Basic Terminology

REVIEW from Data Structures!

$G = (V, E)$; V is set of n nodes, E is set of m edges

- **Node** or **Vertex**: a point in a graph
- **Edge**: connection between nodes
- **Weight**: numerical cost or length of an edge
- **Direction**: arrow on an edge
- **Path**: sequence $(u_0, u_1, \ldots, u_k)$ with every $(u_{i-1}, u_i) \in E$
- **Cycle**: path that starts and ends at the same node

Examples

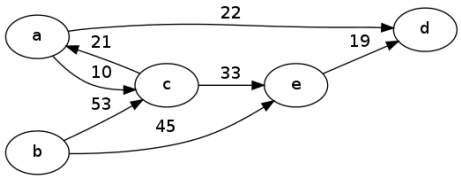
- Roads and intersections
- People and relationships
- Computers in a network
- Web pages and hyperlinks
- Makefile dependencies
- Scheduling tasks and constraints
- (many more!)

Graph Representations

- **Adjacency Matrix**: $n \times n$ matrix of weights.
  $A[i][j]$ has the weight of edge (u_i, u_j).
  Weights of non-existent edges usually 0 or ∞.
  Size:
- **Adjacency Lists**: Array of n lists;
  each list has node-weight pairs for the *outgoing edges* of that node.
  Size:
- **Implicit**: Adjacency lists computed on-demand.
  Can be used for infinite graphs!

Unweighted graphs have all weights either 0 or 1.
Undirected graphs have every edge in both directions.

Simple Example



Adjacency Matrix:

	a	*b*	*c*	*d*	*e*
a					
b					
c					
d					
e					

Adjacency List:

Graph Search

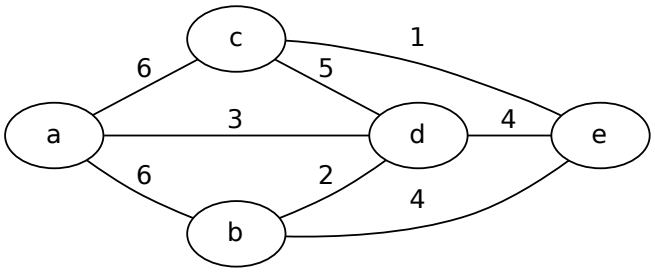
1. Initialize fringe with starting vertex
2. Remove next unvisited vertex from fringe
3. Mark that node as *visited*
4. Add all its neighbors to the fringe
5. Repeat 2-4 until fringe is empty

Search algorithms you know

The previous template covers many algorithms:

- **Depth-first search**
- **Breadth-first search**
- **Dijkstra's Algorithm**

Dijkstra example



All-Pairs Shortest Paths

Problem: All-Pairs Shortest Paths

Input: A graph $G = (V, E)$, weighted, and possibly directed.

Output: Shortest path between every pair of vertices in V

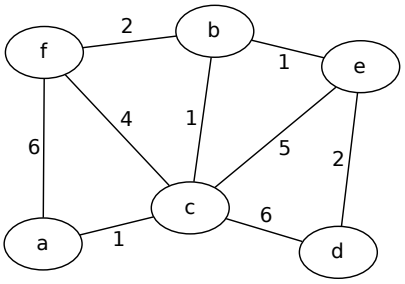
First idea: Run Dijkstra's algorithm from every vertex.
Cost:

Dynamic Programming Solution

Key idea: Keep overwriting shortest paths, using the same memory

This returns a matrix of ALL shortest path lengths at once!

```
def FloydWarshall(AM):
    L = copy(AM)
    n = len(AM)
    for k in range(0, n):
        for i in range(0, n):
            for j in range(0, n):
                L[i][j] = min( L[i][j],
                               L[i][k] + L[k][j]
                             )
    return L
```



	a	b	c	d	e	f
a						
b						
c						
d						
e						
f						

Analysis of Floyd-Warshall

- Time:
- Space:
- **Advantages**:

Transitive Closure

Examples of reachability questions:

- Is there any way out of a maze?
- Is there a flight plan from one airport another?
- Can you tell me a is greater than b without a direct comparison?

Precomputation/query formulation: Same graph, many reachability questions.

