Module Graph Algorithms

Thanks to Anna Lubiw and other previous CS 341 instructors.

- Graph Terminology and Storing Graphs
- Exploring Graphs
- Breadth First Search and Properties of BFS
- Applications of BFS: Bipartite Testing
- Depth First Search and Properties of DFS
- DFS on Directed Graphs and Applications
- Minimum Spanning Trees, Kruskal's Algorithm, Prim's Algorithm
- Short Paths on Weighted Graphs
- Dijkstra's Algorithm
- Dynamic Programming: Bellman-Ford, Floyd-Warshall

Graph Terminology

Definition: A **graph** $G = (V, E)$ where

- V is a set of **vertices** where $|V| = n$ and
- E is a set of **edges**, $E \subseteq V \times V$, where $|E| = m$ and $m \leq n^2$.

Edges can be undirected (unordered pairs) or directed (ordered pairs). A graph with directed edges is called a **directed graph** or **digraph**.

Graph Terminology

Definition: A **graph** $G = (V, E)$ where

- V is a set of **vertices** where $|V| = n$ and
- E is a set of **edges**, $E \subseteq V \times V$, where $|E| = m$ and $m \leq n^2$.

Edges can be undirected (unordered pairs) or directed (ordered pairs). A graph with directed edges is called a **directed graph** or **digraph**.

Basic Notations

- Convention: vertices are $\{1, 2, \ldots, n\}$ but sometimes written $v_1, \ldots, v_n$ or $a, b, c, \ldots$.
- $u, v \in V$ are **adjacent** or **neighbours** if $(u, v) \in E$
- $v \in V$ is **incident** to $e \in E$ if $e = (v, u)$
- $\deg(v)$ = number of incident edges
- For a directed graph, we define indegree(v) and outdegree(v) as the number of incident edges direct into v and directed out of v.

In practice, vertices and edges may have names or other associated information but our algorithms will be for **abstract graphs**.

Graph Terminology Continued

- A **path** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$. A **simple** path does not repeat vertices.
- A **cycle** is a path that starts and ends at the same vertex. A **simple cycle** does not repeat any vertices except the start/end. Note: some sources use "path" to mean a simple path.
- A **tree** is a connected (undirect) graph with no cycles.
- An undirected graph is connected if there exists a path joining all pairs $u, v \in V$.
- A **connected component** of a graph is the maximal connected subgraph.

Graph Terminology Continued

- A **path** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$. A **simple** path does not repeat vertices.
- A **cycle** is a path that starts and ends at the same vertex. A **simple cycle** does not repeat any vertices except the start/end. Note: some sources use "path" to mean a simple path.
- A **tree** is a connected (undirect) graph with no cycles.
- An undirected graph is connected if there exists a path joining all pairs $u, v \in V$.
- A **connected component** of a graph is the maximal connected subgraph.

The study of graphs is know as **Graph Theory**.

- Origin from the Königsberg bridge problem (Euler 1735).
- Many applications! - networks, transportation, social, scheduling, game configurations, etc

Storing Graphs: Adjacency Matrices

There are two main data structures used to store a graph: an **adjacency matrix** and a set of **adjacency lists**.

The **adjacency matrix** of G is an n by n matrix A, requiring $O(n^2)$ space, which is indexed by V, such that

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

There are exactly $2m$ entries of A equal to 1.

If G is a directed graph, then

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

For a directed graph, there are exactly m entries of A equal to 1.

Storing Graphs: Adjacency Lists

An **adjacency list** representation of G consists of n linked lists.

- For every $u \in V$, there is a linked list (called an **adjacency list**) which is named **Adj**$[u]$.
- For every $v \in V$ such that $uv \in E$, there is a node in **Adj**$[u]$ labelled v. (This definition is used for both directed and undirected graphs.)
- In an undirected graph, every edge uv corresponds to nodes in two adjacency lists: there is a node v in **Adj**$[u]$ and a node u in **Adj**$[v]$.
- In a directed graph, every edge corresponds to a node in only one adjacency list.

