Basic Terminology

REVIEW from Data Structures!

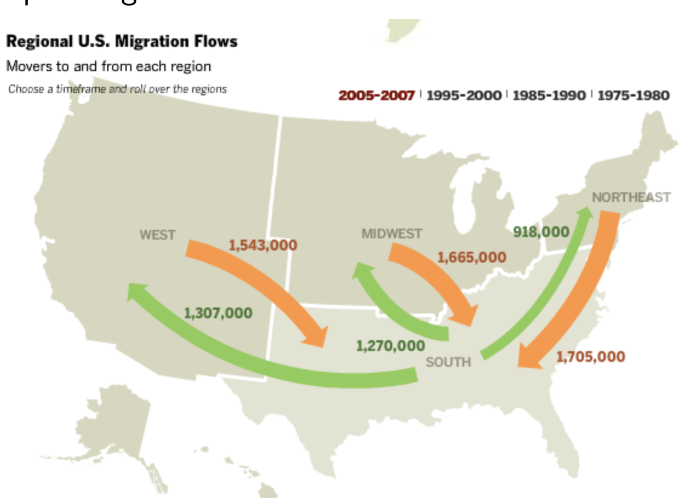
$G = (V, E)$; V is set of n nodes, E is set of m edges

- **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!)

Example: Migration Flows



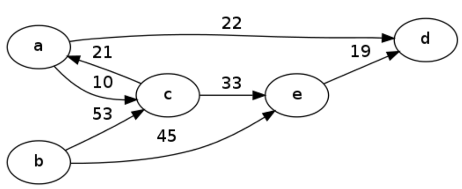
Source: http://www.pewsocialtrends.org/2008/12/17/u-s-migration-flows/

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:

Search Template

```
def genericSearch(G, start, end):
    colors = {}
    for u in G.V:
        colors[u] = "white"
    # initialize fringe with node-weight pairs
    while len(fringe) > 0:
        (u, w1) = fringe.top()
        if colors[u] == "white":
            colors[u] = "gray"
            for (v, w2) in G.edgesFrom(u):
                if colors[v] == "white":
                    fringe.insert((v, w1+w2))
        elif colors[u] == "gray":
            colors[u] = "black"
        else:
            fringe.remove((u, w1))
```

Basic Searches

To find a path from u to v,
initialize `fringe` with $(u, 0)$,
and exit when we color v to "gray".

Two choices:

- **Depth-First Search**
  `fringe` is a stack. Updates are pushes.
- **Breadth-First Search**
  `fringe` is a queue. Updates are enqueues.

DAGs

Some graphs are acyclic by nature.

An acyclic undirected graph is a...

DAGs (Directed Acyclic Graphs) are more interesting:

- Can have more than $n - 1$ edges
- Always at least one "source" and at least one "sink"
- Examples:

Linearization

Problem

Input: A DAG $G = (V, E)$

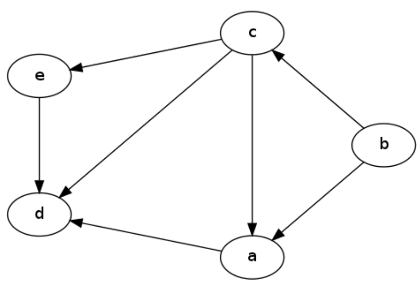
Output: Ordering of the n vertices in V as
$(u_1, u_2, \ldots, u_n)$ such that only "forward edges" exist,
i.e., for all $(u_i, u_j) \in E)$, $i < j$.

(Also called "topological sort".)

Applications:

```
def linearize(G):
    order = [];   colors = {};   fringe = []
    for u in G.V:
        colors[u] = "white"
        fringe.append(u)
    while len(fringe) > 0:
        u = fringe[-1]
        if colors[u] == "white":
            colors[u] = "gray"
            for (v,w2) in G.edgesFrom(u):
                if colors[v] == "white":
                    fringe.append(v)
        elif colors[u] == "gray":
            colors[u] = "black"
            order.insert(0, u)
        else:
            fringe.pop()
    return order
```


Linearization Example



Properties of DFS

- Every vertex in the stack is a child of the first gray vertex below it.
- Every descendant of u is a child of u or a descendant of a child of u.
- In a DAG, when a node is colored gray its children are all white or black.
- In a DAG, every descendant of a black node is black.

Dijkstra's Algorithm

Dijkstra's is a modification of BFS to find shortest paths.

Solves the single source shortest paths problem.

