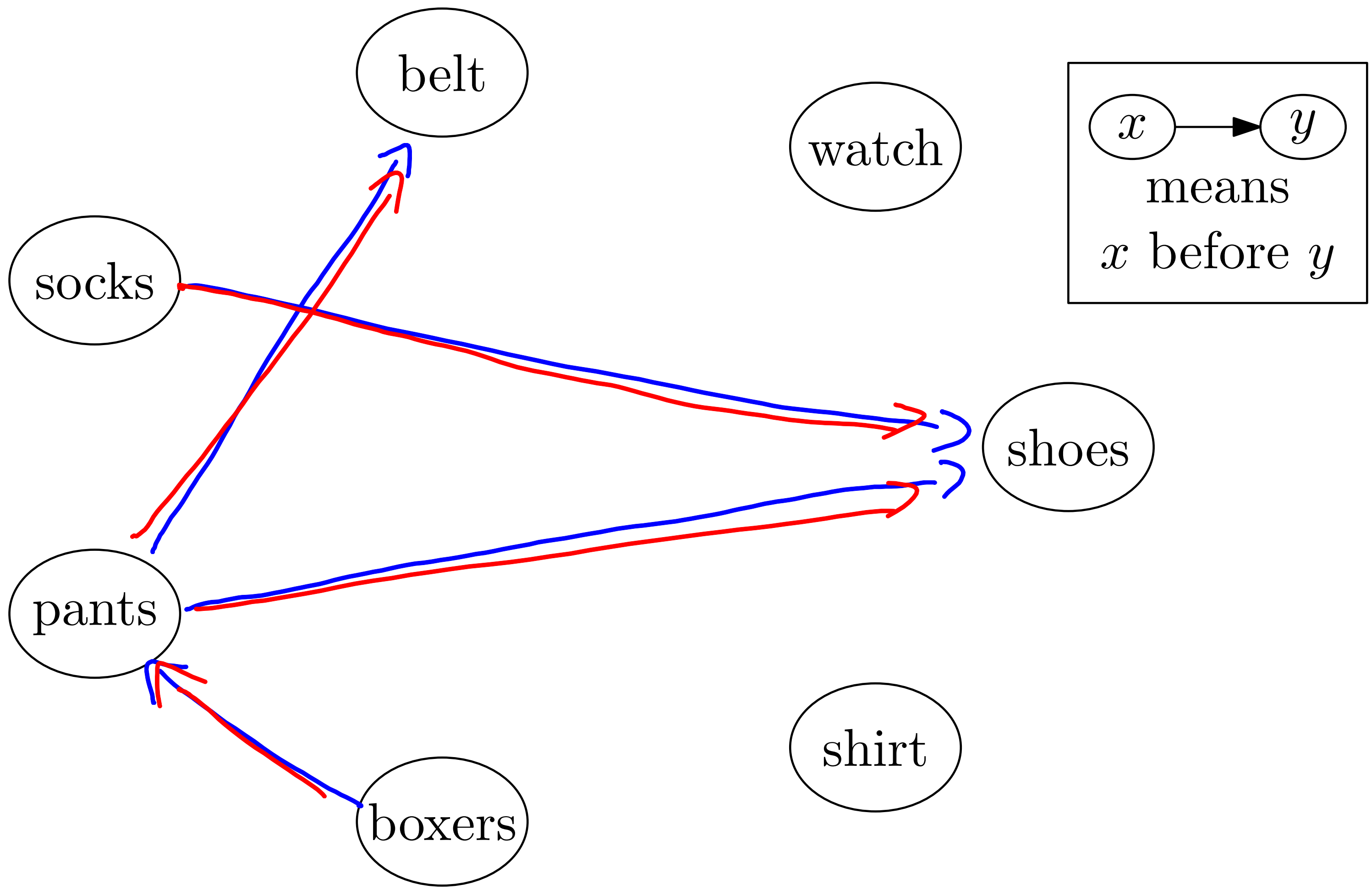
What property does the comparison-based sorting algorithm need to achieve?

- For every 2 elements there is a directed path between them

$\Longleftrightarrow$

- $\exists$ directed path containing all elements

Partial Order: Getting Dressed



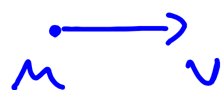
? total order (that satisfies given edges/constraints)

Example: boxers, watch, socks, shirt, pants, belt

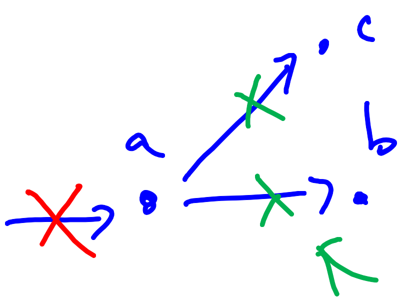
Topological Sort

directed acyclic

A topological sort is a total order of the vertices of a graph $G = (V, E)$ such that if (u, v) is an edge of G then u appears before v in the order.



- Start with a vertex with no incoming edges ("a")



print "a" and remove edges from "a"

and repeat

Topological Sort Algorithm I

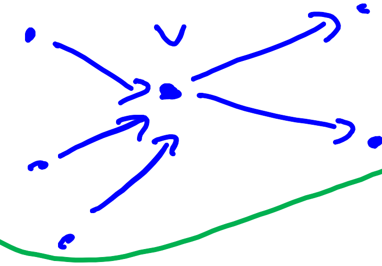
$n = \#$ vertices

$m = \#$ edges

def:

in degree(v) = 3

out degree(v) = 2

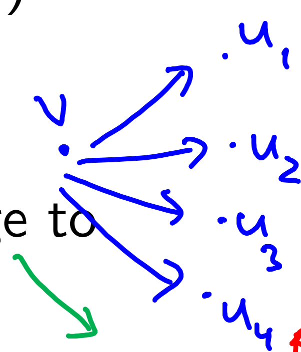


1. Find each vertex's *in-degree* (# of inbound edges) $\therefore \Theta(n)$
2. While there are vertices remaining
   - 2.1 Pick a vertex v with in-degree zero and output it
   - 2.2 Reduce the in-degree of all vertices it has an edge to
   - 2.3 Remove it from the list of vertices

n iterations

# of remaining vertices

$O(n)$



in-degree--

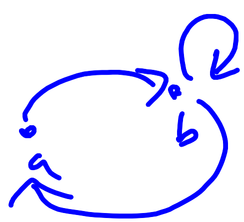
$O(n)$

out-degree(u)

