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 \( \infty \).
  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:

```
 a | b | c | d | e
---|---|---|---|---
a |  |  |  |  |
b |  |  |  |  |
c |  |  |  |  |
d |  |  |  |  |
e |  |  |  |  |
```

Adjacency List:

SI 335 (USNA) Unit 5 Spring 2015 4 / 46

Graph Search

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

SI 335 (USNA) Unit 5 Spring 2015 5 / 46

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!

```python
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
```
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 $u, v \in V$
Transitive Closure

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 $\lor$
- The $\cdot$ operation becomes $\land$

Update step becomes:
Transitive Closure

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...)
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?
Greedy Algorithms

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

Spanning Trees

Back to graphs
Challenge: Connect a network using a minimal amount of wiring.

\[
\begin{array}{c}
\text{f} \\
\text{a} \\
\text{c} \\
\text{d} \\
\text{e} \\
\text{b}
\end{array}
\]

\[
\begin{array}{cccccccc}
f & 2 & b \\
6 & 4 & 1 & e \\
1 & 5 & 2 & c \\
ad & 6 & c \\
\end{array}
\]

MSTs

Recall:
1. A tree is a connected graph with no cycles.
2. 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!

- Start with your tree $T$ being empty
- Add the least-weight edge in $G$ that doesn't introduce a cycle in $T$
- 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?
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 $M$. 
Matchings

How good is the greedy solution?

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

**Proof:**

---

Hard Graph Problems

Vertex Cover

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

![Graph with vertices a, b, c, d, e, f, g, h, i, j, k, l, m connected in a network.]

---

Approximating VC

Approximation algorithm for minimal vertex cover:

- Find a greedy maximal matching
- 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.

![Graph with vertices a, b, c, d, e and edges](image)

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 $(b, d)$