Chapter 12

Greedy Algorithms for Minimum Spanning Trees

CS 473: Fundamental Algorithms, Spring 2011
March 3, 2011

12.1 Minimum Spanning Trees

We are given a connected graph $G = (V, E)$ with edge costs. The Goal is to find $T \subseteq E$ such that (V, T) is connected and total cost of all edges in T is smallest. That is, T is the ***minimum spanning tree*** (MST) of G. See Figure 12.1 for an example.

There are many applications of MST, including:

(A) Network Design
  (A) Designing networks with minimum cost but maximum connectivity
(B) Approximation algorithms
  (A) Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
(C) Cluster Analysis

12.1.1 The Algorithms

The basic template for the MST algorithms would be a greedy algorithm, listed on the left. Our main task, is to decide in what 'order should edges be processed? When should we add edge to spanning tree?

```
Initially $E$ is the set of all edges in $G$
$T$ is empty (* T will store edges of a MST *)
while $E$ is not empty do
    choose $i \in E$
    if ($i$ satisfies condition) then
        add $i$ to $T$
return the set $T$
```

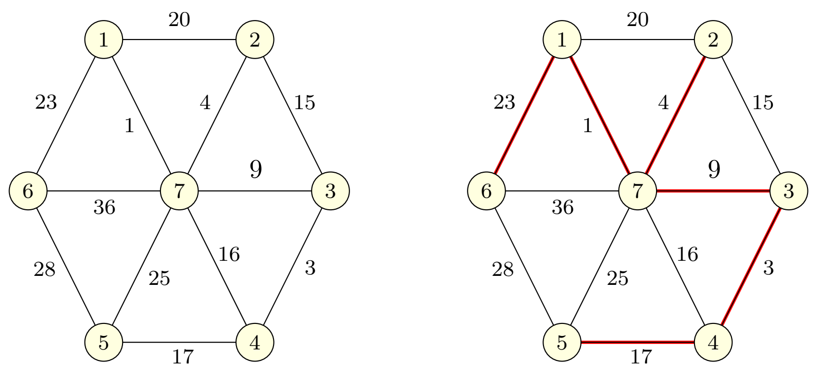
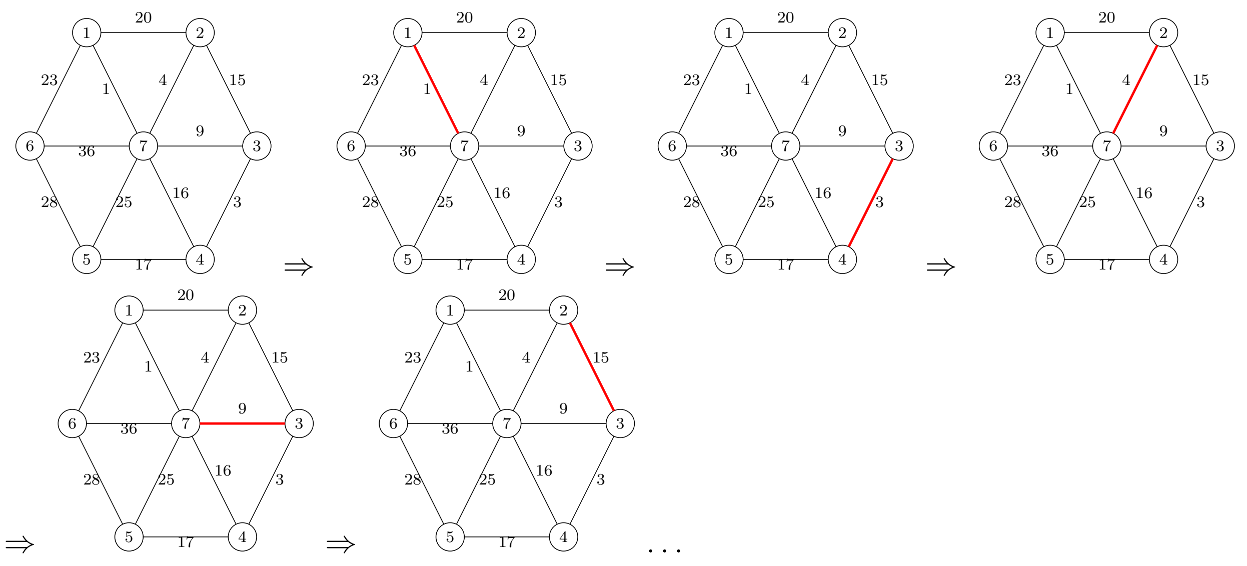


Figure 12.1: A graph and its MST.

