Two techniques for algorithm design

- We have seen recursion techniques so far
- Next few weeks, we will see graph algorithms
- E.g. Dijkstra…
Two techniques of algorithm design

- We have seen recursion techniques so far
- Next few weeks, we will see graph algorithms
- E.g. Dijkstra
Graphs

- 8 vertices
- 10 edges
- 2 components
Graph Representation

- 8 vertices
- 10 edges
- 2 components

Just a representation of the graph!
Graph Definition

- A (simple) graph is
  - Non empty finite set $V$, called vertex set
  - Set $E$ of pairs of vertices, called edges.

  - Undirected $(u,v) = \{u,v\}$
  - Directed $(u,v) = u \rightarrow v$
Graph Representation

- 8 vertices
- 10 edges
- 2 components

Just a representation of the graph!
Graph Representation

• 8 vertices
• 10 edges
• 2 components

Planar Graph! independent of the representation
Graph Representation

- 8 vertices
- 10 edges
- 2 components

Another representation: Intersection graph
Graph Representation

- 8 vertices
- 10 edges
- 2 components

Another representation: Intervals

\[
\begin{align*}
a & \quad b \\
c & \quad d \\
e & \quad f \\
g & \quad h
\end{align*}
\]
For $i < j$

$$LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}$$
For $i < j$

$$LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}$$
Recursion Tree (of Mergesort)
Configuration graph (Tower of Hanoi)

Vertices = legal configurations of discs
Edges = legal move undirected, or directed pointing both ways
The configuration graph of the 4-disk Tower of Hanoi.
DFA as graph

Labeled graph, With conditions

lookup table from transition function is data structure
How to determine if NFA accepts anything? Can s reach an accepting state?

DFS, reachability
NFA to DFA (subset construction)

Some times the graph given is not the right graph!

\[ V = 2^\varnothing \]

\[ E = \{ A \to B \mid \text{for all } u \text{ in } A \text{ there is } v \text{ in } B \text{ such that } u \to v \} \]

This is a 16 node graph!
\( \delta'(P,0) \) and \( \delta'(P,1) \) depend on the input and the node where the transition occurs.

<table>
<thead>
<tr>
<th>( P )</th>
<th>( \mathcal{E} )</th>
<th>( \delta'(P,0) )</th>
<th>( \delta'(P,1) )</th>
<th>( q' \in A' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>as</td>
<td>bs</td>
<td>No</td>
</tr>
<tr>
<td>as</td>
<td>as</td>
<td>ats</td>
<td>bs</td>
<td>No</td>
</tr>
<tr>
<td>bs</td>
<td>bs</td>
<td>as</td>
<td>bts</td>
<td>No</td>
</tr>
<tr>
<td>ats</td>
<td>ats</td>
<td>ats</td>
<td>bts</td>
<td>Yes</td>
</tr>
<tr>
<td>bts</td>
<td>bts</td>
<td>ats</td>
<td>bts</td>
<td>Yes</td>
</tr>
</tbody>
</table>
NFA to DFA (subset construction)

Some times the graph given is not the right graph!

\[ V = 2^Q \]

\[ E = \{ A \rightarrow B \mid \text{for all } u \in A \text{ there is } v \in B \text{ such that } u \rightarrow v \} \]

This is a 16 node graph!

Incremental subset construction was running BFS in the DFA, though we were only explicitly given the NFA graph
Graph Boilerplate

• When I design algorithm on a graph:
  • $V = ?$
  • $E = ?$
  • Problem
  • Algorithm
  • Running time in terms of original input
Graph Algorithms

• “Given a graph G(V,E), do …”

• What does that mean?

• How to represent a graph? (string is represented by array etc..)

• Two standard representations
Adjacency Lists

- Adjacency List (= Array of lists)

```
1
2
...
5
```

undirected graph
Adjacency Lists

- Adjacency List (= Array of lists)

Why Adjacency lists?
- Access each node in $O(1)$ time
- List edges at each node in $O(1)$ time each
- Insert edge
- Hard: is $(u,v)$ in $E$?

More efficient data structure?
- Why use linked lists?

$O(|V|+|E|)$ space
Adjacency Matrix

- Adjacency matrix

$$A(i,j) = 1 \text{ if } (i,j) \text{ edge}$$
$$0 \text{ otherwise}$$

$$O(1) \text{ time to decide if } (u,v) \text{ edge}$$

Always $$O(n^2) \text{ space!}$$

$$O(V) \text{ time to list all the neighbors of a vertex } u$$

even though there are only constant number of edges!
<table>
<thead>
<tr>
<th></th>
<th>Adjacency matrix</th>
<th>Standard adjacency list (linked lists)</th>
<th>Adjacency list (hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(V^2)$</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V + E)$</td>
</tr>
<tr>
<td>Time to test if $uv \in E$</td>
<td>$O(1)$</td>
<td>$O(1 + \min{\deg(u), \deg(v)}) = O(V)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Time to test if $u \rightarrow v \in E$</td>
<td>$O(1)$</td>
<td>$O(1 + \deg(u)) = O(V)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Time to list the neighbors of $v$</td>
<td>$O(V)$</td>
<td>$O(1 + \deg(v))$</td>
<td>$O(1 + \deg(v))$</td>
</tr>
<tr>
<td>Time to list all edges</td>
<td>$\Theta(V^2)$</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V + E)$</td>
</tr>
<tr>
<td>Time to add edge $uv$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)*$</td>
</tr>
<tr>
<td>Time to delete edge $uv$</td>
<td>$O(1)$</td>
<td>$O(\deg(u) + \deg(v)) = O(V)$</td>
<td>$O(1)*$</td>
</tr>
</tbody>
</table>
How to traverse a graph?

- Traversal in general: e.g. you have a data structure with pointers and you want to print it out once

- How to traverse a graph?

```plaintext
RECURSIVELDFS(v):
  if v is unmarked
    mark v
    for each edge vw
      RECURSIVELDFS(w)
```
How to traverse a graph?

**RecursiveDFS(v):**
- if v is unmarked
  - mark v
  - for each edge vw
    - RecursiveDFS(w)

**O(|V|+|E|) time**

**IterativeDFS(s):**
- Push(s)
- while the stack is not empty
  - v ← Pop
  - if v is unmarked
    - mark v
    - for each edge vw
      - Push(w)
How to traverse a graph?

```
TRAVVERSE(s):
  put s into the bag
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      for each edge vw
        put w into the bag
```

- stack = LIFO (DFS)
- Queue = FIFO (BFS)
- Priority Queue = lightest out
  Random, etc
**Whatever First Search**

\[
\text{Traverse}(s):
\]

- put \((\emptyset, s)\) in bag
- while the bag is not empty
  - take \((p, v)\) from the bag
  - if \(v\) is unmarked
    - mark \(v\)
    - \(\text{parent}(v) \leftarrow p\)
    - for each edge \(vw\)
      - put \((v, w)\) into the bag

stack = LIFO (DFS)
Queue = FIFO (BFS)
Priority Queue = lightest out
Random, etc
Traverse(s) marks every vertex in a connected graph exactly once, and the set of pairs \((v, \text{parent}(v))\) with parent\((v)\) not empty, defines a spanning tree of the graph.

BFS tree has shortest paths from s!