Algorithms 2

Section 2: Graphs and Subgraphs

Flow Networks [1]

Flow networks concern weighted, directed graphs $G = (V, E)$.

V contains two distinguished vertices, s and t, known as the source and the sink of the flow.

We require two properties of E:

1. No self loops at any vertex: $\forall v \in V . (v, v) \notin E$
2. No antiparallel edges: $\forall u,v \in V . (u, v) \in E \rightarrow (v, u) \notin E$

The weights, known as **capacities**, are non-negative: $\forall (u, v) \in E . c(u, v) \geq 0$ and it will be convenient to define $c(u, v) = 0$ if $(u, v) \notin E$.

Flow Networks [2]

All vertices are on some path $s \rightsquigarrow v \rightsquigarrow t$ so $|E| \geq |V| - 1$ (every vertex other than s must have at least one inbound edge).

Put another way, we can delete any vertex v (and its incident edges) if v is not reachable from s or t is not reachable from v. For the problems we want to solve, such vertices never alter the solution.

Definition of a flow

A **flow** f(u, v) in G is a function of type $V \times V \to \mathbb{R}$ with two properties:

1. Flows are subject to the **capacity constraint**: $\forall u,v \in V \,.\, 0 \leq f(u, v) \leq c(u, v)$
2. At every vertex $u \in V \setminus \{s, t\}$, we have **flow conservation**:

   $$\Sigma_{v \in V} f(v, u) = \Sigma_{v \in V} f(u, v)$$

where f(u, v) = 0 if $(u, v) \notin E$.

The flow is defined between all pairs of vertices in G and is known as the flow from u to v.

Value of a flow

We denote the value of a flow f as |f| where

$$|f| = \Sigma_{v \in V} f(s, v) - \Sigma_{v \in V} f(v, s)$$

⚠️ |..| is not absolute value or set cardinality!

The second term is usually zero because there is no flow into the source but, as we will see, we want to generalise the networks to which we apply this idea and the edges into the source will not always have zero weight.

Maximum Flow Problem

Input: a flow network, i.e. a directed graph $G = (V, E)$ with edge capacities $c(u, v) \geq 0$, and two distinguished vertices $s,t \in V$ being the source and sink.

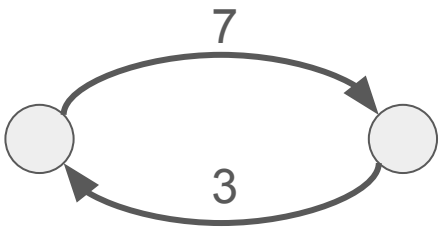
Output: any flow having maximum value.

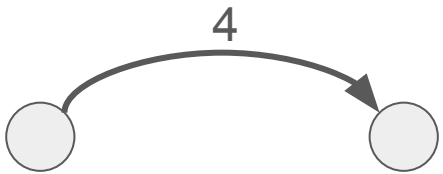
Note that we seek to determine the flow, not just the flow value.

Antiparallel edges [1]

We said that we do not allow having both (u, v) and (v, u) to be edges in E. Several algorithms that solve the Maximum Flow Problem require this.

We cannot simplify antiparallel edges to a single edge with the net capacity because we might want to use only the capacity in the smaller magnitude direction, or all the capacity in the larger direction.





Antiparallel edges [2]

We can handle antiparallel edges by introducing additional vertices to split one of the edges. Two new edges are assigned the same capacity as the original they replace.

This means we can require no antiparallel edges without limiting the set of problems our algorithms can solve.

