Algorithms 10

CS201

Kaustuv Nag

Disjoint-set Data Structure

- A disjoint-set data structure maintains a collection $\mathscr{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is represented by a some representative member of the set.
- Letting x denote an object, it supports the following operations:

MAKE-SET(x)	creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, x must not already be in some other set.
UNION(x, y)	unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y, removing them from the collection $\mathscr{S}$. In practice, we often absorb the elements of one of the sets into the other set.
FIND-SET(x)	returns a pointer to the representative of the (unique) set containing x.

Disjoint-set Data Structure

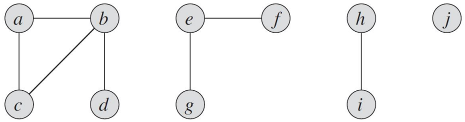
Determining the Connected Components of an Undirected Graph

```
CONNECTED-COMPONENTS$(G)$
1  for each vertex $v \in G.V$
2      MAKE-SET$(v)$
3  for each edge $(u, v) \in G.E$
4      if FIND-SET$(u) \neq$ FIND-SET$(v)$
5          UNION$(u, v)$
```

```
SAME-COMPONENT$(u, v)$
1  if FIND-SET$(u)$ == FIND-SET$(v)$
2      return TRUE
3  else return FALSE
```

Disjoint-set Data Structure

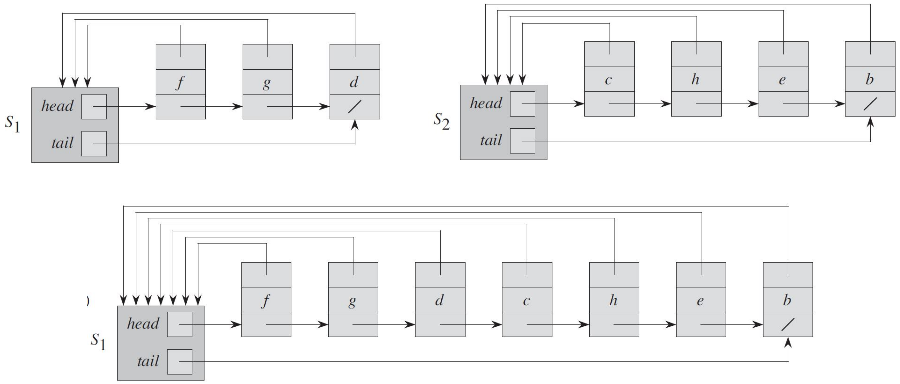
Example



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Disjoint-set Data Structure

Implementation

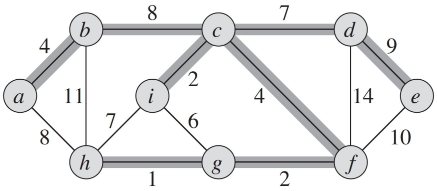


Minimum Spanning Tree

Let there be an undirected graph $G = (V, E)$, where V is the set of vertices, E is the set of edges, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v. We find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it “spans” the graph G. We call the problem of determining the tree T the minimum-spanning-tree problem.



Growing a Minimum Spanning Tree

- We use a greedy strategy.
- We manage a set of edges A, maintaining the following loop invariant:
  Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, we determine an edge (u, v) that we can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.
- We call such an edge a *safe edge* for A, since we can add it safely to A while maintaining the invariant.

```
GENERIC-MST($G, w$)
1  $A = \emptyset$
2  while $A$ does not form a spanning tree
3      find an edge $(u, v)$ that is safe for $A$
4      $A = A \cup \{(u, v)\}$
5  return $A$
```

Use of Loop Invarinat

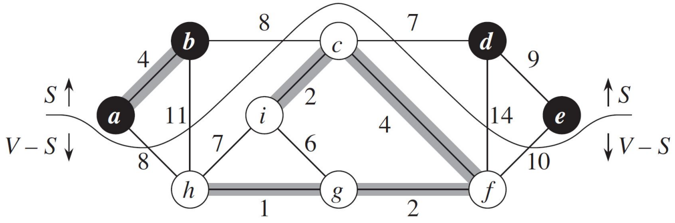
Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

