Remembering the edit distance example we saw in class last time, we formalized the processing of the recursion as a table:

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td></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>
<td></td>
</tr>
<tr>
<td>E</td>
<td></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>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is there an easier way to get the minimum alignment without having to calculate all the values in the cell?
Remembering the edit distance example we saw in class last time, we formalized the processing of the recursion as a table:

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$D$</th>
<th>$R$</th>
<th>$E$</th>
<th>$A$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td></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>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></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>
<td></td>
</tr>
<tr>
<td>$D$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is there an easier way to get the minimum alignment without having to calculate all the values in the cell?
Remembering the edit distance example we saw in class last time, we formalized the processing of the recursion as a table:

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td></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>
<td></td>
</tr>
<tr>
<td>E</td>
<td></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>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Look at the flow of the computation!
Remembering the edit distance example we saw in class last time, we formalized the processing of the recursion as a table:

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td></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>
<td></td>
</tr>
<tr>
<td>E</td>
<td></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>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Look at the flow of the computation!
Remembering the edit distance example we saw in class last time, we formalized the processing of the recursion as a table:

\[
\begin{array}{cccccc}
\varepsilon & D & R & E & A & D \\
\hline
\varepsilon & & & & & \\
D & & & & & \\
E & & & & & \\
E & & & & & \\
D & & & & & \\
\end{array}
\]

We can solve the problem by turning it into a graph!
Graph Basics
Why Graphs?

• Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links), and many problems that don’t even look like graph problems.
• Fundamental objects in Computer Science, Optimization, Combinatorics
• Many important and useful optimization problems are graph problems
• Graph theory: elegant, fun and deep mathematics
An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

- $V$ is a set of vertices (also referred to as nodes/points)
- $E$ is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.

**Example**

In figure, $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$. 
Example: Modeling Problems as Search

State Space Search
Many search problems can be modeled as search on a graph. The trick is figuring out what the vertices and edges are.

Missionaries and Cannibals

• Three missionaries, three cannibals, one boat, one river
• Boat carries two people, must have at least one person
• Must all get across
• At no time can cannibals outnumber missionaries

How is this a graph search problem?
What are the vertices?
What are the edges?
Problems goes back to 800 CE
Versions with brothers and sisters.
Jealous Husbands.
Lions and buffalo
All bad names to a simple problem...

*Omitted states where cannibals outnumber missionaries*
Problems on DFAs and NFA sometimes are just problems on graphs

• M: DFA/NFA is $L(M)$ empty?
• M: DFA is $L(M) = \Sigma^*$?
• M: DFA, and a string $w$. Does $M$ accepts $w$?
• N: NFA, and a string $w$. Does $N$ accepts $w$?
Graph notation and representation
Notation
An edge in an undirected graphs is an unordered pair of nodes and hence it is a set. Conventionally we use $uv$ for $\{u, v\}$ when it is clear from the context that the graph is undirected.

- $u$ and $v$ are the end points of an edge $\{u, v\}$
- Multi-graphs allow
  - loops which are edges with the same node appearing as both end points
  - multi-edges: different edges between same pairs of nodes
- In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.
Adjacency Matrix
Represent $G = (V, E)$ with $n$ vertices and $m$ edges using a $n \times n$ adjacency matrix $A$ where

- Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
- Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$
Graph adjacency matrix example [10 vertices]

```
1 2 3 4 5 6 7 8 9 10
1 0 0 1 1 0 0 0 0 1 0
2 0 0 0 0 0 0 1 1 0 1
3 1 0 0 0 1 1 1 0 0 0
4 1 0 0 0 0 1 0 0 0 1
5 0 0 1 0 0 1 0 1 1 0
6 0 0 1 1 1 0 1 0 0 0
7 0 1 1 0 0 1 0 0 0 1
8 0 1 0 0 1 0 0 0 1 0
9 1 0 0 0 1 0 0 1 0 0
10 0 1 0 1 0 0 1 0 0 0
```
**Adjacency Lists**
Represent $G = (V, E)$ with $n$ vertices and $m$ edges using adjacency lists:

- For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of $u$. Sometimes $\text{Adj}(u)$ is the list of edges incident to $u$.
- Advantage: space is $O(m + n)$
- Disadvantage: cannot “easily” determine in $O(1)$ time whether $\{i, j\} \in E$
  - By sorting each list, one can achieve $O(\log n)$ time
  - By hashing “appropriately”, one can achieve $O(1)$ time

**Note:** In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.
Graph adjacency list example [10 vertices]

<table>
<thead>
<tr>
<th>vertex</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3, 4, 9</td>
</tr>
<tr>
<td>2</td>
<td>7, 8, 10</td>
</tr>
<tr>
<td>3</td>
<td>1, 5, 6, 7</td>
</tr>
<tr>
<td>4</td>
<td>1, 6, 10</td>
</tr>
<tr>
<td>5</td>
<td>3, 6, 8, 9</td>
</tr>
<tr>
<td>6</td>
<td>3, 4, 5, 7</td>
</tr>
<tr>
<td>7</td>
<td>2, 3, 6, 10</td>
</tr>
<tr>
<td>8</td>
<td>2, 5, 9</td>
</tr>
<tr>
<td>9</td>
<td>1, 5, 8</td>
</tr>
<tr>
<td>10</td>
<td>2, 4, 7</td>
</tr>
</tbody>
</table>
Graph adjacency matrix+list example [10 vertices]

**Adjacency Matrix:**

```
   1  2  3  4  5  6  7  8  9 10
1 0  1  1  0  0  0  0  1  0  
2 0  0  0  0  0  0  0  1  1  0  1
3 1  0  0  1  1  1  0  0  0  
4 0  1  0  0  0  0  1  0  0  0  1
5 0  0  1  0  0  1  0  1  1  0  
6 0  0  1  1  1  0  1  0  0  0  
7 0  1  1  0  0  1  0  0  0  1  
8 0  1  0  0  1  0  0  0  1  0  
9 1  0  0  0  1  0  0  1  0  0  
10 0  1  0  1  0  0  1  0  0  0
```

**Adjacency List:**

<table>
<thead>
<tr>
<th>vertex</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3, 4, 9</td>
</tr>
<tr>
<td>2</td>
<td>7, 8, 10</td>
</tr>
<tr>
<td>3</td>
<td>1, 5, 6, 7</td>
</tr>
<tr>
<td>4</td>
<td>1, 6, 10</td>
</tr>
<tr>
<td>5</td>
<td>3, 6, 8, 9</td>
</tr>
<tr>
<td>6</td>
<td>3, 4, 5, 7</td>
</tr>
<tr>
<td>7</td>
<td>2, 3, 6, 10</td>
</tr>
<tr>
<td>8</td>
<td>2, 5, 9</td>
</tr>
<tr>
<td>9</td>
<td>1, 5, 8</td>
</tr>
<tr>
<td>10</td>
<td>2, 4, 7</td>
</tr>
</tbody>
</table>
Graph adjacency matrix example [20 vertices]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph adjacency matrix example [40 vertices]
Graph adjacency list example [40 vertices]

