Lecture #2: Graph Algorithms

COSC 3020: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@uwyo.edu

[1]with material from various sources

Outline

- ▷ Motivating Example: Topological Sort
- ▷ Definitions and Data Structures
- ▷ Search in Graphs
- ▷ Shortest-Path Algorithms
- ▷ Network Flow Problems
- ▷ Minimum Spanning Tree
- ▷ NP-Completeness

Learning Goals

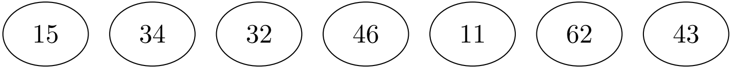
- Be able to represent graphs efficiently.
- Describe graphs, their properties, and applications.
- Recognize graph problems and run algorithms to solve them.

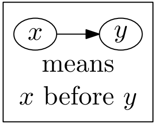
Do try this at home

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- http://algo-visualizer.jasonpark.me/

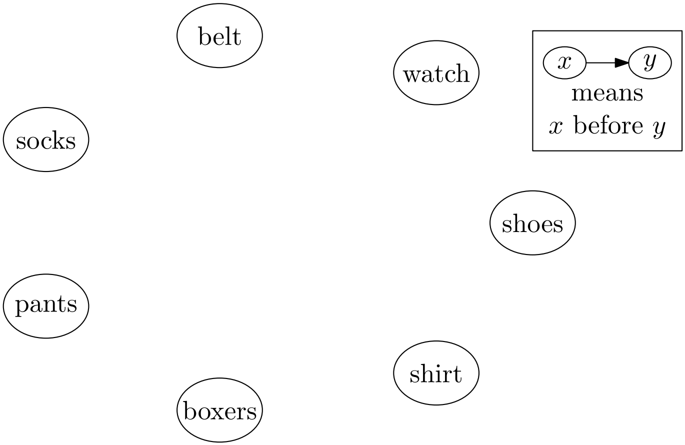
Topological Sort

Total Order: Sorting





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? $\Theta(|V|^2)$

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

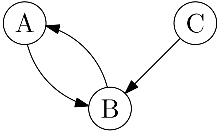
Runtime? $\Theta(|V| + |E|)$

Graph Representation, Properties

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: $e_i \in V \times V$.

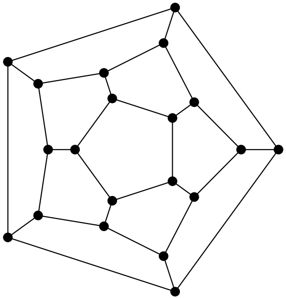


$$V = \{A, B, C\}$$
$$E = \{(A, B), (B, A), (C, B)\}$$

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

Compilers

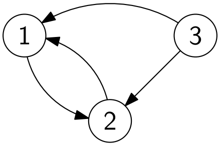
- call graph – which functions call which others
- control flow graph – which fragments of code can follow which others
- dependency graphs – which variables depend on which others

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			
3			

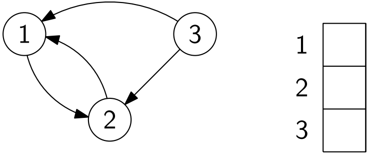
Runtime:

- ▷ iterate over vertices $\Theta(|V|)$
- ▷ iterate over edges $\Theta(|V|^2)$
- ▷ iterate over vertices adj. to a vertex $\Theta(|V|)$
- ▷ check whether an edge exists $\Theta(1)$

Memory: $\Theta(|V|^2)$

Graph Representations: Adjacency List

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



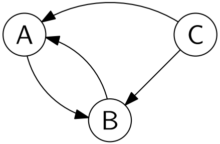
Runtime:

- ▷ iterate over vertices $\Theta(|V|)$
- ▷ iterate over edges $\Theta(|E|)$
- ▷ iterate over vertices adj. to a vertex $\Theta(|E|)$
- ▷ check whether an edge exists $\Theta(|E|)$

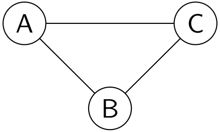
Memory: $\Theta(|E| + |V|)$

Directed vs. Undirected Graphs

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



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

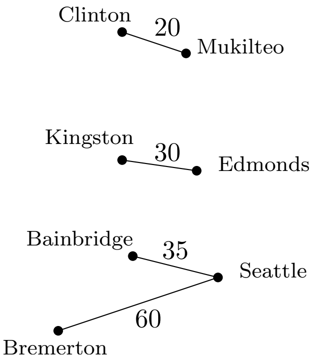


Vertices u and v are **adjacent** if $(u, v) \in E$.

What property do adjacency matrices of undirected graphs have?

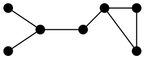
Weighted Graphs

Each edge has an associated weight or cost.

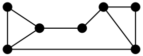


How can we store weights in an adjacency matrix?
In an adjacency list?

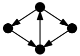
Connectivity



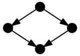
Connected: undirected and there is a path between any two vertices.



