1 Introduction to Graphs

A graph G is a pair (V, E), where V is a set of nodes, called vertices and E is a collection of pairs of vertices, called edges.

Edge Types
A graph can be directed or undirected. In a directed graph, the edges are ordered pairs of vertices (u, v), where u is the origin/tail and v is the destination/head. In an undirected graph, the edges are unordered pairs of vertices (u,v), whereby u and v can access each other either way.

Simple paths
A path is a sequence/set of vertices such that every pair of consecutive vertices is connected by an edge. A simple path is a path with no repeated vertices. As such, all vertices in that path are distinct.

Cycles in graphs
A cycle is a path with at least one edge, whose first and last vertices are the same. A simple cycle is a cycle. A simple cycle is one where all vertices are distinct, except for the first and last vertices. An acyclic graph has no cycle in it, which is impossible for undirected graphs. A directed graph can be acyclic.

Subgraphs
A subgraph S of a graph G is defined as S = (U, F) where U is a subset of vertices in V and F is a subset of edges in E. An induced subgraph of G is a subgraph that is formed by selecting a subset U of vertices and all the edges in G that have both endpoints in U, which is denoted as G[U]. Similarly, an induced subgraph can be formed by selecting a subset F of edges and the vertices that are endpoints of the edges in F, which is denoted as G[F].

Connectivity
A graph G = (V, E) is considered connected if there is a path between every pair of vertices in V. In other words, every vertex in the graph is reachable from every other vertex. A connected component of a graph G is a maximally connected subgraph of G, meaning that it is a subgraph that is connected, and there is no larger connected subgraph that contains it. A graph is considered disconnected if it has two or more connected components, which means that there are pairs of vertices that are not connected by any path in the graph.

Trees and Forests
An unrooted tree is a graph that is connected and has no cycles. A forest is a graph without cycles, and its connected components are trees. Every tree on n vertices has exactly n-1 edges, which is a well-known property of trees in graph theory. A rooted tree is a type of tree that is produced by a directed graph where the edges are directed away from the root.

1.1 Graph Properties

- $\sum_{v \in V}$ deg(v) = 2m, where m is the number of edges in the graph. This states that the sum of the degrees of all the vertices in the graph is equal to twice the number of edges in the graph.
- In a simple undirected graph m <= n (n - 1)/2.
- n represents the number of vertices, m represents the number of edges. And $\triangle$ represents the maximum degree.

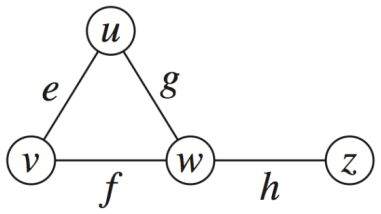
2 Graph ADT

We model the abstraction as a combination of three data types: Vertex, Edge, and Graph. A Vertex stores an associated object (e.g., an airport code) that is retrieved with a getElement() method. An Edge stores an associated object (e.g., a flight number, travel distance) that is retrieved with a getElement() method.

2.1 Edge List Structure

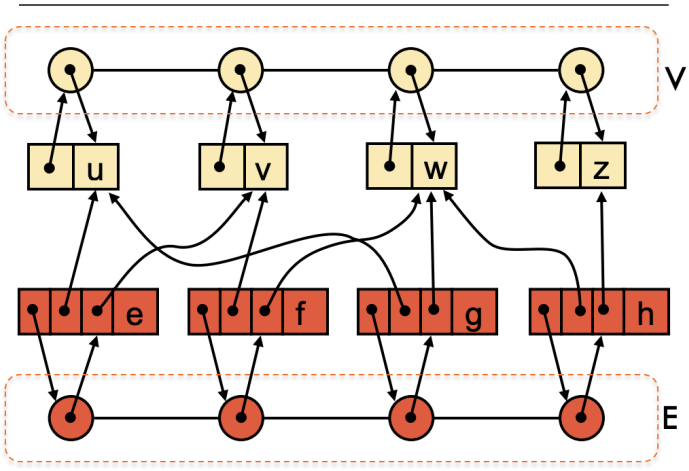
Vertex sequence holds

- sequence of vertices
- vertex object keeps track of its position in the sequence



