More formally, for a flow network $G = (V, E)$ with source s, sink t, and a flow f, consider a pair of vertices $u, v \in V$. We define the ***residual capacity*** $c_f(u, v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\,, \\ f(v,u) & \text{if } (v,u) \in E\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{24.2}$$

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{24.4}$$

Lemma 24.1
Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ defined in equation (24.4) is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

$$\begin{aligned} &\sum_{v \in V} (f \uparrow f')(u,v) - \sum_{v \in V} (f \uparrow f')(v,u) \\ &\quad = \sum_{v \in V_f(u)} (f \uparrow f')(u,v) - \sum_{v \in V_e(u)} (f \uparrow f')(v,u) \\ &\quad = \sum_{v \in V_f(u)} (f(u,v) + f'(u,v) - f'(v,u)) - \sum_{v \in V_e(u)} (f(v,u) + f'(v,u) - f'(u,v)) \end{aligned}$$

smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is in } p\}\,.$$

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{24.7}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. ∎

Corollary 24.3
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (24.7), and suppose that f is augmented by f_p. Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. (This definition is similar to the definition of "cut" that we used for minimum spanning trees in Chapter 21, except that here we are cutting a directed graph rather than an undirected graph, and we insist that $s \in S$ and $t \in T$.) If f is a flow, then the ***net flow*** $f(S, T)$ across the cut (S, T) is defined to be

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)\,. \tag{24.8}$$

The ***capacity*** of the cut (S, T) is

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)\,. \tag{24.9}$$

A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

Lemma 24.4
Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right)\,.$$

Corollary 24.5
The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof Let (S, T) be any cut of G and let f be any flow. By Lemma 24.4 and the capacity constraint,

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\ &\le \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\le \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) . \end{aligned}$$

■

Theorem 24.6 (Max-flow min-cut theorem)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof (1) $\Rightarrow$ (2): Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary 24.3, the flow found by augmenting f by f_p, where f_p is given by equation (24.7), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) $\Rightarrow$ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$S = \{v \in V :$ there exists a path from s to v in $G_f\}$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f. Now consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S. If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, which again would place v in S. Of course, if neither (u, v) nor (v, u) belongs to E, then $f(u, v) = f(v, u) = 0$. We thus have

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S,T) . \end{aligned}$$

By Lemma 24.4, therefore, $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): By Corollary 24.5, $|f| \le c(S, T)$ for all cuts (S, T). The condition $|f| = c(S, T)$ thus implies that f is a maximum flow. ■

If f^* denotes a maximum flow in the transformed network, !

in lines 1–2, making the total running time of the FORD-FULKERSON algorithm $O(E\,|f^*|)$.

The term "minimal" is used as the mst may not be unique.
Let G be an undirected graph.

1. G is connected.
2. G is acyclic.
3. G has exactly $n - 1$ edges.

If any of the two statements hold, then the third statement holds.
To prove statement 3 from statements 1 and 2, consider the following:

- **Statement 1:** G has at least $n - 1$ edges.

  Proof. Basis: A graph with a single vertex clearly has at least 0 edges(Trivial).

  Inductive Hypothesis: A connected graph with k vertices have at least $k - 1$ edges for any positive integer k

  Induction Step: Now, we should show that the inductive hypothesis is true for k+1 vertices.

- **Statement 2:** G has at most $n - 1$ edges.

Minimum Separator Lemma

Assumption: All the edges in the graph have unique weights.
Statement:
Let $U \subseteq V$ and $W = V \setminus U$. Let (u, w) be the minimum cost edge that connects U and W, where $u \in U$ and $w \in W$. Then, the minimum cost edge between $u \in U$ and $w \in W$ will be part of every minimum spanning tree (MST).

Algorithm 1 Prim's Algorithm

Input: Graph G with vertices V, edges E, and weight function $w : E \to \mathbb{R}$
Output: Shortest path from a given vertex to all other vertices.
Procedure: PrimMST(G, w)
Let $n = |V|$
for $i = 1$ **to** n **do**
  $visited[i] = 0$
  $d[i] = \infty$
  $N[i] = \text{NIL}$
end for
$TV = \{1\}$ $TE = \phi$
$visited[1] = 1$
for all $(1, u) \in G.E$ **do**
  $d[u] = w[1, u]$
  $N[u] = 1$
end for
for $u = 2$ **to** n **do**
  Get $u \in V$ such that $d[u]$ is minimum and $visited[u] = 0$
  $TV = TV \cup \{u\}$
  $TE = TE \cup (u, N(u))$
  for all $(u, v) \in G.E$ such that $visited[v] = 0$ **do**
    if $d[v] > w[u, v]$ **then**
      $d[v] = w[u, v]$
      $N[v] = u$
    end if
  end for
end for

Data Structure	Decrease-Key	Extract-Min
Array	$O(1)$	$O(V)$
Binary Heap	$O(\log V)$	$O(\log V)$

1. We maintain three arrays:
   - *visited*: stores the vertices currently in the spanning tree.
   - d: stores the minimum edge weight from the current spanning tree to each vertex.
   - N: stores the other vertex of the minimum weight edge.

