Unit #9: Graphs

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

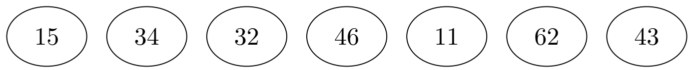
Unit Outline

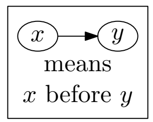
- Topological Sort: Sorting vertices
- Graph ADT and Graph Representations
- Graph Terminology
- More Graph Algorithms
  - Shortest Path (Dijkstra's Algorithm)
  - Minimum Spanning Tree (Kruskal's Algorithm)

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.

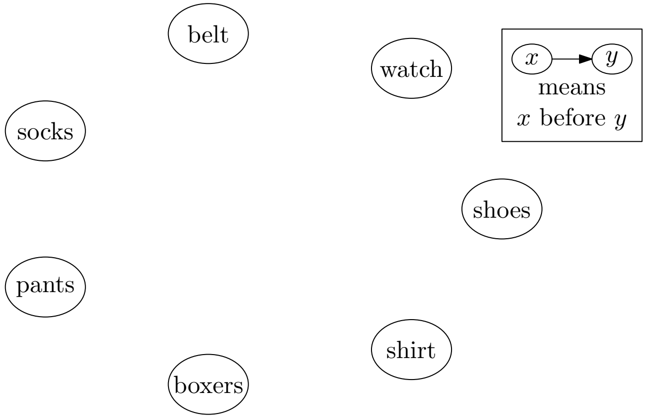
Sorting Total Orders





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

Partial Order: Getting Dressed



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.

Topological Sort Algorithm I

1. Find each vertex's *in-degree* (# of inbound edges)
2. While there are vertices remaining
    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?

Topological Sort Algorithm II

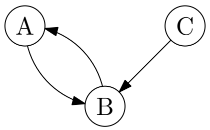
1. Find each vertex's in-degree
2. Initialize a queue to contain all in-degree zero vertices
3. While there are vertices in the queue
    3.1 Dequeue a vertex v (with in-degree zero) and output it
    3.2 Reduce the in-degree of all vertices v has an edge to
    3.3 Enqueue any of these that now have in-degree zero

Runtime?

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: $e_i \in V \times V$.

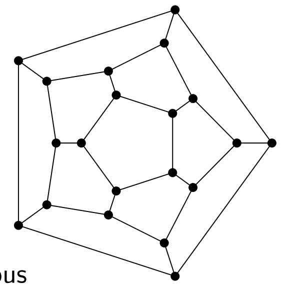


$V = \{A, B, C\}$
$E = \{(A, B), (B, A), (C, B)\}$

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

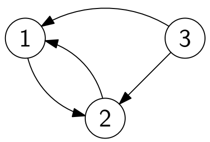
- call graph - which functions call which others
- control flow graph - which fragments of code can follow which others
- dependency graphs - which variables depend on which others

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			
2			

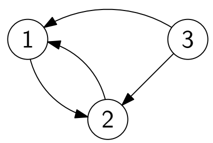
Runtime:

- iterate over vertices
- iterate over edges
- iterate over vertices adj. to a vertex
- check whether an edge exists

Memory:

Graph Representations: Adjacency List

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



1	
2	

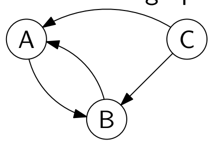
Runtime:

- iterate over vertices
- iterate over edges
- iterate over vertices adj. to a vertex
- check whether an edge exists

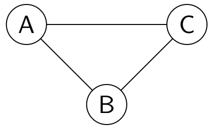
Memory:

Directed vs. Undirected Graphs

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



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

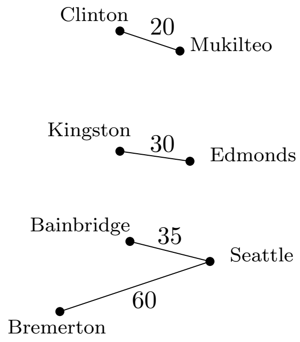


Vertices u and v are **adjacent** if $(u, v) \in E$.