<table>
<thead>
<tr>
<th>vertex</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6, 26, 34, 36</td>
</tr>
<tr>
<td>2</td>
<td>12, 22, 23, 29</td>
</tr>
<tr>
<td>3</td>
<td>14, 15, 21</td>
</tr>
<tr>
<td>4</td>
<td>8, 19, 28, 36</td>
</tr>
<tr>
<td>5</td>
<td>6, 24, 25, 27</td>
</tr>
<tr>
<td>6</td>
<td>1, 5, 7, 23</td>
</tr>
<tr>
<td>7</td>
<td>6, 25, 32, 39</td>
</tr>
<tr>
<td>8</td>
<td>6, 19, 30</td>
</tr>
<tr>
<td>9</td>
<td>10, 16, 28, 35</td>
</tr>
<tr>
<td>10</td>
<td>9, 25, 27, 35</td>
</tr>
<tr>
<td>11</td>
<td>13, 15, 33, 34</td>
</tr>
<tr>
<td>12</td>
<td>2, 33, 37, 38</td>
</tr>
<tr>
<td>13</td>
<td>11, 15, 17, 25</td>
</tr>
<tr>
<td>14</td>
<td>3, 22, 40</td>
</tr>
<tr>
<td>15</td>
<td>3, 11, 13, 22</td>
</tr>
<tr>
<td>16</td>
<td>9, 20, 23, 33</td>
</tr>
<tr>
<td>17</td>
<td>13, 20, 32, 34</td>
</tr>
<tr>
<td>18</td>
<td>20, 30, 34, 60</td>
</tr>
<tr>
<td>19</td>
<td>6, 8, 31, 37</td>
</tr>
<tr>
<td>20</td>
<td>16, 17, 18, 35</td>
</tr>
<tr>
<td>21</td>
<td>3, 31, 38</td>
</tr>
<tr>
<td>22</td>
<td>2, 14, 15</td>
</tr>
<tr>
<td>23</td>
<td>2, 6, 16, 26</td>
</tr>
<tr>
<td>24</td>
<td>1, 5, 31, 38</td>
</tr>
<tr>
<td>25</td>
<td>5, 7, 10, 13</td>
</tr>
<tr>
<td>26</td>
<td>23, 29</td>
</tr>
<tr>
<td>27</td>
<td>5, 10, 40</td>
</tr>
<tr>
<td>28</td>
<td>6, 9, 30, 36</td>
</tr>
<tr>
<td>29</td>
<td>2, 26</td>
</tr>
<tr>
<td>30</td>
<td>8, 18, 28</td>
</tr>
<tr>
<td>31</td>
<td>19, 21, 24, 37</td>
</tr>
<tr>
<td>32</td>
<td>7, 17, 37, 39</td>
</tr>
<tr>
<td>33</td>
<td>11, 12, 16, 39</td>
</tr>
<tr>
<td>34</td>
<td>1, 11, 17, 18</td>
</tr>
<tr>
<td>35</td>
<td>9, 10, 20, 36</td>
</tr>
<tr>
<td>36</td>
<td>1, 4, 28, 35</td>
</tr>
<tr>
<td>37</td>
<td>12, 19, 31, 32</td>
</tr>
<tr>
<td>38</td>
<td>12, 21, 24, 39</td>
</tr>
<tr>
<td>39</td>
<td>7, 32, 33, 38</td>
</tr>
<tr>
<td>40</td>
<td>14, 18, 27</td>
</tr>
</tbody>
</table>
• Assume vertices are numbered arbitrarily as \{1, 2, \ldots, n\}.
• Edges are numbered arbitrarily as \{1, 2, \ldots, m\}.
• Edges stored in an array/list of size \(m\). \(E[j]\) is \(j^{th}\) edge with info on end points which are integers in range 1 to \(n\).
• Array \(Adj\) of size \(n\) for adjacency lists. \(Adj[i]\) points to adjacency list of vertex \(i\). \(Adj[i]\) is a list of edge indices in range 1 to \(m\).
A Concrete Representation

Array of edges $E$

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$e_j$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

information including end point indices

Array of adjacency lists

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

List of edges (indices) that are incident to $v_i$
A Concrete Representation: Advantages

- Edges are explicitly represented/numbered. Scanning/processing all edges easy to do.
- Representation easily supports multigraphs including self-loops.
- Explicit numbering of vertices and edges allows use of arrays: $O(1)$-time operations are easy to understand.
- Can also implement via pointer based lists for certain dynamic graph settings.
Connectivity
Connectivity

Given a graph $G = (V, E)$:

- **path**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from $v_1$ to $v_k$. **Note**: a single vertex $u$ is a path of length 0.
Connectivity

Given a graph $G = (V, E)$:

- **path**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from $v_1$ to $v_k$. Note: a single vertex $u$ is a path of length 0.