Edge sequence

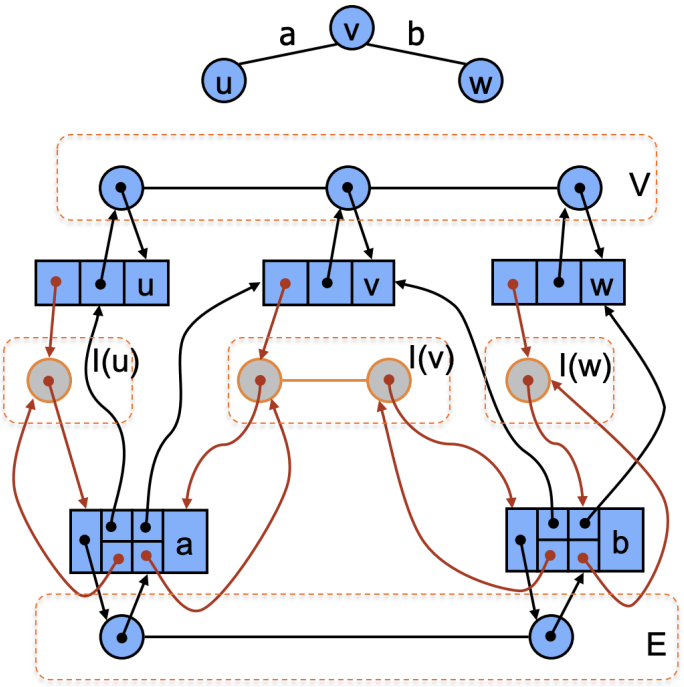
- sequence edges
- edge object keeps track of its position in the sequence
- Edge object points to the two vertices it connects



2.2 Adjacency List

Additionally each vertex keeps a sequence of edges incident on it

Edge objects keep reference to their position in the incidence sequence of its endpoints

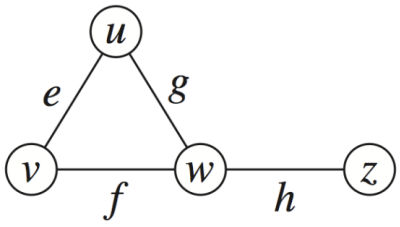


2.3 Adjacency Matrix Structure

Vertex array induces an index from 0 to n-1 for each vertex

2D-array adjacency matrix

- Reference to edge object for adjacent vertices
- Null for nonadjacent vertices



	0	1	2	3
u → 0		e	g	
v → 1	e		f	
w → 2	g	f		h
z → 3			h	

2.4 Asymptotic performance

No parallel edges/s-loops	Edge List	Adjacency List	Adjacency Matrix
Space	O(n + m)	O(n + m)	O(n^2)
incidentEdges(v)	O(m)	O(deg(v))	O(n)
getEdge(u, v)	O(m)	O(min(deg(u), deg(v)))	O(1)
insertVertex(x)	O(1)	O(1)	O(n^2)
insertEdge(u, v, x)	O(1)	O(1)	O(1)
removeVertex(v)	O(m)	O(deg(v))	O(n^2)
removeEdge(e)	O(1)	O(1)	O(1)

3 Depth-First Search (DFS)

This strategy tries to follow outgoing edges leading to yet unvisited vertices whenever possible, and backtrack if "stuck." If an edge is used to discover a new vertex, we call it a DFS edge, otherwise we call it a back edge. The DFS edges form a spanning tree of the connected component being explored.

```
1. def DFS(G):                                  # Total run time is O(n + m)
2.     for u in G.vertices() do                 # Set things up for DFS: O(n)
3.         visited[u] ← false
4.         parent[u] ← null
5.     for u in G.vertices() do                 # Visit all vertices: O(n) not counting DFS_Visit
6.         if not visited[u] then
7.             DFS_Visit(u)
8.     return parent
```

```
1. def DFS_Visit(u):                            # O(deg(u)). Total runtime: O(m)
2.     visited[u] ← true
3.     for v in G.incidentEdges(u) do           # Check out all neighbors of u
4.         if not visited[v] then
5.             parent[v] ← u
6.             DFS_Visit(v)
```