Runtime? $\Theta(n) + \Theta(n)\cdot(O(n) + O(n) + \Theta(1))$

$\Theta(1)$

$= \Theta(n^2)$



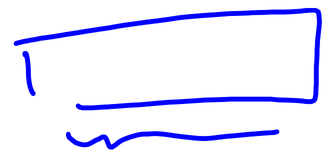
$0 \le m \le n^2$

$O(n^2)$

possible # of edges

Topological Sort Algorithm II

QUEUE



vertices with in-degree = 0
that have not been output

1. Find each vertex's in-degree $\Theta(n)$
2. Initialize a queue to contain all in-degree zero vertices $\leftarrow O(n)$
3. While there are vertices in the queue

n iter

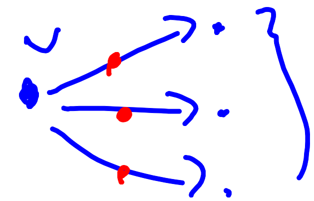
   3.1 Dequeue a vertex *v* (with in-degree zero) and output it $\Theta(1)$
   3.2 Reduce the in-degree of all vertices *v* has an edge to $\leftarrow O(n)$ for one iter. so $O(n^2)$
   3.3 Enqueue any of these that now have in-degree zero

but we can do better! :

Runtime? $\Theta(n) + O(n) + \Theta(n) \cdot \Theta(1) + \Theta(n+m) + \Theta(n)$

over all iterations $\Theta(n)$ | $\Theta(\text{out\_degree}(v))$

v



reduce in-degree by 1

$\underline{\Theta(n+m)}$

over all iteration in 3.2

$\Theta(m)$

output:

$v_1 \quad v_2 \quad v_3 \quad v_4 \quad \ldots\ldots \quad v_n$

overall time @ 3.2:

$\Theta(n) +$ out-degree(v_1) + out-degree(v_2) + ... + out-degree(v_n) $= \Theta(n+m)$

each edge once!

Graph ADT

Graphs are a formalism useful for representing relationships between things.

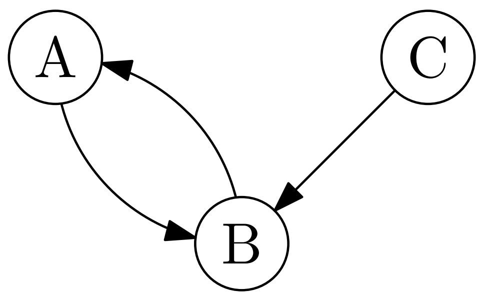
$|V| = n$

A graph is represented as a pair of sets: $G = (V, E)$

$|E| = m$

- V is a set of vertices: $\{v_1, v_2, \ldots, v_n\}$.
- E is a set of edges: $\{e_1, e_2, \ldots, e_m\}$ where each e_i is a pair of vertices: $e_i \in V \times V$.

directed graph



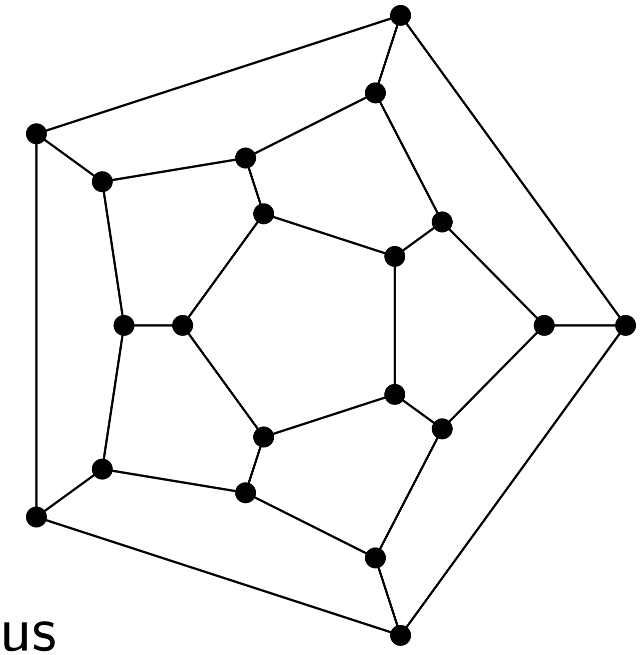
$V = \{A, B, C\}$
$E = \{(A, B), (B, A), (C, B)\}$

Operations may include:

- create (with a certain number of vertices)
- insert/delete a given edge/vertex
- iterate over vertices adjacent to a given vertex
- ask if an edge exists connecting two given vertices

⋮

Graph Applications



Storing things that are graphs by nature

- Road networks
- Airline flights
- Relationships between people, things
- Room connections in Hunt the Wumpus

Compilers

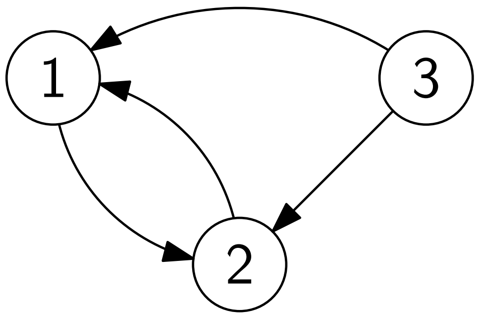
- call graph - which functions call which others
- control flow graph - which fragments of code can follow which others
- dependency graphs - which variables depend on which others

Others

- circuits, class hierarchies, meshes, networks of computers, ...

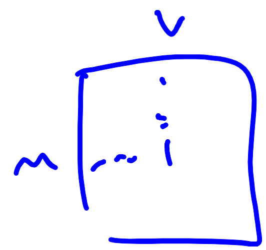
Graph Representations: Adjacency Matrix

A $|V| \times |V|$ array A where $A[u, v] = 1$ if and only if $(u, v) \in E$.

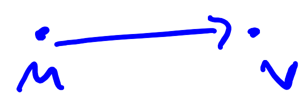


	1	2	3
1	0	1	0
2	1	0	0
3	1	1	0

01 $n \times n$ matrix



$(u, v) \in E$.