Operations: Adjacency Matrix vs Adjacency Lists

Operation	Adjacency Matrix	Adjacency Lists
Space	$O(n^2)$	$O(n+m)$
$(u,v) \in E$?	$\Theta(1)$	$O(1 + \deg(u))$
List $v's$ neighbours		

Operations: Adjacency Matrix vs Adjacency Lists

Operation	Adjacency Matrix	Adjacency Lists
Space	$O(n^2)$	$O(n+m)$
$(u,v) \in E$?	$\Theta(1)$	$O(1+\deg(u))$
List $v's$ neighbours	$\Theta(n)$	$\Theta(1+\deg(v))$
List all edges		

Operations: Adjacency Matrix vs Adjacency Lists

Operation	Adjacency Matrix	Adjacency Lists
Space	$O(n^2)$	$O(n+m)$
$(u,v) \in E$?	$\Theta(1)$	$O(1+\deg(u))$
List $v's$ neighbours	$\Theta(n)$	$\Theta(1+\deg(v))$
List all edges	$\Theta(n^2)$	$\Theta(n+m)$

The algorithms we focus on will typically require listing neighbours and all edges so we'll use adjacency lists.

Exploring Graphs

To explore a graph, we want to visit all vertices, or all vertices starting at some **source**. This will help us find shortest paths and connected components.

Two typical strategies are breadth first search and depth first search. To keep track of where we have been and where we plan to go next, we may mark vertices as **undiscovered/discovered** or **unexplored/explored**.

- A vertex is **discovered** meaning we have identified it as a place we want to go but have not yet visited.
- A vertex has been **explored** once we visit the node and perform the work needed to be done at the node.
- undiscovered $\Rightarrow$ discovered but unexplored $\Rightarrow$ explored

Breadth First Search (BFS)

Breadth First Search (BFS)

- BFS begins at a specified vertex v_0.
- A cautious search: "spreads out" from v_0 by checking everything one edge away, then two, etc.
- First, from v_0 **discover** all neighbours of v_0.
- Next, **explore** all neighbours of v_0 to discover the neighbours of neighbours (2 edges away).
  How do we know if a neighbour of a neighbour (2 edges away) was already handled as a neighbour of v_0?

Breadth First Search (BFS)

Breadth First Search (BFS)

- BFS begins at a specified vertex v_0.
- A cautious search: "spreads out" from v_0 by checking everything one edge away, then two, etc.
- First, from v_0 **discover** all neighbours of v_0.
- Next, **explore** all neighbours of v_0 to discover the neighbours of neighbours (2 edges away).
  How do we know if a neighbour of a neighbour (2 edges away) was already handled as a neighbour of v_0? It is marked as discovered.
- Then, explore all neighbours of neighbours to identify vertices that are 3 edges away from v_0 (but have not previously been discovered).
- This process continues until all vertices have been explored.

Implementation: Use a queue to keep track of the vertices that have been discovered but must still be explored.
Also useful to store **parent** and **level** information.

Breadth First Search (BFS)

Explore(v)

```
1.  for each neighbour u of v do
2.      if mark(u) = undiscovered then
3.          mark(u) ← discovered
4.          parent(u) ← v
5.          level(u) ← level(v) + 1
6.          Add u to Queue
```

BFS

```
1.  Initialization: Mark all vertices as undiscovered
2.  Pick initial vertex v_0
3.  parent(v_0) ← ∅
4.  level(v_0) ← 0
5.  Add v_0 to Queue
6.  Mark(v_0) ← discovered
7.  while Queue is not empty do
8.      v ← remove from Queue
9.      Explore(v)
```

Breadth First Search (BFS)

Runtime: We explore each vertex once and check all incident edges:

Breadth First Search (BFS)

Runtime: We explore each vertex once and check all incident edges: $O(n + \sum_{v \in V} \deg(v)) = O(n + m)$

Note: $\sum_{v \in V} \deg(v) = 2m$ since we count each edge twice.

Properties

- The parent pointers create a directed tree (because each addition adds a new vertex u, with parent v in the tree).
- u is connected to v_0 if and only if BFS from v_0 reaches u.