Let Cv be a connected component of v in our graph G. The DFS_Visit visits all vertices in Cv before returning to the outer loop. Edges {(u, parent[u]): u in Cv} form a spanning tree of C. Edges {(u, parent[u]): u in V} form a spanning forest of G.

DFS runs in O(n + m). It can also solve other graph problems in this time. For example, it can find a path between two given vertices, if any, find a cycle in the graph, test whether a graph is connected, compute connected components of a graph and compute a spanning forest of a graph.

3.1 Cut edges

In a connected graph G=(V, E), we say that an edge (u, v) in E is a cut edge if (V, E / (u, v)) is not connected. The cut edge problem is to identify all cut edges. We can solve this problem by running DFS on G. Trivial O(m2) time algorithm: For each edge (u,v) in E, remove (u,v) and check using DFS if G is still connected, put back (u,v). Better O(nm) time algorithm: Only test edges in a DFS tree of G. Way to do it in O(n + m) but we don't know how to do it yet.

4 Breadth-First Search (BFS)

This strategy tries to visit all vertices at distance k from a start vertex s before visiting vertices at distance k + 1:

- L_0 = s
- L_1 = vertices one hop away from s
- L_2 = vertices two hops away from s but no closer
- k = vertices k hops away from s but no closer

```
1. def BFS(G, s):                                          # Total run time is O(n + m)
2.     for u in G.vertices() do                            # Set things up for BFS: O(n)
3.         visited[u] ← false
4.         parent[u] ← null
5.     seen[s] ← true
6.     layers ← []
7.     current ← [s]
8.     next ← []
9.     while current is not empty do                       # O(m)
10.        layers.append(current)
11.        for u in current do                             # iterate over current layer
12.            for v in G.incidentEdges(u) do              # iterate over neighbors of u
13.                if not seen[v] then
14.                    seen[v] ← true
15.                    parent[v] ← u
16.                    next.append(v)
17.        current ← next                                  # update current and next layer
18.        next ← []
19.    return layers, parent
```

4.1 BFS Properties

- Let C_v be the connected component of v in our graph G.
- Fact: BFS(G, s) visits all vertices in C_s.
- Fact: Edges $\{(u, \text{parent}[u]) : u \in C_s\}$ form a spanning tree T_s of C_s.
- Fact: For each v in L_i, there is a path in T_s from s to v with i edges.
- Fact: For each v in L_i, any path in G from s to v has at least i edges.
- Fact: Assuming adjacency list representation we can perform a BFS traversal of a graph with n vertices and m edges in O(n+m) time
- Fact: Assuming adjacency matrix representation we can perform a BFS traversal of a graph with n vertices and m edges in O(n^2) time

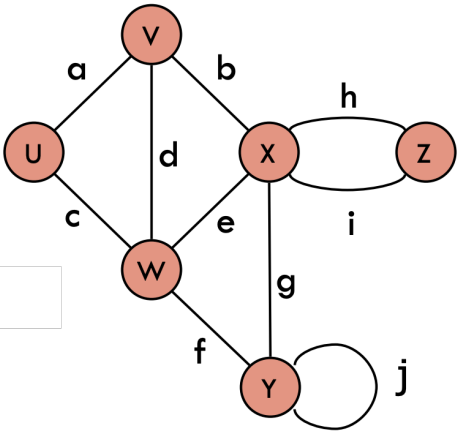
4.2 BFS Applications

BFS can be used to solve other graph problems in O(n + m) time. For example, it can find the shortest path between two given vertices, find a cycle in a graph, test whether a graph is connected or compute a spanning tree of a graph (if connected).