What property do adjacency matrices of undirected graphs have?

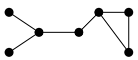
Weighted Graphs

Each edge has an associated weight or cost.

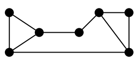


How can we store weights in an adjacency matrix?
In an adjacency list?

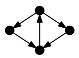
Connectivity



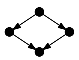
Connected: undirected and there is a path between any two vertices.



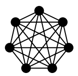
Biconnected: connected even after removing one vertex.



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



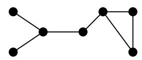
Weakly connected: directed and there is a path between any two vertices, ignoring direction.

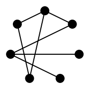


Complete graph: edge between every pair of vertices

Isomorphism and Subgraphs

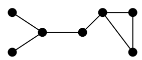
Isomorphic: Two graphs are isomorphic if they have the same structure (ignoring vertex names).





$G_1 = (V_1, E_1)$ is isomorphic to $G_2 = (V_2, E_2)$ if there is a one-to-one and onto function $f : V_1 \rightarrow V_2$ 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

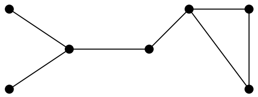
$$\sum_{v \in V} \deg(v) = 2|E|$$

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...



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.

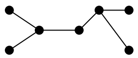
The **out-degree** of a vertex $v \in V$ (denoted $\deg^+(v)$) is the number of edges going out of v.

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

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

Trees as Graphs

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

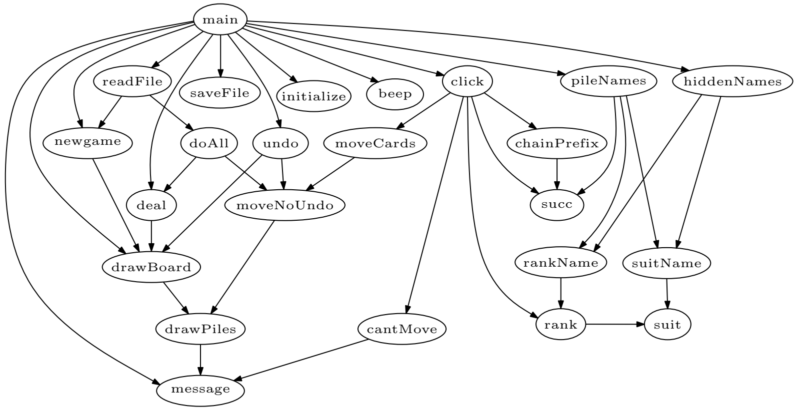


Rooted tree: A rooted tree is a tree with a single distinguished vertex called the root.

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.

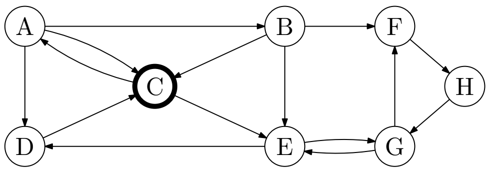
Many variations:

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

Unweighted Single-Source Shortest Path Problem

```
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])
```

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



Weighted Single-Source Shortest Path

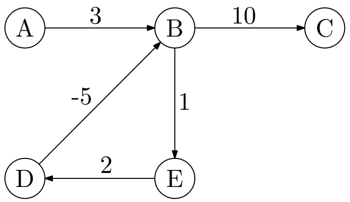
Assumes edge weights are non-negative.

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

Intuition: Find shortest paths in order of length.

