Graphs and Search Algorithms
Social Network Backend

Want to build social network app tracking:

• users

• *friend* relationship between some pairs of users
Graph Basics

Network is a graph:
- **nodes or vertices** \( V = \{A, B, C, D, E, F\} \)
Graph Basics

Network is a graph:

- **nodes or vertices** \( V = \{A, B, C, D, E, F\} \)
- **edges** \( E = \{\{A, B\}, \{A, C\}, \{B, C\}, \ldots\} \)
Graph Basics

Network is a graph:

- **nodes** or **vertices** $V = \{A, B, C, D, E, F\}$
- **edges** $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \ldots\}$
- can be **directed** (one-way) or **undirected**
1. Raw vertex/edge lists

\[ V = \{ A, B, C, D, E, F \} \]
\[ E = \{ \{ A, B \}, \{ A, C \}, \{ B, C \}, \ldots \} \]
Graph Data Structs

1. Raw vertex/edge lists
   \[ V = \{A, B, C, D, E, F\} \]
   \[ E = \{\{A, B\}, \{A, C\}, \{B, C\}, \ldots\} \]

2. Adjacency list
   \[ A : \{B, C\} \]
   \[ B : \{A, C\} \]
   \[ C : \{A, B, E\} \]
   \[ \ldots \]
Graph Data Structs

1. Raw vertex/edge lists
   \[ V = \{A, B, C, D, E, F\} \]
   \[ E = \{\{A, B\}, \{A, C\}, \{B, C\}, \ldots\} \]

2. Adjacency list

3. Adjacency matrix
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
</table>
   A | 0 | 1 | 1 | 0 | 0 | 0 |
   B |   |   |   |   |   |   |
   C |   |   |   |   |   |   |
   D |   |   |   |   |   |   |
   E |   |   |   |   |   |   |
   F |   |   |   |   |   |   |
Graph Data Structs

1. Raw vertex/edge lists
   
   \[ V = \{ A, B, C, D, E, F \} \]
   
   \[ E = \{ \{ A, B \}, \{ A, C \}, \{ B, C \}, \ldots \} \]

2. Adjacency list

3. Adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Data Structs

What is the space cost of each option?

Raw lists:

Adjacency list:

Adjacency matrix:
Graph Data Structs

What is the space cost of each option?

**Raw lists:** \( O(|V| + |E|) \)

**Adjacency list:** \( O(|V| + |E|) \)

**Adjacency matrix:** \( O(|V|^2) \)
Graph Operations

Given two vertices, are they neighbors?

Raw lists:

Adjacency list:

Adjacency matrix:
Graph Operations

Given two vertices, are they neighbors?

**Raw lists:** search entire edge list  $O(|E|)$

**Adjacency list:**

**Adjacency matrix:**
Graph Operations

Given two vertices, are they neighbors?

**Raw lists:** search entire edge list  $O(|E|)$

**Adjacency list:** search one adjacency list
(technically  $O(|E|)$)

**Adjacency matrix:**
Graph Operations

Given two vertices, are they neighbors?

**Raw lists:** search entire edge list \( O(|E|) \)

**Adjacency list:** search one adjacency list (technically \( O(|E|) \))

**Adjacency matrix:** look up entry \( O(1) \)
Graph Operations

Given a vertex, who are the neighbors?

Raw lists: search entire edge list $O(|E|)$

Adjacency list:

Adjacency matrix:
Graph Operations

Given a vertex, who are the neighbors?

Raw lists: search entire edge list $O(|E|)$

Adjacency list: nothing to do… $O(1)$

Adjacency matrix: search row of matrix $O(|V|)$
Friend Network

Given a social network containing people (vertices) and friend relationships (edges), A is in the same friend network as B if

• they are the same person

• A is friends with someone in that is in the same friend network as B
Friend Network

Are A and E in the same friend network?
Friend Network

Are A and E in the same friend network?

Basic idea: start at A and “flood fill” along edges, and see if we ever hit E

(We will need to create a “visited” flag for vertices)
friendNetwork(A, B)
for each vertex v:
  v.visited = false;
return search(A);

search(v)
if(v == B) return true;
if(v.visited) return false;
v.visited = true;
for each neighbor w:
  if(search(w))
    return true;
return false;
friendNetwork(A, B)  
for each vertex v:  
  v.visited = false;  
return search(A);  

(are there potential issues?)

search(v)  
if(v == B) return true;  
if(v.visited) return false;  
v.visited = true;  
for each neighbor w:  
  if(search(w))  
    return true;  
return false;
Iterative Version

friendNetwork(A, B)
for each vertex v:
    v.visited = false;
stack S = {A};

while(!S.empty())
    v = S.pop();
    if(v == B) return true;
    if(v.visited) continue;
    v.visited = true;
    for each neighbor w:
        S.push(w);
return false;
friendNetwork(A, B) 

for each vertex v:
  v.visited = false;

stack S = {A};

while(!S.empty())
  v = S.pop();
  if(v == B) return true;
  if(v.visited) continue;
  v.visited = true;
  for each neighbor w:
    S.push(w);

return false;
Kevin Bacon Problem

Given a social network and two people A, B, what is the shortest chain of friends from A to B?

Ex:

\[ \text{bacon}(A, A) = 0 \]
\[ \text{bacon}(A, E) = 2 \]
\[ \text{bacon}(C, D) = \infty \]
Kevin Bacon Problem

Intuition: when calculating bacon(A, *) we still want to flood-fill, but we need to guarantee we search friends before friends-of-friends
Breadth-First Search

friendNetwork(A, B)
for each vertex v:
  v.visited = false;
queue Q = {A};
while(!Q.empty())
  v = Q.pop();
  if(v == B) return true;
  if(v.visited) continue;
  v.visited = true;
  for each neighbor w:
    Q.push(w);
return false;
Kevin Bacon Problem

\texttt{bacon(A, B)}
for each vertex \texttt{v}:
\hspace{1cm} \texttt{v.visited = false;}
\hspace{1cm} \texttt{v.dist = infinity;}
queue \texttt{Q = \{A\}};
\texttt{A.dist = 0;}
\texttt{while(!Q.empty())}
\hspace{1cm} \texttt{v = Q.pop();}
\hspace{1cm} if(\texttt{v == B}) return \texttt{v.dist;}
\hspace{1cm} if(\texttt{v.visited}) continue;
\hspace{1cm} \texttt{v.visited = true;}
for each neighbor \texttt{w}:
\hspace{1cm} \texttt{Q.push(w);}
\hspace{1cm} \texttt{w.dist = v.dist + 1;}
return infinity;