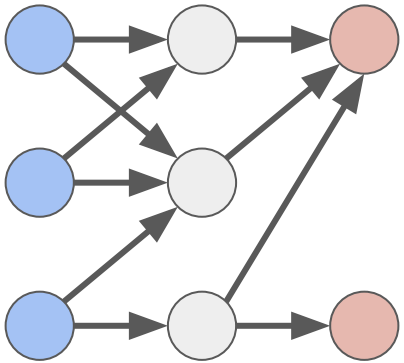
Supersources and Supersinks [1]

If we want to model a system where flow originates from multiple sources (s_1 .. s_m) and is consumed by multiple sinks (t_1 .. t_n), we can add additional vertices and edges:

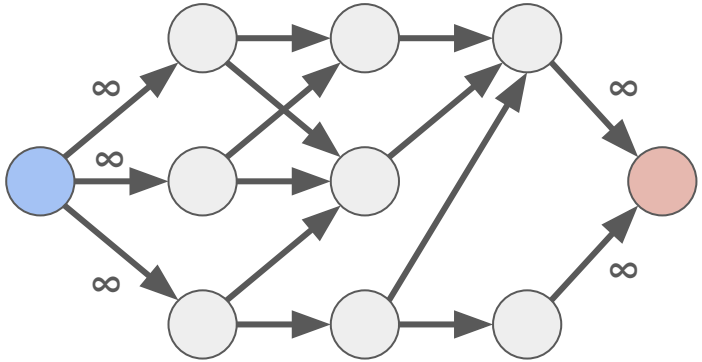
- two additional vertices for the supersource s and supersink t
- edges (s, s_i) for i = 1 .. m and (t_j, t) for j = 1 .. n, all with capacity c = ∞

This reduces the multiple source, multiple sink problem to the single source, single sink problem. We lose no generality by only considering solutions to the single source, single sink problem.

Supersources and Supersinks [2]



3 sources, 2 sinks



1 source, 1 sink

Ford-Fulkerson Methods

Ford and Fulkerson covers several algorithms based on a few key ideas, which we can also use to solve related problems:

1. Residual networks
2. Augmenting paths
3. Cuts

Residual Networks

Given a flow network G = (V, E) and a flow f, the **residual network** G_f contains **residual edges** showing how we can change the flow:

1. If an edge $(u, v) \in E$ and $f(u, v) < c(u, v)$ then we can add more flow to the edge: up to $c(u, v) - f(u, v)$ more. NB: there is no edge if $f(u, v) = c(u, v)$!!

2. If $f(u, v) > 0$ then we can **cancel** flow that is already present by adding flow in the reverse direction: up to $f(u, v)$ along edge (v, u) [note reverse direction]

Residual Capacity

Given a flow network G = (V, E) and a flow f, the residual edges (u, v) in the residual network G_f have **residual capacities** $c_f(u, v)$ where

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

Exactly one case applies because of the antiparallel edge constraint on E.

Augmentation

Any flow f’ in the residual network G_f can be added to the flow f to make a valid flow because the flow assigned to every edge cannot exceed its capacity and cannot become negative. This is **augmentation**, written as f □ f’.

(f □ f’) (u, v) = f(u, v) + f’(u, v) - f’(v, u) if (u, v) ∈ E

(f □ f’) (u, v) = 0 otherwise

The value of the augmented flow, |f □ f’| = |f| + |f’|.

Augmenting Paths

Given a flow network G and a flow f, an **augmenting path** is a simple path p from s to t in the residual network.

The maximum amount by which we can increase the flow along each edge in p is called the **residual capacity** of the path p:

$c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ is on } p\}$

Notice that if we augment flow f with the residual capacities along each edge (u, v) on p, then we get a flow with strictly larger value: $|f \uparrow f_p| = |f| + |f_p| > |f|$.

FORD-FULKERSON(G, s, t)

```
1   Initialise flow f to 0 on all edges
2   while there exists an augmenting path p in
                          the residual network G_f
3       augment the flow f along p
4   return f
```

Why do we get a maximum flow?

We saw that Ford-Fulkerson augments a flow using augmenting paths until no more augmenting paths can be found.

The question to be answered is whether it is guaranteed that Ford-Fulkerson terminates only when a maximum flow has been found.

The Max-Flow Min-Cut Theorem tells us that this technique will work.