5 Terminology

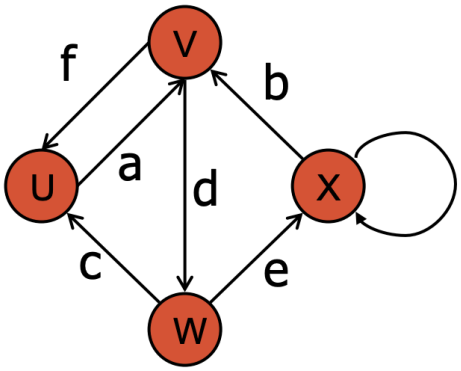
5.1 Undirected graphs

- Edges connect endpoints (e.g., W and Y for edge f)
- Edges are incident on endpoints (e.g., a, d, and b are incident on V)
- Adjacent vertices are connected (e.g., U and V are adjacent)
- Degree is the number of edges on a vertex (e.g., X has degree 5)
- Parallel edges share the same endpoints (e.g., h and i are parallel)
- Self-loop have only one endpoint (e.g., j is a self-loop)
- Simple graphs have no parallel or self-loops



5.2 Directed graphs

- Edges go from tail to head (e.g., W is the tail of c and U its head)
- Out-degree is the number of edges out of a vertex (e.g., W has out-degree 2)
- In-degree is the number of edges into a vertex (e.g., W has in-degree 1)
- Parallel edges share tail and head (e.g., no parallel edge on the right)
- Self-loop have the same head and tail (e.g., X has a self-loop)
- Simple directed graphs have no parallel or self-loops, but are allowed to have antiparallel loops like f and a



6 Weighted Graphs

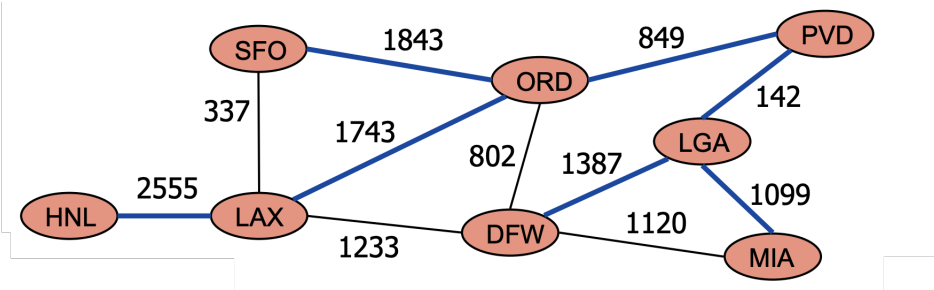
In a weighted graph, each edge has an associated numerical value, called the weight of the edge. Edge weights may represent, distances, costs, etc.

6.1 Shortest Paths

Given an edge weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v, where the weight of a path is the sum of the weights of its edges.

Properties:

- A subpath of the shortest path is itself the shortest path.
- There is a tree of the shortest paths from a start vertex to all the other vertices (Shortest path tree).



7 Dijkstra's Algorithm

Input:

- Graph $G = (V, E)$.
- Edge weights $w : E \to R^+$.
- Start vertex s.

Output:

- Distance from s to all v in V.
- The shortest path tree rooted at s.

Assumptions:

- G is connected and undirected.
- Edge weights are nonnegative.

High level idea:

- Maintain a distance estimate $D[v] \geq dist_w(s, v)$ for all v in V.
- Keep track of a subset S of V such that $D[v] = dist_w(s, v)$ for all v in S.

Initially:

- $D[s] = 0$.
- $D[v] = \infty$ for all v in $V - s$.

