```
def find_cycle(G):
    #make set of all!
    #for each edge, call find_set(v), find_set(u)
    # if find_set(v) < findset(u), has cycle
    # else union(u, v)
```

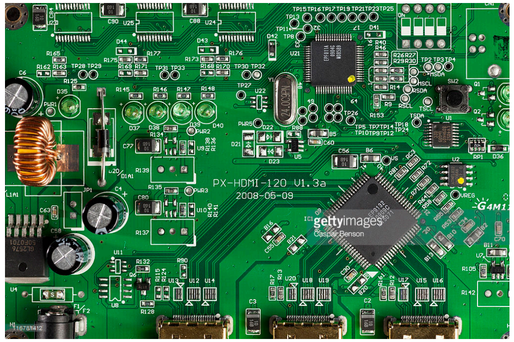
```
def find_cycle(G):
    vetices =
    for i in range(0, vertex):
        make_set(i)
    for i in range(0, vertex):
        for j in range (0, vertex):
            if a[i, j] =
```

Minimum Spanning Tree Overview

A Tree Spanning All Vertices Of A Given Connected Graph With Minimum Total Weight

Minimum Spanning Tree (MST) Overview

- A motivating example: To interconnect a set of n pins in an electronic circuit.



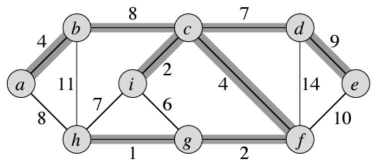
Minimum Spanning Tree (MST) Overview

- A motivating example: To interconnect a set of n pins in an electronic circuit
- Given a connected, undirected graph $G = (V, E)$ and a weight $w(u, v)$ for each edge $(u, v) \in E$, specifying the cost to connect (or traverse) u and v,
- We want to find an acyclic subset of edges $T \subseteq E$ that connects all vertices in V and whose total weight $w(T)$ is minimized:

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

- Since T is acyclic and connects all vertices, it must form a tree, which we call a ***spanning tree***, and a ***minimum***(-weight) ***spanning tree***.

MST Example (CLRS Fig. 23.1)



- Total weight (minimum) is 37. Shaded edges are tree edges.
- The MST is not necessarily unique
  - Remove (b, c) and add (a, h).
  - You still get a spanning tree, and its edge-weight total is also 37.

Strategy To Find an MST

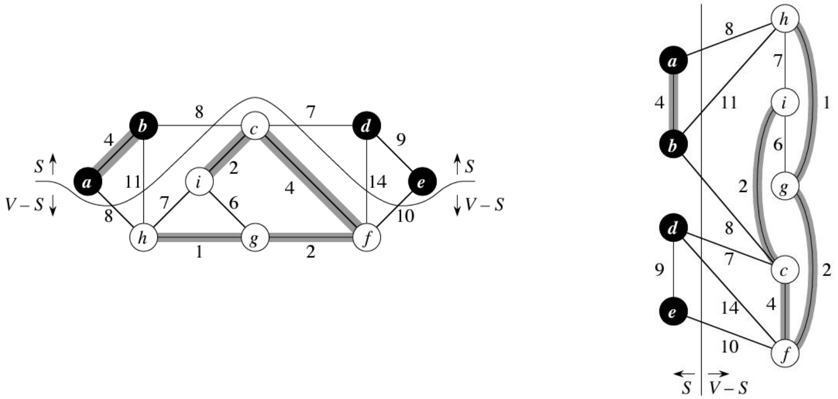
- We actually grow an MST. Of course it's not an MST until it's fully grown. It's just a subgraph of the final MST we'll get.
- Starting from the smallest subset of edges $A = \emptyset \subseteq T \subseteq E$,
  - We grow A (the subgraph that's being grown) to T (a full MST)
  - By adding one edge to A at a time (at each iteration of a loop).
- Surprisingly, there's a greedy choice property that makes this strategy possible.
  - That is, given a subset (of edges) A of an MST, we can always find an edge (u, v) such that $A \cup \{(u, v)\}$ is still a subset of an MST.
  - This edge (u, v) is called a ***safe edge*** for A.
    - Since we can add it safely to A while maintaining the invariant (A being a subset of an MST)

