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)

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

$\forall a, b \quad a < b$ or $a > b$ (or $a = b$)

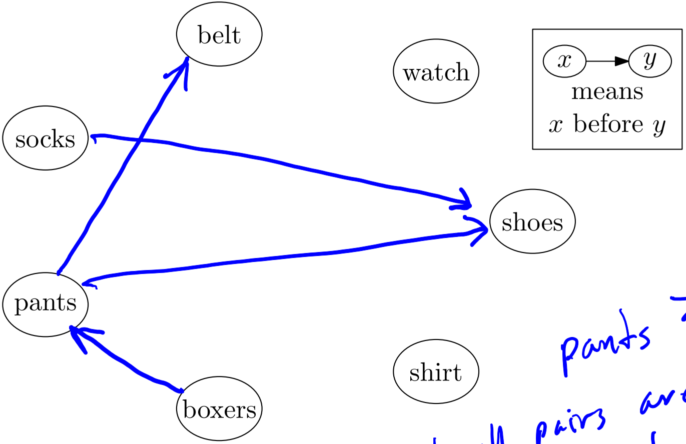
15 34 32 46 11 62 43

$x \rightarrow y$ means x before y

What property does the comparison-based sorting algorithm need to achieve? set of edges contains a directed path between every pair of elements

$\Rightarrow \exists$ one directed path from smallest to largest

Partial Order: Getting Dressed



pants $\gtrless$ shirt

not all pairs are ordered.

Topological Sort

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.

shirt socks boxers pants shoes watch belt

example total order that obeys partial order

Topological Sort Algorithm I

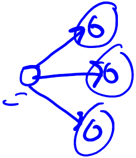
m is at most n^2

n # vertices
m # edges

loops n times

1. Find each vertex's *in-degree* (# of inbound edges) $\Omega(n)$
2. While there are vertices remaining $O(n)$
    2.1 Pick a vertex 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

Runtime?



$n + n-1 + n-2 \ldots$

$= \Theta(n^2)$

takes time prop to # edges coming out

takes time prop. to # remaining vertices

Topological Sort Algorithm II

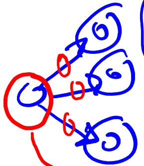
n = # vertices
m = # edges

1. Find each vertex's in-degree — $\Theta(n)$
2. Initialize a queue to contain all in-degree zero vertices — $\Theta(n)$
3. While there are vertices in the queue — n times
   - 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
   - 3.3 Enqueue any of these that now have in-degree zero — $\Theta(1)$

Runtime?

candidates to be enqueued for next iteration

over all iterations # reductions of degree $= m$



$\Theta(n+m)$ ← using adj. List input

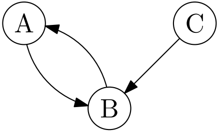
dequeued vertex v

Graph ADT

Graphs are a formalism useful for representing relationships between things.

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

- 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: $e_i \in V \times V$.



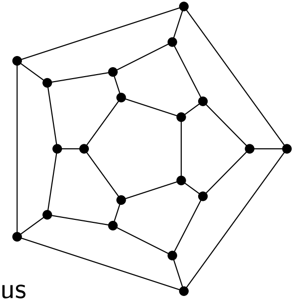
$$V = \{A, B, C\}$$
$$E = \{(A, B), (B, A), (C, B)\}$$

directed edges

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

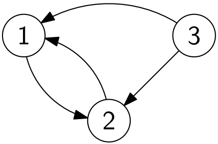
- call graph - which functions call which others
- control flow graph - which fragments of code can follow which others
- dependency graphs - which variables depend on which others

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

n = # vertices
m = # edges

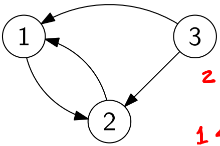
Runtime:

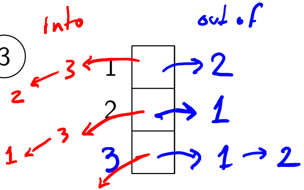
- \+ iterate over vertices $\Theta(n)$
- − iterate over edges $\Theta(n^2)$ examine all matrix locations
- − iterate over vertices adj. to a vertex $\Theta(n)$ examine all entries in a row
- \+ check whether an edge exists $\Theta(1)$

Memory: $\Theta(n^2)$ = # array entries

Graph Representations: Adjacency List

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





Runtime:

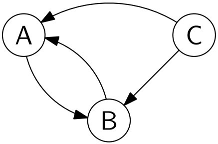
- + iterate over vertices $\Theta(n)$
- + iterate over edges $\Theta(n+m)$
- + iterate over vertices adj. to a vertex u $\Theta(\text{outdegree of } u)$
- − check whether an edge exists (u,v) $\Theta(\text{outdegree of } u)$

+ Memory:

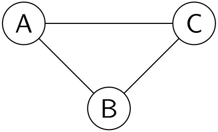
$\Theta(n+m)$

Directed vs. Undirected Graphs

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



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



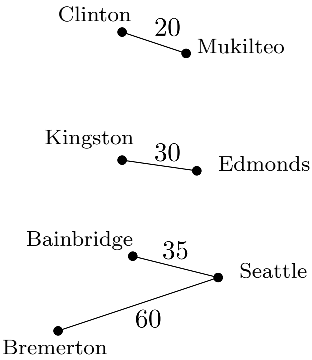
Vertices u and v are **adjacent** if $(u, v) \in E$.

matrix is symmetric
$A[a,b] = A[b,a]$
for all a,b.