   Initializing these arrays takes $O(V)$ time.
2. Mark vertex 1 as visited and update the d and N arrays. This requires traversing all vertices adjacent to vertex 1. (decrease-key operation)
3. Extract the least weight vertex V-1 times (extract-min operation)
4. Iterate through all vertices adjacent to the chosen vertex and update the minimum edge weight connecting current spanning vertex to the vertex (decrease-key operation).

Adjacency list:
The first step takes $O(V)$ time. Checking all the edges of the graph takes O(E) time in adjacency list representation. So second and fourth steps together take $E \cdot O(\text{decrease-key})$ time. The third step takes $V \cdot O(\text{extract-min})$ time.
So the time complexity is:

$$\begin{aligned} &O(V) + E \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \\ &= E \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \end{aligned}$$

We can use a Binary heap to obtain the time complexity $O(V \log V + E \log V) = O((V+E) \log V)$.
We can use a Array to obtain the time complexity $O(V^2 + E) = O(V^2)$.
The optimal time complexity when using adjacency list representation is $O((V+E) \log V)$ which can be obtained by using a binary heap.

Adjacency matrix:
The first step takes $O(V)$ time. Checking all the edges of the graph takes $O(V^2)$ time in adjacency matrix representation. So second and fourth steps together take $V^2 \cdot O(\text{decrease-key})$ time. The third step takes $V \cdot O(\text{extract-min})$ time. So the time complexity is:

$$\begin{aligned} &O(V) + V^2 \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \\ &= V^2 \cdot O(\text{decrease-key}) + V \cdot O(\text{extract-min}) \end{aligned}$$

We can use a Binary heap to obtain the time complexity $O(V \log V + V^2 \log V) = O(V^2 \log V)$.
We can use a Array to obtain the time complexity $O(V^2 + V^2) = O(V^2)$.
The optimal time complexity when using adjacency matrix representation is $O(V^2)$ which can be obtained by using an array.

visited	1	1	1	1	0	0	0
distance	0	1	2	2	1	∞	∞
neighbour	NIL	1	1	3	4	NIL	NIL

1.1 Basic Implementation

Algorithm 1 Make-Set

Input: A graph G containing the nodes whose components set has to be made.
Procedure Make-Set(G)
  int n = $|G.V|$
  int num-components = 0
  Declare component : Integer Array of size n
  Declare size : Integer Array of size n
  Declare members : Array of size n containing empty Integer Arrays
for each vertex in G.V : **do**
    component[vertex] ← num-components
    member[num-components] ← member[num-components] U vertex
    size[num-components] ← 1
    num-components ← num-components + 1
end for=0

Algorithm 2 Kruskal's Algorithm

Input: Graph G with vertices V, edges E, and weight function $w : E \to \mathbb{Z}$
Output: Shortest path from a given vertex to all other vertices.
Procedure: KruskalMST(G, w)
Sort edges E based on edge weight
Let $n = |V|$
for $i = 1$ **to** n **do**
  $c[i] = i$
end for
$TE = \phi$
while $|TE| < (n - 1)$ **do**
  Let $(u, v) = E.next$ // gets the next unprocessed edge
  if $c[u] \neq c[v]$ **then**
    $TE = TE \cup \{(u, v)\}$
    for $j = 1$ **to** $|V|$ **do**
      if $C[j] = C[u]$ **then**
        $C[j] = C[v]$
      end if
    end for
  end if
end while

1. Sort the edges based on their weights. This operation takes $O(E \log E)$ time.

   $$E \log E \leq E \log V^2 = 2E \log V$$

   So time taken to sort edges = O(E log V)
2. Maintain an array c (component array) to keep track of the component each vertex is in. Initializing this array takes $O(V)$ time (make operation).
3. Iterate through the edges and check if the vertices belong to the same component (find operation). This operation is performed for all edges in the worst case.
4. When adding a vertex to the minimum spanning tree , combine the components (union operation). This operation is performed V times.