Breadth First Search (BFS)

Runtime: We explore each vertex once and check all incident edges: $O(n + \sum_{v \in V} \deg(v)) = O(n + m)$

Note: $\sum_{v \in V} \deg(v) = 2m$ since we count each edge twice.

Properties

- The parent pointers create a directed tree (because each addition adds a new vertex u, with parent v in the tree).
- u is connected to v_0 if and only if BFS from v_0 reaches u.

Lemma: The level of a vertex v = length of shortest path from v_0 to v.

Proof:

Claim 1: v in level i implies there is a path v_0 to v of i edges.

Claim 2: v in level i implies every path v_0 to v has $\geq i$ edges.

Proof of Claim 1

Claim 1: v in level i implies there is a path v_0 to v of i edges.

Proof: By induction on the level, i.

Basis: $i = 0$: the only node on level 0 is the root of the tree so $v = v_0$.

Induction hypothesis: For a vertex v on level $i - 1$, the path from v_0 to v is $i - 1$ edges.

Proof of Claim 1

Claim 1: v in level i implies there is a path v_0 to v of i edges.

Proof: By induction on the level, i.

Basis: $i = 0$: the only node on level 0 is the root of the tree so $v = v_0$.

Induction hypothesis: For a vertex v on level $i - 1$, the path from v_0 to v is $i - 1$ edges.

Induction step: Let v be in level i.
Since it is a tree, the parent(v) is in level $i - 1$.
By the induction hypothesis, there is a path from v_0 to parent(v) of $i - 1$ edges. Extending the path to v, adding edge (parent(v), v) gives a path from v_0 to v of i edges.

Proof of Claim 2

Claim 2: v in level i implies every path v_0 to v has $\geq i$ edges.

Proof: We will prove that if there is a path v_0 to v of j edges then v is in level $\leq j$ by induction on j.

Basis: $j = 0$: the only node reachable on 0 edges from v_0 is v_0 itself; which is on level 0.

Induction hypothesis: For a vertex v where the path from v_0 to v is $j - 1$ edges, then v is on level $\leq j - 1$.

Proof of Claim 2

Claim 2: v in level i implies every path v_0 to v has $\geq i$ edges.

Proof: We will prove that if there is a path v_0 to v of j edges then v is in level $\leq j$ by induction on j.

Basis: $j = 0$: the only node reachable on 0 edges from v_0 is v_0 itself; which is on level 0.

Induction hypothesis: For a vertex v where the path from v_0 to v is $j - 1$ edges, then v is on level $\leq j - 1$.

Induction step: Let v be a vertex where the path from v_0 to v is j edges. Let u be the vertex before v in the path so the path from v_0 to u is $j - 1$ edges.
By the induction hypothesis, u must be on level $\leq j - 1$. So, v is only one edge (u, v) further away from v_0 and must be on level $\leq j$.

Breadth First Search (BFS)

Consequences

1. BFS from v_0 finds the connected component of v_0.
2. BFS finds all the shortest paths (number of edges) from v_0.

Exercises

- Enhance BFS to find all connected components in $O(n + m)$ time.
- Use BFS to find if a connected graph has a cycle.
- Prove that if $(u, v) \in E$ then level(u), level(v) differ by 0 or 1.

Applications of BFS

Applications

- Find the shortest path (number of edges) from a root vertex v_0 to any node v = level of v.
- Test if a graph has a cycle.
- Test if a graph is bipartite.

G is **bipartite** if V can be partitioned into $V_1 \cup V_2$ (i.e. $V_1 \cap V_2 = \emptyset$) such that every edge has one end in V_1 and one end in V_2.

Note: A bipartite graph cannot contain an odd cycle.

Recall from previous exercises:

- If $(u, v) \in E$ then level(u) and level(v) differ by 0 or 1.
- G has a cycle $\iff$ we discover an edge (dashed) that is not used in the BFS tree; i.e. we encounter a vertex that has already been discovered.

Testing for Bipartite

Idea: V_1 is all vertices on odd levels and V_2 is all vertices on even levels. If no dashed edges, then this is clearly true.