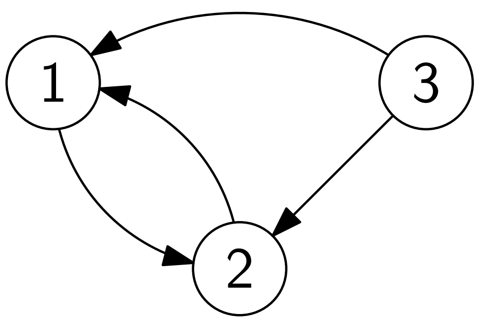
Runtime:

- \+ iterate over vertices $\Theta(n)$
- − iterate over edges $\Theta(n^2)$
- − iterate over vertices adj. to a vertex $\Theta(n)$
- \+ check whether an edge exists $\Theta(1)$

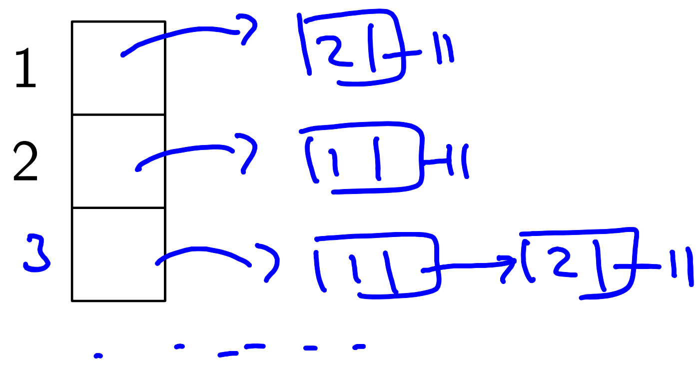
Memory: $\Theta(n^2)$

Graph Representations: Adjacency List

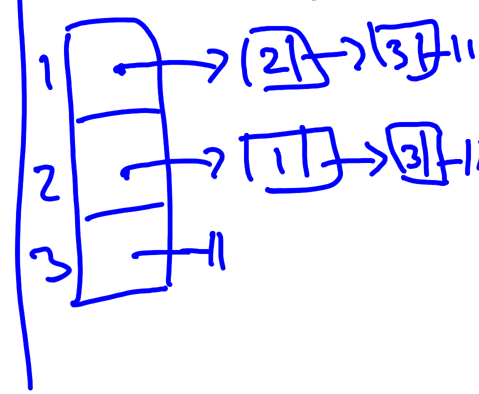
An array L of $|V|$ lists. $L[u]$ contains v if and only if $(u, v) \in E$.



out going edges:



incoming edges:



Runtime:

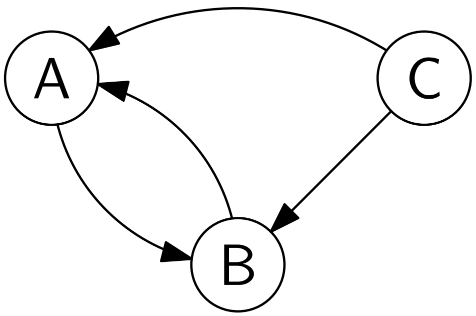
- \+ iterate over vertices $\Theta(n)$
- \+ iterate over edges $\Theta(n+m)$
- \+ iterate over vertices adj. to a vertex u: $\Theta(\text{out-degree}(u))$ → $\Theta(\text{out-degree}(u))$
- − check whether an edge (u,v) exists

Memory:

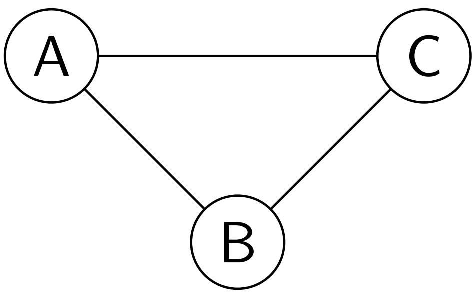
$\Theta(n+m)$

Directed vs. Undirected Graphs

In **directed** graphs, edges have a specific direction:



In **undirected** graphs, they don't (edges are two-way):



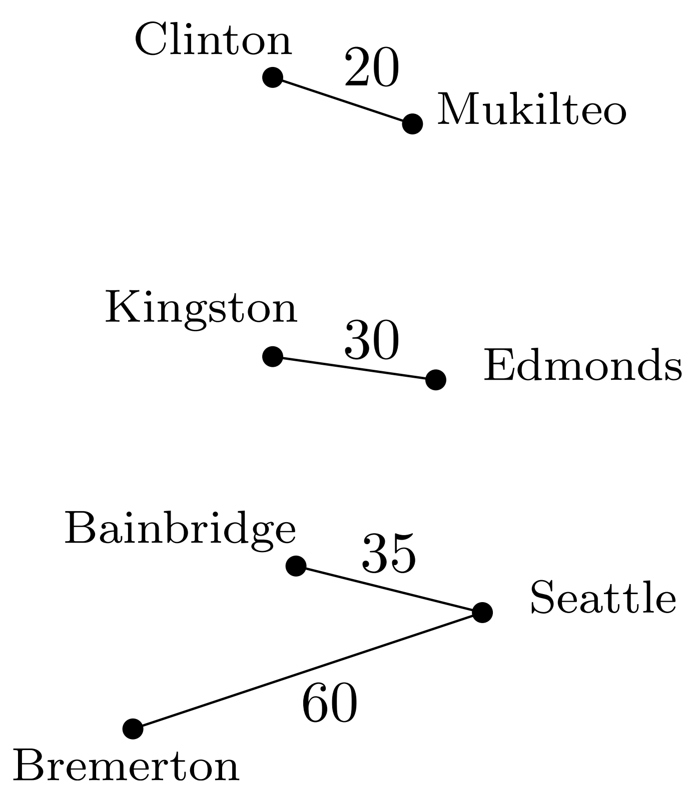
Vertices u and v are **adjacent** if $(u, v) \in E$.

What property do adjacency matrices of undirected graphs have?

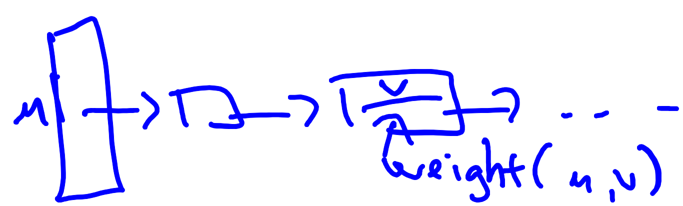
symmetric A $A[x, y] = A[y, x]$

Weighted Graphs

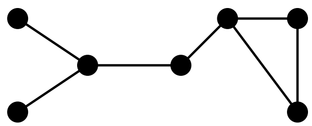
Each edge has an associated weight or cost.



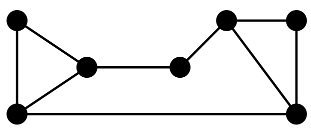
How can we store weights in an adjacency matrix?
In an adjacency list?



Connectivity

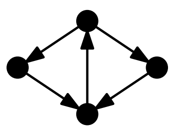


Connected: undirected and there is a path between any two vertices.

