Lecture 34: Graphs II

CS 62

Spring 2019

William Devanny & Alexandra Papoutsaki
Number of Edges

• If $|V| = n$, then:
  • minimum number of edges: 0
    • A graph can have only nodes
  • For simple directed graphs, maximum number: $n(n - 1)$
    • (assuming no self loops)
  • For simple undirected graphs, maximum number: $\frac{n(n-1)}{2}$

• Dense graphs $\rightarrow$ #edges close to maximum
• Sparse graphs $\rightarrow$ #edges close to $n$
Graph Representations

- Adjacency Matrix
- Adjacency List
# Adjacency 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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Good for dense graphs
- Constant time for lookup for edges.
- Constant time for adding/removing an edge
- Symmetric if undirected.
- Can hold weights.
Adjacency Lists

- Good for sparse graphs, saves space.
- Linear time lookup for edges.
## Time complexity comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store graph</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Add vertex</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Add edge</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Remove vertex</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Remove edge</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td>Are two vertices adjacent?</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
</tbody>
</table>
Spanning Trees

- **Tree**: connected undirected graph with no cycles
- **Spanning tree of $G$**: includes every vertex of $G$ and is a subgraph of $G$ (every edge belongs to $G$)
- Often interested in properties like minimum-cost
- Can be constructed by search algorithms

![Graph $G$ and 3 different spanning trees of graph $G$]
Depth-First Search

- Explore the graph without revisiting nodes
- Depth-first means go until you hit a dead end, then back up to branch out

Example from wikipedia
Recursive DFS pseudocode

```java
void DFS(G, v) {
    visited[v] = true;
    for (every edge (v, w)) {
        if (!visited[w]) {
            DFS(G, w);
        }
    }
}
```

Order of visit: A B D F E C G
Practice time

Order of visit: A B I C D E H G F

Example graph from: https://www.youtube.com/watch?v=iaBEKo5sM7w
Non-recursive DFS pseudocode

```java
for (every vertex v)
    visited[v] = false;

s = new Stack();
s.push(v1);

while (!s.isEmpty()) {
    v = s.pop();
    if (!visited[v]) {
        visited[v] = true;
        for (every edge (v, w))
            if (!visited[w])
                s.push(w);
    }
}
```

Order of visit: A E F B D C G
Practice time

Order of visit: A I G H E C F D B

Example graph from: https://www.youtube.com/watch?v=iaBEKo5sM7w
Breadth-First Search

• Replace stack with queue
• Now we explore in order of distance from start

• Algorithm:
  1. Mark start vertex
  2. Add all unmarked neighbors to queue and mark them
  3. Repeat step 2 with next from queue until it’s empty
BFS pseudocode

for (every vertex v)
    visited[v] = false;
q = new Queue();
q.enqueue(v1);
while (!q.isEmpty()) {
    v = q.dequeue();
    if (!visited[v]) {
        visited[v] = true;
        for (every edge (v, w))
            if (!visited[w])
                q.enqueue(w);
    }
}

Order of visit: A B C E D F G
Example graph from: https://www.youtube.com/watch?v=iaBEKo5sM7w

Order of visit: A B I C G D E F H
DFS/BFS traversal

- Can be performed in $O(n + m)$, where $n = |V|, m = |E|$.
- Can:
  - Test if $G$ is connected
    - If traversal visited all vertices, then graph is connected
  - Compute a spanning tree of $G$, if $G$ is connected
  - Find a path between two vertices, if it exits
  - Compute the connected components of $G$
    (needs to loop over all vertices and run DFS/BFS again)
Connectivity in Digraphs

- **reachable vertices**: when there is a directed path from one to another.
- **strongly connected vertices**: if mutually reachable
- **strongly connected digraph**: directed path from every vertex to every other vertex
- **weakly connected graph**: a digraph that would be connected if all of its directed edges were replaced by undirected edges.
Testing connectivity

• For an undirected graph:
  • Run DFS/BFS from any vertex without restarting and see if all vertices are marked

• For strong connectivity on a directed graph:
  • 1. Initialize all vertices are not visited
  • 2. Run DFS/BFS from an arbitrary vertex \( v \).
    • If traversal does not visit all vertices return false
  • 3. Reverse all edges
  • 4. Start from same vertex \( v \) and perform DFS/BFS. Graph is strongly connected iff all vertices are marked as visited again.