Run BFS. For each edge $(u, v) \in E$ check if $u, v \in V_1$ or $u, v \in V_2$.

- This can be done during BFS when we check the neighbours of vertex.

If no such edge is found, then G is bipartite.

If such an edge is found, then G is not bipartite.

Testing for Bipartite

Idea: V_1 is all vertices on odd levels and V_2 is all vertices on even levels.
If no dashed edges, then this is clearly true.

Run BFS. For each edge $(u, v) \in E$ check if $u, v \in V_1$ or $u, v \in V_2$.

- This can be done during BFS when we check the neighbours of vertex.

If no such edge is found, then G is bipartite.

If such an edge is found, then G is not bipartite.

- Since level(u) and level(v) differ by 0 or 1, if they are in the same vertex set, the difference must be $0 \Rightarrow u, v$ are on the same level.
- Let z be the least (closest) common ancestor of u, v. Then a path from u to z and z to v are the same length, say k and edge (u, v) creates a cycle of odd length $2k + 1$.

Lemma: G is bipartite $\iff$ G has no odd cycle.
Note: The algorithm either finds a bipartition OR an odd cycle.

Depth First Search (DFS)

Depth First Search (DFS)

- DFS begins at a specified vertex v_0.
- A bold search: go as far away as you can; when nothing new to discover, retrace your steps to find something new.
- From v_0 discover a new neighbour and go explore it.
- From the neighbour, discover a new neighbour (of the neighbour) and go explore it.
- Repeat this until you reach a vertex with no undiscovered neighbours.
- Backtrack your path 1 edge and check for a new undiscovered neighbour then go back 2 bullets; otherwise repeat this step.

Implementation: Marked a vertex as **finished** when all of its neighbours have been **explored**.
Useful to store parent and also if an edge is a tree edge or non-tree edge.

Depth First Search (DFS)

DFS(v)
1. Mark(v) $\leftarrow$ discovered
2. **for** $u \in$ AdjacencyList(v)) **do**
3. **if** u is undiscovered **then**
4. DFS(u)
5. **parent(u)** $\leftarrow v$
6. (u, v) **is a tree edge**
7. **else** (u, v) **is a non-tree edge** if $u \neq$ parent(v)
8. Mark(v) $\leftarrow$ finished

DFS Main
1. Mark all vertices undiscovered
2. **for** $v \in V$ **do** // Handles multiple components
3. **if** v is undiscovered **then** // Start a new tree rooted at v
4. DFS(v)

Runtime: $O(n + m)$, similar to BFS

Depth First Search (DFS)

DFS gives us some nice structural properties:

- Partitions G into separate trees (connected components).
- Gives an edge classification.
- Vertex orderings: order of discovery, order of finishing

Lemma: DFS(v_0) reaches all vertices connected to v_0.

Proof: Suppose there is a path $v_0, v_1, \ldots, v_i, \ldots, v_k$ but we only discover the vertices upto v_i; i.e. we don't reach v_k.
DFS would explore all neighbours of v_i including v_{i+1} (a contradiction).
We can also repeat this process (math induction) to show we extend the path to reach v_k.

Depth First Search (DFS)

Lemma: All non-tree edges join an ancestor and a descendant (vertices on the same branch).

Proof: Suppose a non-tree edge $(x, y) \in E$ does not join an ancestor and a descendant and WLOG, suppose x is discovered first.
Then, in DFS(x), we would discover and explore neighbour y before x finishes.
Thus, y would appear in the subtree of x:
x would be the ancestor of y and y would be a descendant of x (a contradicton).

Enhancing DFS

Exercise: Enhance DFS code to number the connected components and store the component number for each vertex.

Enhance DFS to compute discovery and finish times

```
Initialize a clock: time ← 1
DFS(v)
 1.    Mark(v) ← discovered
 2.    discover(v) ← time
 3.    time ← time + 1
 4.    for u ∈ AdjacencyList(v)) do
 5.        if u is undiscovered then
 6.            DFS(u)
 7.    Mark(v) ← finished
 8.    finish(v) ← time
 9.    time ← time + 1
```

Cut Vertices

A vertex v is a **cut vertex** if removing v makes G disconnected.