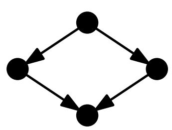


k - connectivity (k-1)

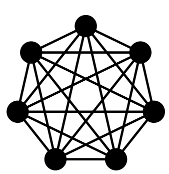
Biconnected: connected even after removing one vertex.



Strongly connected: directed and there is a path from any one vertex to any other.



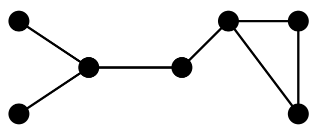
Weakly connected: directed and there is a path between any two vertices, ignoring direction.

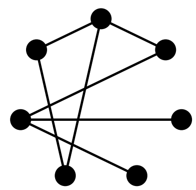


Complete graph: edge between every pair of vertices

Isomorphism and Subgraphs

Isomorphic: Two graphs are isomorphic if they have the same structure (ignoring vertex names).

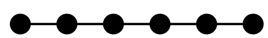


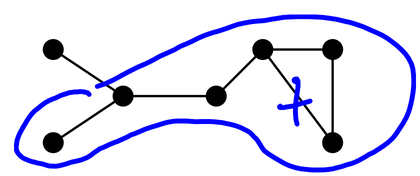


$G_1 = (V_1, E_1)$ is isomorphic to $G_2 = (V_2, E_2)$ if there is a one-to-one and onto function $f : V_1 \to V_2$ such that $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$.

bijection ← n! of them

Subgraph: One graph is a subgraph of another if it is some part of the other graph.

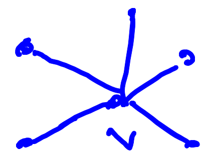




$G_1 = (V_1, E_1)$ is a subgraph of $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.
Note: We sometimes say H is a subgraph of G if H is isomorphic to a subgraph (in the above sense) of G.

Degree

The degree of a vertex $v \in V$ is denoted $\deg(v)$ and represents the number of edges incident on v. (An edge from v to itself contributes 2 towards the degree.)



$\deg(v) = 5$

Handshaking Theorem:

If $G = (V, E)$ is an undirected graph, then

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

sum of all endpoints of edges

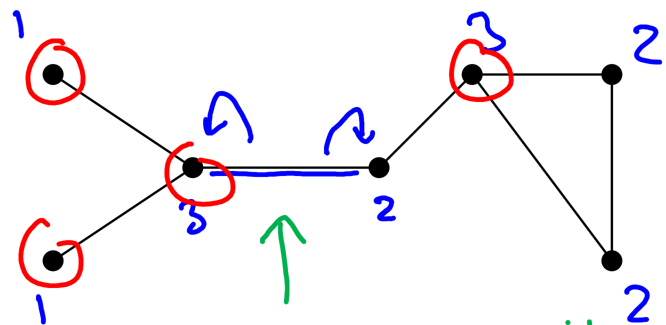
Corollary

An undirected graph has an even number of vertices of odd degree.

Degree/Handshake Example

The degree of a vertex $v \in V$ is the number of edges incident on v.

Let's label each vertex with its degree and calculate the sum...



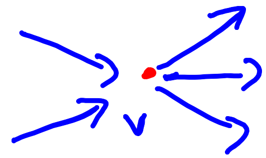
the edge contributes to degree of each endpoint

Sum = 14

|E| = 7

Degree for Directed Graphs

$\deg^-(v) = 2$



$\deg^+(v) = 3$

The **in-degree** of a vertex $v \in V$ (denoted $\underline{\deg^-(v)}$) is the number of edges coming in to v.

The **out-degree** of a vertex $v \in V$ (denoted $\deg^+(v)$) is the number of edges going out of v.

So, $\deg(v) = \deg^+(v) + \deg^-(v)$, and

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = \frac{1}{2} \sum_{v \in V} \deg(v). \quad = |E|$$

heads

tails

heads & tails

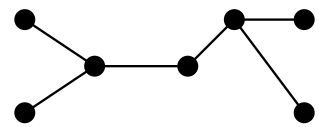
$\text{tail} \longrightarrow \text{head}$

edge

Trees as Graphs

$n = \#$ vertices

Tree: A tree is a connected, acyclic, undirected graph.



$m = \#$edges $= n - 1$

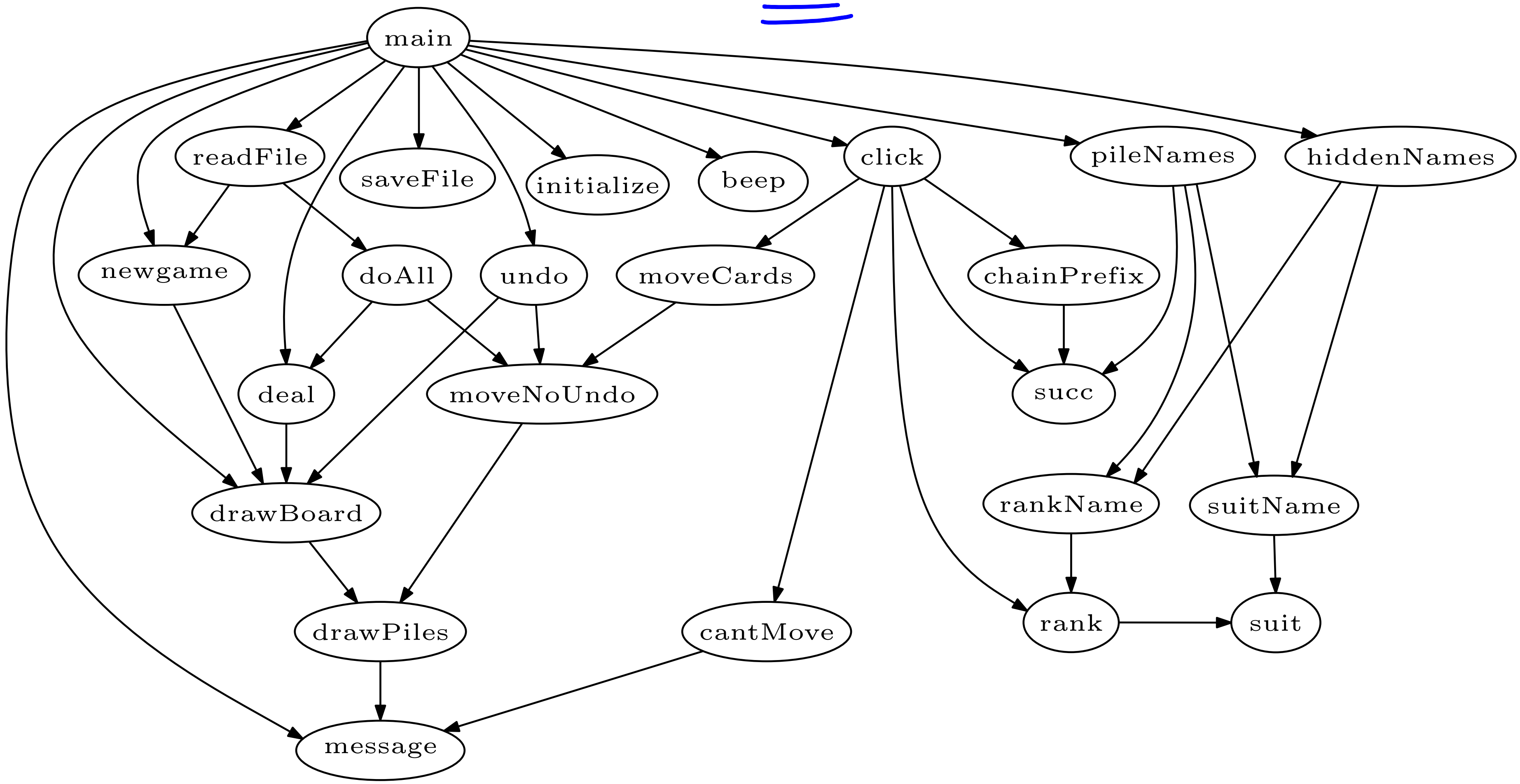
Proof: By induction on n: remove a leaf, you get a tree with one less vertex and one less edge.

