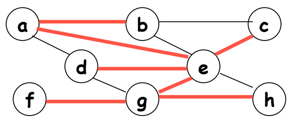
Spanning Trees (Ch. 23)

Definition: Given an undirected, unweighted graph G = (V, E), a **spanning tree** of G is any subgraph of G that is a tree



Weighting edges

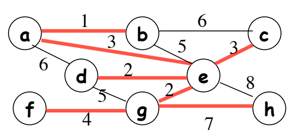
Assign a weight (a numerical value) to each edge of the graph.
Examples:

1. a road network, the weights could represent the length of each road;
2. a network of connecting flights, weights could represent flight time;
3. a computer network, the weights could represent the bandwidth of each bus and link.
4. length of wire needed to connect gates in a circuit.

Minimum Spanning Trees

Definition: Given an undirected graph G = (V, E) with weights on the edges, a **minimum spanning tree** of G is a subgraph T ⊆ E such that T:

- is a spanning tree of G
- has no cycles (i.e., is a tree), and
- has a sum of edge weights that is minimum over all possible spanning trees of G.



MST Property

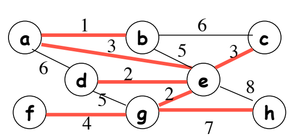
MST property: Let G = (V, E) and let T be any spanning tree of G. Suppose that for every edge (u,v) of G that is not in T, if (u,v) is added to T it creates a cycle such that (u,v) is a maximum weight edge on that cycle. Then T has the MST property.

If there are 2 spanning trees T_1 and T_2 on G that both have the MST property, then T_1 and T_2 have same total weight.

Minimum Spanning Trees

We will look at two "greedy algorithms" to find an MST of a weighted graph: **Kruskal's** and **Prim's** algorithms

A greedy algorithm makes choices in sequence such that each choice is best according to some limited "short-term" criterion that is not too expensive to evaluate (no look-ahead is involved).



MST Uses

Finding an MST has proven to be a useful technique in finding maximum bandwidth channels, circuit design, and in finding neural pathways in the brain.

Finding maximum bandwidth channels in networks requires a simple change to Kruskal's algorithm:

N. Malpani, J. Chen / Information Processing Letters 83 (2002) 175–180

Kruskal's MST Algorithm

Idea:

- use a greedy strategy
- consider edges in increasing order of weight (sort edges)
- add edge to spanning forest F if adding the edge doesn't create a cycle.

```
Algorithm MST-Kruskal (G)
   R = E  // R is initially set of all edges
   F = ∅ // F is set of edges in a spanning tree of a sub-graph of G
1.  sort all edges of R in increasing order of weight
2.  while (R is not empty)
3.      remove the lightest-weight edge, (v,w), from R
4.      if (v,w) does not make a cycle in F
5.          add (v,w) to F
6.  return F
```

Kruskal's MST Algorithm

Complexity:

line 1- Sorting edges = ?? time
lines 2-5 - Keep edges in a structure with O(1) time
line 3- Removal = ?? time per removal
line 4 - Checking to see if edge creates a cycle = ?? time

```
Algorithm MST-Kruskal (G)
   R = E  // R is initially set of all edges
   F = ∅ // F is set of edges in a spanning tree of a sub-graph of G
1.  sort all edges of R in non-decreasing order of weight
2.  while (R is not empty)
3.      remove the lightest-weight edge, (v,w), from R
4.      if (v,w) does not make a cycle in F
5.          add (v,w) to F
6.  return F
```

Disjoint Sets (Ch. 21)

A **disjoint-set data structure**

- maintains a collection of disjoint subsets $C = s_1, s_2, \ldots, s_m$, where *each s_i is identified by a representative element (set id).*

Operations on C:

- **Make-Set**(x): creates singleton set {x}
- **Union**(x,y): x and y are id's of their resp. sets, s_x and s_y; union operation replaces sets s_x and s_y with a set that is $s_x \cup s_y$ and returns the id of the new set.
- **Find-Set**(x): returns the id of the set containing x.

Data Structures for Disjoint Sets

Applications include network algorithms such as finding the connected components of a graph.

PROBLEM IS HOW TO KEEP ELEMENTS IN A SET IN SUCH A WAY THAT IT IS A FAST (SUB-LINEAR TIME) OPERATION TO DETERMINE WHETHER A NEW EDGE WILL CREATE A CYCLE.