Cuts [1]

A **cut** (S, T) of a 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.

For a flow f, we define the **net flow** f(S, T) across the cut (S, T) as

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

Given a flow network G with source s and sink t, and a flow f, let (S, T) be any cut of G. The net flow across (S, T) is f(S, T) = |f|. (The proof follows from the definition of flow conservation.)

Cuts [2]

The **capacity** of the cut (S, T) is $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$.

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

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

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; and
3. |f| = c(S, T) for some cut (S, T) of G.

Proof of Max-Flow Min-Cut Theorem [1]

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

Proof that 1 => 2:

Suppose f is a maximum flow in G but G_f contains an augmenting path p. The flow found by augmenting f using p has value $|f| + |f_p| > |f|$, which contradicts that f was maximum.

Note that $|f_p| > 0$ because we did not add edges to G_f with zero capacity.

Proof of Max-Flow Min-Cut Theorem [2]

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

Proof that 3 => 1:

Remember that the value of any flow f in a flow network G is bounded from above by the capacity of any cut of G, i.e. $|f| \leq c(S, T)$.

If $|f| = c(S, T)$ then f must be a maximum flow.

Proof of Max-Flow Min-Cut Theorem [3]

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

Proof that 2 => 3:

Suppose G_f has no augmenting paths (so no paths from s to t). Consider the partition (S, T) where $S = \{v \in V \mid \exists \text{ path from s to v in } G_f\}$, and $T = V \setminus S$.

(S, T) is a cut because $s \in S$ and $t \in T$.

Consider $u \in S$ and $v \in T$. Is $(u, v) \in E$, or is $(v, u) \in E$, or is neither in E?

Proof of Max-Flow Min-Cut Theorem [4]

If $(u, v) \in E$ then we must have $f(u, v) = c(u, v)$ since, otherwise, there would be residual capacity on the edge and (u, v) would be in E_f. That would place $v \in S$.

If $(v, u) \in E$ then we must have $f(v, u) = 0$ because, otherwise, $c_f(u, v) = f(v, u) > 0$ and we would have $(u, v) \in E_f$. That also places $v \in S$.

If neither is in E then $f(u, v) = f(v, u) = 0$.

$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) = \sum_{u \in S} \sum_{v \in T} c(u, v) - 0 = c(S, T)$

We know that $|f| = f(S, T) = c(S, T)$, which proves statement 3.

Proof of Max-Flow Min-Cut Theorem [5]

We have proven that 1 => 2 and that 2 => 3 and that 3 => 1, which suffices to show the equivalence of all three statements in the Max-Flow Min-Cut Theorem.

Basic FORD-FULKERSON(G, s, t)

```
1   for (u, v) in G.E    (u, v).f = 0
2   while there exists a path p from s to t in G_f
3       c_f(p) = min{c_f(u, v) | (u, v) is in p}
4       for (u, v) in p
5           if (u, v) ∈ G.E
6               (u, v).f = (u, v).f + c_f(p)
7           else
8               (v, u).f = (v, u).f - c_f(p)
```

Termination and Analysis

Interestingly, Ford-Fulkerson can fail to terminate if the edge capacities are irrational numbers: augmenting paths can add tiny amounts of additional flow in a series that is not convergent.

If all the capacities are integers, this cannot occur. We can find augmenting paths using breadth-first search or depth first search, costing $O(|E_f|)$ each time, which is $O(|E|)$ each time.

With integral capacities, the flow must increase by at least 1 each iteration so the cost of FORD-FULKERSON is $O(|E|\ |f^*|)$, where f^* is the maximum flow.

Optimisation: EDMUNDS-KARP(G, s, t)

We find *shortest* augmenting paths using breadth-first search on the residual network but with edge weights all set to 1.

It can be shown that this algorithm has $O(|V| \, |E|^2)$ running time.

(Proof is in CLRS3, pp 728–729.)