Transitive Closure Problem

Input: A graph $G = (V, E)$, unweighted, possibly directed
Output: Whether u is reachable from v, for every $u, v \in V$

TC with APSP

One vertex is reachable from another if the shortest path isn't infinite.

Therefore transitive closure can be solved with repeated Dijkstra's or Floyd-Warshall. Cost will be $\Theta(n^3)$.

Why *might* we be able to beat this?

Another Dynamic Solution

What if every path can only have at most k edges?

Let L_k be the reachability matrix using only k-length paths at most.

- **Base case**: $k = 1$, then $L_1 = A$, the adjacency matrix itself!
- **Recursive step**: A length-$(k+1)$ path exists, if there is a length-k path, followed by a single edge.
- **Termination**: Every path has length at most $n-1$.
  So L_{n-1} is the final answer.

Boolean Arithmetic

Update step: $L_{k+1}[i,j] =$

Boolean Algebra

- The $+$ operation becomes $\vee$
- The $\cdot$ operation becomes $\wedge$

Update step becomes:

TC with Boolean Matrix Multiplication

We start with
$T_0 =$
$T_1 =$

We want to compute $T_{n-1} =$

How to do each multiplication?

The most amazing connection

(Pay attention. Minds will be blown in 3...2...1...)

Optimization Problems

An optimization problem is one where there are many solutions, and we have to find the "best" one.

Examples we have seen:

Optimal solution can often be made as a series of "moves"
(Moves can be parts of the answer, or general decisions)

Greedy Design Paradigm

A greedy algorithm solves an optimization problem by a sequence of "greedy moves".

Greedy moves:

- Are based on "local" information
- Don't require "looking ahead"
- Should be fast to compute!
- Might **not** lead to optimal solutions

Example: Counting change

Appointment Scheduling

Problem

Given n requests for EI appointments, each with start and end time, how to schedule the maximum number of appointments?

For example:

Name	Start	End
Billy	8:30	9:00
Susan	9:00	10:00
Brenda	8:00	8:20
Aaron	8:55	9:05
Paul	8:15	8:45
Brad	7:55	9:45
Pam	9:00	9:30

Greedy Scheduling Options

How should the greedy choice be made?

1. First come, first served
2. Shortest time first
3. Earliest finish first

Which one will lead to optimal solutions?

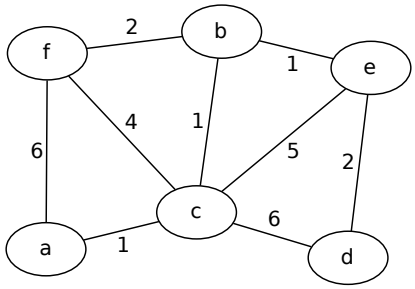
Proving Greedy Strategy is Optimal

Two things to prove:

1. Greedy choice is always part of an optimal solution
2. Rest of optimal solution can be found recursively

Back to graphs

Challenge: Connect a network using a minimal amount of wiring.



MSTs

Recall:

- A tree is a connected graph with no cycles.
- A tree with n vertices always has $n - 1$ edges, exactly.

Spanning tree: a tree within a larger graph, that includes all the vertices

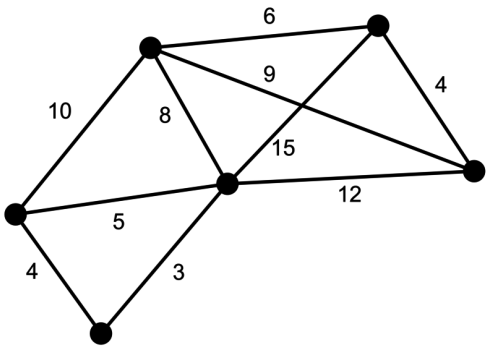
Minimum spanning tree: A spanning tree with the least possible total edge weight

Prim's Algorithm

A greedy algorithm for MST.

1. Start at any vertex. That's your initial tree T
2. Add the least-weight edge from T to the rest of the graph.
3. Keep going until T has n vertices.

Prim's Example



What algorithm does this remind you of?

Correctness of Prim's algorithm

Theorem

For any vertex v in a graph G, the MST of G always contains v's least-weight neighboring edge.