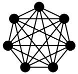
Biconnected: connected even after removing any one vertex with adjacent edges.



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



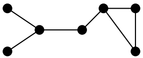
Weakly connected: directed and there is a path between any two vertices, ignoring direction.

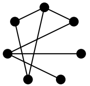


Complete graph: edge between every pair of vertices.

Isomorphism and Subgraphs

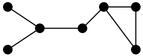
Isomorphic: Two graphs are isomorphic if they have the same structure (ignoring vertex names).





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

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

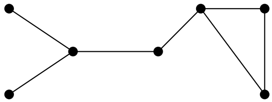
Corollary

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.

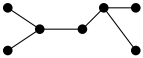
The **out-degree** of a vertex $v \in V$ (denoted $\deg^+(v)$) is the number of edges going out of v.

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.

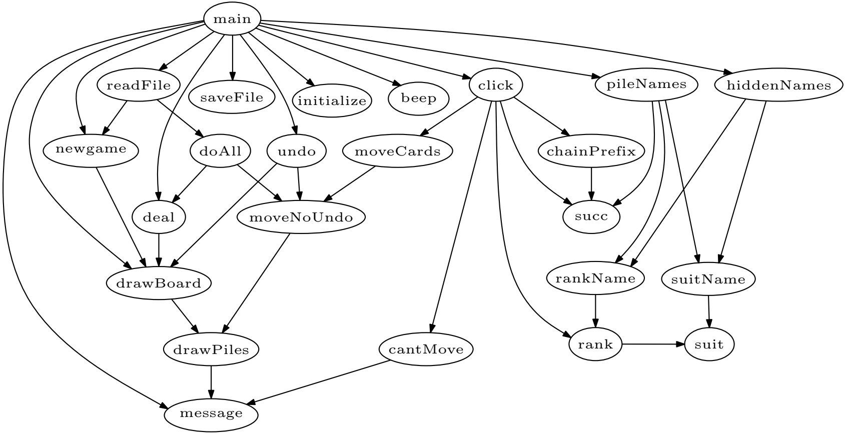


Rooted tree: A rooted tree is a tree with a single distinguished vertex called the root.

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.

Search in Graphs

Search

- find a node in a graph, or traverse all nodes in a graph if the node is not there
- need to take care if there are cycles
- all graph algorithms perform some kind of search
- e.g. path between nodes, which edge to cut to separate graph into two

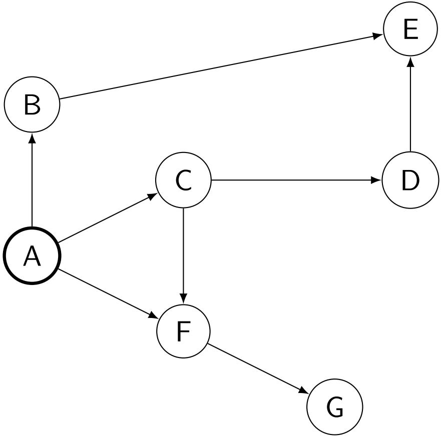
Depth-First Search Pseudocode

Given a graph and a node:

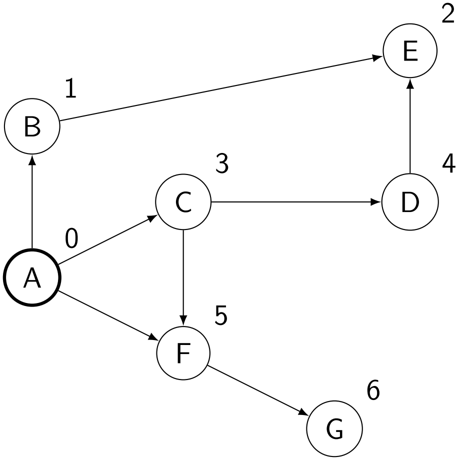
- while unvisited nodes remain
  - if current vertex v is the node we're looking for, return it
  - mark v as visited
  - for each edge (v, w)
    - recursively process w unless marked visited

Data structure?

Depth-First Search Example



Depth-First Search Example



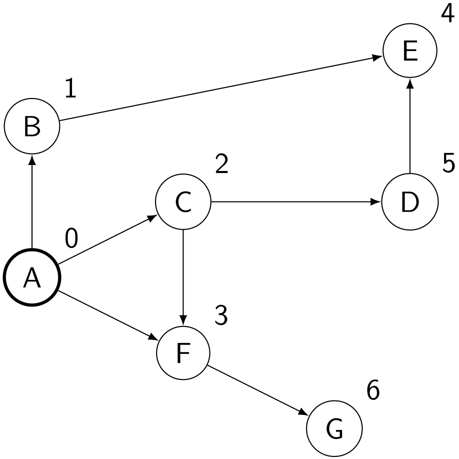
Breadth-First Search Pseudocode

Given a graph and a node:

- while unvisited nodes remain
  - if current vertex v is the node we're looking for, return it
  - mark v as visited
  - for each edge (v, w)
    - enqueue w for processing unless marked visited

Data structure(s)?

