Lecture 19: Basic Graph Algorithms
Processing Graphs

- Graphs model many scenarios
  - Many problems are presented as graph problems
  - Can then use known general graph algorithms to solve those problems

- Data is inputted as adjacency matrix or, more commonly, an adjacency lists

- To start processing the data, we often need some way to derive structure from this input

- Breadth First Search and Depth First Search are the most common simple ways of imposing structure.
Breadth First Search

BFS idea. Explore outward from $s$ in all possible directions, adding nodes one “layer” at a time.

BFS.

- $L_0 = \{s\}$.
- $L_1 =$ all neighbors of $L_0$.
- $L_2 =$ all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
- $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

Def: The distance from $u$ to $v$ is the number of edges on the shortest path from $u$ to $v$.

Theorem. For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
**BFS Algorithm**

```plaintext
BFS(G,s):
for each vertex u ∈ V - {s}
    u.color ← white
    u.d ← ∞
    u.p ← nil
s.color ← gray
s.d ← 0
initialize an empty queue Q
Enqueue(Q,s)
```

Every node stores a color, a distance and a parent

**Distance (d):** the length of shortest path from s to u

**Parent (p):** u’s predecessor on the shortest path from s to u

**Color:** indicates status
- white: (initial value) undiscovered
- gray: discovered, but neighbors not fully processed
- black: discovered and neighbors fully processed

**Note:** Assume, initially, that G is connected (will fix later)
BFS Algorithm Complete

\[
\text{BFS}(G,s): \\
\text{for each vertex } u \in V - \{s\} \\
\hspace{1em} u.\text{color} \leftarrow \text{white} \\
\hspace{1em} u.d \leftarrow \infty \\
\hspace{1em} u.p \leftarrow \text{nil} \\
\] 
\[s.\text{color} \leftarrow \text{gray} \]
\[s.d \leftarrow 0\]

1. initialize an empty queue \(Q\)
2. \(\text{Enqueue}(Q,s)\)
3. \(\text{while } Q \neq \emptyset \text{ do}\)
4. \(u \leftarrow \text{Dequeue}(Q)\)
5. \(\text{for each } v \in \text{Adj}[u]\)
6. \(\text{if } v.\text{color} = \text{white} \text{ then}\)
7. \(v.\text{color} \leftarrow \text{gray}\)
8. \(v.d \leftarrow u.d + 1\)
9. \(v.p \leftarrow u\)
10. \(\text{Enqueue}(Q,v)\)
11. \(u.\text{color} \leftarrow \text{black}\)

- Algorithm keeps current active nodes in a (FIFO) Queue \(Q\)
- Starts by inserting \(s\) in \(Q\)
- At each step takes node \(u\) off \(Q\)
  - Checks all neighbors \(v\) of \(u\)
  - If \(v\) has not been seen yet
    - Marks \(v\) as seen (gray)
    - Says that distance from \(s\) to \(v\) is \(1 + \text{dist to } u\)
    - Makes \(u\) the parent of \(v\)
    - inserts \(v\) in queue
- Marks \(u\) as being fully processed

Note: Nodes in Queue \(Q\)
- Are ones that have been seen but are unprocessed (gray)
BFS Algorithm Complete

BFS($G, s$):
for each vertex $u \in V - \{s\}$
    $u$.color $\leftarrow$ white
    $u.d$ $\leftarrow$ $\infty$
    $u.p$ $\leftarrow$ nil
$s$.color $\leftarrow$ gray
$s.d$ $\leftarrow$ 0
initialize an empty queue $Q$
Enqueue($Q, s$)
while $Q \neq \emptyset$ do
    $u \leftarrow$ Dequeue($Q$)
    for each $v \in Adj[u]$
        if $v$.color = white then
            $v$.color $\leftarrow$ gray
            $v.d$ $\leftarrow$ $u.d + 1$
            $v.p$ $\leftarrow$ $u$
            Enqueue($Q, v$)
    $u$.color $\leftarrow$ black

Parent pointers:
- Pointing to the node that leads to its discovery
- Parent must be in $L_{i-1}$
- Can follow parent pointers to find the actual shortest path
- The pointers form a BFS tree, rooted at $s$

Running time:
$\sum_u (1 + \deg(u)) = \Theta(V + E)$, which is $\Theta(E)$ if the graph is connected.
BFS Tree

Note: BFS finds the shortest path from $s$ to every other node.
Connected Components

Connected component containing $s$. All nodes reachable from $s$.

Connected component containing node 1 = \{1, 2, 3, 4, 5, 6, 7, 8\}.

