Lecture 12:
• Applications of BFS: 2-Coloring, Connected Components, Topological Sort

Oct 19, 2018
Recap: Graphs/BFS
Graphs: Key Definitions

• **Definition:** A directed graph \( G = (V, E) \)
  - \( V \) is the set of nodes/vertices, \( |V| = n \)
  - \( E \subseteq V \times V \) is the set of edges, \( |E| = m \)
  - An edge is an ordered \( e = (u, v) \) “from \( u \) to \( v \)”

• **Definition:** An undirected graph \( G = (V, E) \)
  - Edges are unordered \( e = (u, v) \) “between \( u \) and \( v \)”

• **Simple Graph:**
  - No duplicate edges
  - No self-loops \( e = (u, u) \)
Breadth-First Search (BFS)

- **Definition:** the distance between \( s, t \) is the number of edges on the shortest path from \( s \) to \( t \).
- **Thm:** BFS finds distances from \( s \) to other nodes.
  - \( L_i \) contains all nodes at distance \( i \) from \( s \).
  - Nodes not in any layer are not reachable from \( s \).

![Diagram of BFS](image)
Adjacency Matrices

- The **adjacency matrix** of a graph $G = (V, E)$ with $n$ nodes is the matrix $A[1:n, 1:n]$ where

$$A[i,j] = \begin{cases} 
1 & (i, j) \in E \\
0 & (i, j) \notin E
\end{cases}$$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Cost
Space: $\Theta(n^2)$

Lookup: $\Theta(1)$ time
List Neighbors: $\Theta(n)$ time
Adjacency Lists (Undirected)

• The **adjacency list** of a vertex \( v \in V \) is the list \( A[v] \) of all \( u \) s.t. \( (v, u) \in E \)

\[
\begin{align*}
A[1] &= \{2, 3\} \\
A[2] &= \{1, 3\} \\
A[3] &= \{1, 2, 4\} \\
\end{align*}
\]

**Cost**

- **Space:** \( \Theta(n + m) \)

- **Lookup:** \( \Theta(\deg(u) + 1) \) time

- **List Neighbors:** \( \Theta(\deg(u) + 1) \) time
Breadth-First Search Implementation

BFS\((G = (V,E), s)\):

Let \(\text{found}[v] \leftarrow \text{false} \ \forall v, \ \text{found}[s] \leftarrow \text{true}\)
Let \(\text{layer}[v] \leftarrow \infty \ \forall v, \ \text{layer}[s] \leftarrow 0\)
Let \(i \leftarrow 0, L_0 = \{s\}, T \leftarrow \emptyset\)

While (\(L_i\) is not empty):

Initialize new layer \(L_{i+1}\)
For (\(u\) in \(L_i\)):
For ((\(u,v\)) in \(E\)):
If (\(\text{found}[v] = \text{false}\)):
\(\text{found}[v] \leftarrow \text{true}, \ \text{layer}[v] \leftarrow i+1\)
Add \((u,v)\) to \(T\) and add \(v\) to \(L_{i+1}\)
\(i \leftarrow i+1\)

Implements BFS in \(O(n + m)\) time

If \(n_s\) is the \# of nodes reachable from \(s\)
\(m_s\) "edges"

\[ \Rightarrow \text{time} \ O(n_s + m_s) \]
Bipartiteness / 2-Coloring
2-Coloring

- **Problem:** Tug-of-War Rematch
  - Need to form two teams $R, P$
  - Some students are still mad from last time
- **Input:** Undirected graph $G = (V, E)$
  - $(u, v) \in E$ means $u, v$ won't be on the same team
- **Output:** Split $V$ into two sets $R, P$ so that no pair in either set is connected by an edge
2-Coloring (Bipartiteness)

• **Equivalent Problem:** Is the graph $G$ bipartite?
  
  - A graph $G$ is bipartite if I can split $V$ into two sets $L$ and $R$ such that all edges $(u, v) \in E$ go between $L$ and $R$.

```
L
2
3
5
R
1
4
```

$L \cap R = \emptyset$

$L \cup R = V$
Designing the Algorithm

- **Key Fact:** If $G$ contains a cycle of odd length, then $G$ is not 2-colorable/bipartite
Designing the Algorithm

**Idea for the algorithm:**
- BFS the graph, coloring nodes as you find them
- Color nodes in layer $i$ **purple** if $i$ even, **red** if $i$ odd
- See if you have succeeded or failed
Designing the Algorithm

- **Claim:** If BFS 2-colored the graph successfully, the graph has been 2-colored successfully

- **Key Question:** Suppose you have not 2-colored the graph successfully, maybe someone else can do it?
Designing the Algorithm

• Claim: If BFS fails, then G contains an odd cycle
  • If G contains an odd cycle then G can’t be 2-colored!
  • Example of a phenomenon called duality

• Every edge in the BFS tree is colored correctly
• Dotted edge from L_i to L_{i+1} are colored correctly

• If the 2-coloring is not correct then there is an edge from L_i to L_{i+1}
Designing the Algorithm

- **Claim:** If BFS fails, then G contains an odd cycle
  - If G contains an odd cycle then G can’t be 2-colored!
  - Example of a phenomenon called **duality**

```
Claim: If G contains an edge from L; to itself, then
G contains an odd cycle.
```

