Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge $(v, w)$: $\Theta(\text{deg}(v))$
Remove Edge $(v, w)$: $\Theta(\text{deg}(v))$
Check if Edge $(v, w)$ 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: $\Theta(n^2)$
Add Edge $(v, w)$: $\Theta(1)$
Remove Edge $(v, w)$: $\Theta(1)$
Check if Edge $(v, w)$ Exists: $\Theta(1)$
Get Neighbors (incoming): $\Theta(n)$
Get Neighbors (outgoing): $\Theta(n)$

$|V| = n$  $|E| = m$
Comparison

- **Adjacency List:**
  - Less memory when $|E| < |V|^2$
  - Operations with running time linear in degree of source node
    - Add an edge
    - Remove an edge
    - Check for edge
    - Get neighbors

- **Adjacency Matrix:**
  - Similar amount of memory when $|E| \approx |V|^2$
  - Constant time operations:
    - Add an edge
    - Remove an edge
    - Check for an edge
  - Operations running with linear time in $|V|$
    - Get neighbors

Adjacency List is more common in practice:
- Most graphs have $|E| \ll |V|^2$
  - Saves memory
  - Most nodes will have small degree
- Getting neighbors is a common operation
- Adjacency Matrix may be better if the graph is “dense” or if its edges change a lot
Breadth-First Search

• Input: a node $s$
• Behavior: Start with node $s$, visit all neighbors of $s$, then all neighbors of neighbors of $s$, ...
• Visits every node reachable from $s$ in order of distance
• 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){
    Stack found = new Stack();
    found.push(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);
            }
        }
    }
}

Running time: Θ(|V| + |E|)
DFS Recursively (more common)

```c
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
    • \(t_{done}(b) < t_{visited}(a)\)
Back Edges

• 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!
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;
}
Topological Sort

• A Topological Sort of a **directed acyclic graph** $G = (V, E)$ is a permutation of $V$ such that if $(u, v) \in E$ then $u$ is before $v$ in the permutation.
DFS Recursively

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

```java
List topSort(graph){
    List<Nodes> done = new List<>();
    for (Node v : graph.vertices){
        if (!v.visited){
            finishTime(graph, v, finished);
        }
    }
    done.reverse();
    return done;
}

void finishTime(graph, curr, finished){
    curr.visited = true;
    for (Node v : curr.neighbors){
        if (!v.visited){
            finishTime(graph, v, finished);
        }
    }
    done.add(curr)
}
```

Idea: List in reverse order by “done” time
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

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

Node | Distance
---|---
0 | 0
1 | 10
2 | 12
3 | ∞
4 | ∞
5 | ∞
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

<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>∞</td>
</tr>
<tr>
<td>4</td>
<td>18</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

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

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

```
int dijkstras(graph, start, end){
    distances = [\(\infty\), \(\infty\), \(\infty\),...]; // 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.extract();
        done[current] = true;
        for (neighbor : current.neighbors){
            if (!done[neighbor]){ // new_dist = weight(current, neighbor)
                new_dist = distances[current] + weight(current, neighbor);
                if(distances[neighbor] == \(\infty\)){
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]){ // PQ.decreaseKey(new_dist, 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?
    • At most once
  • How many times might a node’s priority be changed?
    • Indegree of that node

• What’s the running time of each priority queue operation?
  • $\log |V|$

• Overall running time:
  • $|V| \log |V| + |E| \log |V|$
  • $\Theta(|E| \log |V|)$
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!
Definition: Tree

A connected graph with no cycles

Note: A tree does not need a root, but they often do!
Definition: Tree

A connected graph with no cycles

Pick some arbitrary root node and rearrange tree
Definition: Spanning Tree

A Tree $T = (V_T, E_T)$ which connects (“spans”) all the nodes in a graph $G = (V, E)$

Any set of $V-1$ edges that doesn’t have any cycles is guaranteed to be a spanning tree!

