Announcements

- Reading
  - Chapter 3 (Mostly review)
  - Start on Chapter 4
- No class on Monday

Review from Wednesday

- Run time function \( T(n) \)
  - \( T(n) \) is the maximum time to solve an instance of size \( n \)
- Disregard constant functions
- \( T(n) = O(f(n)) \) \( [T : \mathbb{Z}^+ \to \mathbb{R}^+] \)
  - If \( n \) is sufficiently large, \( T(n) \) is bounded by a constant multiple of \( f(n) \)
  - Exist \( c, n_0 \) such that for \( n > n_0 \), \( T(n) < c f(n) \)

Graph Theory

- \( G = (V, E) \)
  - \( V \) - vertices
  - \( E \) - edges
- Undirected graphs
  - Edges sets of two vertices \( \{u, v\} \)
- Directed graphs
  - Edges ordered pairs \( (u, v) \)
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops

Definitions

- Path: \( v_1, v_2, ..., v_k \) with \( \{v_i, v_{i+1}\} \) in \( E \)
  - Simple Path
  - Cycle
  - Simple Cycle
- Neighborhood
  - \( N(v) \)
- Distance
- Connectivity
  - Undirected
  - Directed (strong connectivity)
- Trees
  - Rooted
  - Unrooted

Graph Representation

- \( V = \{a, b, c, d\} \)
- \( E = \{(a, b), (a, c), (a, d), (b, d)\} \)

Adjacency List

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Incidence Matrix

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph search

- Find a path from \( s \) to \( t \)

\[
S = \{s\}
\]

while \( S \) is not empty

\[
\begin{align*}
&u = \text{Select}(S) \\
&\text{visit } u \\
&\text{foreach } v \text{ in } N(u) \\
&\quad \text{if } v \text{ is unvisited} \\
&\qquad \text{Add}(S, v) \\
&\qquad \text{Pred}(v) = u \\
&\quad \text{if } (v = t) \text{ then path found}
\end{align*}
\]

Breadth First Search

- Explore vertices in layers
  - \( s \) in layer 1
  - Neighbors of \( s \) in layer 2
  - Neighbors of layer 2 in layer 3 . . .

Key Observation

- All edges go between vertices on the same layer or adjacent layers

Bipartite Graphs

- A graph \( V \) is bipartite if \( V \) can be partitioned into \( V_1, V_2 \) such that all edges go between \( V_1 \) and \( V_2 \)
- A graph is bipartite if it can be two colored

Can this graph be two colored?

Algorithm

- Run BFS
- Color odd layers red, even layers blue
- If no edges between the same layer, the graph is bipartite
- If edge between two vertices of the same layer, then there is an odd cycle, and the graph is not bipartite
Theorem: A graph is bipartite if and only if it has no odd cycles.

Lemma 1
- If a graph contains an odd cycle, it is not bipartite.

Lemma 2
- If a BFS tree has an \textit{intra-level edge}, then the graph has an odd length cycle.

Lemma 3
- If a graph has no odd length cycles, then it is bipartite.

Intra-level edge: both end points are in the same level.

Graph Search
- Data structure for next vertex to visit determines search order.

Graph search

Breadth First Search
\begin{verbatim}
S = \{s\}
\text{while } S \text{ is not empty}
\text{\hspace{1em} } u = \text{Dequeue}(S)
\text{\hspace{1em} } if \text{ } u \text{ is unvisited}
\text{\hspace{2em} } visit \text{ } u
\text{\hspace{2em} } \text{foreach } v \text{ in N}(u)
\text{\hspace{3em} } \text{Enqueue}(S, v)
\end{verbatim}

Depth First Search
\begin{verbatim}
S = \{s\}
\text{while } S \text{ is not empty}
\text{\hspace{1em} } u = \text{Pop}(S)
\text{\hspace{1em} } if \text{ } u \text{ is unvisited}
\text{\hspace{2em} } visit \text{ } u
\text{\hspace{2em} } \text{foreach } v \text{ in N}(u)
\text{\hspace{3em} } \text{Push}(S, v)
\end{verbatim}
Breadth First Search
• All edges go between vertices on the same layer or adjacent layers

Depth First Search
• Each edge goes between vertices on the same branch
• No cross edges

Connected Components
• Undirected Graphs
Computing Connected Components in $O(n+m)$ time
• A search algorithm from a vertex $v$ can find all vertices in $v$'s component
• While there is an unvisited vertex $v$, search from $v$ to find a new component

Directed Graphs
• A Strongly Connected Component is a subset of the vertices with paths between every pair of vertices.
Identify the Strongly Connected Components
Strongly connected components can be found in $O(n+m)$ time

- But it’s tricky!
- Simpler problem: given a vertex $v$, compute the vertices in $v$’s scc in $O(n+m)$ time

Topological Sort

- Given a set of tasks with precedence constraints, find a linear order of the tasks

Find a topological order for the following graph

If a graph has a cycle, there is no topological sort

- Consider the first vertex on the cycle in the topological sort
- It must have an incoming edge

Lemma: If a graph is acyclic, it has a vertex with in degree 0

- Proof:
  - Pick a vertex $v_1$, if it has in-degree 0 then done
  - If not, let $(v_2, v_1)$ be an edge, if $v_2$ has in-degree 0 then done
  - If not, let $(v_p, v_2)$ be an edge . . .
  - If this process continues for more than $n$ steps, we have a repeated vertex, so we have a cycle

Topological Sort Algorithm

While there exists a vertex $v$ with in-degree 0

- Output vertex $v$
- Delete the vertex $v$ and all outgoing edges
Details for O(n+m) implementation

- Maintain a list of vertices of in-degree 0
- Each vertex keeps track of its in-degree
- Update in-degrees and list when edges are removed
- m edge removals at O(1) cost each