CSE 332 Autumn 2023
Lecture 18: Graphs

Nathan Brunelle

http://www.cs.uw.edu/332
Some Graph Terms

- **Adjacent/Neighbors**
  - Nodes are adjacent/neighbors if they share an edge

- **Degree**
  - Number of “neighbors” of a vertex

- **Indegree**
  - Number of incoming neighbors

- **Outdegree**
  - Number of outgoing neighbors
Graph Operations

• To represent a Graph (i.e. build a data structure) we need:
  • Add Edge
  • Remove Edge
  • Check if Edge Exists
  • Get Neighbors (incoming)
  • Get Neighbors (outgoing)
Adjacency List

Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(\deg(v))$
Check if Edge Exists: $\Theta(\deg(v))$
Get Neighbors (incoming): $\Theta(n + m)$
Get Neighbors (outgoing): $\Theta(\deg(v))$

$|V| = n$
$|E| = m$
Adjacency List (Weighted)

Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(\text{deg}(v))$
Check if Edge Exists: $\Theta(\text{deg}(v))$
Get Neighbors (incoming): $\Theta(n + m)$
Get Neighbors (outgoing): $\Theta(\text{deg}(v))$

$|V| = n$
$|E| = m$
Adjacency Matrix

Time/Space Tradeoffs
Space to represent: $\Theta(\cdot)$
Add Edge: $\Theta(\cdot)$
Remove Edge: $\Theta(\cdot)$
Check if Edge Exists: $\Theta(\cdot)$
Get Neighbors (incoming): $\Theta(\cdot)$
Get Neighbors (outgoing): $\Theta(\cdot)$

$|V| = n$
$|E| = m$
Adjacency Matrix (weighted)

**Time/Space Tradeoffs**
- Space to represent: $\Theta(n^2)$
- Add Edge: $\Theta(1)$
- Remove Edge: $\Theta(1)$
- Check if Edge Exists: $\Theta(1)$
- Get Neighbors (incoming): $\Theta(n)$
- Get Neighbors (outgoing): $\Theta(n)$

$|V| = n$

$|E| = m$
Aside

• Almost always, adjacency lists are the better choice
• Most graphs are missing most of their edges, so the adjacency list is much more space efficient and the slower operations aren’t that bad
Definition: Path

A sequence of nodes \((v_1, v_2, ..., v_k)\) s.t. \(\forall 1 \leq i \leq k - 1, (v_i, v_{i+1}) \in E\)

Simple Path:
A path in which each node appears at most once

Cycle:
A path which starts and ends in the same place
Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$
Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$

Connected

Not (strongly) Connected
Definition: Weakly Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$ ignoring direction of edges.
Definition: Complete Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is an edge from $v_1$ to $v_2$
Graph Density, Data Structures, Efficiency

- The maximum number of edges in a graph is $\Theta(|V|^2)$:
  - Undirected and simple: $\frac{|V|(|V|-1)}{2}$
  - Directed and simple: $|V|(|V| - 1)$
  - Direct and non-simple (but no duplicates): $|V|^2$

- If the graph is connected, the minimum number of edges is $|V| - 1$

- If $|E| \in \Theta(|V|^2)$ we say the graph is dense

- If $|E| \in \Theta(|V|)$ we say the graph is sparse

- Because $|E|$ is not always near to $|V|^2$ we do not typically substitute $|V|^2$ for $|E|$ in running times, but leave it as a separate variable
Definition: Tree

A Graph $G = (V, E)$ is a tree if it is undirected, connected, and has no cycles (i.e. is acyclic). Often one node is identified as the “root”

A Tree

A Rooted Tree
Breadth-First Search

• Input: a node $s$

• Behavior: Start with node $s$, visit all neighbors of $s$, then all neighbors of neighbors of $s$, ...

• Output:
  • How long is the shortest path?
  • Is the graph connected?
void bfs(graph, s){
    found = new Queue();
    found.enqueue(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.dequeue();
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                found.enqueue(v);
            }
        }
    }
}

Running time: $\Theta(|V| + |E|)$
int shortestPath(graph, s, t){
    found = new Queue();
    layer = 0;
    found.enqueue(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.dequeue();
        layer = depth of current;
        for (v : neighbors(current)){
            if (! v marked “visited”){
                mark v as “visited”;
                depth of v = layer + 1;
                found.enqueue(v);
            }
        }
    }
    return depth of t;
}
Depth-First Search
Depth-First Search

• Input: a node \( s \)

• Behavior: Start with node \( s \), visit one neighbor of \( s \), then all nodes reachable from that neighbor of \( s \), then another neighbor of \( s \),...
  • Before moving on to the second neighbor of \( s \), visit everything reachable from the first neighbor of \( s \)

• Output:
  • Does the graph have a cycle?
  • A **topological sort** of the graph.
void dfs(graph, s){
    found = new Stack();
    found.pop(s);
    mark s as “visited”;
    While (!found.isEmpty()){  
        current = found.pop();
        for (v : neighbors(current)){
            if (! v marked “visited”){
                mark v as “visited”;
                found.push(v);
            }
        }
    }
}
DFS Recursively (more common)

```java
void dfs(graph, curr){
    mark curr as “visited”;  
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;  
}
```
Using DFS

• Consider the “visited times” and “done times”