In each iteration we:

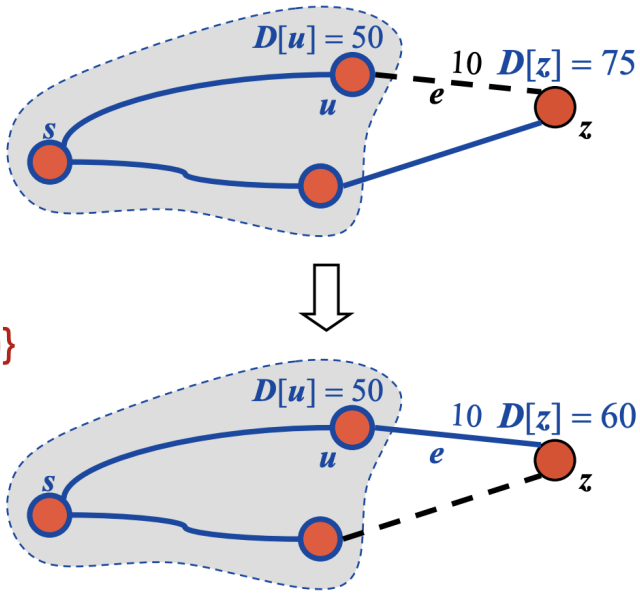
- add to S vertex u in $V - S$ with smallest $D[u]$.
- update D-values for vertices adjacent to u.

7.1 Edge Relaxation

Consider edge e = (u, z) such that:
- u is the last vertex added to S
- z is not in S

The relaxation of edge (u, z) updates D[z] as follows:

D[z] ← min{D[z], D[u] + w(u, z)}



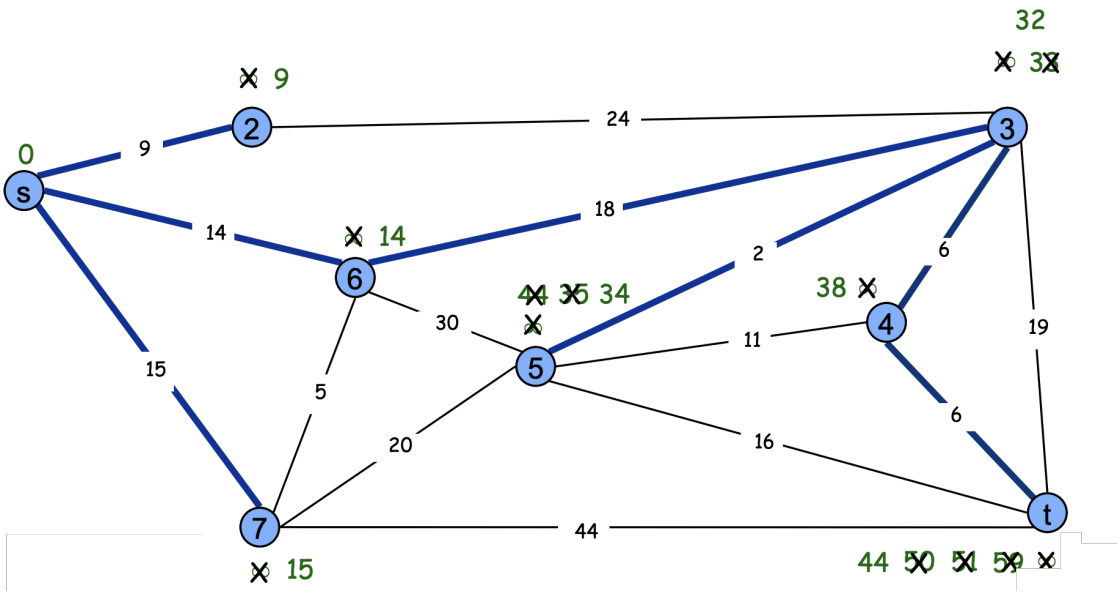
7.2 Dijkstra's Algorithm pseudocode

```
1. def Dijkstra(G, w, s):                                  # Total run time is O(n + m log n)
2.     for v in G.vertices() do                            # Set things up for Dijkstra: O(n)
3.         D[v] ← ∞
4.         parent[v] ← null
5.     D[s] ← 0
6.     Q ← new PriorityQueue() for {(v, D[v]) : v in V}    # Key is depth estimate

7.     while Q is not empty do                             # iteratively add vertices to S: O(m log n)
8.         u ← Q.removeMin()
9.         for z in G.incidentEdges(u) do                  # relax edges out of u: O(deg(u))
10.            if D[z] > D[u] + w(u, z) then
11.                D[z] ← D[u] + w(u, z)
12.                parent[z] ← u
13.                Q.update_PriorityQueue(z, d[z])
14.    return D, parent   # dictionaries (vertex to shortest path from s) (vertex to parent)
```

