Graphs

- Representation of things and relationships between them
  - People (vertices)
  - Relationships (edges)

Diagram:

- Simple graph: at most one copy of edges
- Simple path: no repeated vertices
- No self-loops

Paths:
- Seq of vertices s.t. edge exists between adjacent vertices
  - J6 SG AS DT VP

Simple paths:

Paths:

Simple paths:

Cycles:

Simple cycles:
int foo(int n, int c)
{
    if (n == c)
    {
        return 0;
    }
    int i = 1;
    while (i < n)
    {
        if (i % c == 3)
        {
            if (n % 2 == 1)
            {
                return 0;
            }
        }
        i++;
    }
    return 0;
}

Flow Control Graph

- **Vertices:** Lines of code
- **Edges:** u → v if v can immediately follow u

Is there a path from beginning to end that doesn't go through return? Yes
Feedback Arc Set

- Vertices: teams
- Edges: \( u \rightarrow v \) if \( u \) lost to \( v \)

Feedback Arc Set: What is minimum number of edges you need to remove to make graph acyclic (no cycles)?

1. If there is a cycle? [Easy]
2. If not, find ordering so all edges go in same dir [also easy]
3. If so, find ordering to minimize # of wrong-way edges [hard]

- Brute force: For each ordering of \( n \) teams, count # of wrong-way edges and keep track of ordering giving minimum so far.

NP-complete: No one has done much better than inefficient brute force (like TSP). No one has proved it is impossible to get poly-time algorithm.
Weighted Graph

- Each edge labelled with weight
- Vertices: cities
- Edges: roads between cities
- Weights: travel times

Given start and end, find path that minimizes total weight of edges in path.

Vertices: intersections
Edges: road segments between intersections
Graph Representation

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>Col</th>
<th>D</th>
<th>Pr</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Col</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>D</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Pr</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>H</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Map vertex labels to indices

| Y  | 0   |
| Col | 1   |
| D  | 2   |
| Pr | 3   |
| H  | 4   |

Adjacency List

<table>
<thead>
<tr>
<th>address</th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Y</td>
<td>Col</td>
<td>D</td>
<td>Pr</td>
<td>H</td>
</tr>
</tbody>
</table>

To determine if edge exists

- O(1) expected translate label to index
- O(1) get list at resulting index
- O(n) worst search list for to vertex
- O(n) worst-case

Has-edge (Pr, Col)

Adj Set: use hash table for set representation

- O(1) expected
- O(1)
- O(1) expected
- O(1) expected
for each edge

\[
\text{adj matrix: } \text{for each row } r \text{ and each col } c \to O(n^2) \text{ iterations total}
\]

\[
\text{if } \text{adj}[r][c] = T \rightarrow \text{process edge } (r, c) \quad O(1) \text{ (assuming } O(1) \text{ for)}
\]

\[
O(n^2) \text{ total}
\]

\[
\text{adj list: } \text{for each list } u \to O(n) \text{ iterations}
\]

\[
\sum_{i=1}^{n} 1 + \sum_{j=1}^{c} \rightarrow \text{ for each } v \text{ on } \text{adj}[u] \quad \text{process edge } (u, v) \rightarrow O(1)
\]

\[
\sum_{i=1}^{n} (1 + c \cdot \text{outdegree}(i)) = \sum_{i=1}^{n} 1 + c \cdot \sum_{i=1}^{n} \text{outdegree}(i) = n + c \cdot \sum_{i=1}^{n} \text{outdegree}(i)
\]

\[
n - 1 \leq m \leq n(n - 1)
\]