What property do adjacency matrices of undirected graphs have?

Weighted Graphs

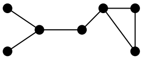
Each edge has an associated weight or cost.



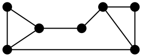
in the matrix rather than 1

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

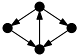
Connectivity



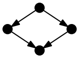
Connected: undirected and there is a path between any two vertices.



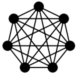
Biconnected: connected even after removing one vertex.



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



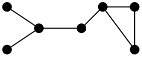
Weakly connected: directed and there is a path between any two vertices, ignoring direction.

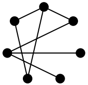


Complete graph: edge between every pair of vertices

Isomorphism and Subgraphs

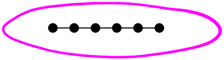
Isomorphic: Two graphs are isomorphic if they have the same structure (ignoring vertex names).

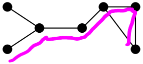




$G_1 = (V_1, E_1)$ is isomorphic to $G_2 = (V_2, E_2)$ if there is a one-to-one and onto function $f : V_1 \to V_2$ such that $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$.

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.)

Handshaking Theorem:

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

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

every edge contributes two to the sum of degrees

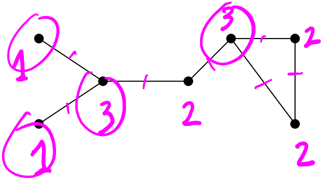
Corollary

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...



$\sum \deg = 14$

$= 2|E|$

Degree for Directed Graphs

The **in-degree** of a vertex $v \in V$ (denoted $\deg^-(v)$) 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

$$\underbrace{\sum_{v\in V}\deg^-(v)}_{\#\text{ heads}} = \underbrace{\sum_{v\in V}\deg^+(v)}_{\#\text{ tails}} = \frac{1}{2}\sum_{v\in V}\deg(v). \quad = \frac{1}{2}\, 2|E|$$

$$= |E|$$

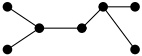
directed edge =

$\longrightarrow$

tail head

Trees as Graphs

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



num vertices

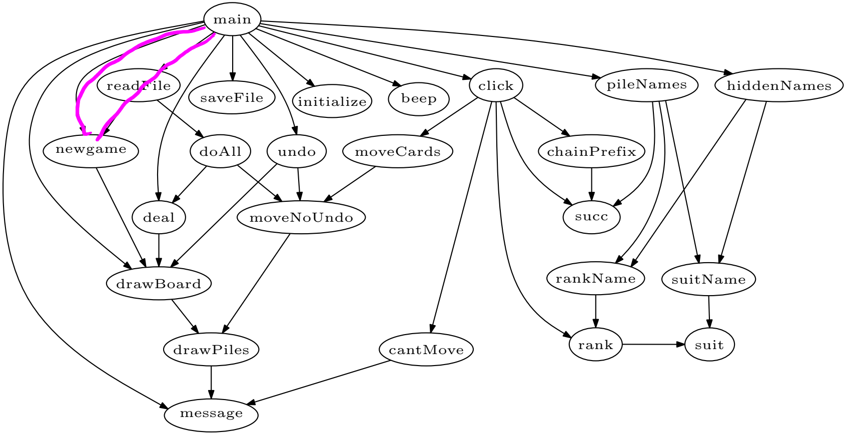
= num edges + 1

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 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:

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

Unweighted Single-Source Shortest Path Problem

queue

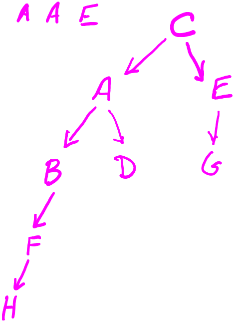
front

1 1 2 2 2

~~A~~ ~~E~~ ~~B~~ D G

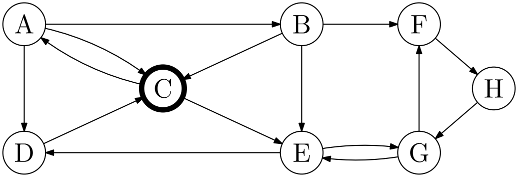
C C A A E

```
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])
```

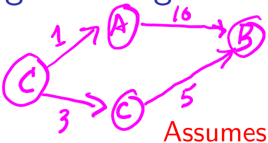


BFS tree

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



Weighted Single-Source Shortest Path



Assumes edge weights are non-negative. why?

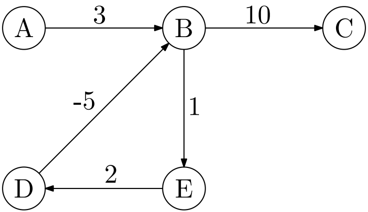
Dijkstra's algorithm is a **greedy algorithm** (makes the current best choice without considering future consequences).