7.3 Dijkstra's Algorithm Analysis

- Assuming the graph is connected (so $m \geq n - 1$), the algorithm spends $O(m)$ time on everything except priority queue operations.
- The priority queue operations involve n inserts, m decrease_key operations, and n remove_min operations.
- Using a binary heap for the priority queue, the total time complexity is $O(m \log n)$.
- Using a Fibonacci heap for the priority queue, the total time complexity is $O(m + n \log n)$, due to the $O(1)$ amortized time for decrease_key operations.



7.4 Correctness of Dijkstra's Algorithm

To prove Dijkstra's correctness, we can use induction on the number of vertices visited by the algorithm. Let's denote the set of visited vertices as S and the set of unvisited vertices as V-S.

Base case:
At the start, the distance of the starting vertex s is 0, which is the correct shortest path distance from s to itself.

Inductive step
Assume that the algorithm maintains the correct shortest path distances for all vertices visited so far. Now we'll show that after visiting the next vertex u with the smallest distance among the unvisited vertices, the algorithm still maintains the correct shortest path distances for all visited vertices.

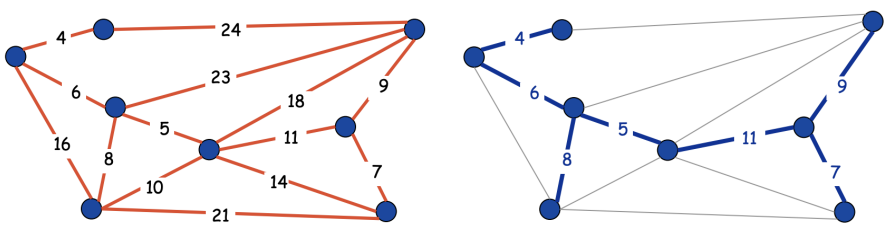
1. Consider the shortest path P from the starting vertex s to u. We claim that P does not contain any other unvisited vertices besides u. If P did contain another unvisited vertex w, the distance from s to w would be shorter than the distance from s to u, which contradicts our choice of u as the closest unvisited vertex.

2. After visiting u, we update the distance estimates for the neighbors of u. Since we've assumed that the shortest path from s to u is correct, any updated distance estimate for a neighbor v of u will also be correct. This is because the new distance estimate for v takes into account the correct shortest path distance from s to u, and the shortest path from s to v through u does not contain any other unvisited vertices (from point 1).

By induction, Dijkstra's algorithm maintains the correct shortest path distances for all visited vertices. When all vertices are visited, the algorithm has computed the correct shortest path distances for the entire graph.

8 Minimum Spanning Trees

Given a connected graph $G = (V, E)$ with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



8.1 MST properties

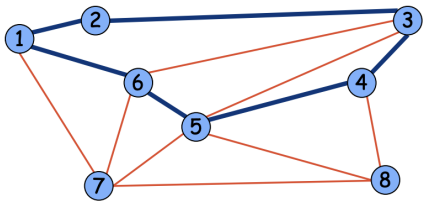
Simplifying assumption: All edge costs c_e are distinct.

Cut property: Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property: Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.