We determine membership using representative node names for each connected component.

Data Structures for Disjoint Sets

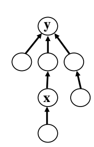
Comment 1: The Make-Set operation is only used during the initialization of a particular algorithm.

Comment 2: We assume there is an array of pointers to each $x \in U$ (so we never have to search for a particular element, just for the id of the set x is in).

Thus the problems we're trying to solve are how to join the sets (Union) and how to find the id of the set containing a particular element (Find-Set) efficiently.

Rooted Tree Representation of Sets

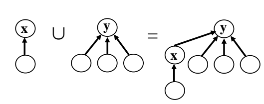
Idea: Organize elements of each set as a tree with id = element at the root, and a pointer from every child to its parent. Also assume we have an array of pointers to each element in the tree.



Make-Set(x): (initial) O(1) time

Find-Set(x):
- start at x (using pointer provided to find x) and follow pointers up to the root.
- return id of root

w-c running time is O(n)



Union(x, y):
- x and y are ids (roots of trees).
- make x a child of y and return y

running time is O(1)

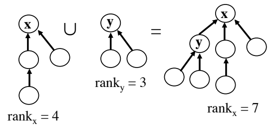
Weighted Union Implementation for Trees

Idea: Add **rank** field to each node x holding the number of nodes in subtree rooted at x (only care about weight field of roots, even though other nodes maintain rank value too). When doing a Union, make the smaller tree (with lower rank at the root) a subtree of the larger tree (with greater rank at the root).

Make-Set(x): O(1)

Find-Set(x):
- ??? See next slide, O(n) w.c.

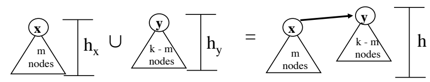
Union(x, y):
- x and y are ids (roots of trees).
- make node (x or y) with smaller rank the child of the other
- O(1) time



Weighted Union

Theorem: Any k-node tree created by k-1 weighted Unions has height O(lg k) (assume we start with Make-Set on k singleton sets). We want to show that trees stay "short".

Proof: By induction on k, the number of nodes in forest.
Basis: k = 1, height = 0 = lg 1 <true>
Inductive Hypothesis: Assume true for all i < k.
Inductive Step: Show true for k. Suppose the last operation performed was union(x,y) and that if m = wt(x) and wt(x) ≤ wt(y), that m ≤ k/2.

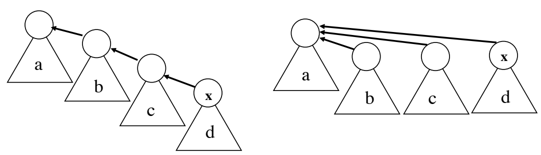


Show $h = \max(h_x + 1, h_y) \leq \lg k$. The IHOP must hold for trees x and y.

- $h_x + 1 \leq \lg(m) + 1 \leq \lg(k/2) + 1 = \lg k - 1 + 1 = \lg k$

Path Compression Implementation

Idea: extend the idea of weighted union (i.e., unions still weighted), but on a Find-Set(x) operation, make every node on the path from x to the root (the node with the set id) a child of the root.



Find-Set(x) still has worst-case time of O(lgn), but subsequent Find-Sets for nodes that used to be ancestors of x (or subsequent finds for x itself) will now be very fast: O(1).

Path Compression Analysis

The running time for *m* (find or union) disjoint-set operations on *n* elements is

$$O(m \lg^* n)$$

The full proof is given in our textbook.

Kruskal's MST Algorithm

Idea: Make each node a singleton set.
Sort edges, then add the minimum-weight edge (u,v) to the MST if u and v are not already in same sub-graph.
Use weighted union with path compression during find-set operations to determine when nodes are in same sub-graph.

```
MST-Kruskal (G) /** G = (V, E) **/
1.  T = ∅
2.  for each v ∈ V
        make-set(v)
3.  sort edges in E by increasing (non-decreasing) weight
4.  for each (u,v) ∈ E
        if find-set(u) ≠ find-set(v)
            T = T ∪ {(u,v)}     /** add edge to MST **/
            union(find-set(u), find-set(v))
5.  return t
```

Kruskal's MST Algorithm

Running Time

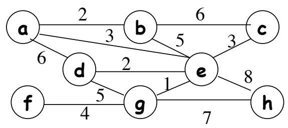
*** initialization (lines 1-3) –**
O(1) + O(V) + O(E lg E) = O(V + E lg E)