- If the graph represents a network, the breakdown at the cut vertex disconnects the network.
- We can use DFS to find cut vertices.

Characterizing Cut Vertices

Claim: The root is a cut vertex $\iff$ it has > 1 child.

Lemma: A non-root v is a cut vertex $\iff$ v has a subtree T with no non-tree edge going to a proper ancestor of v.

Cut Vertices

A vertex v is a **cut vertex** if removing v makes G disconnected.

- If the graph represents a network, the breakdown at the cut vertex disconnects the network.
- We can use DFS to find cut vertices.

Characterizing Cut Vertices

Claim: The root is a cut vertex $\iff$ it has > 1 child.

Lemma: A non-root v is a cut vertex $\iff$ v has a subtree T with no non-tree edge going to a proper ancestor of v.

Proof:
($\Leftarrow$) Removing v disconnects T from the rest of G.
($\Rightarrow$) Since removing v disconnects G, some subtree must get disconnected.

Algorithm to Identify Cut Vertices

How do we check if a vertex in subtree T of v has a non-tree edge going to a proper ancestor of v or not? Use the discovery times

- Recall previous Lemma: All non-tree edges join an ancestor and a descendant (vertices on the same branch).
- A proper ancestor of v would have a discovery time $<$ discovery(v).
- Check vertices in T, if on a non-tree edge, does the other endpoint have a discovery time $<$ discovery(v).

How do we implement this?

DFS on Directed Graphs

Classifying Edges

- An edge in the DFS tree is a **tree edge**.
  Note: these edge are directed from an ancestor to a descendant.
- **Forward edge**: a non-tree edge (v, u) where u is a descendant of v.
- **Back edge**: a non-tree edge (v, u) where u is an ancestor of v.
- **Cross edge**: a non-tree edge (v, u) where u is not a descendant of v and v is not a descendant of u.

DFS on Directed Graphs

```
DFS(v)
 1.    mark(v) ← discovered
 2.    discover(v) ← time
 3.    time ← time + 1
 4.    for u ∈ AdjacencyList(v)) do
 5.        if u is undiscovered then
 6.            DFS(u)
 7.            (v, u) is a tree edge
 8.        else // Not a tree edge
 9.            if u is not finished then
10.                (v, u) is a back edge
11.            else if discover(u) > discover(v) then
12.                (v, u) is a forward edge
13.            else
14.                (v, u) is a cross edge
15.    mark(v) ← finished
16.    finish(v) ← time
17.    time ← time + 1
```

Applications of DFS

Lemma: A directed graph has a (directed) cycle $\iff$ DFS has a back edge.

Topological sort of a directed acyclic (no directed cycles) graph.

- A directed edge (a, b) means a must come before b.
- Find a linear order of vertices satisfying all edge constraints.
  Note: this is possible $\iff$ G has no directed cycle.

$\Rightarrow$ Reverse finish order.

Finding strongly connected components in a directed graph

- Strongly connected: for all vertices u, v there is a path from u to v.
- Let s be a vertex. G is strongly connected $\iff$ for all vertices v, there is a path from s to v and a path from v to s.

Minimum Spanning Tree

Problem

Minimum Spanning Tree (MST)

Instance: *Given a connected graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ on the edges.*

Find: *A subset of the edges of size $n - 1$ that connects all the vertices and has minimum weight. The edge subset is called a* ***minimum spanning tree****.*

Recall: Any connected graph on n vertices and $n - 1$ edges is a tree.

There are several greedy approaches to find a MST:

- **Kruskal's Algorithm**: Always choose cheapest edge available that doesn't build a cycle.
- **Prim's Algorithm**: from a vertex, grow a connected graph by choosing the least expensive edge that connects to a new vertex.
- Remove the most expensive edge that doesn't disconnect the graph.