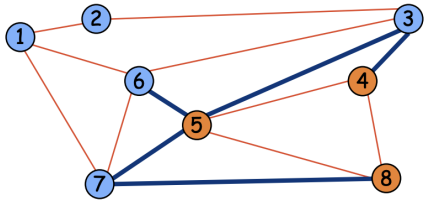
8.2 Cycles and Cuts

Cycle. Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



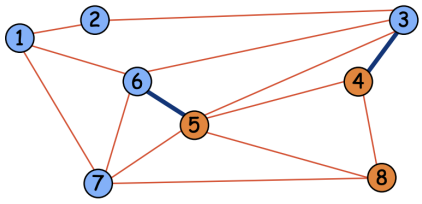
Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



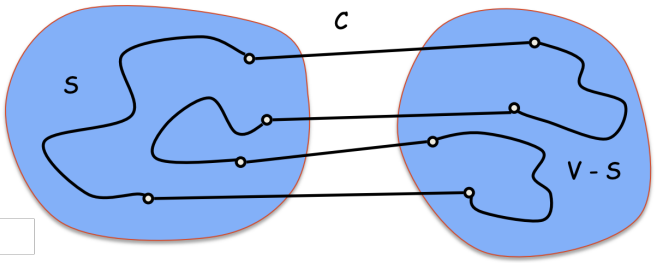
Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Proof:



9 Prim's Algorithm

Every time we add an edge we follow cut property!

```
1. def Prim(G, c):                    # C is the cost function. Total run time is O(n + m log n)
2.     for v in G.vertices() do                                  # Set things up for Prim: O(n)
3.         D[v] ← ∞
4.         parent[v] ← null
5.     u ← any vertex in V
6.     D[u] ← 0
7.     Q ← new PriorityQueue() for {(v, D[v]) : v in V}          # Key is depth estimate
8.     S ← []

9.     while Q is not empty do                  # iteratively add vertices to S: O(m log n)
10.        u ← Q.removeMin()
11.        S.append(u)
12.        for z in G.incidentEdges(u) do                # relax edges out of u: O(deg(u))
13.            if z not in S and D[z] > c(u, z) then
14.                parent[z] ← u
15.                decrease PriorityQueue d[z] to c(u, z)
16.    return parent              # parent is a dictionary that maps each vertex to its parent
```

9.1 Prim's Algorithm Analysis

Similar analysis to Dijkstra's algorithm: O(m log n) using a heap and O(m + n log n) using Fibonacci heap.

9.2 Dijkstra vs Prim and Kruskal

MPT algorithms finds the minimum spanning tree of a connected, undirected graph, while Dijkstra's algorithm computes the shortest paths from a source vertex to all other vertices in a directed or undirected graph.

- Shortest Path: The goal of finding the shortest path from a source vertex to all other vertices is to identify the path with the minimum total weight between the source vertex and each of the other vertices in the graph. The output of a shortest path algorithm (e.g., Dijkstra's algorithm) is a set of paths from the source vertex to all other vertices in the graph.
- Minimum Spanning Tree: The goal of finding an MST is to create a tree that spans all vertices in the graph while minimizing the sum of the edge weights. It connects all vertices without cycles and minimizes the total edge weight. The output of an MST algorithm (e.g., Prim's or Kruskal's algorithm) is a tree that spans all vertices in the graph with the minimum total edge weight.

9.3 Correctness of Prim's Algorithm

We'll prove that the tree constructed by Prim's algorithm is a minimum spanning tree (MST) by induction using the cut property.

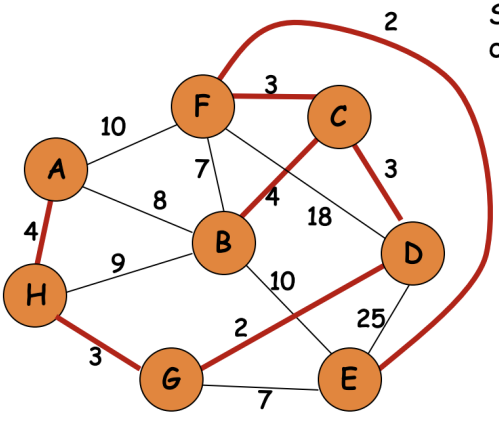
Base case:
Initially, the set T is empty, which is a valid subset of the MST.