Analysis of Prim's algorithm

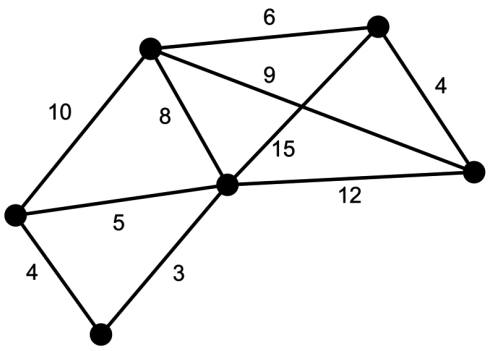
- Which data structures should we use?
- How many times are each operation performed?
- **Total cost**:

Kruskal's Algorithm

A different greedy algorithm for the same problem!

1. Start with your tree T being empty
2. Add the least-weight edge in G that doesn't introduce a cycle in T
3. Repeat!

Kruskal's Example



Disjoint-set data structure

How to keep track of the "connected components" of T?

Disjoint Set ADT

- **create(items)**:
- **find(x)**:
- **union(x,y)**:

Data structure ideas?

Analysis of Kruskal's algorithm

- Which data structures should we use?
- How many times are each operation performed?
- **Total cost**:

Another paradigm?

Prim's and Kruskal's utilize the Greedy paradigm.

They also depend heavily on **data structures**.

How would you make these algorithms faster?

Matchings

Pairing up people or resources is a common task.

We can model this task with graphs:

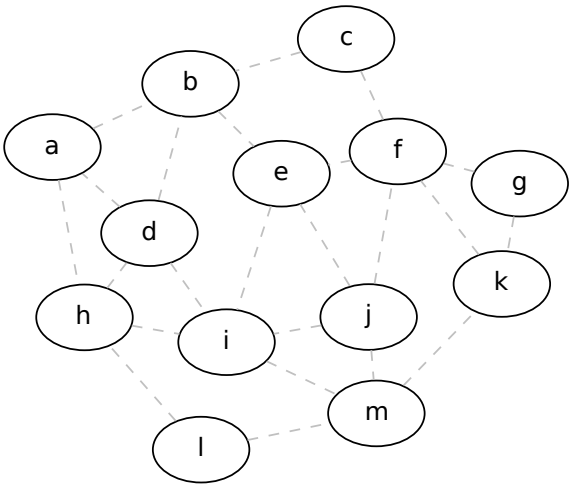
Maximum Matching Problem

Given an undirected, unweighted graph $G = (V, E)$, find a subset of edges $M \subseteq E$ such that:

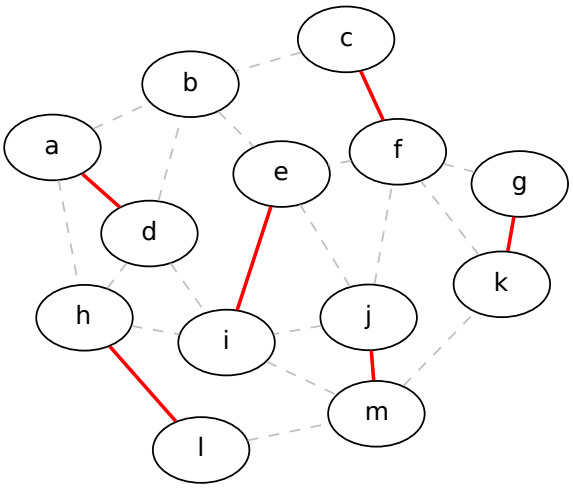
- Every vertex touches at most one edge in M
- The size of M is as large as possible

Greedy Algorithm: Repeatedly choose any edge that goes between two unpaired vertices and add it to M.

Greedy matching example



Maximum matching example



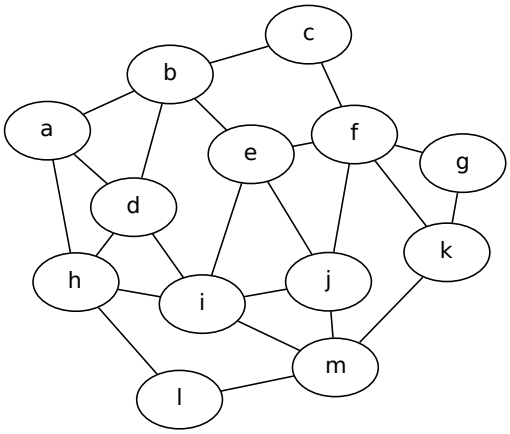
How good is the greedy solution?