Generic MST Finding Logic

```
GENERIC-MST(G, w)          // G = (V, E), w: E → ℝ (edge-weight function)
1  A = ∅
2  while A does not form a spanning tree          // i.e., while |A| < |G.V| − 1
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

- Of course line 3 is not trivial.
- Theorem 23.1 gives us a general strategy to find a safe edge.
  - And it's a greedy choice (thus all MST algorithms we will learn are greedy)
  - Need to understand the following:
    - A cut of vertices
    - An edge crossing a cut
    - A set of edges respecting a cut
    - A light edge crossing a cut.

Cut, Cut-Crossing Edge, Mutually Respecting Cut & Edge Set, Light Cut-Crossing Edge (CLRS Fig. 23.2)

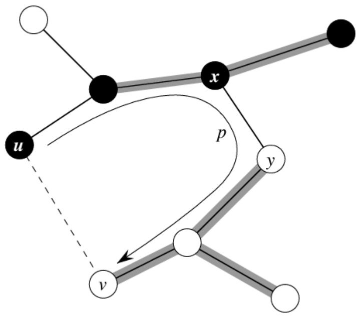


Greedy Choice Property of MST Algorithms (Theorem 23.1)

- Given a cut $(S, V - S)$ and a cut-respecting edge set A which is also a subgraph of an MST, a light cut-crossing edge (u, v) is safe!
- A lot of terminology, but bottom line is:
  - Given a subgraph A of an MST, ***find a cut (any cut) that respects A*** (no edges in A crossing that cut). Let's call the cut $(S, V - S)$.
  - Find all edges crossing the cut. Pick a light edge of those edges (minimum weight). Call such a light edge (u, v).
  - Then $A \cup \{(u, v)\}$ is still a subgraph of an MST.
  - Repeat this $|V| - 1$ times from $A = \emptyset$, and you get an MST!
- It's a greedy strategy (picking a minimum-weight crossing edge).

Proof Sketch (CLRS Fig. 23.3)

- “Cut-and-paste” technique.
- Assume T is an MST which A is a subgraph of, but not including a light edge (u, v) for a cut $(S, V - S)$.
- Then we can “cut” the edge in the MST T crossing the cut. Remove (“cut”) that edge, and add (“paste”) the light edge (u, v). Call the resulting tree T'.
- Then we can show that T' is still an MST.



Multiple MST Algorithms Possible

- Because there are many ways to ***form A and find a cut (any cut) that respects A***!
- Kruskal’s strategy:
  - Given a ***sub-forest*** of an MST, find all edges that connect two trees, pick a minimum-weight edge and add it to the forest.
  - Starting from $|V|$ singleton trees, reduce # trees by 1 at every iteration, ending with only 1 spanning tree, which must be an MST (Theorem 23.1).
  - The cut here is a partition of the forest of all trees, which the minimum-weight edge crosses.

Sub-Forest (Kruskal) vs. Sub-Tree (Prim)

- Prim’s strategy:
  - Given a ***sub-tree*** of an MST, find all edges that connect a vertex in the tree to a non-tree vertex, pick a minimum-weight edge of all and add it to the tree.
  - Starting from one singleton tree (root), grow the tree’s size by 1 at every iteration, ending with 1 spanning tree, which must be an MST (Theorem 23.1).
  - The cut here is the growing sub-tree of an MST, and rest of vertices.

Kruskal’s MST Algorithm