Kruskal's Algorithm

1. Order edges by weight: $e_1, \ldots, e_m$ s.t. $w(e_i) \leq w(e_{i+1})$
2. $T \leftarrow \emptyset$
3. **for** $i \leftarrow 1$ **to** m **do**
4. **if** e_i does not make a cycle with T **then**
5. $T \leftarrow T \cup \{e_i\}$

Edge e makes a cycle with $T \iff e$ joins vertices in the same connected component.

Correctness: An exchange proof.

Let T have edges $t_1, \ldots, t_{n-1}$.
Prove by induction on i that there is a MST matching T on the first i edges.

Kruskal's Algorithm

Analysis: Recall, graph $G = (V, E)$ where $|V| = n$ and $|E| = m$.
$O(m \log m)$ to sort edges but $m \leq n^2$, so $\log m \leq 2 \log n \in O(\log n)$
$\Rightarrow O(m \log n)$

Need to maintain connected components as edges are added. Also test:

- if (a, b) has a, b in the same component (don't add edge), or
- if (a, b) have different components (add the edge).

Kruskal's Algorithm

Analysis: Recall, graph $G = (V, E)$ where $|V| = n$ and $|E| = m$.
$O(m \log m)$ to sort edges but $m \leq n^2$, so $\log m \leq 2 \log n \in O(\log n)$
$\Rightarrow O(m \log n)$

Need to maintain connected components as edges are added. Also test:

- if (a, b) has a, b in the same component (don't add edge), or
- if (a, b) have different components (add the edge).

Union-Find Problem: Maintain a collection of disjoint sets with operations:

- **Find(v)** - determine which set contains element v.
- **Union(X, Y)** - unite two sets X and Y.

For MST, elements are vertices and sets are connect components of T, the tree so far. The simple implementation of this ADT gives $O(m \log n)$ for Kruskal.

Prim's Algorithm

Build a single connected component C (that will eventually be the MST T) by choosing a vertex $v \notin C$ with a minimum weight edge (u, v) where $u \in C$.

1. $C \leftarrow \{s\}$
2. $T \leftarrow \emptyset$
3. **while** $C \neq V$ **do**
4. Find a vertex $v \in V - C$ such that there exists a $u \in C$ with $e = (u, v)$ a minimum weight edge leaving C
5. $C \leftarrow C \cup \{v\}$
6. $T \leftarrow T \cup \{e\}$

Correctness: An exchange proof (similar to one for Kruskal's Algorithm).

What data structure should we use to help choose v?
Should it store vertices or edges?

Prim's Algorithm - Implementation

Find vertex $v \in V - C$ such that $e = (u, v)$, $u \in C$ and e is a minimum weight edge leaving C. Define:

$$weight(v) = \begin{cases} \infty \text{ if no edge } (u, v) \text{ with } u \in C \\ min\{w(e) : e = (u, v) \in E \text{ and } u \in C\} \text{ otherwise} \end{cases}$$

Prim's Algorithm - Implementation

Find vertex $v \in V - C$ such that $e = (u, v)$, $u \in C$ and e is a minimum weight edge leaving C. Define:

$$weight(v) = \begin{cases} \infty \text{ if no edge } (u, v) \text{ with } u \in C \\ min\{w(e) : e = (u, v) \in E \text{ and } u \in C\} \text{ otherwise} \end{cases}$$

PriorityQueue (heap):

- Maintain a set $V - C$ as an array in heap order, according to *weight*
- **ExtractMin**(): remove and return vertex with minimal *weight*
- **Insert**(v, $weight(v)$)
- **Delete**(v) - v may be any vertex in $V - C$
- Want all operations to be $O(\log k)$ where $k = |V - C|$

Careful! Implementation is tricky!