Rooted tree: A rooted tree is a tree with a single distinguished vertex called the root.

We can imagine directing the edges of a rooted tree away from the root, to form a connected, acyclic, directed graph, in which there is a path from the root to every vertex.

Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no directed cycles.



We can topo-sort DAGs.

Single Source, Shortest Path

Given a graph $G = (V, E)$ and a vertex $s \in V$, find the shortest path from s to every vertex in V.

Many variations:

length of path = # edges

- weighted vs. unweighted
- no cycles vs. cycles allowed
- positive weights vs. negative weights allowed

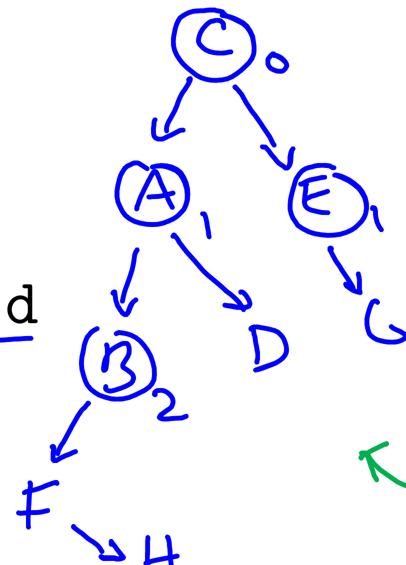
Unweighted Single-Source Shortest Path Problem

distance = # edges

Q: [C,∅] [A,1], [E,1], [B,2], [C,2], [D,2], [D,2], [G,2] . . .

```
BreadthFirstSearch(G, s)
  Q.enqueue([s,0])
  while Q is not empty
    [v,d] = Q.dequeue()
    if v is unmarked
      mark v with distance d
      for each edge (v,w)
        Q.enqueue([w,d+1])
```

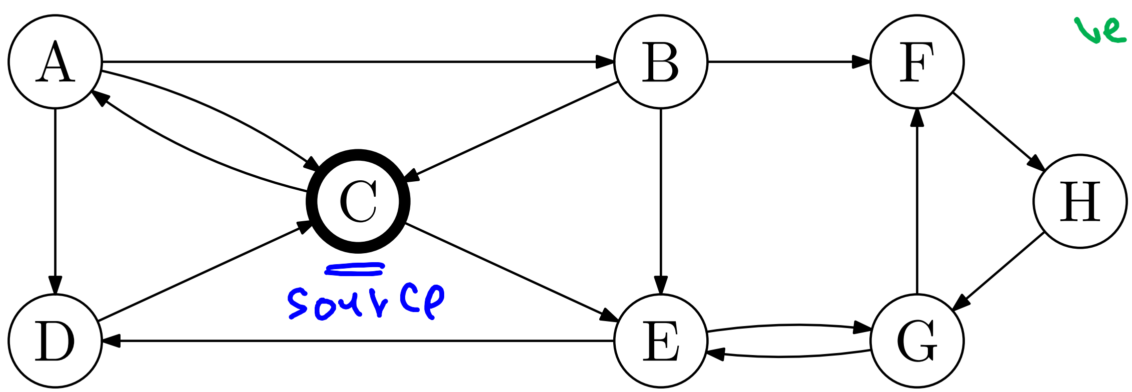
final distance → mark v with distance d



BFS Tree

we can use it to construct the shortest path from the source to any vertex

(Replace the queue with a stack to get depth-first search.)



Weighted Single-Source Shortest Path

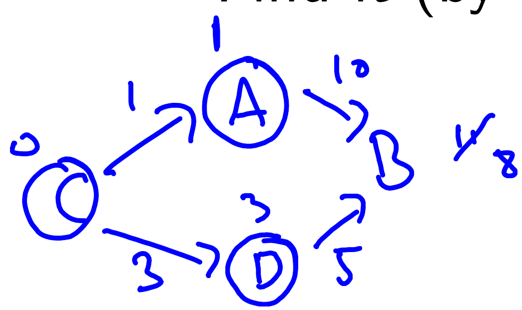
Assumes edge weights are non-negative. $\xrightarrow[\text{why?}]{\text{see next slide}}$

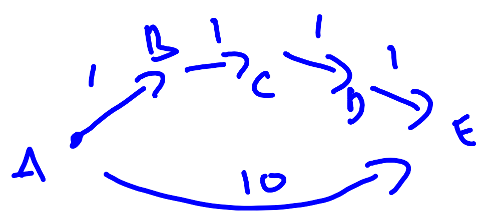
use PRIORITY QUEUE with priority = current distance from the source

Dijkstra's algorithm is a **greedy algorithm** (makes the current best choice without considering future consequences).

Intuition: Find shortest paths in order of length.

- Start at the source vertex (shortest path length $= 0$)
- The next shortest path extends some already discovered shortest path by one edge.
- Find it (by considering all one-edge extensions) and repeat.