Maximum Bipartite Matchings

Given an undirected graph G = (V, E), a **matching** M ⊆ E contains at most one edge that is incident at each vertex v ∈ V.

A vertex v ∈ V is **matched** if some edge in M is incident on v.
Other vertices are **unmatched**.

A **maximum matching** is a matching of maximum cardinality.

We are most interested in finding matchings within *bipartite graphs*.

Bipartite Graphs

An undirected bipartite graph $G = (V, E)$ where $V = V_1 \cup V_2$ and $E \subseteq V_1 \times V_2$.

We also assume that every vertex has at least one incident edge.

Example: your college's undergraduate rooms ballot can be modelled as a bipartite graph where the vertex set is $\{u \in \text{undergraduates}\} \cup \{r \in \text{rooms}\}$, and the edges represent the students' choices for which rooms they might like to live in next year.

Maximum Bipartite Matching Problem

Input: an undirected bipartite graph G = (V, E) where $V = V_1 \cup V_2$ and $E \subseteq V_1 \times V_2$.

Output: a matching $M \subseteq E$ with maximum cardinality.

We can use Ford-Fulkerson to find a maximum matching by transforming G into a flow network. Set the weight of every edge to 1 and add vertices {s, t} with edges {s, u} for $u \in V_1$ and {v, t} for $v \in V_2$ (these edges have infinite capacity). The solution requires O(|V||E|) time but the proof of correctness requires the Integrality Theorem.

It's worth converting the algorithm rather than the data…

Augmenting Paths in Unweighted Bipartite Graphs

An **Augmenting Path** with respect to a matching M in a bipartite graph G is an alternating path that starts at an unmatched vertex in V_1 and ends at unmatched vertex in V_2.

An **Alternating Path** with respect to a matching M in a bipartite graph G is a sequence of edges that are all in the graph's edge set and are alternately in the matching, not-in, in, not-in, …

Maximum Matchings in Unweighted Bipartite Graphs

MAXIMUM-MATCHING(G):

```
1   let M = ∅
2   do
3       let a = FIND-AUGMENTING-PATH(G, M)
4       M = M ⊕ a
5   while (a != NULL)
6   return M
```

Proof: ∃ augmenting path until M is maximum [1]

Let M’ be a maximum matching. M is the matching we have at the moment.

Consider the symmetric difference of M’ and M: every edge that is in M’ or M, but not in both. You can think of this as the XOR of the edge sets.

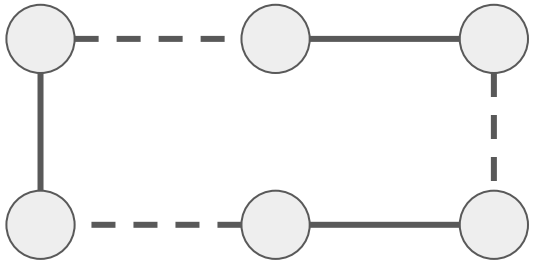
Notice that we cannot have two edges from M’ meeting at a vertex, nor two from M, since M’ and M are both matchings.

What structures can you find in the symmetric difference?

Proof: ∃ augmenting path until M is maximum [2]

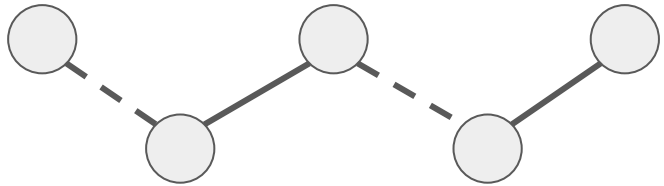
We can find isolated vertices.

We can find closed loops. Any closed loops can have any even length because, otherwise, we would have two edges from the same matching sharing a vertex.