- **cycle**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition. **Caveat**: Some times people use the term cycle to also allow vertices to be repeated; we will use the term **tour**.


Connectivity

Given a graph $G = (V, E)$:

- **path**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from $v_1$ to $v_k$. **Note**: a single vertex $u$ is a path of length 0.

- **cycle**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition. **Caveat**: Some times people use the term cycle to also allow vertices to be repeated; we will use the term **tour**.

- A vertex $u$ is **connected** to $v$ if there is a path from $u$ to $v$. 
Connectivity

Given a graph $G = (V, E)$:

- **path**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from $v_1$ to $v_k$. **Note**: a single vertex $u$ is a path of length 0.

- **cycle**: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition. **Caveat**: Some times people use the term cycle to also allow vertices to be repeated; we will use the term **tour**.

- A vertex $u$ is **connected** to $v$ if there is a path from $u$ to $v$.

- The **connected component** of $u$, $\text{con}(u)$, is the set of all vertices connected to $u$. Is $u \in \text{con}(u)$?
Define a relation $C$ on $V \times V$ as $uCv$ if $u$ is connected to $v$

- In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
- Graph is connected if there is only one connected component.
Connectivity Problems

Algorithmic Problems

• Given graph $G$ and nodes $u$ and $v$, is $u$ connected to $v$?
• Given $G$ and node $u$, find all nodes that are connected to $u$.
• Find all connected components of $G$. 

Can be accomplished in $O(m + n)$ time using BFS or DFS. BFS and DFS are refinements of a basic search procedure which is good to understand on its own.
Algorithmic Problems

- Given graph $G$ and nodes $u$ and $v$, is $u$ connected to $v$?
- Given $G$ and node $u$, find all nodes that are connected to $u$.
- Find all connected components of $G$.

Can be accomplished in $O(m + n)$ time using **BFS** or **DFS**. **BFS** and **DFS** are refinements of a basic search procedure which is good to understand on its own.
Computing connected components in undirected graphs using basic graph search
Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

**Explore**($G, u$):

- $Visited[1..n] \leftarrow FALSE$
- // **ToExplore, S**: Lists
- Add $u$ to **ToExplore** and to **S**
- $Visited[u] \leftarrow TRUE$
- **while** (**ToExplore** is non-empty) **do**
  - Remove node $x$ from **ToExplore**
  - **for each edge** $xy$ in $Adj(x)$ **do**
    - **if** ($Visited[y] = FALSE$) **then**
      - $Visited[y] \leftarrow TRUE$
      - Add $y$ to **ToExplore**
      - Add $y$ to **S**
- Output **S**
Example
Properties of Basic Search

Running Time:
Properties of Basic Search

Running Time:

**BFS** and **DFS** are special case of BasicSearch.

- Breadth First Search (**BFS**): use **queue** data structure to implementing the list \textit{ToExplore}
- Depth First Search (**DFS**): use **stack** data structure to implement the list \textit{ToExplore}
Search Tree

One can create a natural search tree $T$ rooted at $u$ during search.

\begin{center}
\begin{algorithm}
\textbf{Explore}(G, u):

\begin{algorithmic}
\State array \textit{Visited}[1..n]
\State \textbf{Initialize:} \textit{Visited}[i] \leftarrow \text{FALSE} \; \text{for} \; i = 1, \ldots, n
\State \textbf{List:} \; \textit{ToExplore}, \; \textit{S}
\State \text{Add $u$ to $\textit{ToExplore}$ and to $\textit{S}$, $\textit{Visited}[u] \leftarrow \text{TRUE}$}
\State \text{Make tree $T$ with root as $u$}
\State \textbf{while (ToExplore is non-empty) do}
\State \hspace{1em} \text{Remove node $x$ from $\textit{ToExplore}$}
\State \hspace{1em} \text{for each edge $(x, y)$ in $\text{Adj}(x)$ do}
\State \hspace{2em} \text{if ($\textit{Visited}[y] = \text{FALSE}$) }
\State \hspace{3em} \text{\textit{Visited}[y] \leftarrow \text{TRUE}}
\State \hspace{3em} \text{Add $y$ to $\textit{ToExplore}$}
\State \hspace{2em} \text{Add $y$ to $\textit{S}$}
\State \hspace{1em} \text{Add $y$ to $T$ with $x$ as its parent}
\State \textbf{Output $\textit{S}$}
\end{algorithmic}
\end{algorithm}
\end{center}
Finding all connected components