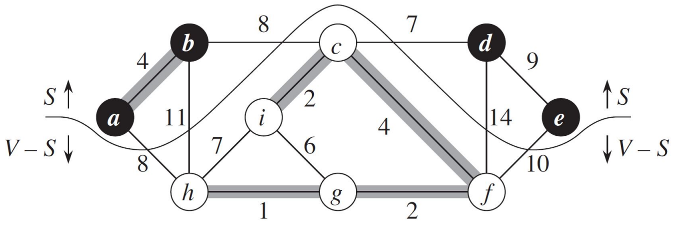
Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

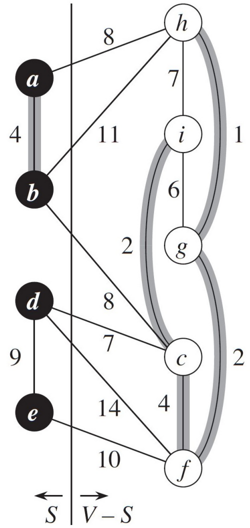
A Cut of an Undirected Graph

- A *cut* $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.
- An edge $(u, v) \in E$ *crosses* the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$.
- A cut *respects* a set A of edges if no edge in A crosses the cut.
- An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut.



A Cut of an Undirected Graph

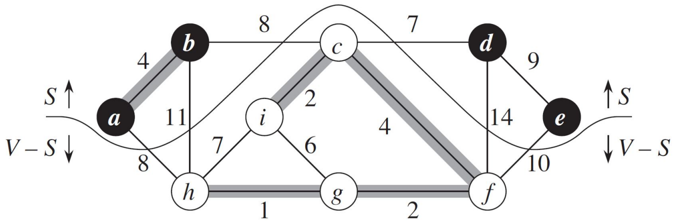




A Cut of an Undirected Graph

Theorem

Let $G = (V, E)$ be a connected, undirected graph with a 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, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.



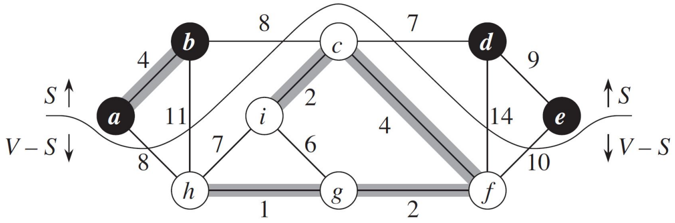
A Cut of an Undirected Graph

Corollary

Let $G = (V, E)$ be a connected, undirected graph with a 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 safe for A.

Proof

The cut $(V_C, V - V_C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A.



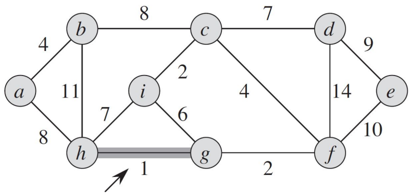
The algorithms of Kruskal and Prim

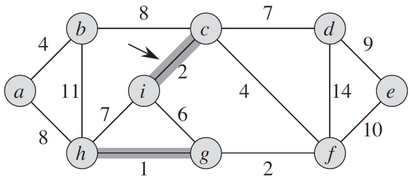
- In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.
- In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

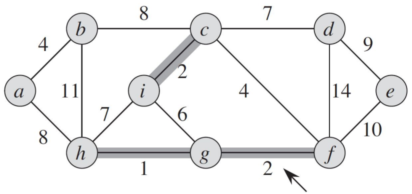
Kruskal’s Algorithm

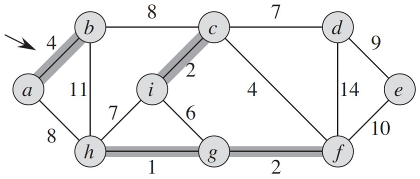
```
MST-KRUSKAL($G, w$)
$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET($v$)
sort the edges of $G.E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET($u$) $\neq$ FIND-SET($v$)
        $A = A \cup \{(u, v)\}$
        UNION($u, v$)
return $A$
```