Breadth-First Search Example



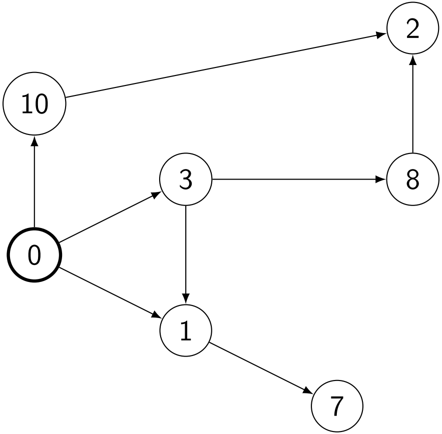
Best-First Search Pseudocode

Given a graph and a node:

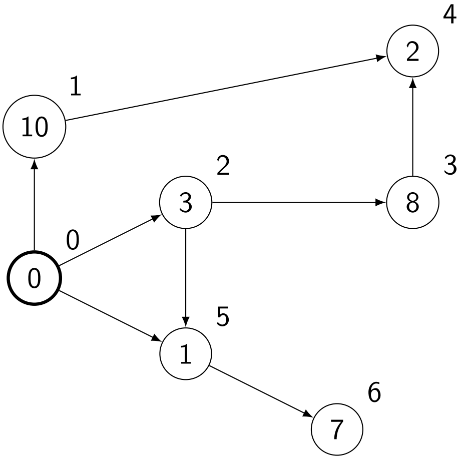
- while unvisited nodes remain
  - if current vertex v is the node we're looking for, return it
  - mark v as visited
  - for each edge (v, w)
    - determine score s_w of w
    - enqueue w with priority s_w unless marked visited

Data structure(s)?

Best-First Search Example



Best-First Search Example



Shortest Paths

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

Weighted Single-Source Shortest Path

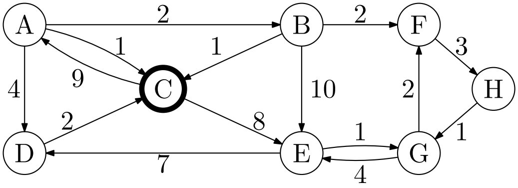
Assumes edge weights are non-negative.

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.

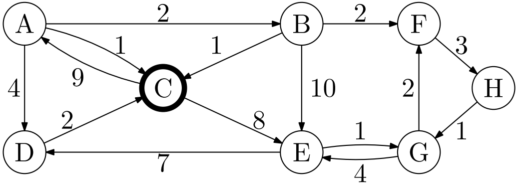
Intuition in Action



Dijkstra's Algorithm Pseudocode

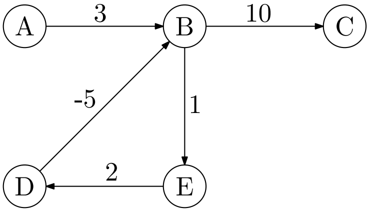
- Initialize the dist to each vertex to ∞, source to 0
- While there are unmarked vertices left in the graph
  - Select the unmarked vertex v with the lowest dist
  - Mark v with distance dist
  - For each edge (v, w)
    - dist(w) $=$ min $\{$dist(w), dist(v) $+$ weight of $(v, w)\}$

Dijkstra's Algorithm in Action



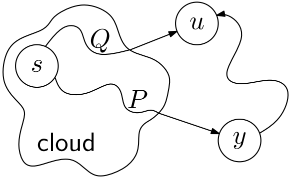
vertex	A	B	C	D	E	F	G	H
dist								
distance								

The Trouble with Negative Weight Cycles



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

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

Runtime of Dijkstra's Algorithm

- Initialize the dist to each vertex to ∞ — $O(|V|)$
- Initialize the dist to the source to 0 — $O(1)$
- While there are unmarked vertices left in the graph — $O(|V|)$
  - Select the unmarked vertex v with the lowest dist — $O(|V|)$
  - Mark v with distance dist — $O(1)$
  - For each edge (v, w) — $O(|V|)$
    - dist(w) $=$ min $\{$dist(w), dist(v) $+$ weight of $(v, w)\}$ — $O(1)$

Runtime of Dijkstra's Algorithm

- Initialize the dist to each vertex to ∞ — $O(|V|)$
- Initialize the dist to the source to 0 — $O(1)$
- While there are unmarked vertices left in the graph — $O(|V|)$
  - Select the unmarked vertex v with the lowest dist — $O(|V|)$
  - Mark v with distance dist — $O(1)$
  - For each edge (v, w) — $O(|V|)$
    - dist(w) $=$ min $\{$dist(w), dist(v) $+$ weight of $(v, w)\}$ — $O(1)$

$O(|V| + |V| \cdot (|V| + |V|)) = O(|V|^2)$ (adjacency matrix)

