CSE 332 Summer 2024
Lecture 16: Graphs

Nathan Brunelle

http://www.cs.uw.edu/332
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): $\Theta(\text{deg}(v))$

$|V| = n$
$|E| = m$
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;
}
Find the quickest way to get from UW 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

```java
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]){ // if neighbor hasn’t been processed yet
                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]){ // if the new distance is shorter
                    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:
  • When 1 node has been removed, its priority matches the weight of its shortest path
  • That one node is the start, it had priority 0
  • 0 is the cost of the shortest path from start to start

• Inductive Step:
  • Assume that $k$ nodes have been removed so far
  • Assume that for all them, their priority matched their shortest path cost
  • Show that the next node that’s removed ($k + 1$) also had its priority match its shortest path cost.
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 priority queue. What do we know about $a$?
  • We have a path from $s$ to $a$
  • $a$ has the lowest priority of all “incomplete” nodes, e.g. $b$
  • $a$ has an edge with a already completed node
Dijkstra’s Algorithm: Correctness

• Suppose $a$ is the next node removed from the priority queue.
  • No other incomplete node has a shorter path discovered so far (e.g. $b$)

• 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
  • Thus no path that includes $b$ can be a shorter path to $a$
    • Only because no path from $b$ to $a$ can have negative weight!
  • Therefore the shortest path to $a$ must use only complete nodes, and therefore we have found it already!
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.
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”; 
}
```
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

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

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

Idea: List in reverse order by “done” time

finished: 9 1 2 5 8 3 4 6 7
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 $G = (V, E)$

How many edges does $T$ have?
$V - 1$

Any set of $V-1$ edges in the graph 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.

$$Cost(T) = \sum_{e \in E_T} w(e)$$
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 $(S, V - S)$ 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 $(S, V - S)$ 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]
}
```
Prims's Algorithm

```java
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);
                }
                if (new_dist < distances[neighbor]){ // Weight is less than current edge
                    distances[neighbor] = new_dist;
                    PQ.decreaseKey(new_dist,neighbor); // Decrease key in the priority queue
                }
            }
        }
    }
    return distances[end]
}
```
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);
                } else 
                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]
}