Modify Basic Search to find all connected components of a given graph $G$ in $O(m + n)$ time.
Directed Graphs and Directed Connectivity
Definition
A directed graph \( G = (V, E) \) consists of

- set of vertices/nodes \( V \)
- a set of edges/arcs \( E \subseteq V \times V \).

An edge is an ordered pair of vertices. \((u, v)\) different from \((v, u)\).
Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

- Road networks with one-way streets.
- Web-link graph: vertices are web-pages and there is an edge from page $p$ to page $p'$ if $p$ has a link to $p'$. Web graphs used by Google with PageRank algorithm to rank pages.
- Dependency graphs in variety of applications: link from $x$ to $y$ if $y$ depends on $x$. Make files for compiling programs.
- Program Analysis: functions/procedures are vertices and there is an edge from $x$ to $y$ if $x$ calls $y$. 
Graph $G = (V, E)$ with $n$ vertices and $m$ edges:


- **Adjacency Lists**: for each node $u$, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from $u$.

Default representation is adjacency lists.
Concrete representation discussed previously for undirected graphs easily extends to directed graphs.

Array of edges $E$

|   |   | $e_j$ |   |   |

information including end point indices

Array of adjacency lists

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$v_i$</td>
</tr>
</tbody>
</table>

List of edges (indices) that are incident to $v_i$
Directed Connectivity

Given a graph $G = (V, E)$:

- A (directed) path is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from $v_1$ to $v_k$. By convention, a single node $u$ is a path of length 0.
Directed Connectivity

Given a graph \( G = (V, E) \):

- A (directed) path is a sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \((v_i, v_{i+1}) \in E\) for \( 1 \leq i \leq k - 1 \). The length of the path is \( k - 1 \) and the path is from \( v_1 \) to \( v_k \). By convention, a single node \( u \) is a path of length 0.

- A cycle is a sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \((v_i, v_{i+1}) \in E\) for \( 1 \leq i \leq k - 1 \) and \((v_k, v_1) \in E\). By convention, a single node \( u \) is not a cycle.
Directed Connectivity

Given a graph $G = (V, E)$:

- A **(directed) path** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from $v_1$ to $v_k$. By convention, a single node $u$ is a path of length 0.

- A **cycle** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$ and $(v_k, v_1) \in E$. By convention, a single node $u$ is not a cycle.

- A vertex $u$ can **reach** $v$ if there is a path from $u$ to $v$. Alternatively $v$ can be reached from $u$. 
Directed Connectivity

Given a graph $G = (V, E)$:

- A **(directed) path** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from $v_1$ to $v_k$. By convention, a single node $u$ is a path of length 0.
- A **cycle** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$ and $(v_k, v_1) \in E$. By convention, a single node $u$ is not a cycle.
- A vertex $u$ can **reach** $v$ if there is a path from $u$ to $v$.
- Alternatively $v$ can be reached from $u$.
- Let $rch(u)$ be the set of all vertices reachable from $u$. 
Asymmetricity: $D$ can reach $B$ but $B$ cannot reach $D$
Asymmetricity: $D$ can reach $B$ but $B$ cannot reach $D$

Questions:

- Is there a notion of connected components?
- How do we understand connectivity in directed graphs?
Strong connected components
Definition
Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$. 

Proposition
$C$ is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of $C$: strong connected components of $G$. They partition the vertices of $G$.

$\text{SCC}(u)$: strongly connected component containing $u$. 


Definition
Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in rch(u)$ and $u \in rch(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$. 