Queue: $\overbrace{A, B, E, C, D}^{\text{order of enquing}}$

out: 1^{st} 2^{nd} 5^{th} 3^{rd} 4^{th}

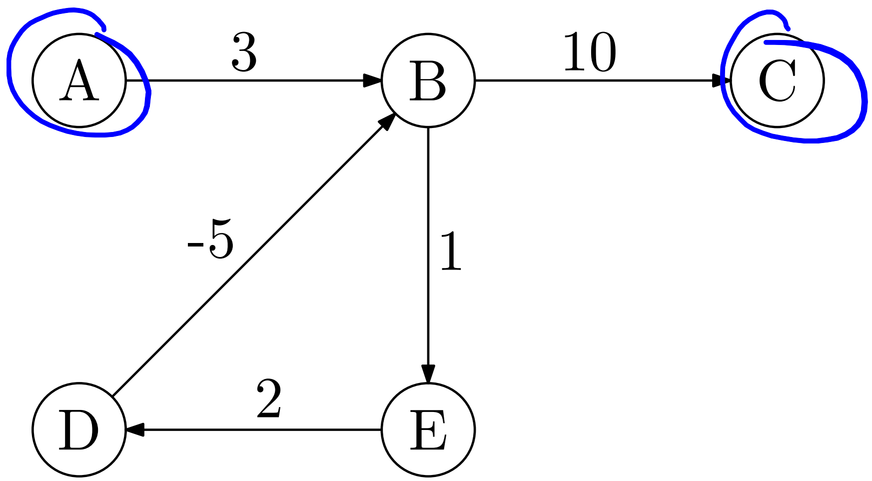
The Trouble with Negative Weight Cycles

A B C ... 13

A B E D B C ... 11

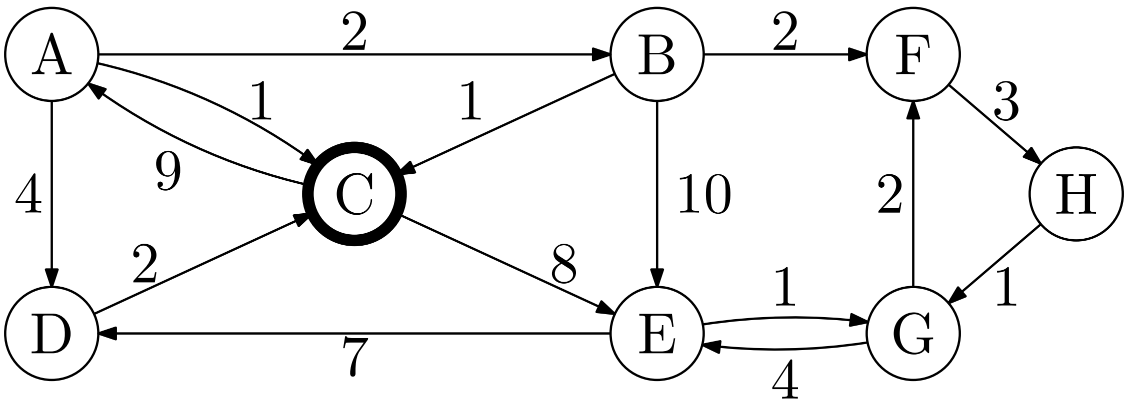
⋮

∴ no shortest path !



What's the shortest path from A to B (or C or D or E)?

Intuition in Action



Dijkstra's Algorithm Pseudocode

tentative distance

- Initialize the dist to each vertex to ∞
- Initialize the dist to the source to 0
- While there are unmarked vertices left in the graph
  - Select the unmarked vertex v with the lowest dist
  - Mark v with distance dist — final distance
  - For each edge (v, w) — Adj. matrix: $O(n)$ Adj. list: $O(\deg^{+}(v))$
    - $\text{dist}(w) = \min\{\text{dist}(w), \text{dist}(v) + \text{weight of } (v, w)\}$

$\qquad\qquad 8 \quad , \quad 3 \quad + \quad 2$

$\text{dist}(w) = 8$

$\downarrow$ ← update

$\text{dist}(w) = 5$

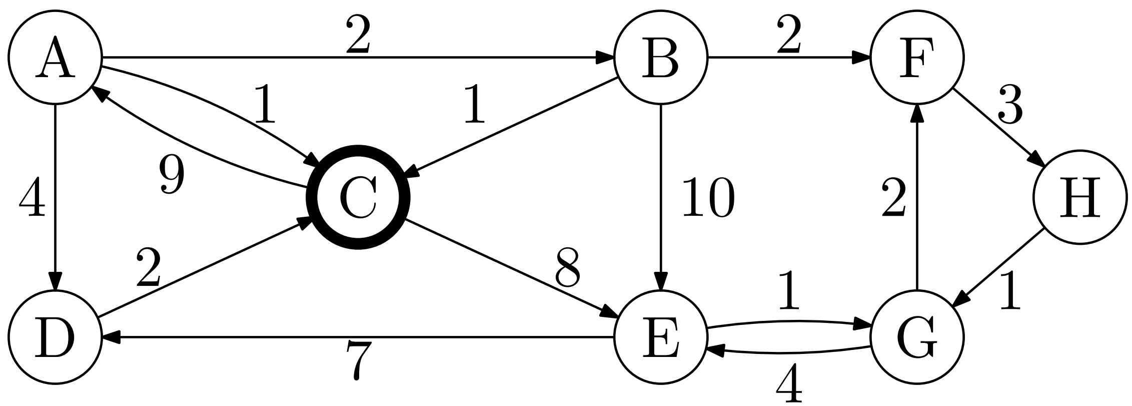
$v \xrightarrow{2} w$

$\text{dist}(v) = 3$

mark:

$\text{distance}(v) = 3$

Dijkstra's Algorithm in Action



marked →

vertex	A	B	C	D	E	F	G	H
dist	9	11	0	~~15~~ 13	8	11	9	
distance	9		0		8		9	

Step 1

Step 2

Step 3

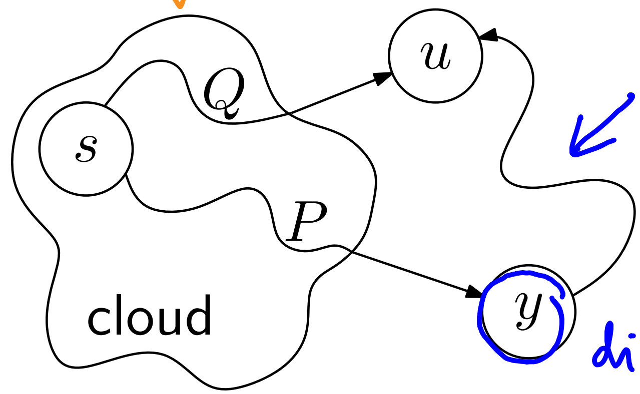
Step 4

⋮

The Cloud Proof

marked vertices

dist(u)



no negative weights

$\Rightarrow$ length(P) $\geq$ dist(y)

dist(y) $\geq$ dist(u)

since Dijkstra choose unmarked vertex with smallest dist

- Assume Dijkstra's algorithm finds the correct shortest path to the first k vertices it visits (the **cloud**).
- But it fails on the $(k+1)$st vertex u.
- So there is some shorter path, P, from s to u.
- Path P must contain a first vertex y not in the cloud.
- But since the path, Q, to u is the shortest path out of the cloud, the path on P upto y must be at least as long as Q.
- Thus the whole path P is at least as long as Q. Contradiction

(What did I use in that last step?)