Prim's Algorithm - Analysis

- Need to **ExtractMin** each vertex to add it to C
- When v is extracted, search v's adj list to find the edge $e = (v, u)$ with weight=$weight(v)$ and $u \in C \rightarrow$ **add this edge to MST**
- Also, for each $v' \in V - C$ in v's adj list, update/reduce $weight(v')$ $\rightarrow$ **Delete and re-Insert v' with updated $weight(v')$**

Prim's Algorithm - Analysis

- Need to **ExtractMin** each vertex to add it to C
- When v is extracted, search v's adj list to find the edge $e = (v, u)$ with weight=$weight(v)$ and $u \in C \rightarrow$ **add this edge to MST**
- Also, for each $v' \in V - C$ in v's adj list, update/reduce $weight(v')$ $\rightarrow$ **Delete and re-Insert v' with updated** $weight(v')$

Size of heap: $O(n)$

- $n - 1$ **ExtractMin** operations
- $O(m)$ update *weight*, **Delete** and re-**Insert** operations

Prim's Algorithm - Analysis

- Need to **ExtractMin** each vertex to add it to C
- When v is extracted, search v's adj list to find the edge $e = (v, u)$ with weight=$weight(v)$ and $u \in C \rightarrow$ **add this edge to MST**
- Also, for each $v' \in V - C$ in v's adj list, update/reduce $weight(v')$ $\rightarrow$ **Delete and re-Insert v' with updated $weight(v')$**

Size of heap: $O(n)$

- $n - 1$ **ExtractMin** operations
- $O(m)$ update *weight*, **Delete** and re-**Insert** operations

Total: $O(m \log n)$, we assume the graph is connected; i.e., $m \geq n - 1$

Shortest Paths in Edge Weighted Graphs

BFS finds the shortest path from some vertex v (root of BFS tree) to other connect vertices in an unweighted undirected graph.

General Input: Directed or undirected graph with weights on edges.
Note: In directed graphs, typically do not allow negative weight cycles.

There are many different problems related to shortest paths.

- Given u, v, find shortest uv path.
- Given u, find shortest uv path $\forall v$ - "single source shortest path problem".
- Find shortest uv path $\forall u, \forall v$ - "all pairs shortest path problem".

Dijkstra's Algorithm (1959)

Input: Graph or directed graph $G = (V, E)$, $w : E \to \mathbb{R}^{\geq 0}$ and source vertex $s \in V$

Output: Shortest path from s to every other vertex v.

Choose edge $(x, y), x \in B, y \notin B$ to minimize $d(s, x) + w(x, y)$ where $d(s, x)$ is the (known) minimum distance from s to x.
Call this minimum distance d.

```
1.  d(v) ← ∞ ∀v ≠ s
2.  d(s) ← 0
3.  B ← {s}
4.  while |B| < n do
5.      y ← vertex of V − B with minimum d value
6.      for z ∈ AdjacencyList(y) do
7.          if d(y) + w(y, z) < d(z) then
8.              d(z) ← d(y) + w(y, z)
9.              Parent(z) ← y
```

Edsger W. Dijkstra (1930 - 2002)

Dijkstra was known for many contributions to computer science, e.g. structured programming, concurrent programming. He designed the previous algorithm to demonstrate the capabilities of a new computer (to find railway journeys in the Netherlands). At that time (50s) the result was not considered important. He wrote:

At the time, algorithms were hardly considered a scientific topic. I wouldn't have known where to publish it... The mathematical culture of the day was very much identified with the continuum and infinity. Could a finite discrete problem be of any interest? The number of paths from here to there on a finite graph is finite; each path is a finite length; you must search for the minimum of a finite set. Any finite set has a minimum - next problem, please. It was not considered mathematically respectable.

Single Source Shortest Paths in a DAG

A *directed acyclic graph* (DAG) has no directed cycle.

Idea: Use topological sort $v_1 v_2 \ldots v_n$ so every edge (v_i, v_j) has $i < j$.

If v comes before s, there is no path $s \to v$ so remove all such vertices, relabel, let $s = v_1$.

```
1.   d_i ← ∞ ∀i
2.   d_1 ← 0
3.   for i from 1 to n do
4.       for every edge (v_i, v_j) do
5.           if d_i + w(v_i, v_j) < d_j then
6.               d_j ← d_i + w(v_i, v_j)
```

Analysis: $O(n + m)$

Claim: This finds shortest paths from s.
Exercise: Proof by induction on i.