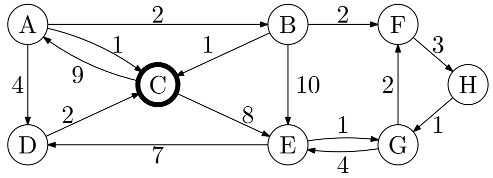
- Start at the source vertex (shortest path length $= 0$)
- The next shortest path extends some already discovered shortest path by one edge.
- Find it (by considering all one-edge extensions) and repeat.

The Trouble with Negative Weight Cycles



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

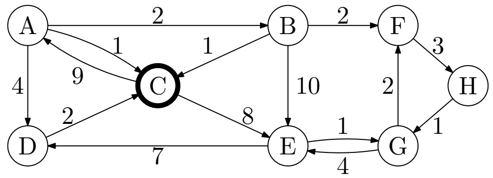
Intuition in Action



Dijkstra's Algorithm Pseudocode

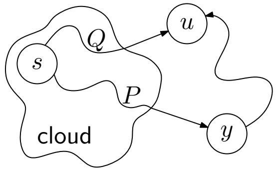
- Initialize the dist to each vertex to ∞
- Initialize the dist to the source to 0
- While there are unmarked vertices left in the graph
  - Select the unmarked vertex v with the lowest dist
  - Mark v with distance dist
  - For each edge (v, w)
    - $\text{dist}(w) = \min\ \{\text{dist}(w), \text{dist}(v) + \text{weight of } (v, w)\}$

Dijkstra's Algorithm in Action



vertex	A	B	C	D	E	F	G	H
dist								
distance								

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?)

Data Structures for Dijkstra's Algorithm

$|V|$ times: Select the unknown vertex with the lowest dist.
findMin/deleteMin

$|E|$ 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)

Runtime: (adjacency matrix or adjacency list?)

Fibonacci Heaps

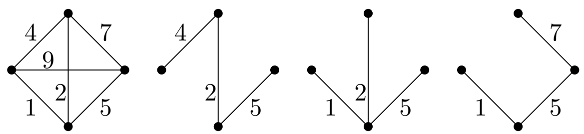
- Very cool variation on Priority Queues
- Amortized $O(1)$ time for decreaseKey.
- $O(\log n)$ time for deleteMin

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).



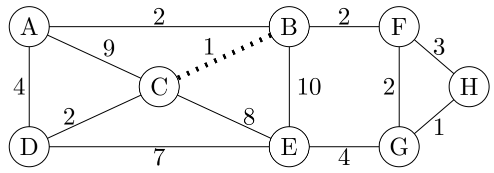
Minimum spanning tree: the spanning tree with the least total edge dist.

Kruskal's Algorithm for Minimum Spanning Trees

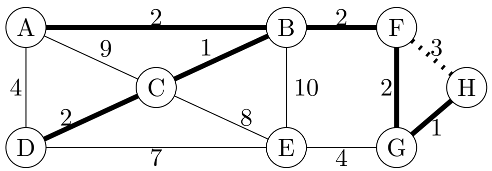
Yet another greedy algorithm:

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

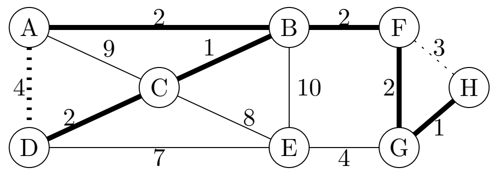
Kruskal's Algorithm in Action (1/5)



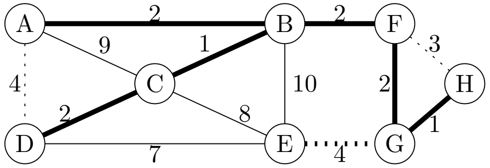
Kruskal's Algorithm in Action (2/5)



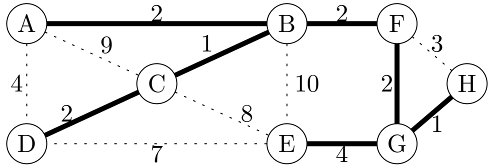
Kruskal's Algorithm in Action (3/5)



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.