Equivalence classes of $C$:
- Strong connected components of $G$.
- They partition the vertices of $G$.

$SCC(u)$: strongly connected component containing $u$. 
Connectivity and Strong Connected Components

**Definition**
Given a directed graph $G$, $u$ is **strongly connected** to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$.

**Proposition**
$C$ is an equivalence relation, that is reflexive, symmetric and transitive.
Connectivity and Strong Connected Components

Definition
Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$.

Proposition
$C$ is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of $C$: strong connected components of $G$. They partition the vertices of $G$.

$\text{SCC}(u)$: strongly connected component containing $u$. 
Strongly Connected Components: Example
Strongly Connected Components: Example
Strongly Connected Components: Example
Strongly Connected Components: Example
Strongly Connected Components: Example
Directed Graph Connectivity Problems

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $rch(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.
- Find the strongly connected component containing node $u$, that is $SCC(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strongly connected components of $G$. 
Graph exploration in directed graphs
Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$.

**Explore**$(G, u)$:

- array $Visited[1..n]$
- Initialize: Set $Visited[i] \leftarrow$ FALSE for $1 \leq i \leq n$
- List: $ToExplore, S$
- Add $u$ to $ToExplore$ and to $S$, $Visited[u] \leftarrow$ TRUE
- Make tree $T$ with root as $u$
- while ($ToExplore$ is non-empty) do
  - Remove node $x$ from $ToExplore$
  - for each edge $(x, y)$ in $Adj(x)$ do
    - if ($Visited[y] =$ FALSE)
      - $Visited[y] \leftarrow$ TRUE
      - Add $y$ to $ToExplore$
      - Add $y$ to $S$
      - Add $y$ to $T$ with edge $(x, y)$
- Output $S$
Example
Example
Example
Example
Example
Example
Properties of Basic Search

Proposition
Explore\((G, u)\) terminates with \(S = rch(u)\).

Proof Sketch.

\begin{itemize}
\item Once \textit{Visited}[i] is set to \textit{TRUE} it never changes. Hence a node is added only once to \textit{ToExplore}. Thus algorithm terminates in at most \(n\) iterations of while loop.
\item By induction on iterations, can show \(v \in S \Rightarrow v \in rch(u)\)
\item Since each node \(v \in S\) was in \textit{ToExplore} and was explored, no edges in \(G\) leave \(S\). Hence no node in \(V - S\) is in \(rch(u)\). Caveat: In directed graphs edges can enter \(S\).
\item Thus \(S = rch(u)\) at termination.
\end{itemize}
Directed Graph Connectivity Problems

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $rch(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.
- Find the strongly connected component containing node $u$, that is $SCC(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strongly connected components of $G$. 

First five problems can be solved in $O(n + m)$ time by via Basic Search (or BFS/DFS). The last one can also be done in linear time but requires a rather clever DFS based algorithm (next lecture).
Directed Graph Connectivity Problems

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $rch(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.
- Find the strongly connected component containing node $u$, that is $SCC(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strongly connected components of $G$.

First five problems can be solved in $O(n + m)$ time by via Basic Search (or BFS/DFS). The last one can also be done in linear time but requires a rather clever DFS based algorithm (next lecture).
Algorithms via Basic Search
• Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
• Given $G$ and $u$, compute $rch(u)$. 
• Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
• Given $G$ and $u$, compute $rch(u)$.

Use $Explore(G, u)$ to compute $rch(u)$ in $O(n + m)$ time.
• Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$. 

Definition (Reverse graph.)

Given $G = (V, E)$, $G^\text{rev}$ is the graph with edge directions reversed $G^\text{rev} = (V, E')$ where $E' = \{(y, x) | (x, y) \in E\}$.

Compute $rch(u)$ in $G^\text{rev}!