Any set of $V-1$ edges that connects all the nodes in the graph is guaranteed to be a spanning tree!
Definition: Minimum Spanning Tree

A Tree $T = (V_T, E_T)$ which connects (“spans”) all the nodes in a graph $G = (V, E)$, that has minimal cost

$\text{Cost}(T) = \sum_{e \in E_T} w(e)$
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Kruskal’s Algorithm

Start with an empty tree $A$
Add to $A$ the lowest-weight edge that does not create a cycle
Definition: Cut

A Cut of graph $G = (V, E)$ is a partition of the nodes into two sets, $S$ and $V - S$.

Edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$ (or opposite), e.g. $(A, C)$.

A set of edges $R$ Respects a cut if no edges cross the cut, e.g. $R = \{(A, B), (E, G), (F, G)\}$.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V \setminus S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V \setminus S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Cut Theorem

If a set of edges $A$ is a subset of a minimum spanning tree $T$, let $(S, V - S)$ be any cut which $A$ respects. Let $e$ be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.
Proof of Kruskal’s Algorithm

Start with an empty tree $A$
Repeat $V - 1$ times:
    Add the min-weight edge that doesn’t cause a cycle

Proof: Suppose we have some arbitrary set of edges $A$ that Kruskal’s has already selected to include in the MST. $e = (F, G)$ is the edge Kruskal’s selects to add next

We know that there cannot exist a path from $F$ to $G$ using only edges in $A$ because $e$ does not cause a cycle

We can cut the graph therefore into 2 disjoint sets:
- nodes reachable from $G$ using edges in $A$
- All other nodes

$e$ is the minimum cost edge that crosses this cut, so by the Cut Theorem, Kruskal’s is optimal!
Kruskal’s Algorithm Runtime

Start with an empty tree $A$
Repeat $V - 1$ times:
Add the min-weight edge that doesn’t cause a cycle

Keep edges in a Disjoint-set data structure (very fancy)
$O(E \log V)$
General MST Algorithm

Start with an empty tree $A$
Repeat $V - 1$ times:
Pick a cut $(S, V - S)$ which $A$ respects (typically implicitly)
Add the min-weight edge which crosses $(S, V - S)$
Prim’s Algorithm

Start with an empty tree $A$

Repeat $V - 1$ times:

- Pick a cut $(S, V - S)$ which $A$ respects
- Add the min-weight edge which crosses $(S, V - S)$

$S$ is all endpoint of edges in $A$

$e$ is the min-weight edge that grows the tree
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm
Start with an empty tree $A$
Pick a start node
Repeat $V - 1$ times:
Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm

Start with an empty tree \( A \)

Pick a start node

Repeat \( V - 1 \) times:

Add the min-weight edge which connects to node in \( A \) with a node not in \( A \)
Prim’s Algorithm
Start with an empty tree $A$
Pick a start node
Repeat $V - 1$ times:
Add the min-weight edge which connects to node in $A$ with a node not in $A$
Prim’s Algorithm

Start with an empty tree $A$

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in $A$ with a node not in $A$

Keep edges in a Heap $O(E \log V)$
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]){
                new_dist = distances[current]+weight(current,neighbor);
                if(distances[neighbor] == ∞){
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]){  
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);  
                
                }
            }
        }
    }
    return distances[end] 
}
```
int primss(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]) {
                new_dist = weight(current, neighbor);
                if (distances[neighbor] == ∞) {
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                } else if (new_dist < distances[neighbor]) {
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);
                }
            }
        }
    }
    return distances[end]
}
Dijkstra’s Algorithm

```java
int dijkstras(graph, start, end){
    distances = [\(\infty, \infty, \infty, \ldots\)];  // one index per node
    done = [False, False, False, \ldots];  // 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]){  // Explore neighbors
                new_dist = distances[current]+weight(current,neighbor);
                if (distances[neighbor] == \(\infty\)){
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]){  // Update distance
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);
                }
            }
        }
    }
    return distances[end]
}
```
int primss(graph, start, end) {
    distances = \([\infty, \infty, \infty, \ldots]\); // one index per node
    done = \([\text{False}, \text{False}, \text{False}, \ldots]\); // 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]) {
                new_dist = weight(current, neighbor);
                if (distances[neighbor] == \(\infty\)) {
                    distances[neighbor] = new_dist;
                    PQ.insert(new_dist, neighbor);
                }
                if (new_dist < distances[neighbor]) {
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist, neighbor);
                }
            }
        }
    }
    return distances[end]
}