Data Structures for Dijkstra's Algorithm

priority queue .. implemented using heap

$n =$ $|V|$ times: Select the unknown vertex with the lowest dist.

findMin/deleteMin $\Theta(\log n)$

$m =$ $|E|$ times: $\text{dist}(w) = \min\ \{\text{dist}(w), \text{dist}(v) + \text{weight of } (v, w)\}$

for each edge (v, w) computed exactly once

decreaseKey (i.e., change a key and fix the heap)

find by name (dictionary lookup)

swap up .. $\Theta(\log n)$

Runtime: (adjacency matrix or adjacency list?)

$\underline{\Theta((n+m)\log n)}$

extra overhead:

$\Theta((n+m)\log n + n^2)$

find outgoing edges from each v takes time $\Theta(n)$

Fibonacci Heaps

- Very cool variation on Priority Queues
- Amortized $O(1)$ time for decreaseKey.
- $O(\log n)$ time for deleteMin

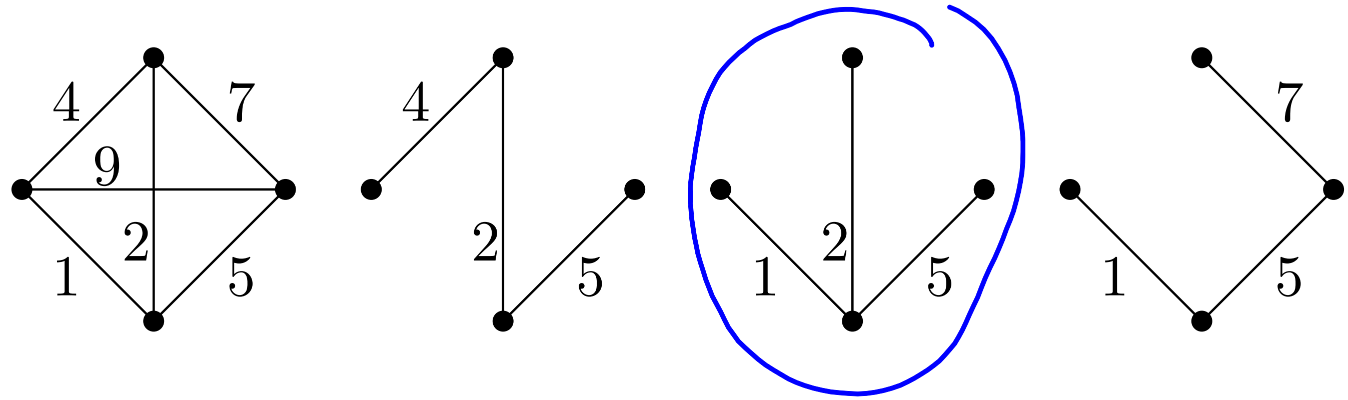
Dijkstra's uses $|V|$ deleteMins and $|E|$ decreaseKeys
Runtime with Fibonacci heaps:

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

Spanning Tree

Spanning tree: a subset of the edges from a connected graph that

- touches all vertices in the graph (spans the graph) and
- forms a tree (is connected and contains no cycles).



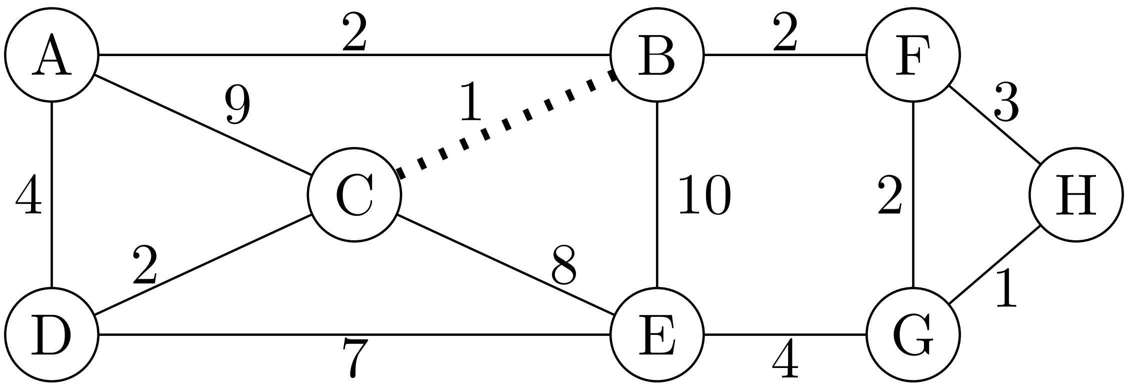
Minimum spanning tree: the spanning tree with the least total edge dist.

Kruskal's Algorithm for Minimum Spanning Trees

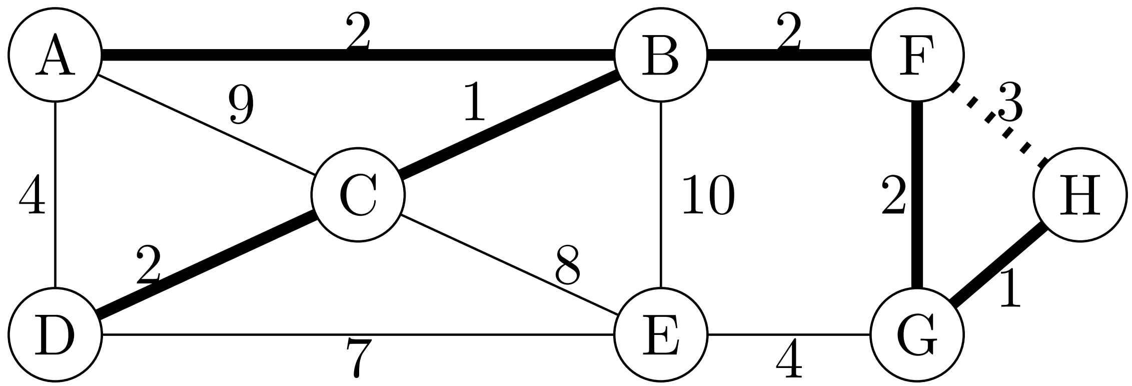
Yet another greedy algorithm:

- Start with an empty tree T
- Repeat: Add the minimum weight edge to T **unless** it forms a cycle.