Add Minimum-Weight Edge To Ongoing MST Subgraph (Forest) If It Doesn’t Form A Cycle. Discard It If It Forms A Cycle. Then Proceed To Next Smallest-Weight Edge.

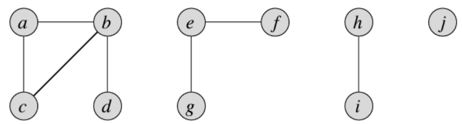
Strategy of Kruskal's MST Algorithm

- Finding a cut that respects the ongoing MST subgraph is not really important.
- Focus on a light edge. What's the first candidate?
  - Minimum-weight edge of all remaining.
- If a minimum-weight edge doesn't form a cycle, it must be a safe (because it doesn't form a cycle) and light (because it's minimum-weight) edge!
- If it forms a cycle, throw it away (can't use it anyway) and try the next minimum-weight edge.
  - It would be handy to sort edges in non-decreasing order of weights, then scan them one by one.
- The key point here is ***how to check if adding an edge forms a cycle***.

Disjoint-Set Data Structure (CLRS Ch. 21)

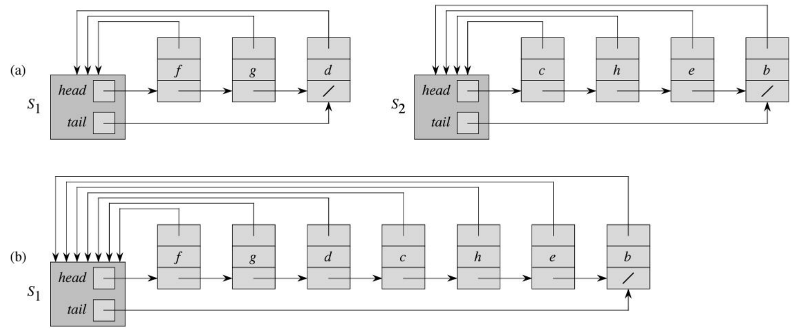
- Efficiently maintains disjoint sets of elements (vertices).
- Supports three operations:
  - MAKE-SET(x): Make a singleton set with one element x. Trivial ($O(1)$).
  - FIND-SET(x): Returns a representative (unique) element from the set that contains x. Time complexity depends on how to implement the data structure.
  - UNION(x, y): Returns the union of two ***disjoint*** sets S_x and S_y (where $x \in S_x$ and $y \in S_y$). Time complexity depends on how to implement.
- Then, the idea is to form disjoint sets of vertices each of which corresponds to a tree in the ongoing forest in MST algorithm.
  - Then checking if adding (u, v) to the forest would form a cycle is equivalent to check whether FIND-SET(u)=FIND-SET(v)!

Disjoint-Set Concept & Example: Connected Components (CLRS Fig. 21.1)



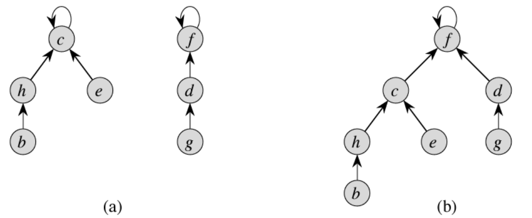
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$

Linked-List Representation Of Disjoint Sets (CLRS Ch. 21-2, Fig. 21.2)



- FIND-SET(x) is $O(1)$, UNION(x, y) is $O(\max(|S_x|, |S_y|)$.

Disjoint-Set Forests (CLRS Ch. 21-3, Fig. 21.4)

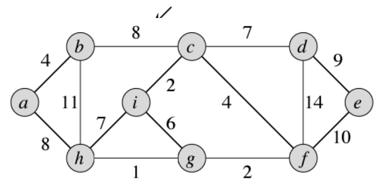


(a) (b)

- FIND-SET(x) is $O(h)$ (where h is height of the tree).
- UNION(x, y) is $O(h)$ for the above straightforward idea.
- But this is NOT an improvement over linked-list representation.
- There are heuristics to improve running time (study CLRS Ch. 21-3).
  - Giving $O(m\alpha(n))$, where $\alpha(n)$ is a very slowly growing function (Ch. 21-4: optional)
    - m: sum of # MAKE-SET, UNION, and FIND-SET ops. n: # MAKE-SET ops.

Kruskal's Algorithm Example (CLRS Fig. 23.4)

Sorted edges: (h, g): 1, (c, i): 2, (g, f): 2, (a, b): 4, (c, f): 4, (g, i): 6, (c, d): 7, (h, i): 7, (a, h): 8: (b, c): 8, (d, e): 9, (e, f): 10, (b, h): 11, (d, f): 14



MST-KRUSKAL(G, w)

1 $A = \emptyset$
2 **for** each vertex $v \in G.V$
3 MAKE-SET(v)
4 sort the edges of $G.E$ into nondecreasing order by weight w