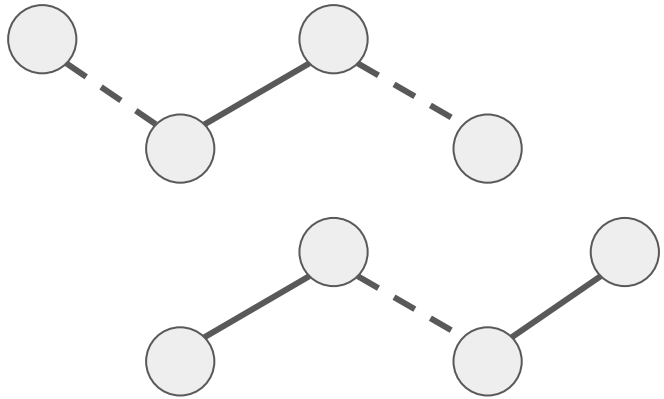


Proof: ∃ augmenting path until M is maximum [3]

We can have chains of even length.



We can have chains of odd length, in two ways: one more edge from M’ or one more edge from M.



Proof: ∃ augmenting path until M is maximum [4]

There are no other options because the maximum degree of any vertex is two: if we had three or more incident edges at any vertex, at least two would need to come from M or M', which is impossible because both are matchings.

We know that |M'| > |M| if M is not maximum, and in that case there must be at least one more edge from M' than from M in the symmetric difference.

The loops and even chains use the same number of edges from M' and M so there must be at least one odd-length chain with one more edge from M' than M.

That odd-length chain is an augmenting path with respect to M for G!

Finding Augmenting Paths

A simple method is to run a variant of BFS or DFS starting from each unmatched vertex in whichever of V_1 and V_2 has fewer unmatched vertices.

The search is constrained to taking edges $(u, v) \notin M$ for its first step.

If v is unmatched then we have an augmenting path, otherwise, we follow the edge $(v, w) \in M$ and allow the search to explore edges $(w, x) \notin M$ as the next step.

Repeat until either an augmenting path is found or the search gets stuck with no further edges to follow. If so, start a new search from the next unmatched starting vertex. If no search finds an augmenting path, there is none to be found.

Cost of Finding Augmenting Paths

The algorithm can find at most |V|/2 augmenting paths.

Each search costs O(|V| + |E|) = O(|E|) here (because the graph is connected).

Total cost is O(|V||E|).

We can do better…

HOPCROFT-KARP(G)

```
1   let M = ∅
2   do
3       a[] = ALL-VERTEX-DISJOINT-SHORTEST-AUGMENTING-PATHS(G, M)
4       M = M ⊕ a_1 ⊕ a_2 ⊕ … ⊕ a_a.length
5   while (a.length != 0)
6   return M
```

ALL-VERTEX-DISJOINT-SHORTEST-AUGMENTING-PATHS(G, M)

These are minimum length augmenting paths for M, with no common vertices.

Find them using a combination of a depth-first and breadth-first search, marking nodes on augmenting paths as they are found (to avoid finding multiple augmenting paths using the same vertices).

It can be shown that the WHILE loop requires only $\sqrt{|V|}$ iterations (by considering the maximum number of augmenting paths that can be found in each iteration).

HOPCROFT-KARP(G) requires $O(|E| \sqrt{|V|})$ running time!

Beware: Maximum and Maximal Matchings

A *maximum* matching is what we have been finding: a largest cardinality subset of non-adjacent edges in an input graph.

A *maximal* matching is a subset of non-adjacent edges that cannot be extended.

These are not the same!

Example: by making poor choices about the edges to include, it might be impossible to add more non-adjacent edges, but if you removed some and added different edges, it might be possible to get more in total.

Minimum Spanning Trees (MSTs)

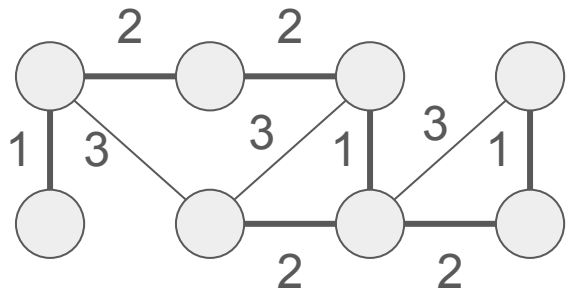
Input: a connected, undirected graph G = (V, E) with weight function w: E→ℝ.