Used millions of times every day (!) for packet routing

Main idea: Use a minimum priority queue for the fringe

Requires all edge weights to be non-negative

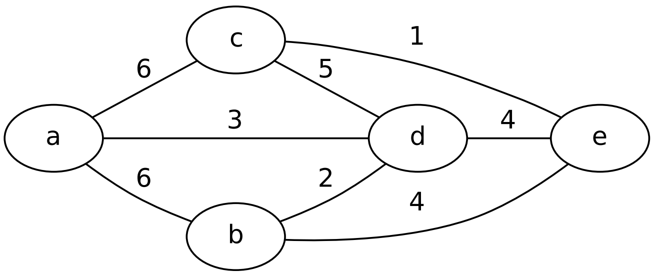
Differences from the search template

- `fringe` is a priority queue
- No gray nodes! (No post-processing necessary.)

Useful variants:

- Keep track of the actual paths as well as path lengths
- Stop when a destination vertex is found

Dijkstra example




```
def dijkstraHeap(G, start):
    shortest = {}
    colors = {}
    for u in G.V:
        colors[u] = "white"
    fringe = [(0, start)] # weight goes first for ordering.
    while len(fringe) > 0:
        (w1, u) = heappop(fringe)
        if colors[u] == "white":
            colors[u] = "black"
            shortest[u] = w1
            for (v, w2) in G.edgesFrom(u):
                heappush(fringe, (w1+w2, v))
    return shortest
```

```
def dijkstraArray(G, start):
    shortest = {}
    fringe = {}
    for u in G.V:
        fringe[u] = infinity
    fringe[start] = 0
    while len(fringe) > 0:
        w1 = min(fringe.values())
        for u in fringe:
            if fringe[u] == w1:
                break
        del fringe[u]
        shortest[u] = w1
        for (v, w2) in G.edgesFrom(u):
            if v in fringe:
                fringe[v] = min(fringe[v], w1+w2)
    return shortest
```

Dijkstra Implementation Options

	Heap	Unsorted Array
Adj. Matrix		
Adj. List		

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

Matchings

Pairing up people or resources is a common task.

We can model this task with graphs:

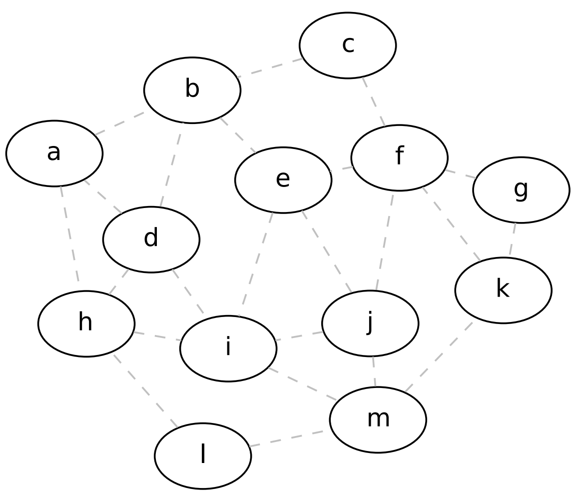
Maximum Matching Problem

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

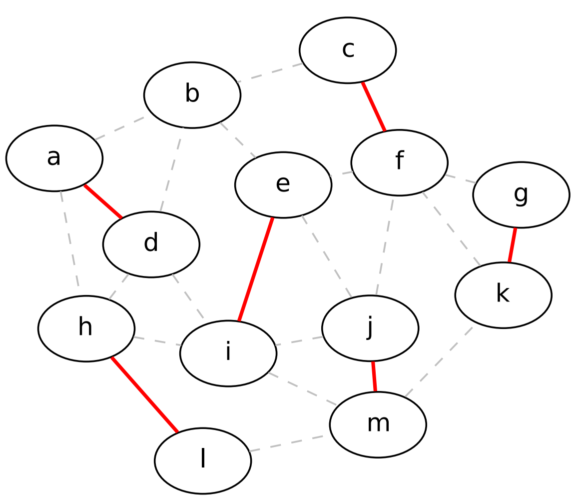
- Every vertex touches at most one edge in M
- The size of M is as large as possible

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:

Spanning Trees

A *spanning tree* in a graph is a connected subset of edges that touches every vertex.

Dijkstra's algorithm creates a kind of spanning tree.
This tree is created by **greedily** choosing the "closest" vertex at each step.

We are often interested in a minimal spanning tree instead.