Theorem: The optimal solution is at most ___ times the size of one produced by the greedy algorithm.

Proof:

Vertex Cover

Problem: Find the smallest set of vertices that touches every edge.



Approximating VC

Approximation algorithm for minimal vertex cover:

1. Find a greedy maximal matching
2. Take both vertices in every edge in the matching

Why is this always a vertex cover?

How good is the approximation?

Traveling Salesman Problem

TSP Definition

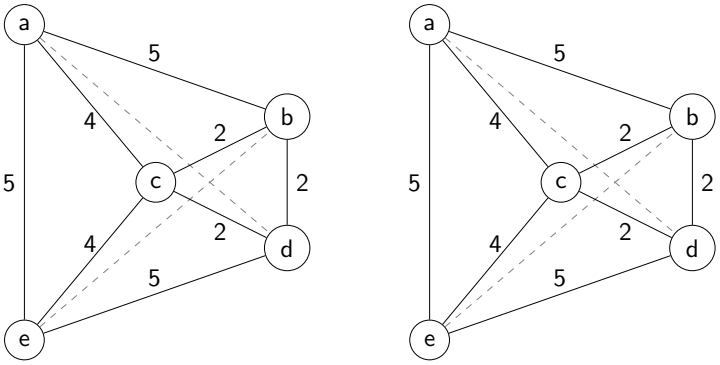
Input: Graph $G = (V, E)$
Output: The shortest cycle that includes every vertex exactly once, or `FAIL` if none exist.

- Sample applications:

How do you confront a problem that seems impossibly hard?

MSTs and TSP

Theorem: Length of TSP tour is at least the size of a MST.



Branch and Bound

How to compute the optimal TSP?

1. Pick a starting vertex
2. Explore every path, depth-first
3. Return the least-length Hamiltonian cycle

This is really slow (*of course*!)

Branch and bound idea:

- Define a quick lower bound on remaining subproblem (MST!)
- Stop exploring when the lower bound exceeds the best-so-far

Simplified TSP

Solving the TSP is really hard; some special cases are a bit easier:

Metric TSP

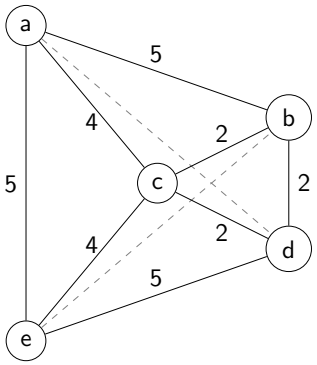
- Edge lengths "obey the triangle inequality":
  $w(a, b) + w(b, c) \geq w(a, c) \forall a, b, c \in V$
- What does this mean about the graph?

Euclidean TSP

- Graph can be drawn on a 2-dimensional map.
- Edge weights are just distances!
- (Sub-case of Metric TSP)

Approximating Metric TSP

Idea: Turn any MST into a TSP tour.

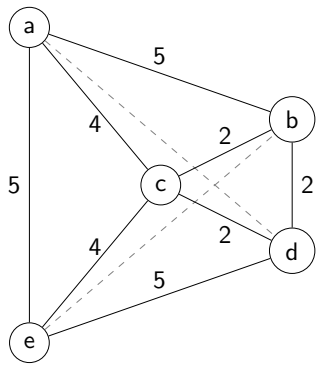


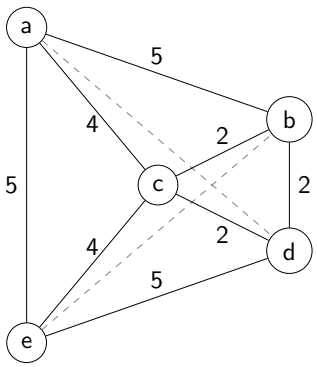
How good is the approximation?

Greedy TSP

Greedy strategies:

- Nearest neighbor
- Smallest "good" edge





Local Refinement

Idea: Take any greedy solution, then make it better.

2-OPT refinement:

- Take a cycle with (a, b) and (c, d)
- Replace with (a, c) and (b, d)