Runtime of Dijkstra's Algorithm

- Initialize the dist to each vertex to ∞ — $O(|V|)$
- Initialize the dist to the source to 0 — $O(1)$
- While there are unmarked vertices left in the graph — $O(|V|)$
  - Select the unmarked vertex v with the lowest dist — $O(|V|)$
  - Mark v with distance dist — $O(1)$
  - For each edge (v, w) — $O(|E|)$
    - dist(w) = min $\{$dist(w), dist(v) + weight of $(v, w)\}$ — $O(1)$

$O(|V| + |E| + |V| \cdot |V|) = O(|E| + |V|^2)$ (adjacency list)

Runtime of Dijkstra's Algorithm

- Initialize the dist to each vertex to ∞ $O(|V|)$
- Initialize the dist to the source to 0 $O(1)$
- While there are unmarked vertices left in the graph $O(|V|)$
  - Select the unmarked vertex v with the lowest dist $O(\log |V|)$
  - Mark v with distance dist $O(1)$
  - For each edge (v, w) $O(|E|)$
    - dist(w) = min $\{$dist(w), dist(v) + weight of $(v, w)\}$ $O(1)$

with heaps and sparse (connected) graphs:
$O(|V| + |E| \log |V| + |V| \log |V|) = O((|E| + |V|) \log |V|)$

Fibonacci Heaps

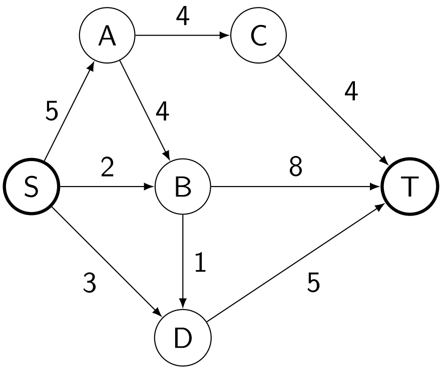
- Variation on Priority Queues
- Amortized $O(1)$ time for decreaseKey
- $O(\log n)$ time for deleteMin

Dijkstra's uses $|V|$ deleteMins and $|E|$ decreaseKeys

Runtime with Fibonacci heaps: $O(|V| + |E| + |V| \log |V|) = O(|E| + |V| \log |V|)$

Network Flow Problems

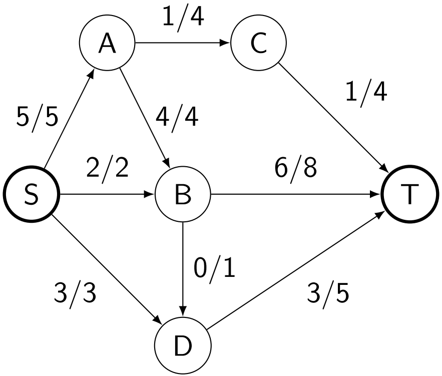
Network Flow



- graph with edge capacities
- designated "source" and "target" vertices
- flow into vertex = flow out of vertex (except for source and target)
- e.g. water network, roads, LAN cables...

Maximum Flow

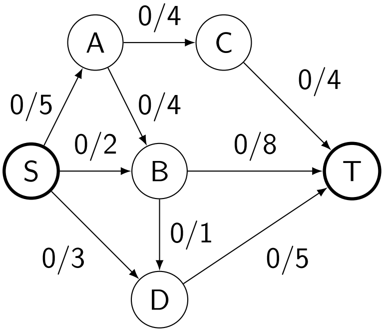
How much can we push from source to target?

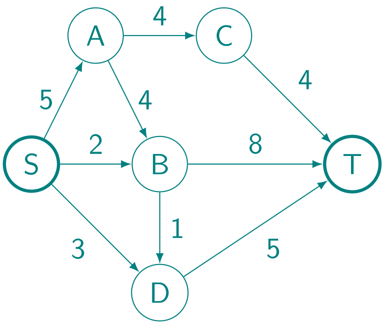


Finding Maximum Flow (Ford-Fulkerson-Algorithm)

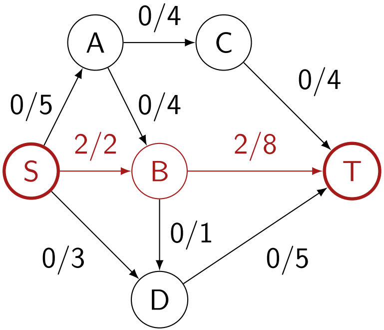
- set flow to 0 for all edges
- construct residual graph with remaining capacity for all edges
- while there is a path from source to target (augmenting path) in the residual graph
  - find the maximum amount of flow for the path (edge with lowest capacity)
  - add the flow to each edge on the path in the original graph
  - in the residual graph
    - reduce the remaining capacities for each edge on the path
    - add a “return edge” with the same amount of flow for each edge on the path
    - delete edges with residual capacity 0