Inductive hypothesis:
Assume after k iterations, the set T contains k edges of the MST.

Inductive step:
In the (k+1)-th iteration, Prim's algorithm selects the lightest edge e crossing a cut C. We want to show that e is part of the MST.

1. If no MST M exists that contains edges in T but not edge e, then Prim's algorithm has already built the MST, and the proof is complete.
2. If an MST M exists, find another edge f in M connecting the same vertex sets as edge e. Adding f to the tree formed by Prim's algorithm creates a cycle crossing the cut C.
3. Edge e has the minimum weight among edges crossing the cut C, so the weight of e is less than or equal to the weight of f.
4. Construct a new spanning tree M_1 by replacing f with e in M. The weight of M_1 is less than or equal to the weight of M.
5. Since M is an MST and the weight of M_1 is no greater than M, M_1 is also an MST containing edge e.

Thus, the edge e selected by Prim's algorithm in the (k+1)-th iteration is part of an MST. Therefore, by induciton, Prim's algorithm correctly constructs an MST.



Select node with minimum distance

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

10 Kruskal's Algorithm

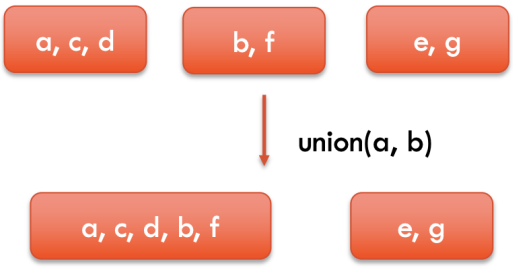
Kruskal's Algorithm works by examining the edges in ascending order of their weights and adding them to the MST if they do not create a cycle. Here's an expanded explanation of the algorithm:

1. Sort all the edges in the graph in ascending order of their weights.
2. Initialize the set T as an empty set. This set will store the edges of the MST.
3. For each edge e in the sorted list, do the following:
   (a) Case 1 (Cycle): If adding edge e to T results in a cycle, discard e according to the cycle property. The cycle property states that in an MST, there cannot be any cycles. Discarding e ensures that the resulting MST remains acyclic.
   (b) Case 2 (Cut): If adding edge e to T does not create a cycle, insert $e = (u, v)$ into T according to the cut property. The cut property states that for any partition of the graph's vertices into two non-empty sets (a cut), the lightest edge crossing the partition must be included in the MST. In this case, S represents the set of nodes in the connected component of vertex u. By adding edge e, we are connecting the components of vertices u and v while maintaining the properties of an MST.
4. Repeat step 3 for all edges in the sorted list.
5. Once all edges have been processed, the set T will contain the edges of the MST.

10.1 Union Find ADT

Data structure defined on a ground set of elements A used to keep track of an evolving partition.

- make_sets(A): makes |A| singleton sets with elements in A. O(n)
- find(a): returns an id for the set element a belongs to. O(1)
- union(a,b): union the sets elements a and b belong to. O(1)



10.2 Kurskal's algorithm implementation

```
1. def Kruskal(G, c):                    # c is the cost function. Total run time is O(m log n)
2.     sort E by increasing c-value order                                  # O(m log m)
3.     T ← []
4.     comp ← make_sets(V)                                                 # O(n)
5.     for e in E do                                                       # O(m)
6.         if comp.find(e.u) != comp.find(e.v) then
7.             T.append(e)
8.             comp.union(e.u, e.v)
9.     return T
```

10.3 Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct, we can perturb all edge costs by tiny amounts to break any ties. The impact of this perturbation is minimal because Kruskal and Prim's algorithms only interact with costs through pairwise comparisons. If the perturbations are sufficiently small, the MST with perturbed costs will be the same as the MST with the original costs. For instance, if we assume that all costs are integral, adding i/n^2 to each edge e_i would not significantly alter the MST. Under these perturbed weights, any MST found would still be an MST under the original weights, ensuring that the algorithms continue to function correctly despite the small changes in edge costs.