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.

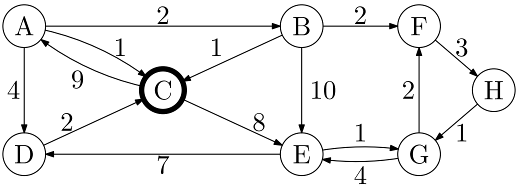
like BFS but with priority queue

The Trouble with Negative Weight Cycles



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

Intuition in Action

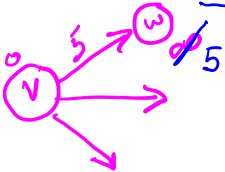


Dijkstra's Algorithm Pseudocode

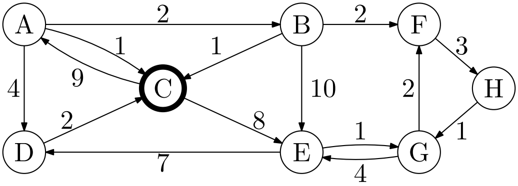
priority

- 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
  - For each edge (v, w)
    - $\text{dist}(w) = \min\{\text{dist}(w), \text{dist}(v) + \text{weight of } (v, w)\}$

$\infty \quad , \quad 0 \quad + \quad 5$



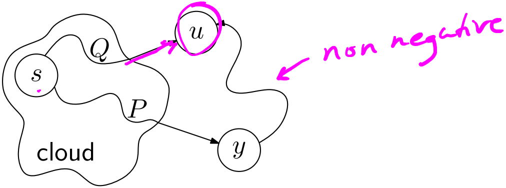
Dijkstra's Algorithm in Action



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

Keep going ...

The Cloud Proof



- 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?) non negative edge wts.

Data Structures for Dijkstra's Algorithm

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

findMin/deleteMin

$O(\log(n))$

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

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

find by name (dictionary lookup)

$O(\log(n))$

Runtime: (adjacency matrix or adjacency list?)

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

$O((n+m)\log n + n^2) = O(m \log n + n^2)$

Fibonacci Heaps

$O(m + n\log n)$

Fibonacci Heaps

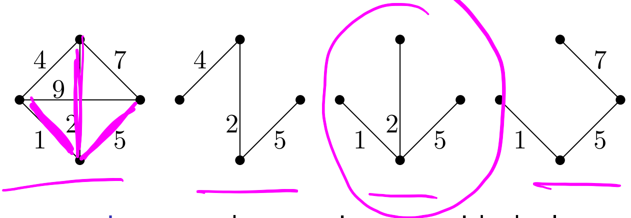
- Very cool variation on Priority Queues
- Amortized $O(1)$ time for decreaseKey.
- $O(\log n)$ time for deleteMin

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

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).



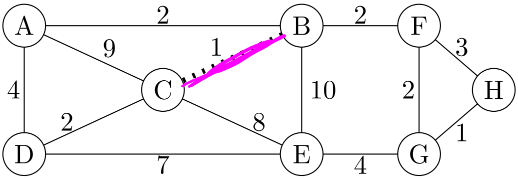
Minimum spanning tree: the spanning tree with the least total edge dist.

Kruskal's Algorithm for Minimum Spanning Trees

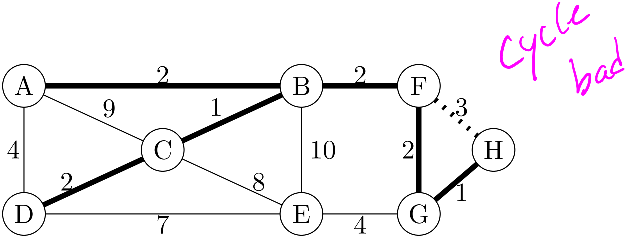
Yet another greedy algorithm:

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

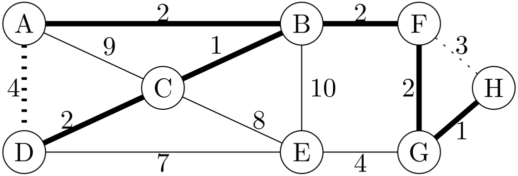
Kruskal's Algorithm in Action (1/5)



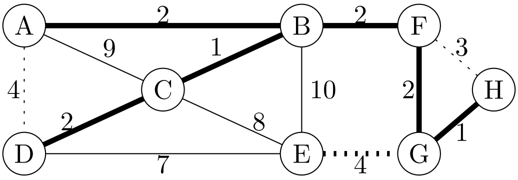
Kruskal's Algorithm in Action (2/5)



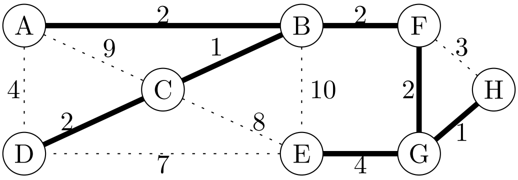
Kruskal's Algorithm in Action (3/5)



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.

Data Structures for Kruskal's Algorithm

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

findMin/deleteMin

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

find representative

union

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

$O(|E| \lg |E|)$