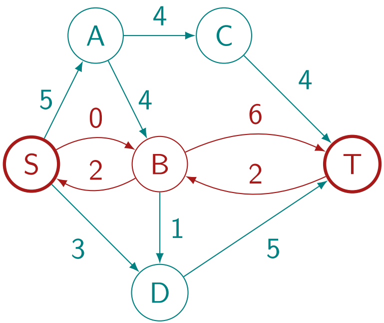
Ford-Fulkerson in Action



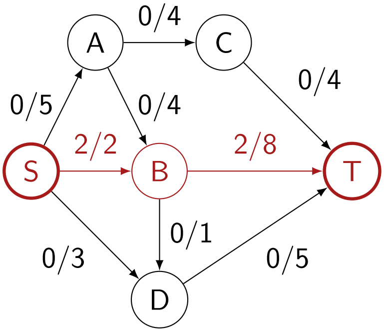


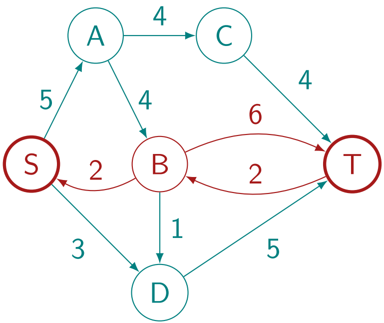
Ford-Fulkerson in Action



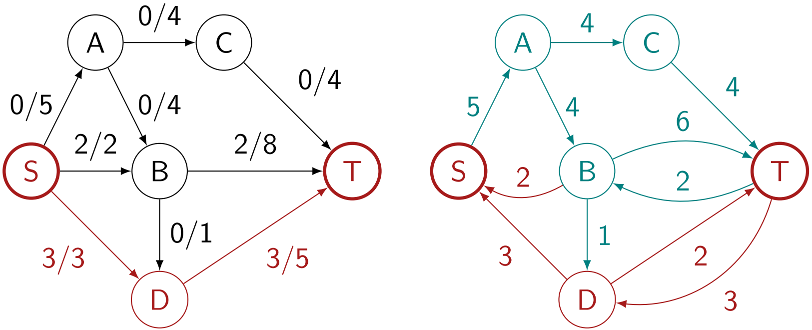


Ford-Fulkerson in Action

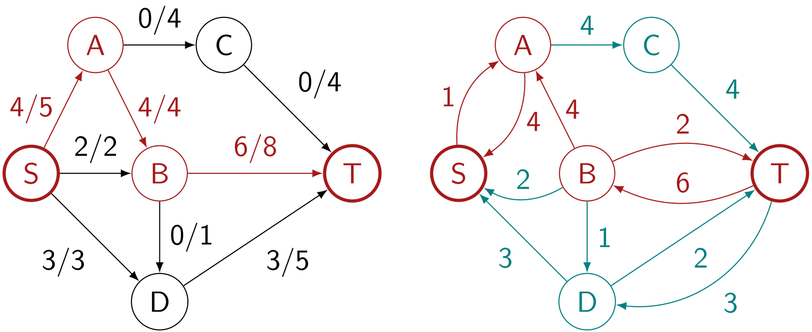




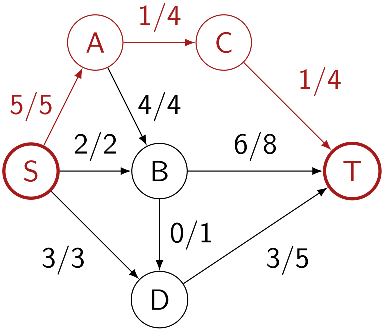
Ford-Fulkerson in Action

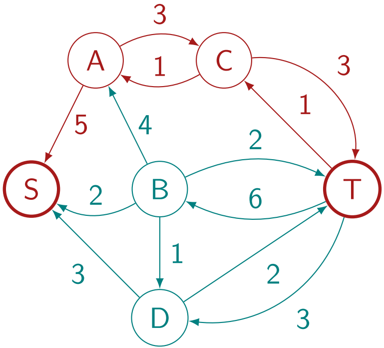


Ford-Fulkerson in Action

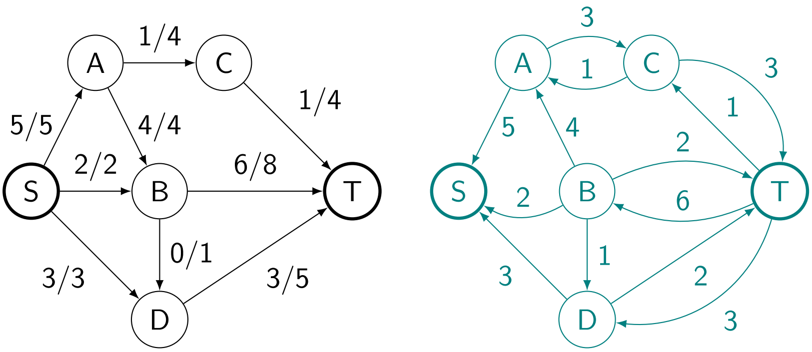


Ford-Fulkerson in Action

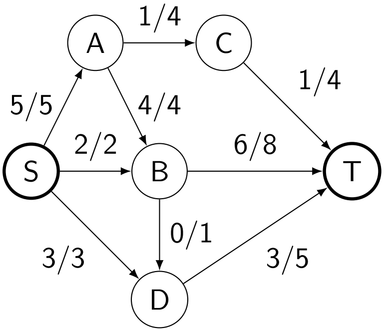


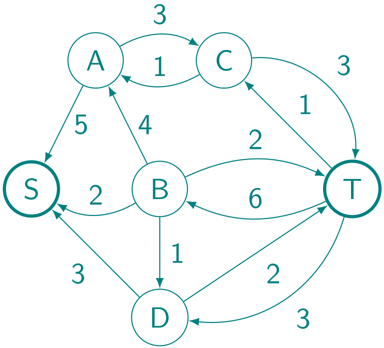


Ford-Fulkerson Done



Ford-Fulkerson Done

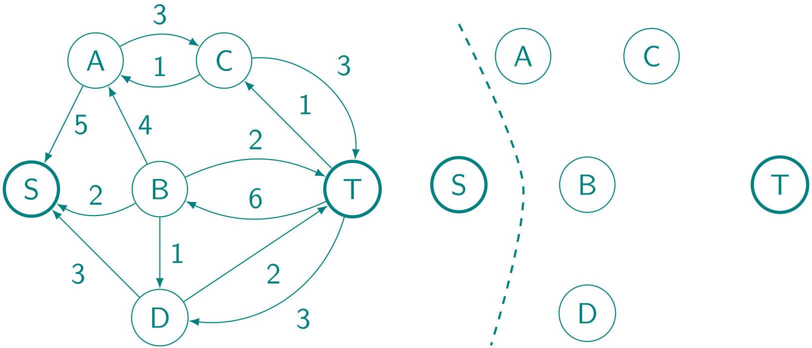




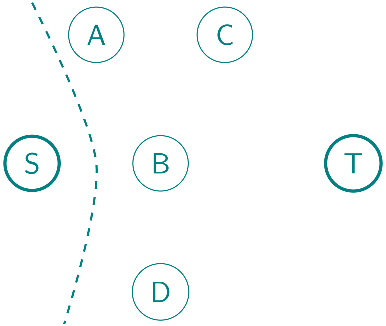