*** E iterations of for-loop (line 4)**
- **2E finds – O(E lg*E) time**
- **O(V) unions = O(V) time (at most V – 1 unions)**

- **total: O(V + E lg E) = O(E lg V) time**

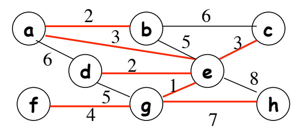
- (note lg E = O(lgV) since E = O(V^2), so lgE = 2 lg V).

```
MST-Kruskal (G)
1.  T = ∅
2.  for each v ∈ V
        makeset(v)
3.  sort edges in E by increasing weight
4.  for each (u,v) ∈ sorted E
        if find (u) ≠ find(v) /** doesn't create a cycle **/
            T = T ∪ {(u,v)}  /** add edge to MST **/
            union(find(u), find(v))
5.  return T
```



List the edges in the above graph in a possible order they are added to the MST by Kruskal's algorithm. Which edges would not be added?

```
MST-Kruskal (G)
1.  T = ∅
2.  for each v ∈ V
        makeset(v)
3.  sort edges in E by increasing weight
4.  for each (u,v) ∈ sorted E
        if find (u) ≠ find(v) /** doesn't create a cycle **/
            T = T ∪ {(u,v)}  /** add edge to MST **/
            union(find(u), find(v))
5.  return t
```



Correctness of Kruskal's Algorithm

Theorem: Kruskal's algorithm produces an MST on G = (V, E).

Proof: Clearly, the algorithm produces a spanning tree. We need to argue that it is an MST.

Suppose, in contradiction, the algorithm does not produce an MST. Suppose that the algorithm adds edges to the tree T' in order

$e_1, e_2, \ldots, e_i, \ldots, e_{n-1}$.

Let i be the value such that $e_1, e_2, \ldots, e_{i-1}$ is a subset of some MST T, but $e_1, e_2, \ldots, e_{i-1}, e_i$ is not a subset of any MST.

Consider T ∪ {e_i}
- T ∪ {e_i} must have a cycle c involving e_i
- In the cycle c there is at least one edge that is not in $e_1, e_2, \ldots, e_{i-1}$ (since the algorithm doesn't pick an edge that creates a cycle and it picked e_i).

Correctness of Kruskal's Algorithm (cont.)

- let e* be the edge in T ∪ {e_i} that forms a cycle when e_i is added to T that is not in $e_1, e_2, \ldots, e_{i-1}$

  Then $wt(e_i) < wt(e^*)$, otherwise the algorithm would have picked e* next in sorted order when it picked e_i (by assumption that T, with e*, is not an MST because the algorithm does not find an MST).

Claim: T' = T − {e*} ∪ {e_i} is a MST
- T' is a spanning tree since it contains all nodes and has no cycles.
- wt(T') < wt(T), so T is not a MST

This contradiction means our original assumption must be wrong and therefore the algorithm always finds an MST. ■

Prim's MinST Algorithm

Algorithm starts by selecting an arbitrary starting vertex, and then "branching out" from the part of the tree constructed so far by choosing a new vertex and edge at each iteration.

Idea:
- **always maintains one connected subgraph (different from Kruskal's)**
- **at each iteration, chooses the lowest weight edge that goes out from the current tree (a greedy strategy).**

Prim's MST Algorithm

Idea: Use a **min priority queue** PQ that uses the wt field as a key.

Associate with each node v two fields:
- *v.wt :* if v isn't in T, then holds the min wt of all the edges from v to a node in T.
- *v.π:* if v isn't in T, holds the name of the node u in T such that wt(u,v) is v's best edge to node in T.