Kruskal’s Algorithm

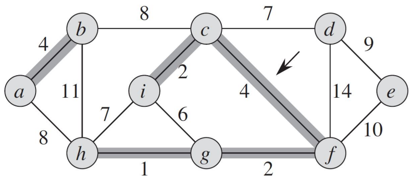


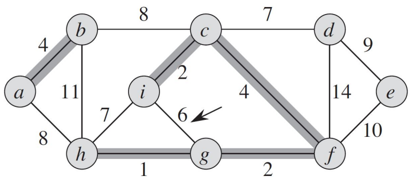


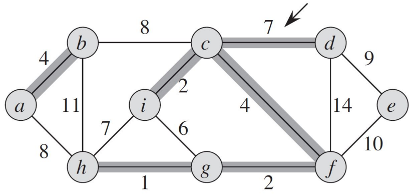


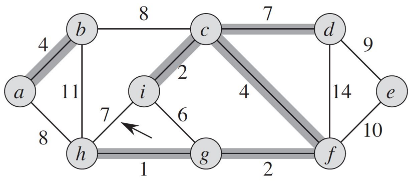


Kruskal’s Algorithm

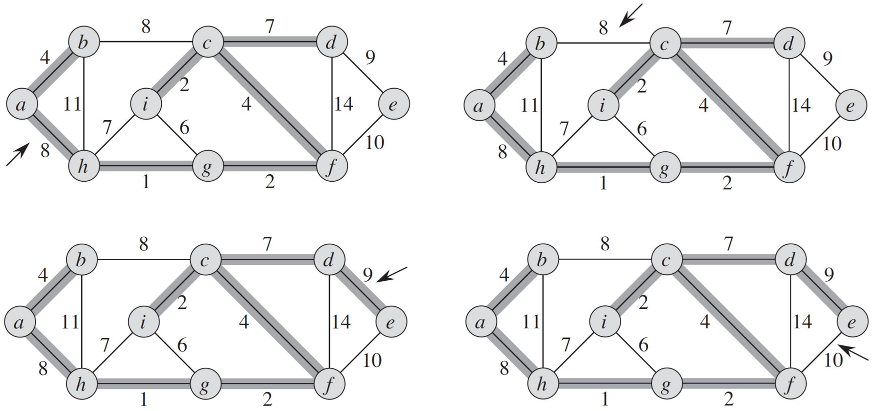




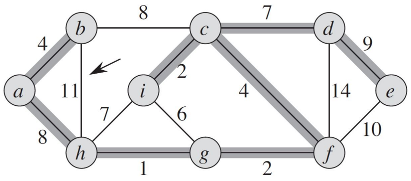


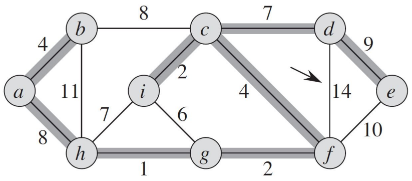


Kruskal’s Algorithm



Kruskal's Algorithm





Kruskal's Algorithm

Runtime

- The running time of Kruskal's algorithm depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation with the *union-by-rank and path-compression heuristics*, which is the asymptotically fastest implementation known.
- Initializing the set A in line 1 takes $O(1)$ time, and the time to sort the edges in line 4 is $O(E \lg E)$.
- The **for** loop of lines 5–8 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these take a total of $O((V+E)\alpha(V))$ time, where $\alpha(V)$ is a very slowly growing function. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time.
- Since $\alpha(V) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

Prim’s Algorithm

- Prim’s algorithm has the property that the edges in the set A always form a single tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.
- Each step adds to the tree A a light edge that connects A to an isolated vertex—one on which no edge of A is incident.
- By the last Corollary that we learnt, this rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree.
- This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight.