BFS starting from $s$ finds the connected component containing $s$.

Repeatedly running BFS from an undiscovered node finds all the connected components.
The old BFS(G,s) algorithm is renamed BFS-Visit(G,s).

A new upper-level BFS(G) is created.

BFS(G) initializes all vertices to white (unvisited)
It then calls all vertices s, passing them to BFS-visit(s), if s was not already seen while traversing a previously visited connected component.
**Connected Components**

*Connected component containing s.* All nodes reachable from s.

BFS-Visit(1) would turn all nodes in leftmost component black.
**Connected Components**

**Connected component containing** \( s \).  All nodes reachable from \( s \).

BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black
Connected Components

Connected component containing $s$. All nodes reachable from $s$.

BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black
Connected Components

Connected component containing $s$. All nodes reachable from $s$.

BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black

BFS-Visit($i$) for $3 \leq i \leq 9$ would do nothing.

BFS-Visit(10) would then turn all nodes in middle component black
**Connected Components**

**Connected component containing** $s$. All nodes reachable from $s$.

BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black

BFS-Visit($i$) for $3 \leq i \leq 9$ would do nothing.

BFS-Visit(10) would then turn all nodes in middle component black
s-t connectivity and shortest path in directed graphs

s-t connectivity (often called reachability for directed graphs). Given two nodes \( s \) and \( t \), is there a path from \( s \) to \( t \)?
- Undirected graph: \( s \) can reach \( t \) \( \iff \) \( t \) can reach \( s \)
- Directed graph: Not necessarily true

s-t shortest path problem. Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) and \( t \)?
- Undirected graph: \( p \) is the shortest path from \( s \) to \( t \) \( \iff \) \( p \) is the shortest path from \( t \) to \( s \)
- Directed graph: Not necessarily true

BFS on a directed graph. Same as in undirected case
- Ex: Web crawler. Start from web page \( s \). Find all web pages linked from \( s \), either directly or indirectly.
Strong Connectivity in Directed Graphs

Def. Node \( u \) and \( v \) are **mutually reachable** if there is a path from \( u \) to \( v \) and also a path from \( v \) to \( u \).

Def. A graph is **strongly connected** if every pair of nodes is mutually reachable.

Definition: vertex \( s \) is "strong" in Graph \( G \) if, for every vertex \( t \), there is a path from \( s \) to \( t \) and from \( t \) to \( s \).

Observation 1: If graph \( G \) has a strong vertex \( s \) then EVERY vertex in \( G \) is strong

Observation 2: A graph \( G \) is strongly connected if and only if every vertex in \( G \) is strong
Strong Connectivity in Directed Graphs

Def. Node $u$ and $v$ are **mutually reachable** if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

Def. A graph is **strongly connected** if every pair of nodes is mutually reachable.

Algorithm for checking strong connectivity
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- **Reverse all edges in** $G$, and run BFS from $s$.
- Return true iff all nodes reached in both BFS executions.
Strongly Connected Components

**Strongly-Connected-Components**(\(G\)):
- create \(G^{rev}\) which is \(G\) with all edges reversed
- while there are nodes left do
  - \(u \leftarrow\) any node
  - run BFS in \(G\) starting from \(u\)
  - run BFS in \(G^{rev}\) starting from \(u\)
  - \(C \leftarrow\) {nodes reached in both BFSs}
- output \(C\) as a strongly connected component
- remove \(C\) and its edges from \(G\) and \(G^{rev}\)

**Running time:** \(O(VE)\)

See text book for a \(\Theta(V + E)\) algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components\((G)\):
create \(G^{rev}\) which is \(G\) with all edges reversed
while there are nodes left do
    \(u \leftarrow\) any node
    run BFS in \(G\) starting from \(u\)
    run BFS in \(G^{rev}\) starting from \(u\)
    \(C \leftarrow\) {nodes reached in both BFSs}
output \(C\) as a strongly connected component
remove \(C\) and its edges from \(G\) and \(G^{rev}\)

Running time: \(O(VE)\)

See text book for a \(\Theta(V + E)\) algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components\( (G) \):
create \( G^{rev} \) which is \( G \) with all edges reversed
while there are nodes left do
  \( u \leftarrow \) any node
  run BFS in \( G \) starting from \( u \)
  run BFS in \( G^{rev} \) starting from \( u \)
  \( C \leftarrow \{ \text{nodes reached in both BFSs} \} \)
output \( C \) as a strongly connected component
remove \( C \) and its edges from \( G \) and \( G^{rev} \)

Running time: \( O(VE) \)