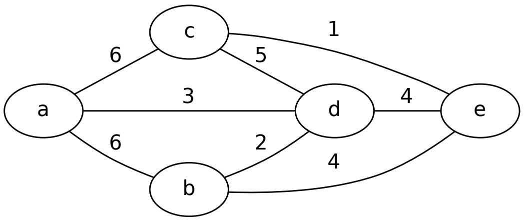
MST Algorithms

There are two **greedy** algorithms for finding MSTs:

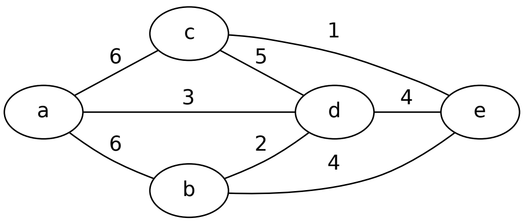
- **Prim's**. Start with a single vertex, and grow the tree by choosing the least-weight fringe edge.
  Identical to Dijkstra's with different weights in the "update" step.
- **Kruskal's**. Start with every vertex (a *forest* of trees)
  and combine trees by using the lease-weight edge between them.

MST Examples

- Prim's:



- Kruskal's:



All-Pairs Shortest Paths

Let's look at a new problem:

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

Many applications in the precomputation/query model:

Repeated Dijkstra's

First idea: Run Dijkstra's algortihm from every vertex.

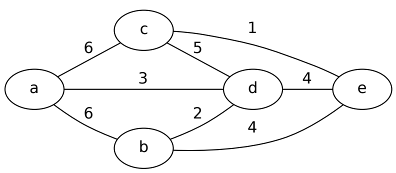
Cost:

- Sparse graphs:

- Dense graphs:

Storing Paths

- Naïve cost to store all paths:
- Memory wall
- Better way:



	a	b	c	d	e
a					
b					
c					
d					
e					

Recursive Approach

Idea for a simple recursive algortihm:

- New parameter k: The highest-index vertex visited in any shortest path.
- Basic idea: Path either contains k, or it doesn't.

Three things needed:

1. **Base case**: $k = -1$. Shortest paths are just single edges.
2. **Recursive step**: Use basic idea above.
   Compare shortest path containing k to shortest path without k.
3. **Termination**: When $k = n$, we're done.

Recursive Shortest Paths

Shortest path from i to j using only vertices 0 up to k.

```
def recShortest(AM, i, j, k):
    if k == -1:
        return AM[i][j]
    else:
        option1 = recShortest(AM, i, j, k-1)
        option2 = recShortest(AM, i, k, k-1) + recShorte
        return min(option1, option2)
```

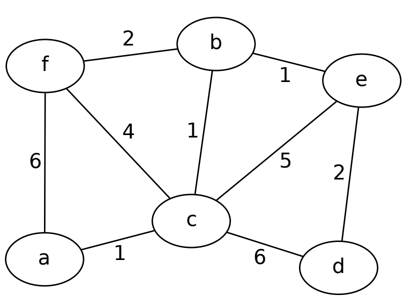
Analysis:

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**:

Another Dynamic Solution

What if k is the greatest number of edges in each shortest path?

Let L_k be the matrix of shortest-path lengths with at most k edges.

- **Base case**: $k = 1$, then $L_1 = A$, the adjacency matrix itself!
- **Recursive step**: Shortest $(k+1)$-edge path is the minimum of k-edge paths, plus a single extra edge.
- **Termination**: Every path has length at most $n - 1$.
  So L_{n-1} is the final answer.

Min-Plus Arithmetic

Update step: $L_{k+1}[i,j] = \min_{0 \le \ell < n} \left(L_k[i,\ell] + A[\ell,j] \right)$

Min-Plus Algebra

- The $+$ operation becomes "min"
- The $\cdot$ operation becomes "plus"

Update step becomes:

APSP with Min-Plus Matrix Multiplication

We want to compute A^{n-1}.

- Initial idea: Multiply $n-1$ times.
- Improvement:
- Further improvement?

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?

Back to Algebra

Define T_k as the reachability matrix **using at most k edges in a path**.

What is T_0?
What is T_1?

Formula to compute T_{k+1}:

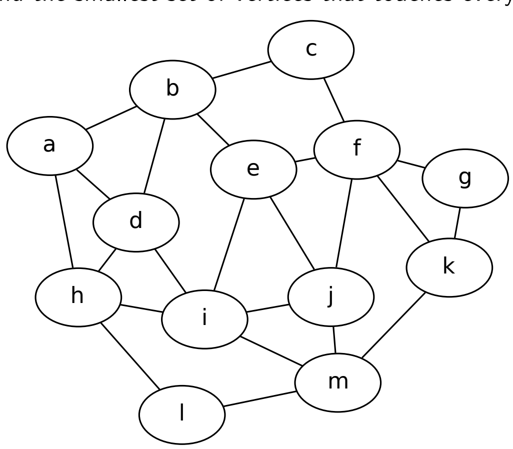
Therefore transitive closure is just:

The most amazing connection

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

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?