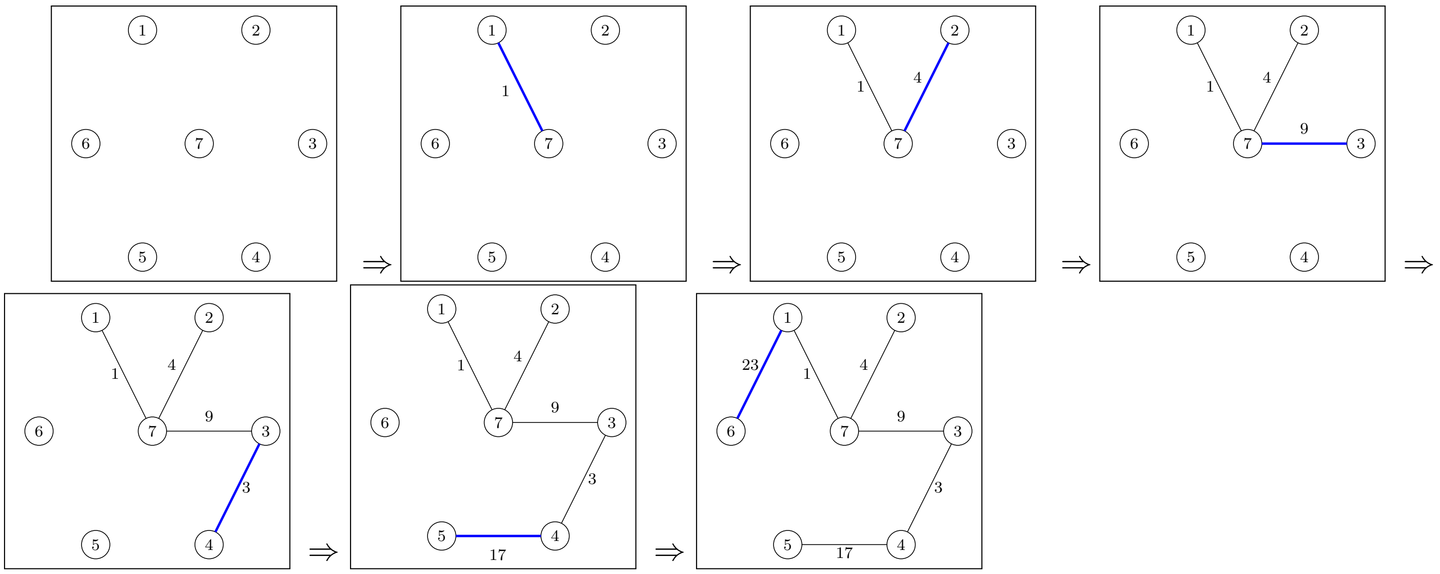
12.1.1.1 Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they do not form a cycle. As such, for the example, the edges would be considered in the following order:



12.1.1.2 Prim's Algorithm

The set T is maintained by algorithm and in the end it will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T. The constructed tree for the example, would be built as follows:



12.1.1.3 Reverse Delete Algorithm

Another curious MST algorithm works by deleting the edges in reverse order from the initial graph, deleting from the most expensive edges to the cheapest, maintaining connectivity.

```
Initially $E$ is the set of all edges in $G$
$T$ is $E$ (* $T$ will store edges of a MST *)
**while** $E$ is not empty **do**
    choose $i \in E$ of largest cost
    **if** removing $i$ does not disconnect $T$ **then**
        remove $i$ from $T$
**return** the set $T$
```

12.1.1.4 Borøuvka's algorithm (*)

Maybe the most elegant algorithm for MST is ***Borøuvka's algorithm***. The algorithm compute for every vertex the cheapest edge adjacent to it. It then add all these edges to the forest being constructed. It computes the connected components of this forest. In the next iteration, for any connected component, it computes the cheapest edge adjacent to it, and these edges to the forest, and repeat this till it ends up with a single tree. Since the number of connected components go down by a factor of two in each iteration, and each iteration can be done in linear time, it follows that this algorithm runs in $O((n + m) \log n)$ time. Otakar Borøuvka discovered this algorithm in 1926 (!).

12.1.2 Correctness

While there are many different MST algorithms, all of them rely on some basic properties of MSTs, in particular the ***Cut Property*** to be seen shortly.

For the sake of simplicity of exposition, we assume the following:

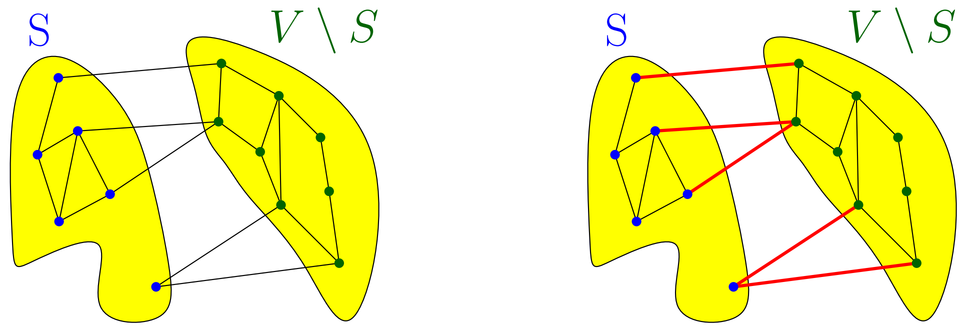


Figure 12.2: A cut and the cut edges.

Assumption 12.1.1 *Edge costs are distinct, that is no two edge costs are equal.*

Definition 12.1.2 *Given a graph $G = (V, E)$, a* **cut** *is a partition of the vertices of the graph into two sets $(S, V \setminus S)$. Edges having an endpoint on both sides are the* **edges of the cut**. *A cut edge is* **crossing** *the cut.*

See Figure 12.1.2 for an example.