Runtime? $O(|E|f)$

Maximum Flow Proof

Cut residual graph into vertices reachable from source and not reachable from source.

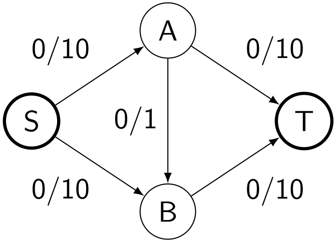


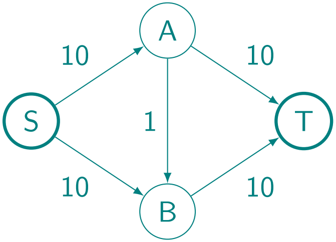
Maximum Flow Proof



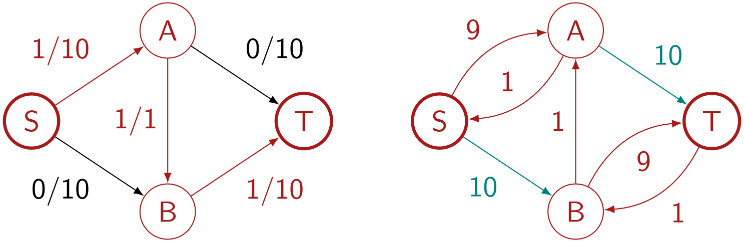
- edges that cross cut in original graph must be saturated
- therefore flow is equal to capacity of cut
- must have maximum flow

What is the “return edge” for?

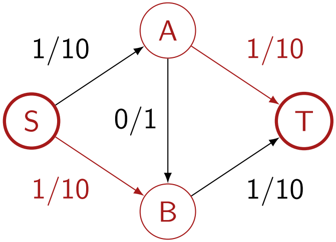


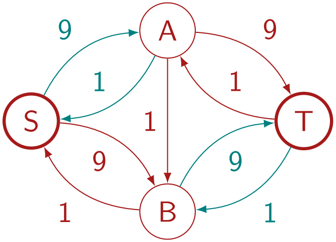


What is the “return edge” for?

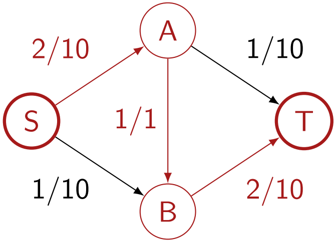


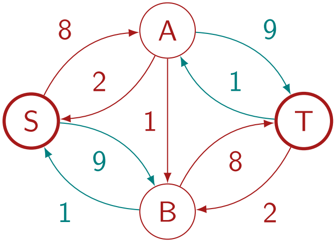
What is the “return edge” for?





What is the “return edge” for?