• Edges can be categorized:
  • Tree Edge
    • \((a, b)\) was followed when pushing
    • \((a, b)\) when \(b\) was unvisited when we were at \(a\)
  • Back Edge
    • \((a, b)\) goes to an “ancestor”
    • \(a\) and \(b\) visited but not done when we saw \((a, b)\)
    • \(t_{visited}(b) < t_{visited}(a) < t_{done}(a) < t_{done}(b)\)
  • Forward Edge
    • \((a, b)\) goes to a “descendent”
    • \(b\) was visited and done between when \(a\) was visited and done
    • \(t_{visited}(a) < t_{visited}(b) < t_{done}(b) < t_{done}(a)\)
  • Cross Edge
    • \((a, b)\) goes to a node that doesn’t connect to \(a\)
    • \(b\) was seen and done before \(a\) was ever visited
    • \(t_{done}(b) < t_{visited}(a)\)
BackEdges

• Behavior of DFS:
  • “Visit everything reachable from the current node before going back”

• Back Edge:
  • The current node’s neighbor is an “in progress” node
  • Since that other node is “in progress”, the current node is reachable from it
  • The back edge is a path to that other node
  • Cycle!
Cycle Detection

boolean hasCycle(graph, curr){
    mark curr as “visited”;
    cycleFound = false;
    for (v : neighbors(current)){
        if (v marked “visited” && ! v marked “done”){
            cycleFound = true;
        }
        if (! v marked “visited” && ! cycleFound){
            cycleFound = hasCycle(graph, v);
        }
    }
    mark curr as “done”;
    return cycleFound;
}
Single-Source Shortest Path

Find the quickest way to get from UVA to each of these other places

Given a graph $G = (V, E)$ and a start node $s \in V$, for each $v \in V$ find the least-weight path from $s \rightarrow v$ (call this weight $\delta(s, v)$)

(assumption: all edge weights are positive)
Dijkstra’s Algorithm

• Input: graph with no negative edge weights, start node $s$, end node $t$
• Behavior: Start with node $s$, repeatedly go to the incomplete node “nearest” to $s$, stop when
• Output:
  • Distance from start to end
  • Distance from start to every node
Dijkstra’s Algorithm

Start: 0

End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path.
Dijkstra’s Algorithm

Start: 0
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path.
Dijkstra’s Algorithm

Start: 0
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>\infty</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>\infty</td>
</tr>
<tr>
<td>6</td>
<td>\infty</td>
</tr>
<tr>
<td>7</td>
<td>\infty</td>
</tr>
<tr>
<td>8</td>
<td>\infty</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path
Dijkstra’s Algorithm

Start: 0

End: 8

Node | Done?
---|---
0 | T
1 | T
2 | T
3 | F
4 | F
5 | F
6 | F
7 | F
8 | F

Node | Distance
---|---
0 | 0
1 | 10
2 | 12
3 | 15
4 | 18
5 | 13
6 | ∞
7 | ∞
8 | ∞

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path
Dijkstra’s Algorithm
Start: 0
End: 8

Node | Done?
---|---
0  | T
1  | T
2  | T
3  | F
4  | F
5  | T
6  | F
7  | F
8  | F

Node | Distance
---|---
0  | 0
1  | 10
2  | 12
3  | 14
4  | 18
5  | 13
6  | ∞
7  | 20
8  | ∞

Idea: When a node is the closest undiscovered thing to the start, we have found its shortest path.
Dijkstra’s Algorithm

```java
int dijkstras(graph, start, end){
    distances = [∞, ∞, ∞,...]; // one index per node
    done = [False,False,False,...]; // one index per node
    PQ = new minheap();
    PQ.insert(0, start); // priority=0, value=start
    distances[start] = 0;
    while (!PQ.isEmpty){
        current = PQ.deleteMin();
        done[current] = true;
        for (neighbor : current.neighbors){
            if (!done[neighbor]){ //if it is not visited
                new_dist = distances[current]+weight(current,neighbor);
                if new_dist < distances[neighbor]{
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist,neighbor); }
            }
        }
    }
    return distances[end]
}
```
Dijkstra’s Algorithm: Running Time

• How many total priority queue operations are necessary?
  • How many times is each node added to the priority queue?
  • How many times might a node’s priority be changed?

• What’s the running time of each priority queue operation?

• Overall running time:
  • $\Theta(|E| \log |V|)$
Dijkstra’s Algorithm: Correctness

• Claim: when a node is removed from the priority queue, we have found its shortest path

• Induction over number of completed nodes

• Base Case:

• Inductive Step:
Dijkstra’s Algorithm: Correctness

• Claim: when a node is removed from the priority queue, its distance is that of the shortest path

• Induction over number of completed nodes

• Base Case: Only the start node removed
  • It is indeed 0 away from itself

• Inductive Step:
  • If we have correctly found shortest paths for the first $k$ nodes, then when we remove node $k + 1$ we have found its shortest path
Dijkstra’s Algorithm: Correctness

• Suppose $a$ is the next node removed from the queue. What do we know bout $a$?
Dijkstra’s Algorithm: Correctness

• Suppose $a$ is the next node removed from the queue.
  - No other node incomplete node has a shorter path discovered so far

• Claim: no undiscovered path to $a$ could be shorter
  - Consider any other incomplete node $b$ that is 1 edge away from a complete node
  - $a$ is the closest node that is one away from a complete node
  - Thus no path that includes $b$ can be a shorter path to $a$
  - Therefore the shortest path to $a$ must use only complete nodes, and therefore we have found it already!
Dijkstra’s Algorithm: Correctness

• Suppose $a$ is the next node removed from the queue.
  • No other node incomplete node has a shorter path discovered so far
• Claim: no undiscovered path to $a$ could be shorter
  • Consider any other incomplete node $b$ that is 1 edge away from a complete node
  • $a$ is the closest node that is one away from a complete node
  • No path from $b$ to $a$ can have negative weight
  • Thus no path that includes $b$ can be a shorter path to $a$
  • Therefore the shortest path to $a$ must use only complete nodes, and therefore we have found it already!