5 **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6 **if** FIND-SET$(u) \neq$ FIND-SET(v)
7 $A = A \cup \{(u, v)\}$
8 UNION(u, v)
9 **return** A

Time Complexity of Kruskal's Algorithm

- $O(E)$ FIND-SET and UNION operations in for loop (line 5-8).
- $|E| \geq |V| - 1$, because G is assumed to be connected.
- Total $O(E\alpha(V))$ (disjoint-set forest representation with heuristics).
- Since $\alpha(V) = O(\lg V)$, total is $O(E \lg V)$.

MST-KRUSKAL(G, w)

1 $A = \emptyset$
2 **for** each vertex $v \in G.V$
3 MAKE-SET(v)
4 sort the edges of $G.E$ into nondecreasing order by weight w
5 **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6 **if** FIND-SET$(u) \neq$ FIND-SET(v)
7 $A = A \cup \{(u, v)\}$
8 UNION(u, v)
9 **return** A

Prim's MST Algorithm

Ongoing MST Subgraph Is Always a Tree. Add To The Tree a Minimum-Weight Edge that Will Still Form a Tree.

Strategy Of Prim's MST Algorithm

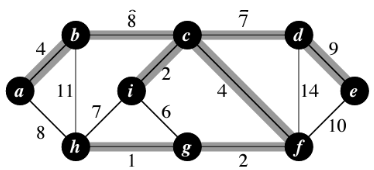
- The ongoing MST subgraph always forms a regular tree (MST subtree).
- Then at each iteration, we add to the tree a new *connected* edge which will still form a tree (no cycle).
- Since the newly added edge should be safe (i.e., the resulting bigger tree should still be a subgraph of an MST), it must be a minimum-weight one of all possible such edges.
- Starting from a singleton MST subtree (root vertex only), add one edge at a time until the tree includes all vertices.
- The key point here is how to find such an edge.

Maintaining Not-Yet-Included Vertices In The Order Of Proximity To Current MST Subtree

- Observation: When a new vertex is added (moved) to the ongoing/growing MST subtree, only its adjacent vertices might get new proximity values (minimum weight to any tree vertex).
- The not-yet-included vertices can be maintained in a min-priority queue (heap).
- When the minimum is extracted from the heap, adjust the proximity values of all its (the minimum's) adjacent vertices in the heap.

Prim's MST Algorithm And Example

$Q =$



MST-PRIM(G, w, r)

```
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)  // This assignment may cause rearrangement of vertices in Q.
```

Time Complexity Of Prim's Algorithm

- Depends on how to implement the min-priority queue Q.
- If we use a binary min-heap (CLRS Ch. 6),
  - Line 1-5: BUILD-MIN-HEAP for $O(V)$ time.
  - Line 6 while loop iterates $|V|$ times. Line 7 EXTRACT-MIN for $O(\lg V)$.
    - Giving $O(V \lg V)$ for EXTRACT-MIN.
  - Line 8 for loop iterates $O(E)$ times all together (amortized): $\sum|Adj(v)| = 2|E|$
  - Line 9 (Q membership test) can be $O(1)$ (maintain flags)
  - Line 11: DECREASE-KEY on the min-heap: $O(\lg V)$
  - $\therefore O(V \lg V + E \lg V) = O(E \lg V)$.

MST-PRIM(G, w, r)

```
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)  // This assignment may cause rearrangement of vertices in Q.
```