Output: an acyclic subset T ⊆ E that connects all the vertices and whose total weight w(t) is minimal, where $w(t) = \sum_{(u,v) \in T} w(u,v)$.

Because T *does* connect all the vertices and T *is acyclic*, it must be that the edges in T form a tree. Any such tree is a **spanning tree**. T is not (necessarily) rooted.

A **minimum spanning tree** is a spanning tree with minimum total edge weights, and need not be unique.

Minimum Spanning Tree Example



Computing Minimum Spanning Trees

We will see two iterative, greedy algorithms that exploit *safe edges*.

Both algorithms iteratively increase a set (w.r.t. subset inclusion) A of edges, maintaining the property that $A \subseteq T$, for *some* T that is a minimum spanning tree.

As the algorithms run, edges $(u, v) \in E$ are added to A, always preserving the property that $A \subseteq T$, for *some* T that is a minimum spanning tree. A **safe edge** is one that can be added without violating the property.

Iteration continues until there are no more safe edges, at which point, A = T.

Cut, Cross, Respect, and Light

A **cut** $(S, V \setminus S)$ of an undirected graph $G = (V, E)$ is a partition of V.

An edge $(u,v) \in E$ **crosses** the cut $(S, V \setminus S)$ if $u \in S$ and $v \in V \setminus S$.

A cut **respects** a set A of edges if no edge in A crosses the cut.

An edge crossing a cut is a **light edge** if its weight is minimum of any edge crossing the cut. The minimum weight crossing the cut is unique but light edges are not necessarily unique: multiple crossing edges might have the same weight.

Safe Edge Theorem

Let G = (V, E) be a connected, undirected graph with real-values weight function w defined on E.

Let A be a subset of E that is included in some minimum spanning tree for G.

Let (S, V \ S) be *any* cut of G that respects A.

Let (u, v) $\in$ E be a light edge crossing (S, V \ S).

$\Rightarrow$ (u, v) is a safe edge for A.

Proof of the Safe Edge Theorem [1]

Let T be a minimum spanning tree that includes A.

If T contains (u, v) then we are done.

If T does not contain (u, v), we can show that another minimum spanning tree T' exists and includes $A \cup \{(u, v)\}$. This makes (u, v) a safe edge for A.

Proof of the Safe Edge Theorem [2]

Add (u, v) to T and note that this forms a cycle (since T is a spanning tree and must already contain some unique path p = u $\rightsquigarrow$ v).

(u, v) crosses the cut (S, V \ S), and there must be at least one edge in T, on the path p, that also crosses the cut (since T is connected).

Let (x, y) be such an edge. (x, y) is not in A because the cut respects A.

Remove (x, y) from T and add (u, v) instead: call this T'. T' must be connected and acyclic (a tree).

Proof of the Safe Edge Theorem [3]

Calculate a bound on the weight of edges in T':

$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$

The final inequality is because (u, v) is a light edge crossing $(S, V \setminus S)$, i.e. for any other edge (x, y) crossing the cut, $w(u, v) \leq w(x, y)$.

Since T was a minimum spanning tree, T' must also be a minimum spanning tree.

So why is (u, v) a safe edge for A? That's because $A \subseteq T'$ since $A \subseteq T$ and the removed edge $(x, y) \notin A$, so $A \cup \{(u, v)\} \subseteq T'$. Because T' is an MST, (u, v) is a safe edge for A.

Corollary

The G = (V, E) be a connected, undirected graph with real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let C = (V_C, E_C) be a connected component (tree) in the forest G_A = (V, A). If (u, v) is a light edge connecting C to some other component in G_A then (u, v) is a safe edge for A.