• Running time: $O(n + m)$ to obtain $G^\text{rev}$ from $G$ and $O(n + m)$ time to compute $rch(u)$ via Basic Search. If both $Out(v)$ and $In(v)$ are available at each $v$ then no need to explicitly compute $G^\text{rev}$. Can do $Explore(G, u)$ in $G^\text{rev}$ implicitly.
• Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$. Naive: $O(n(n + m))$
• Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$.    Naive: $O(n(n + m))$

**Definition (Reverse graph.)**
Given $G = (V, E)$, $G^{rev}$ is the graph with edge directions reversed
$G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}
• Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$. Naive: $O(n(n + m))$

**Definition (Reverse graph.)**
Given $G = (V, E)$, $G^{\text{rev}}$ is the graph with edge directions reversed $G^{\text{rev}} = (V, E')$ where $E' = \{(y, x) | (x, y) \in E\}$

Compute $\text{rch}(u)$ in $G^{\text{rev}}$!

• **Running time:** $O(n + m)$ to obtain $G^{\text{rev}}$ from $G$ and $O(n + m)$ time to compute $\text{rch}(u)$ via Basic Search. If both $\text{Out}(v)$ and $\text{In}(v)$ are available at each $v$ then no need to explicitly compute $G^{\text{rev}}$. Can do $\text{Explore}(G, u)$ in $G^{\text{rev}}$ implicitly.
$SCC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}$
SCC(G, u) = \{v \mid u \text{ is strongly connected to } v\}

- Find the strongly connected component containing node u. That is, compute SCC(G, u).
$SCC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}$

- Find the strongly connected component containing node $u$. That is, compute $SCC(G, u)$.

$SCC(G, u) = rch(G, u) \cap rch(G^{rev}, u)$
$SCC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}$

- Find the strongly connected component containing node $u$. That is, compute $SCC(G, u)$.

$SCC(G, u) = \text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$

Hence, $SCC(G, u)$ can be computed with $\text{Explore}(G, u)$ and $\text{Explore}(G^{rev}, u)$. Total $O(n + m)$ time.

Why can $\text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$ be done in $O(n)$ time?
Graph $G$ and its reverse graph $G^{rev}$
Graph $G$ a vertex $F$ and its reachable set $rch(G, F)$

Reachable set of vertices from $F$
Graph G a vertex $F$ and the set of vertices that can reach it in $G:rch(G^{rev}, F)$
Graph $G$ a vertex $F$ and its strong connected component in $G$: \[ \text{SCC}(G, F) \]

Graph $G$ \[ rch(G, F) \]

Graph $G^{\text{rev}}$ \[ rch(G^{\text{rev}}, F) \]
• Is $G$ strongly connected?
• Is $G$ strongly connected?

Pick arbitrary vertex $u$. Check if $SCC(G, u) = V$. 
• Find all strongly connected components of $G$. 

Running time: $O(n(m + n))$. 

Question: Can we do it in $O(n(m + n))$ time?
• Find all strongly connected components of $G$.

```plaintext
While $G$ is not empty do
    Pick arbitrary node $u$
    find $S = SCC(G,u)$
    Remove $S$ from $G$
```
• Find all strongly connected components of \( G \).

\[
\text{While } G \text{ is not empty do}
\]
\[
\begin{align*}
\text{Pick arbitrary node } u \\
\text{find } S &= SCC(G, u) \\
\text{Remove } S \text{ from } G
\end{align*}
\]

**Question:** Why doesn’t removing one strong connected component affect the other strong connected components?
• Find all strongly connected components of $G$.

While $G$ is not empty do
   Pick arbitrary node $u$
   find $S = SCC(G, u)$
   Remove $S$ from $G$

**Question:** Why doesn’t removing one strong connected components affect the other strong connected components?

Running time: $O(n(n + m))$. 
• Find all strongly connected components of $G$.

```latex
\text{While } G \text{ is not empty do}
\begin{align*}
\text{Pick arbitrary node } u \\
\text{find } S = \text{SCC}(G, u) \\
\text{Remove } S \text{ from } G
\end{align*}
```

**Question:** Why doesn’t removing one strong connected component affect the other strong connected components?

Running time: $O(n(n + m))$.

**Question:** Can we do it in $O(n + m)$ time?
Find out next time.....