Basic Terminology

**REVIEW from Data Structures!**

\[ G = (V, E); V \text{ is set of } n \text{ nodes, } E \text{ is set of } m \text{ edges} \]

- **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**


Regional U.S. Migration Flows

Movers to and from each region

Choose a time period (if not, use the region)

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 \( \infty \). 
  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: 

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency List:

```
Simple Example
Adjacency Matrix:
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Template

```

```python
search(G)
1 colors := size-n array of "white"s
2 fringe := new collection
3 // initialize fringe with node-weight pairs
4 while fringe not empty do
5   (u, w1) := fringe.top()
6   if colors[u] = "white" then
7     colors[u] := "gray"
8     for each outgoing edge (u, v, w2) of u do
9       fringe.update(v, w1+w2)
10   end for
11 else if colors[u] = "gray" then
12   colors[u] := "black"
13   fringe.remove(u, w1)
14 end if
15 end while
```
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:
linearize(G)

1 order := empty list
2 colors := size -n array of "white"s
3 fringe := new stack
4 add every node in V to fringe
5 while fringe not empty do
6   (u,w1) := fringe.top()
7   if colors[u] = "white" then
8     colors[u] := "gray"
9       for each outgoing edge (u,v,w2) of u do
10          fringe.push(v,w2)
11     end for
12   else if colors[u] = "gray" then
13     colors[u] := "black"
14     order := u, order
15     fringe.remove(u,w1)
16   end if
17 end while

Linearization Example

```
        c
       /|
      /  |
     e   d
       |
       |
        a
```

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**

dijkstra(G,u)

1. colors := size-n array of "white"s
2. fringe := new minimum priority queue
3. for each \( v \) in \( V \) do
   4. add (\( v \), infinity) to fringe
   5. fringe.update(\( u \), 0)
4. while fringe not empty do
   5. (\( u \), \( w_1 \)) := fringe.removeMin()
   6. colors[\( u \)] := "black"
   7. print (\( u \), \( w_1 \))
   8. for each edge (\( u \), \( v \), \( w_2 \)) with colors[\( v \)]="white" do
      9. fringe.update(\( v \), \( w_1 + w_2 \))
9. end for
10. end while
Differences from the search template

- fringe is a priority queue
- fringe is initialized with every node
- Updates are done to existing fringe elements
- 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

Dijkstra Implementation Options

<table>
<thead>
<tr>
<th></th>
<th>Heap</th>
<th>Unsorted Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adj. Matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adj. List</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Billy</td>
<td>8:30</td>
<td>9:00</td>
</tr>
<tr>
<td>Susan</td>
<td>9:00</td>
<td>10:00</td>
</tr>
<tr>
<td>Brenda</td>
<td>8:00</td>
<td>8:20</td>
</tr>
<tr>
<td>Aaron</td>
<td>8:55</td>
<td>9:05</td>
</tr>
<tr>
<td>Paul</td>
<td>8:15</td>
<td>8:45</td>
</tr>
<tr>
<td>Brad</td>
<td>7:55</td>
<td>9:45</td>
</tr>
<tr>
<td>Pam</td>
<td>9:00</td>
<td>9:30</td>
</tr>
</tbody>
</table>
Greedy Scheduling Options

How should the greedy choice be made?

- First come, first served
- Shortest time first
- Earliest finish first

Which one will lead to optimal solutions?

---

Proving Greedy Strategy is Optimal

Two things to prove:

- Greedy choice is always part of an optimal solution
- 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 least-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 algorithm from every vertex.

**Cost:**
- Sparse graphs:

Storing Paths

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

Idea for a simple recursive algorithm:
- 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:
- Base case: $k = -1$. Shortest paths are just single edges.
- Recursive step: Use basic idea above. Compare shortest path containing $k$ to shortest path without $k$.
- Termination: When $k = n$, we’re done.

Recursive Shortest Paths

$\text{rshort}(A, i, j, k)$

**Input**: Adjacency matrix $A$ and indices $i, j, k$

**Output**: Shortest path from $i$ to $j$ that only goes through vertices $0$–$k$

```
1 if $k = -1$ then
2    return $A[i, j]$
3 else
4    option1 := $\text{rshort}(A, i, j, k-1)$
5    option2 := $\text{rshort}(A, i, k, k-1) + \text{rshort}(A, k, j, k-1)$
6    return min(option1, option2)
7 end if
```

Analysis:
Dynamic Programming Solution

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

**FloydWarshall(A)**

**Input:** Adjacency matrix $A$

**Output:** Shortest path lengths between every pair of vertices

```plaintext

L = copy(A)

for k from 0 to n do
    for i from 0 to n - 1 do
        for j from 0 to n - 1 do
            L[i,j] := min (L[i,j], L[i,k] + L[k,j])

end for
end for
return L
```

---

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

Min-Plus Algebra
- The + operation becomes “min”
- The · 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?