Review: BFS

BFS(s):
mark s as "discovered" ⊿ 1
L[0] ← {s}, i ← 0 ⊿ 1
while L[i] is not empty do
L[i + 1] ← empty list ⊿ ≤ n
for all nodes v in L[i] do
for all neighbors w of v do
if w is not marked "discovered" then
mark w as "discovered" ⊿ n
put w in L[i + 1] ⊿ n
end if
end for
end for
i ← i + 1 ⊿ ≤ n
end while

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

Hidden assumption: can iterate over neighbors of v efficiently.

BFS Tree

We can use BFS to make a tree. (blue: "tree edges", dashed: "non-tree edges")

Claim: let \(T\) be the tree discovered by BFS on graph \(G = (V, E)\), and let \((x, y)\) be any edge of \(G\).
Then the layers of \(x\) and \(y\) in \(T\) differ by at most 1.

Proof

Let \((x, y)\) be an edge
Assume \(x\) is discovered first and placed in \(L_i\)
Then \(y \in L_j\) for \(j \geq i\)
When neighbors of \(x\) are explored, \(y\) is either already in \(L_i\) or \(L_{i+1}\), or is discovered and added to \(L_{i+1}\)
Clicker Question 1

Suppose in BFS that the nodes in each layer are explored in a different order (e.g. reverse). Which of the following are true?

A. The nodes that appear in each layer may change
B. The BFS tree may change
C. Both A and B
D. Neither A nor B

BFS and Connectivity

Can we use BFS to detect cycles?

What if we find an already discovered neighbor?

only if it is not the parent (same edge backwards)

for all nodes \( v \) in \( L[i] \) do
  for all neighbors \( w \) of \( v \) do
    if \( w \) is not marked "discovered" then
      mark \( w \) as "discovered"
      put \( w \) in \( L[i] \)
      \( \text{parent}[w] = v \)
    else if \( w \neq \text{parent}[v] \) then
      output "graph has cycle"
  end if
end for

DFS: Recursive Implementation

DFS(\( u \))
mark \( u \) as "explored"
for all edges \((u, v)\) do
  if \( v \) is not "explored" then
    call DFS(\( v \)) recursively
  end if
end for

Running time: \( O(m + n) \) same complexity as BFS
Same assumptions: can traverse neighbor list in time proportional to node degree

Claim: Non-tree edges lead to (indirect) ancestors
### DFS: Non-tree edges lead to ancestors

**Claim:** Let $T$ be the tree discovered by DFS, and let $(x, y)$ be an edge of $G$ that is not in $T$. Then $x$ or $y$ is an ancestor of the other.

**Proof:**
- Let $x$ be the first of the two nodes explored.
- Is $y$ explored at beginning of DFS($x$)? No.
- At some point during DFS($x$), we examine the edge $(x, y)$.
- Is $y$ explored then? Yes. Otherwise we would put $(x, y)$ in $T$.
- $\Rightarrow y$ was explored during DFS($x$)
- $\Rightarrow y$ is a descendant of $x$.

We first see the edge as $(y, x)$ when exploring $y$: ancestor $x$ is already marked explored, so $(y, x)$ is a back edge.

$x$ can’t be parent of $y$, since then $(x, y)$ is a tree edge.

### Graph Representation

- What data structure do we use to represent a graph?
- How do we iterate through nodes, edges, neighbors?
- Has impact on memory efficiency and running time.

### Graph Representation: Adjacency Matrix

An $n$-by-$n$ matrix with $A_{uv} = 1$ if $(u, v)$ is an edge (symmetric)

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

- Space proportional to $n^2$
- Checking if $(u, v)$ is an edge takes $\Theta(1)$ time (lookup)
- Iterating through all neighbors takes $\Theta(n)$
- Iterating through all edges takes $\Theta(n^2)$

### Traversal Implementations

**Generic Graph Traversal**

Let $A$ = data structure of discovered nodes

```
Traverse(s)
   put s in A
   while A is not empty do
      take a node v from A
      if v is not marked "explored" then
         mark v "explored"
         for each edge $(v, w)$ incident to v do
            put w in A
         end for
      end if
   end while
```

**BFS:** $A$ is a queue (FIFO)  
**DFS:** $A$ is a stack (LIFO)

Can a node be discovered (placed in $A$) multiple times? Yes. For BFS, node is explored from parent that added it last (LIFO). For BFS, can avoid by not adding discovered nodes.

**Explored** = have seen this node and explored its outgoing edges

**Discovered** = the "frontier". Have seen the node, but not explored its outgoing edges.

### Generic Graph Traversal

```
Traverse(s)
   put s in A
   while A is not empty do
      take a node v from A
      if v is not marked "explored" then
         mark v "explored"
         for each edge $(v, w)$ incident to v do
            put w in A
         end for
      end if
   end while
```

**BFS:** $A$ is a queue (FIFO)  
**DFS:** $A$ is a stack (LIFO)

Can a node be discovered (placed in $A$) multiple times? Yes. For BFS, node is explored from parent that added it last (LIFO). For BFS, can avoid by not adding discovered nodes.

**Explored** = have seen this node and explored its outgoing edges

**Discovered** = the "frontier". Have seen the node, but not explored its outgoing edges.
Clicker Question 2

```
Clicker Question 2

put s in A
while A is not empty do
    take a node v from A
    if v is not marked "explored" then
        mark v "explored"
        for each edge (v, w) incident to v do
            put w in A
        end for
    end if
end while

What is the maximum number of times a node w can be put in A?

A: once
B: degree(w) + 1 times
C: 2 \cdot degree(w) times
D: |V| times
```

Clicker Question 3

```
Clicker Question 3

Put s in A
DFS(u)
Mark u as "explored"
for each edge (u, v) do
    if v is not "explored" then
        Call DFS(v) recursively
    end if
end for

while A is not empty do
    Take a node v from A
    if v is not "explored" then
        Mark v as "explored"
        for each edge (v, w) do
            put w in A
        end for
    end if
end while

Suppose we have a tree with n nodes, height h and degree d.
Compare recursive and non-recursive DFS in terms of memory used for the stack

A: recursive: \Theta(hd), non-recursive: \Theta(h)
B: recursive: \Theta(h), non-recursive: \Theta(hd)
C: recursive: \Theta(h), non-recursive: \Theta(d)
D: recursive: \Theta(n), non-recursive: \Theta(hd)
```

Exploring all Connected Components

How to explore entire graph even if it is disconnected?

```
while there is some unexplored node s do
    Traverse(s) \rightarrow Run BFS/DFS starting from s.
    Extract connected component containing s
end while

Running time? Does it change?

Naive: O(m + n) per component \Rightarrow O(c(m + n)) if c components.

Better: Search on component C only works on nodes/edges in C
- Time for component C: O(#edges in C + #nodes in C)
- O(n) to detect all components
- Total time: O(m + n)

Usually OK to assume graph is connected.
State if you are doing so and why it does not trivialize the problem.
```

Review and Outlook

```
Review and Outlook

- Graph traversal by BFS/DFS
  - Different versions of general exploration strategy
  - O(m + n) time
  - Produce trees with useful properties (for other problems)
  - Basic algorithmic primitive — used in many other algorithms
    path from s to t, connected components

- Bipartite testing

- Directed graphs
  - Traversal
  - Strong connectivity
  - Topological sorting
```