See text book for a \( \Theta(V + E) \) algorithm (not required)
Strongly Connected Components

**Strongly-Connected-Components**(\(G\)):
create \(G^{rev}\) which is \(G\) with all edges reversed
while there are nodes left do
  \(u \leftarrow\) any node
  run BFS in \(G\) starting from \(u\)
  run BFS in \(G^{rev}\) starting from \(u\)
  \(C \leftarrow\) {nodes reached in both BFSs}
output \(C\) as a strongly connected component
remove \(C\) and its edges from \(G\) and \(G^{rev}\)

**Running time:** \(O(VE)\)

See text book for a \(\Theta(V + E)\) algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components($G$):
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
    $u \leftarrow$ any node
    run BFS in $G$ starting from $u$
    run BFS in $G^{rev}$ starting from $u$
    $C \leftarrow \{\text{nodes reached in both BFSs}\}$
    output $C$ as a strongly connected component
    remove $C$ and its edges from $G$ and $G^{rev}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components($G$):
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
    $u \leftarrow$ any node
    run BFS in $G$ starting from $u$
    run BFS in $G^{rev}$ starting from $u$
    $C \leftarrow \{\text{nodes reached in both BFSs}\}$
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{rev}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

**Running time:** $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components(G):
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
    $u \leftarrow$ any node
    run BFS in $G$ starting from $u$
    run BFS in $G^{rev}$ starting from $u$
    $C \leftarrow \{\text{nodes reached in both BFSs}\}$
    output $C$ as a strongly connected component
    remove $C$ and its edges from $G$ and $G^{rev}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components(G):
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
  $u \leftarrow$ any node
  run BFS in $G$ starting from $u$
  run BFS in $G^{rev}$ starting from $u$
  $C \leftarrow \{\text{nodes reached in both BFSs}\}$
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{rev}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly Connected Components ($G$):
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
  $u \leftarrow$ any node
  run BFS in $G$ starting from $u$
  run BFS in $G^{rev}$ starting from $u$
  $C \leftarrow \{\text{nodes reached in both BFSs}\}$
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{rev}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
**Exercise on Chess**

Find the minimum number of steps taken by a knight to reach a destination \((x',y')\) from an input position \((x,y)\) on a chess board.

From some position \((x, y)\) the knight can move to the following positions provided that they are within the board limits:

\[
\begin{align*}
& (x + 2, y - 1) \\
& (x + 2, y + 1) \\
& (x - 2, y + 1) \\
& (x - 2, y - 1) \\
& (x + 1, y + 2) \\
& (x + 1, y - 2) \\
& (x - 1, y + 2) \\
& (x - 1, y - 2)
\end{align*}
\]

Start from position \((x,y)\) and apply BFS, considering that the neighbors of \((x,y)\) are all the positions that can be reach with one move (as above).

Continue this process for each neighbor

The first time that you reach \((x',y')\) corresponds to the minimum number of steps.
Exercise on Binary Maze

Given a binary rectangular maze, find the shortest path’s length from a position \((x,y)\) to position \((x',y')\).

The path can only contain cells having value 1, and at position \((x,y)\) the valid moves are:
- Go Up: \((x - 1, y)\)
- Go Left: \((x, y - 1)\)
- Go Down: \((x + 1, y)\)
- Go Right: \((x, y + 1)\)

Start from position \((x,y)\) and apply BFS, considering that the neighbors of \((x,y)\) are all the positions that can be reach with one move (as above).

Continue this process for each neighbor

The first time that you reach \((x',y')\) corresponds to the minimum number of steps.
Exercise on Bipartite Graphs

A bipartite graph is a graph whose vertices can be divided into two disjoint sets $U$ and $V$ such that every edge connects a vertex in $U$ to one in $V$.

Problem: Given a connected undirected graph determine whether it is bipartite or not.

- Run BFS from any vertex: $d[v]$ stores the shortest distance from the root to $v$. Set $S$ to be the set of all vertices with even $d[v]$, and $V-S$ all vertices with odd $d[v]$.
- $G$ is bipartite if and only if all edges $(u,v)$ in the graph satisfy that the parity of $d[v]$ and $d[u]$ are not the same, i.e., $d[v]$ is odd and $d[u]$ is even or vice versa.
- Alternatively: If a graph contains an odd cycle, we cannot divide the graph such that every adjacent vertex has a different parity. To check if a given graph contains an odd-cycle or not, do a breadth-first search starting from an arbitrary vertex $v$. If in the BFS, we find an edge, both of whose endpoints are at the same level, then the graph is not Bipartite, and an odd-cycle is found.