Prim's Algorithm

- We assume that each vertex $v \in V$ has two attributes: $v.key$ and $v.\pi$.
- The attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $v.key = \infty$ if there is no such edge.
- During execution of the algorithm, each vertex v that is not in the tree resides in a min-priority queue Q based on $v.key$.
- The attribute $v.\pi$ names the parent of v in the tree.
- The algorithm implicitly maintains the set A from GENERIC-MST as

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

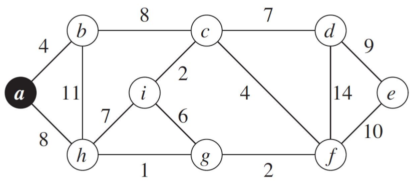
- When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus

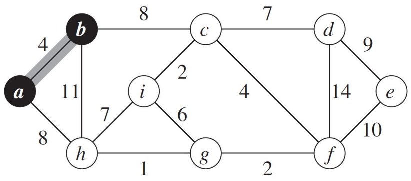
$$A = \{(v, v.\pi) : v \in V - \{r\}\}$$

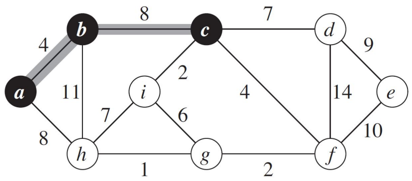
Prim's Algorithm

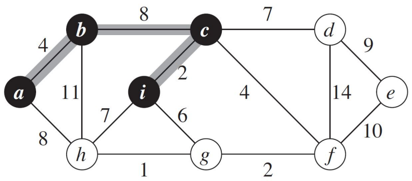
```
MST-PRIM$(G, w, r)$
1  for each $u \in G.V$
2      $u.key = \infty$
3      $u.\pi = \text{NIL}$
4  $r.key = 0$
5  $Q = G.V$
6  while $Q \neq \emptyset$
7      $u = \text{EXTRACT-MIN}(Q)$
8      for each $v \in G.Adj[u]$
9          if $v \in Q$ and $w(u, v) < v.key$
10             $v.\pi = u$
11             $v.key = w(u, v)$
```

Prim’s Algorithm

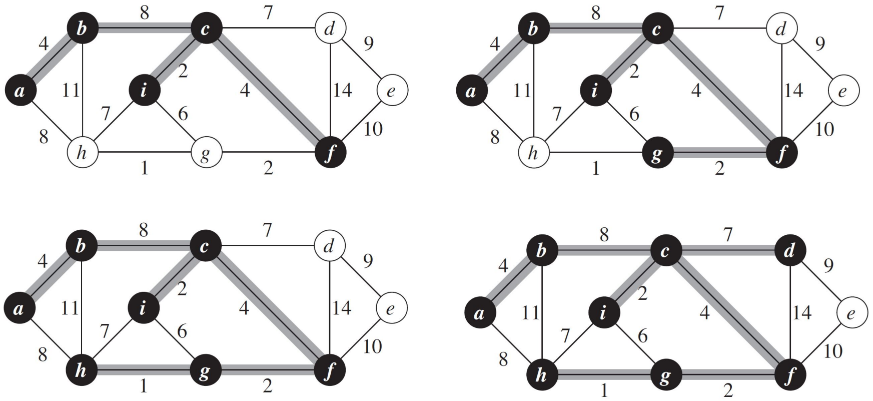




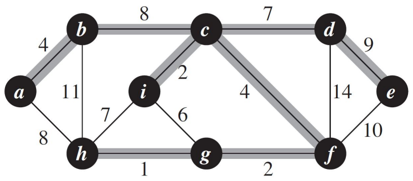




Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

Runtime

- The running time of Prim's algorithm depends on how we implement the min-priority queue Q. If we implement Q as a binary min-heap, we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in $O(V)$ time. The body of the while loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$.
- The for loop in lines 8–11 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the for loop, we can implement the test for membership in Q in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in $O(\lg V)$ time.
- Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.
- We can improve the asymptotic running time to $O(V \lg V)$ by using Fibonacci heaps.

White Board

White Board