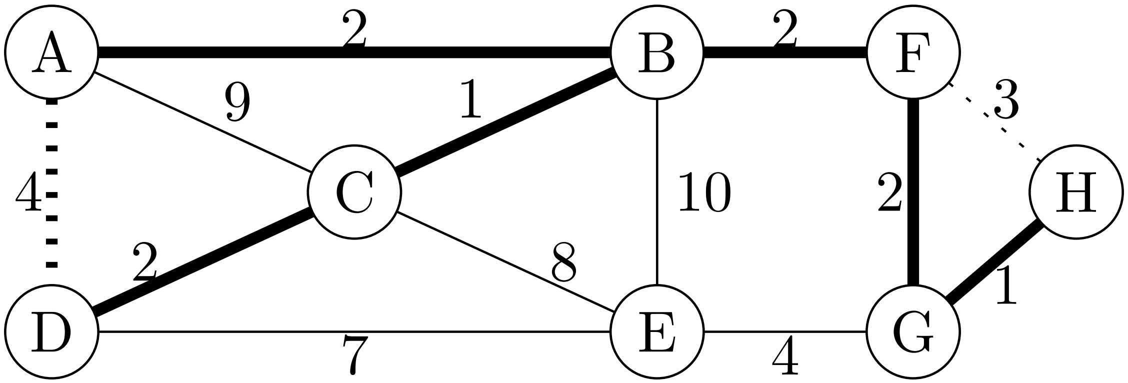
Kruskal's Algorithm in Action (1/5)



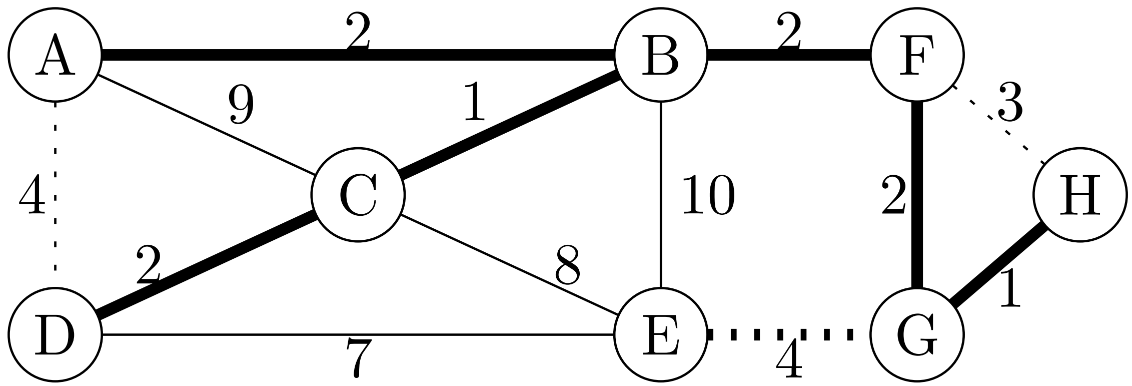
Kruskal's Algorithm in Action (2/5)



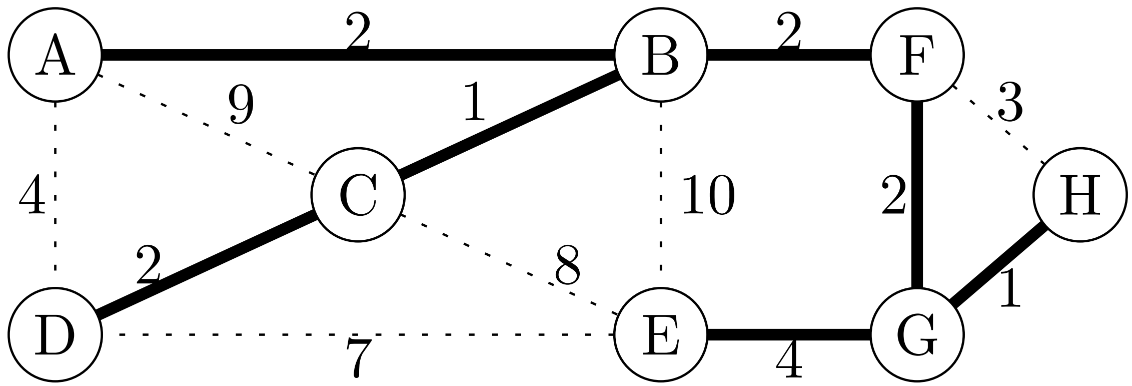
Kruskal's Algorithm in Action (3/5)



Kruskal's Algorithm in Action (4/5)



Kruskal's Algorithm Completed (5/5)



Proof of Correctness

Part I: Kruskal's finds a spanning tree. Why?

Part II: Kruskal's finds a minimum one.
Proof by contradiction.
Assume another spanning tree, T, has lower cost than Kruskal's tree K. (Pick T to be as similar to Kruskal's as possible.)
Pick an edge $e = (u, v)$ in T that's not in K.
Kruskal's rejected e because u and v were already connected by lesser (or equal) weight edges.
Take e out of T and add one of these lesser weight edges to make a new spanning tree. Why does this work?
The new spanning tree still has lower cost than K and it's more like K. Contradiction.

Wiki Epp .. wrong proof

Data Structures for Kruskal's Algorithm

$|E|$ times: Pick the lowest cost edge.

findMin/deleteMin

or just sort edges by weight

$\underline{\Theta(m \cdot \log m)}$

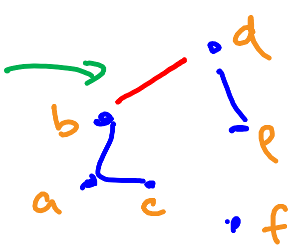
in either case (total time)

$|E|$ times: If u and v are not already connected, connect them.

find representative

union

edge added



connected components

With "disjoint-set" data structure, $O(|E| \log |E|)$ time.

$= O(|E| \cdot \log |V|)$

$\log |E| \le \log |V|^2 = 2 \log |V|$

{a, b, c}, {d, e}, {f} becomes

union

{a, b, c, d, e}, {f}