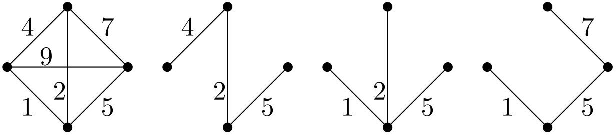


Minimum Spanning Trees

Spanning Tree

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

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



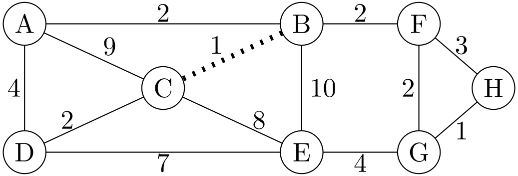
Minimum spanning tree: the spanning tree with the least total edge dist.

Kruskal's Algorithm for Minimum Spanning Trees

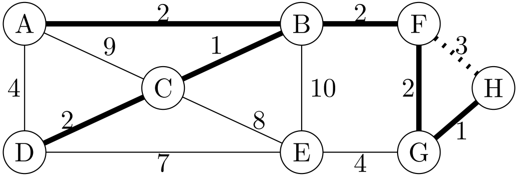
Yet another greedy algorithm:

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

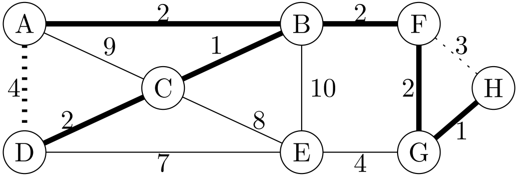
Kruskal's Algorithm in Action



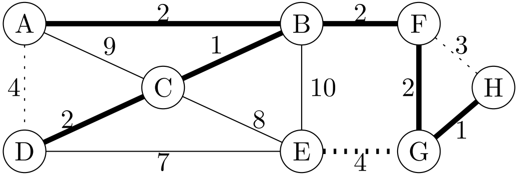
Kruskal's Algorithm in Action



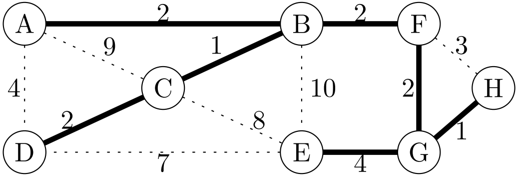
Kruskal's Algorithm in Action



Kruskal's Algorithm in Action



Kruskal's Algorithm Completed



Proof of Correctness

Part I: Kruskal's finds a spanning tree T of graph G.

- T is a tree – no cycles.
- T is spanning – any vertex v not on an edge in T must have incident edges that were considered by the algorithm and would have been included.
- T is connected – if T was not connected, it must have two or more components that are connected in G by one or more edges. One of these edges would have been included by the algorithm, as it does not create a cycle.

Proof of Correctness

Part II: Kruskal's finds a minimum spanning tree.
Let S be another spanning tree with weight less than T.

- ▷ Let e be the edge of least weight in T that is not in S.
- ▷ Add e to S.
  - ▷ This creates a cycle C, and C contains e.
  - ▷ Cycle C contains an edge e', where e' is not in T. Otherwise all edges in $C - e$ are already in T, and T would also contain a cycle, and would not be a tree.
  - ▷ If we replace e' in S by e we get a spanning tree S' where
    - ▷ weight of $e \leq$ weight of e' and Kruskal's algorithm would have chosen e in preference to e' to create T.
    - ▷ S' is now one edge closer to being T than S is to T.
- ▷ weight of $S' \leq$ weight of S. Now repeat until $S' = T$.
- ▷ Process terminates with $S' = T$ and weight of $T \leq$ weight of S. 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
union

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

NP-Completeness

Some Problems are Hard

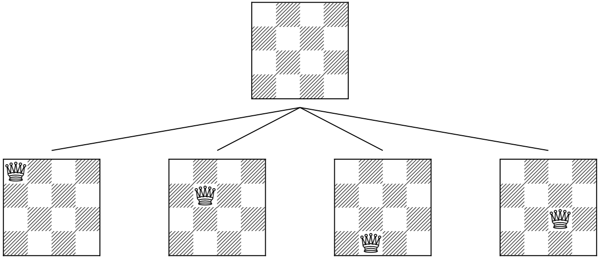
n-queens problem:

- place n queens on an $n \cdot n$ chess board such that no queen is attacking another queen

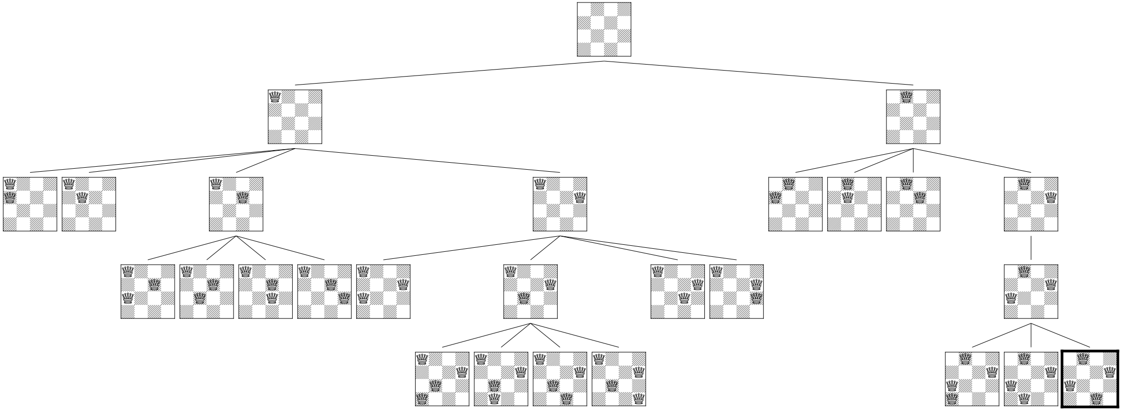
Complexity?