Kruskal’s Algorithm

Kruskal’s algorithm finds safe edges to add to a growing forest of trees by finding least-weight edges that connect any two trees in the forest.

The corollary tells us that that any such edge must be a safe edge (for either tree) because it is the lightest edge crossing the cut that separates that tree from the rest of the graph.

The algorithm resembles that used to find connected components.

MST-KRUSKAL(G, w)

```
1   A = ∅
2   S = new DisjointSet; for v in G.V   MAKE-SET(S, v)
3   MERGE-SORT(G.E)          // Or any other non-decreasing sort
4   for (u, v) in G.E        // Can also stop if |A|=|V|-1
5       if !IN-SAME-SET(S, u, v)
6           A = A ∪ {(u, v)}
7           UNION(S, u, v)
8   return A
```

Analysis of MST-KRUSKAL(G, w)

Creating a disjoint set with |V| separate sets costs Θ(|V|).

In the worst case, the FOR loop runs to completion: |E| iterations performing 1 IN-SAME-SET check each, and |V|-1 calls to UNION across all the iterations.

The total cost is ~O(|E| + |V|) since both disjoint-set representation operations cost ~O(1).

The sort costs O(|E| lg |E|), which is O(|E| lg |V|) since G is connected.

The cost of sorting dominates and we state that MST-KRUSKAL costs O(|E| lg |V|).

Prim's Algorithm

Prim's algorithm maintains that A is a single tree (not a forest), and adds safe edges between the tree and an isolated vertex, to increase the size of the tree until |A| = |V|. Prim's algorithm starts from an arbitrary vertex $r \in V$.

The corollary tells us that that any such edge must be a safe edge because they are the lightest edges crossing the cut that separates the tree from the rest of the graph.

The algorithm resembles Dijkstra's algorithm, used to find single-source shortest paths.

MST-PRIM(G, w, r)

```
1   Q = new PriorityQueue
2   for v in G.V    v.key = ∞; v.π = NIL; PQ-ENQUEUE(Q, v)
3   PQ-DECREASE-KEY(Q, r, 0)
4   while !PQ-IS-EMPTY(Q)
5       u = PQ-EXTRACT-MIN(Q)
6       for v in G.E.adj[u]
7           if v ∈ Q && w(u, v) < v. key
8               v.π=u; v.key=w(u,v); PQ-DECREASE-KEY(Q,v,v.key)
```

Analysis of MST-PRIM(G, w, r) [1]

If we use a Fibonacci Heap as the implementation of the Priority Queue ADT then the |V| calls to PQ-ENQUEUE cost O(1) amortised each (FH-INSERT).

The WHILE loop executes |V| times and each call to FH-EXTRACT-MIN costs O(lg V) time so the total time is O(|V| lg |V|). (We should sum the costs as the size of the PQ decreases but this over-approximation turns out to be asymptotically accurate.)

Analysis of MST-PRIM(G, w, r) [2]

Across all iterations of the WHILE loop, the FOR loop covers every edge exactly twice (once in each direction).

We need to test for membership of the Priority Queue, which is not a supported operation in the ADT. We can implement this with a bit string: one bit per vertex, initialise to 11..1 and set bits to zero when extracted; test membership looks at the corresponding bit. This test for membership becomes O(1) time, and the updates do not add to the corresponding big-O costs because they are O(1) per vertex.

The calls PQ-DECREASE-KEY, cost O(1) amortised for the Fibonacci Heap implementation.

Analysis of MST-PRIM(G, w, r) [3]

The total cost of MST-PRIM(G, w, r) is O(|E| + |V| lg |V|), which is better than MST-KRUSKAL.

Either term could be dominant, depending on the size of the edge set.

If we used a binary heap, MST-PRIM(G, w, r) would cost O(|E| lg |V| + |V| lg |V|), which is O(|E| lg |V|) if G is connected so |E| > |V|-1.