As min wt edges are discovered they are added to T.

```
MST-Prim (G, r)
1.  insert each v ∈V into PQ with v.wt = ∞,
     v.π = ∅
2.  r.wt = 0 // root of MST
3.  while PQ ≠ ∅
4.      u = PQ.extract-min()
5.      add edge (u. π, u) to T
6.      for each neighbor v of u
7.          if v ∈ PQ and wt(u,v) < v.wt
8.              v.π = u
9.              v.wt = wt(u,v)
```

Prim's MST Algorithm

Start at node a: PQ contains all nodes

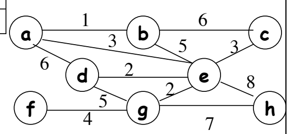
PQ	a	b	c	d	e	f	g	h
π	∅	∅	∅	∅	∅	∅	∅	∅
wt	0	∞	∞	∞	∞	∞	∞	∞

iteration 1: PQ = PQ – {a}

PQ	~~a~~	b	c	d	e	f	g	h
π	∅	a	∅	a	a	∅	∅	∅
wt	0	1	∞	6	3	∞	∞	∞

T = {∅} (change b, d, e) because of edges (a,b), (a,e), and (a,d)

```
MST-Prim (G, r)
insert each v ∈ V into PQ with
  v.wt = ∞, v.π = ∅
r.wt = 0 // root of MST
while PQ ≠ ∅
  u = PQ.extract-min()
  add edge (u. π, u) to T
  for each neighbor v of u
    if v ∈ PQ and wt(u,v) < v.wt
      v.π = u
      v. wt = wt(u,v)
```



iteration 2: PQ = PQ – {b}

PQ	~~a~~	~~b~~	c	d	e	f	g	h
π	∅	a	b	a	a	∅	∅	∅
wt	0	1	6	6	3	∞	∞	∞

T = {(a,b)} (change c due to edge (b,c))

iteration 3: PQ = PQ – {e}

PQ	~~a~~	~~b~~	c	d	~~e~~	f	g	h
π	∅	a	e	e	a	∅	e	e
wt	0	1	3	2	3	∞	2	8

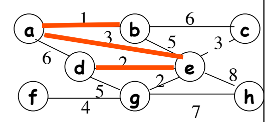
T = {(a,b), (a,e)} (change c, d, g, h) due to (c,e), (d,e), (e,g), and (e,h)

iteration 4: PQ = PQ – {d}

PQ	~~a~~	~~b~~	c	~~d~~	~~e~~	f	g	h
π	∅	a	e	e	a	∅	e	e
wt	0	1	3	2	3	∞	2	8

T = {(a,b), (a,e), (d,e)} (no change to wt field at any node)


```
MST-Prim (G, r)
insert each v ∈ V into PQ with
  v.wt = ∞, v.π = ∅
r.wt = 0 // root of MST
while PQ ≠ ∅
  u = PQ.extract-min()
  add edge (u. π, u) to T
  for each neighbor v of u
    if v ∈ PQ and wt(u,v) < v.wt
      v.π = u
      v. wt = wt(u,v)
```

iteration 5: PQ = PQ – {g}

PQ	~~a~~	~~b~~	c	~~d~~	~~e~~	f	~~g~~	h
π	∅	a	e	e	a	g	e	g
wt	0	1	3	2	3	4	2	7

T = {(a,b), (a,e), (d,e), (e,g)} (change f & h) due to edges (f,g) and (g,h)

iteration 6: PQ = PQ – {c}

PQ	~~a~~	~~b~~	~~c~~	~~d~~	~~e~~	f	~~g~~	h
π	∅	a	e	e	a	g	e	g
wt	0	1	3	2	3	4	2	7

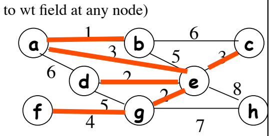
T = {(a,b), (a,e), (d,e), (e,g), (c,e)} (no change to wt field at any node)

iteration 6: PQ = PQ – {f}

PQ	~~a~~	~~b~~	~~c~~	~~d~~	~~e~~	~~f~~	~~g~~	h
π	∅	a	e	e	a	g	e	g
wt	0	1	3	2	3	4	2	7

T = {(a,b), (a,e), (d,e), (e,g), (c,e), (f,g)} (no change to wt field at any node)


```
MST-Prim (G, r)
insert each v ∈ V into PQ with
  v.wt = ∞, v.π = ∅
r.wt = 0 // root of MST
while PQ ≠ ∅
  u = PQ.extract-min()
  add edge (u. π, u) to T
  for each neighbor v of u
    if v ∈ PQ and wt(u,v) < v.wt
      v.π = u
      v. wt = wt(u,v)
```