- \( w \rightarrow u \rightarrow v \rightarrow w \) is an odd cycle
  - length \( i-j \)
  - edge length \( i-j \) \( \Rightarrow \) length = \( 2(i-j) + 1 \)

- There are paths of length \( i \)
  - For some node \( w \in L_j \) for \( j<i \)

- \( \exists \) an \( s \rightarrow u \) path of length \( i \)
  - \( \exists \) an \( s \rightarrow v \) path of length \( i \)
Topological Sort
Acyclic Graphs

- **Acyclic Graph**: An undirected graph with no cycles
  - Also known as a forest
  - If it’s connected then it’s known as a tree
- Can test if a graph has a cycle in $O(n + m)$ time
  - Run BFS
  - If there are any edges that are not in the BFS tree, then they form a cycle
Directed Acyclic Graphs (DAGs)

- **DAG**: A directed graph with no directed cycles
- Can be much more complex than a forest
Directed Acyclic Graphs (DAGs)

- **DAG**: A directed graph with no directed cycles
- DAGs represent **precedence** relationships

- A topological ordering of a directed graph is a labeling of the nodes from $v_1, \ldots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - $G$ has a topological ordering $\Rightarrow G$ is a DAG
  - $G$ is not a DAG $\Rightarrow G$ cannot be top. ordered
Directed Acyclic Graphs (DAGs)

• **Problem 1**: given a digraph $G$, is it a DAG?
• **Problem 2**: given a digraph $G$, can it be topologically ordered?

• **Thm**: $G$ has a topological ordering $\iff G$ is a DAG
  • We will design one algorithm that either outputs a topological ordering or finds a directed cycle
  • *Another example of duality*
Topological Ordering

- If every node has \( \geq 1 \) in-edge, then \( G \) cannot be TO'd.

- **Observation:** the first node must have no in-edges

- **Observation:** In any DAG, there is always a node with no incoming edges

**Proof:** Suppose every node has \( \geq 1 \) in-edge

- Consider this chain of length \( n+1 \)
- The same node must appear twice
- The node that appears twice starts and ends a directed cycle,
Topological Ordering

• **Fact:** In any DAG, there is a node with no incoming edges

• **Thm:** Every DAG has a topological ordering

• **Proof (Induction):**  
  
  \[ H(n) : \forall \text{ DAG with } n \text{ nodes, there exists a topological ordering.} \]

  • To prove: \( \forall n \in \mathbb{N} \) \( H(n) \) is true
  
  • Base Case: \( H(1) \) is true
Inductive Step:

- To prove: \( H(n-1) \Rightarrow H(n) \)

  - By [Fact], there exists a node \( u \) with no incoming edges, call it \( u_1 \)

  - Consider the graph \( G \setminus \{u_1, \ldots, u_n\} \), this graph is a DAG with \( n-1 \) nodes.

  - By \( H(n-1) \), there exists an ordering of \( G \setminus \{u_1, \ldots, u_n\} \), call it \( u_2, u_3, \ldots, u_n \)

  - \( u_1, u_2, \ldots, u_n \) is a TD of \( G \)

  - By induction, all edges in the box go left to right.

There are no in-edges, so all edges go left to right.
Implementing Topological Ordering

**SimpleTopOrder(G):**
1. Set $i \leftarrow 1$
2. Until (G has no more nodes):
   1. Find a node $u$ with no incoming edges
   2. Label $u$ as node $i$, increment $i \leftarrow i+1$
   3. Remove $u$ and its edges from $G$
Implementing Topological Ordering

SimpleTopOrder(G):
    Set $i \leftarrow 1$
    Until (G has no more nodes):
        Find a node $u$ with no incoming edges
        Label $u$ as node $i$, increment $i \leftarrow i+1$
        Remove $u$ and its edges from $G$
Implementing Topological Ordering

SimpleTopOrder(G):
Set \( i \leftarrow 1 \)
Until (G has no more nodes):
① Find a node \( u \) with no incoming edges
② Label \( u \) as node \( i \), increment \( i \leftarrow i+1 \)
③ Remove \( u \) and its edges from \( G \)

- Go around the loop \( n \) times
- Step ① takes \( O(n) \) time
- Step ② takes \( O(1) \) time
- Step ③ takes \( O(m) \) time

\[
n \times O(n+m) = O(n^2+nm) = O(nm)
\]
Fast Topological Ordering

\textbf{FastTopOrder}(G):

Mark all nodes with their # of in-edges
Call a node INACTIVE if it’s mark is 0
Call a node ACTIVE otherwise
Let \( i = 1 \)
Until (all node are INACTIVE):
  Let \( u \) be an INACTIVE
  Label \( u \) as node \( i \) in the top. order
  Let \( i = i + 1 \)
  For (every \((u,v)\) in E):
    Decrease \( v \)’s mark by 1
Fast Topological Ordering Example
Topological Ordering Summary

- **DAG**: A directed graph with no directed cycles
- Any DAG can be **topologically ordered**
  - There is an algorithm that either outputs a topological ordering or finds a directed cycle in time $O(n + m)$