n-Queens as Graph Search

- each state of the board is a vertex
- edges connect vertices that differ in the position of one queen
- for example: start with an empty board, then place one queen on it
- stop as soon as a queen is attacking another queen and try something else
- how to decide what the "best" next position for a queen is?



n-Queens as Graph Search



Solution Approaches

Generate-and-Test Put all queens somewhere (randomly), then check whether it's a solution

Backtracking Search Put each queen on the board one after another, undoing the last assignment if we find a non-solution

Forward Checking Backtracking search + rule out positions that can't be part of solution after placing each queen

What makes this problem hard?

- need to make decision (put queen where)
- "quality" of decision only becomes apparent after making the choice (place left for another queen?)
- may only become apparent after many more choices
- may need to backtrack many times

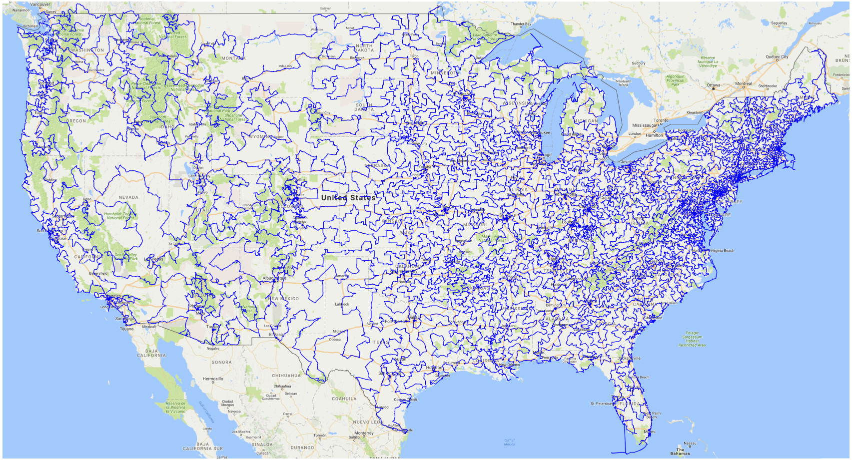
P vs. NP

- some problems are easy – solvable in polynomial time (P)
- problem hardness comes from choices
- if we knew what choice to make, problem would be easy
- assume we know what choice to make – non-deterministic machine
- problem becomes non-deterministic easy (NP)

Properties of NP problems

- hard to solve
- easy to check – polynomial time
- any NP problem can be expressed in terms of another NP problem (“reduces” to it)
- NP-hard: problem at least as hard as hardest NP problem (but could be more difficult)
- NP-complete: NP-hard and in complexity class NP
- P = NP? $1,000,000 question
- in practice, many NP problems can be solved efficiently

Traveling Salesman Problem



Shortest tour visiting 49,603 sites from the National Register of Historic Places

http://www.math.uwaterloo.ca/tsp/us/index.html

Problem Complexity

Searching and Sorting P, tractable

Traveling Salesman Problem NP, intractable[2]

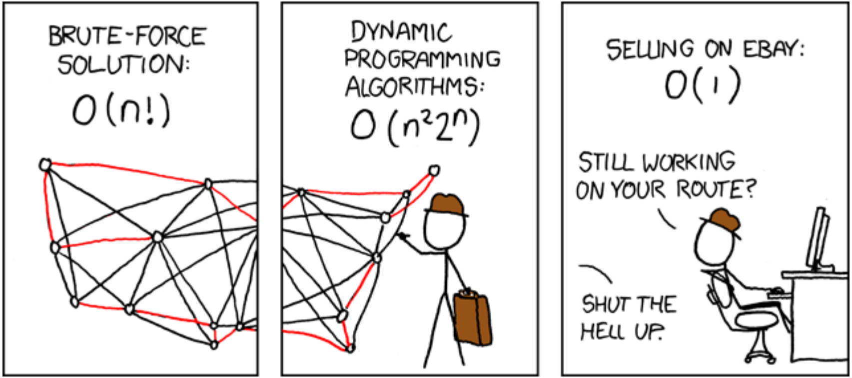
Kolmogorov Complexity uncomputable (and also NP-hard)

Kolmogorov Complexity of a string is the length of the shortest description of it.

Can't be computed. Pithy but hand-wavy proof: What is

> *The smallest positive integer that cannot be described in fewer than fourteen words.*

[2] Assuming $P \neq NP$.



https://xkcd.com/399/