In this algorithm, we directly work on the list of edges, so there is no need to create an adjacency list or matrix representation. When find and union operations are implemented naively as in the algorithm, the time complexity of the find operation is $O(1)$ and the time complexity of the union operation is $O(V)$ in the worst case. So the overall time complexity is:

$$\begin{aligned} &O(E \log V) + O(V) + E \cdot O(\text{find}) + V \cdot O(\text{union}) \\ &= O(E \log V) + O(V) + O(E) + O(V^2) \\ &= O(E \log V + V^2) \end{aligned}$$

- The **make** function takes $O(1)$ time.
- The **find** function takes $O(1)$ time if we maintain the component of each element.
- The **union** function takes amortized $O(\log n)$ time. This is achieved with the help of small-to-large merging. When taking a union, the smaller set is always added to the larger set. We also need to maintain a size array.

Using this data structure, time complexity of Kruskal's algorithm is:

$$\begin{aligned} &O(E \log E) + O(V) + E \cdot O(\text{find}) + V \cdot O(\text{union}) \\ &= O(E \log E) + O(V) + O(E) + O(V \log V) \\ &= O(E \log E + V \log V) \\ &= O(E \log V^2 + V \log V) \\ &= O((2E + V) \log V) \\ &= O((E + V) \log V) \end{aligned}$$

Algorithm 2 Find-Set(x)

```
Procedure Find-Set(u)
    componentID ← component[vertex]
    return componentID =0
```

Algorithm 3 Union(S_x , S_y)

```
Procedure Union(S_x , S_y)
    componentID1 ← S_X
    componentID2 ← S_y
if  componentID1 == componentID2  then
      return
end if
if  size[componentID1] < size[componentID2]  then
    smallerComponentID := componentID1
    largerComponentID := componentID2
else
    smallerComponentID := componentID2
    largerComponentID := componentID1
end if
for element in members[smallerComponentID] do
    members[largerComponentID] ← {element} ∪ members[largerComponentID]
    component[element] ← largerComponentID
end for
size[largerComponentID] ← size[largerComponentID] + size[smallerComponentID]
size[smallerComponentID] ← 0
members[smallerComponentID] ← ∅ =0
```

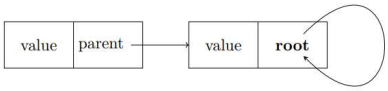
The Make-Set operation takes $O(n)$ time where n is the number of nodes in G which is due to the initialization of the component array and hash map. Find-Set operation runs in $O(1)$ time returning the corresponding component index.

The Union operation involves the merging of the smaller component with the bigger component. Let the size of the components be k_1 and k_2 where $k_1 > k_2$. Since in this operation we have to iterate over the complete smaller component therefore the time complexity of this operation will be $O(k_2)$.

Also in the worst case scenario the size of k_2 can be $\lfloor \frac{n}{2} \rfloor$, hence in the worst case the time complexity of the union function becomes $O(n/2)$ which simplifies to $O(n)$.

Therefore amortised time complexity of single union operation becomes $O(\log n)$.

1.3 Dual Implementation



node:	1	2	3	4	5	6	7	8	9
size:	1	4	1	1	1	5	1	1	1
parent:	2	2	2	2	6	6	6	7	7

Algorithm 4 Make-Set

```
1. Procedure Make-Set(x):
2.         parent[x] ← x
3.         size[x] ← 1
4. end Procedure
```

Algorithm 5 Find-Set

```
1. Procedure Find-Set(x):
2.         if x ≠ parent[x] then
3.                 parent[x] ← Find-Set(parent[x])
4.         end if
5.         return parent[x]
6. end Procedure
```

Algorithm 6 Union

```
1. Procedure Union(x, y):
2.         x_root ← Find-Set(x)
3.         y_root ← Find-Set(y)
4.         if x_root ≠ y_root then
5.                 if size[x_root] > size[y_root] then
6.                         parent[y_root] ← x_root
7.                         size[x_root] ← size[x_root] + size[y_root]
8.                 else
9.                         parent[x_root] ← y_root
10.                        size[y_root] ← size[y_root] + size[x_root]
11.                end if
12.        end if
13. end Procedure
```

Find-Set(x) takes $O(\log n)$ amortized time which is proportional to the height of the tree, if small-to-large merging is followed. This is because, during merging, we are increasing the degree of branching at the root of the larger component, which facilitates the reduction in the height of the tree.
The **Union(x, y)** operation is hence also of $O(\log n)$ time complexity, while the **Make-Set(x)** operation is of $O(1)$ time complexity.

The space complexity is $O(n)$ due to the extra memory used for the storage of the *size* array and the *parent* values at each node.

However, the time complexity of **Find-Set(x)** can further be improved to $O(\alpha(n))$ where $\alpha(n)$ represents the **inverse Ackermann function,** which grows incredibly slowly. In fact, it grows so slowly that its value does not exceed 4 for all reasonable n (approximately $n < 10^{600}$). This can be achieved using union by size, coupled with **path compression** in the recursive traversal towards the root of a component.