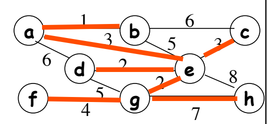

```
MST-Prim (G, r)
insert each v ∈ V into PQ with
  v.wt = ∞, v.π = ∅
r.wt = 0 // root of MST
while PQ ≠ ∅
  u = PQ.extract-min()
  add edge (u. π, u) to T
  for each neighbor v of u
    if v ∈ PQ and wt(u,v) < v.wt
      v.π = u
      v. wt = wt(u,v)
```


iteration 7: PQ = PQ – {h} = ∅

PQ	~~a~~	~~b~~	~~c~~	~~d~~	~~e~~	~~f~~	~~g~~	h
π	∅	a	e	e	a	g	e	g
wt	0	1	3	2	3	4	2	7

T = {(a,b), (a,e), (d,e), (e,g), (c,e), (f,g), (g,h)}

DONE



Running Time of Prim's MST Algorithm

- Assume PQ is implemented with a binary min-heap
- How can we tell if v ∈ PQ without searching heap?


```
MST-Prim (G, r)
insert each v ∈ V into PQ with
  v.wt = ∞, v.π = ∅
r.wt = 0 // root of MST
while PQ ≠ ∅
  u = PQ.extract-min()
  add edge (u. π, u) to T
  for each neighbor v of u
    if v ∈ PQ and wt(u,v) < v.wt
      v.π = u
      v. wt = wt(u,v)
```


Running Time of Prim's MST Algorithm

- Assume PQ is implemented with a binary min-heap
- How can we tell if v ∈ PQ without searching heap?

Keep an array of booleans indexed by the nodes indicating if node is in heap


```
MST-Prim (G, r)
insert each v ∈ V into PQ with
  v.wt = ∞, v.π = ∅
r.wt = 0 // root of MST
while PQ ≠ ∅
  u = PQ.extract-min()
  add edge (u. π, u) to T
  for each neighbor v of u
    if v ∈ PQ and wt(u,v) < v.wt
      v.π = u
      v. wt = wt(u,v)
```

Running Time of Prim's MST Algorithm

```
MST-Prim (G, r)
1.   insert each v ∈ V into PQ with
       v.wt = ∞, v.π = ∅
2.   r.wt = 0 // root of MST
3.   while PQ ≠ ∅
4.       u = PQ.extract-min()
5.       add edge (u. π, u) to T
6.       for each neighbor v of u
7.           if v ∈ PQ and wt(u,v) < v.wt
8.               v.π  = u
9.               v. wt = wt(u,v)
```

Running time:
• initialize PQ: O(V) time

• while loop...
  in each of V iterations of while loop:
    extract min = O(lg V) time
    update T = O(1) time
    ==> O(V lg V) total

  over all iterations (combined):
    check neighbors of u (line 6-9): O(E) iterations
    condition test and update π = O(1) time
    decreasing v's wt= O(lg V) time = O(E lg V)

So, the grand total is:
  O(V lg V + E lg V) = O(E lg V) (asymptotically, the same as Kruskal's)

Correctness of Prim's Algorithm

Let T_i be the tree after the ith iteration of the while loop

Lemma: For all i, T_i is a subtree of some MST of G.

Proof: by induction on i, the number of iterations

Basis: when i = 0, $T_0 = \emptyset$, ok - because empty is trivial MST subtree

IHOP: Assume T_i is a subtree of some MST M

Induction Step: Show that T_{i+1} is a subtree of some MST

Correctness of Prim's Algorithm

Let (u,v) be the edge added in iteration i + 1. Then there are 2 cases:

case 1: (u, v) is an edge of M (the ultimate MST).
Then clearly T_{i+1} is a subtree of M (ok)

case 2: (u, v) is not an edge of M
We know there is a path p in M from u to v (because M is a ST)

Let (x, y) be the first edge in p with x in T_i and y not in T_i. We know this edge exists because the algorithm will not add edge (u,v) to a cycle.

M' = M – {(x, y)} ∪ {(u, v)} is another spanning tree.

Now we note that

$$wt(M') = wt(M) - wt(x, y) + wt(u, v) \leq wt(M)$$

since (u, v) is the minimum weight outgoing edge from T_i

Therefore, M' is also a MST of G and T_{i+1} is a subtree of M'. ■

Maximum Spanning Trees

Definition: Given an undirected graph G = (V, E) with weights on the edges, a **maximum spanning tree** of G is a subgraph T ⊆ E such that T:

- connects all nodes in V,
- has no cycles (i.e., is a tree), and
- has a sum of edge weights that is maximum over all possible spanning trees of G.