Definition 12.1.3 *An edge $e = (u, v)$ is a* **safe** *edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).*

An edge $e = (u, v)$ is an **unsafe** *edge if there is some cycle C such that e is the unique maximum cost edge in C.*

Lemma 12.1.4 (Unsafe is not safe.) *If an edge e is unsafe then it can not be safe.*

Proof: If an edge e is unsafe then it is contained in a cycle, where all the other edges are cheaper. But then, any cut containing this edge would contain some other edge of this cycle, and this other edge would be cheaper. Namely, the edge e can not be safe. ■

Proposition 12.1.5 *If edge costs are distinct then every edge is either safe or unsafe.*

Proof: Consider an edge uv, and consider the cut $S_1 = \{u\}$. If uv is the cheapest edge in the cut $(S_1, V \setminus S_1)$ then it is safe and we are done. Otherwise, there is a cheaper edge $e_1 = uv_1$ in this cut than uv. We add v_1 to the set S_1 to form the new set S_2.

We continue in this fashion, at the ith iteration we have a set S_i that contains u, and does not contain v. If uv is the cheapest edge in the cut $(S_i, V \setminus S_i)$ then we are done, as uv is safe for this cut. Otherwise, there is a cheaper edge $u_u v_i$ in this cut than uv. We add v_i to the set S_i to form the set S_{i+1}.

Clearly, all the edges being considered by the algorithm before the beginning of the ith iteration form a spanning tree of the vertices of S_i. And all these edges are more expensive that uv. As such, when we add v to the set S_i (which must happen as the given graph

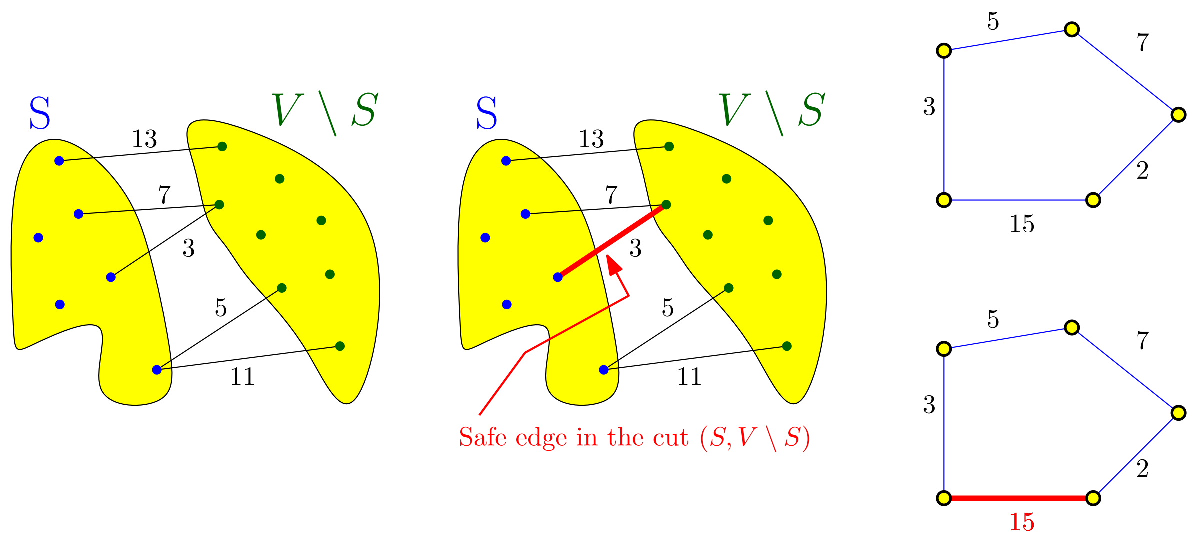


Figure 12.3: Another way to think about safe and unsafe edges – every cut identify one safe edge, and every cycle identify another

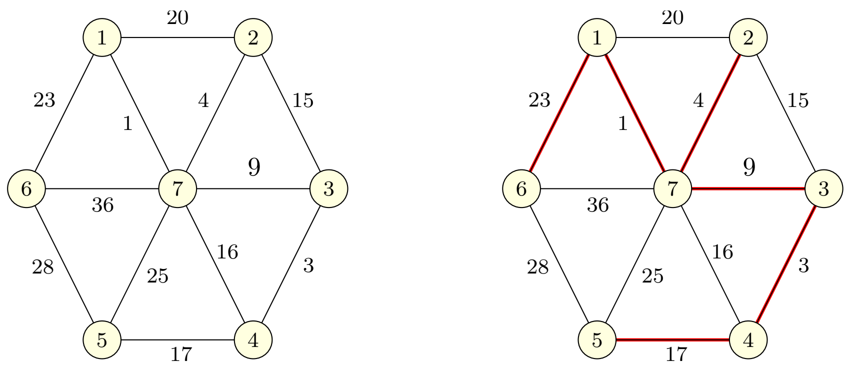


Figure 12.4: Graph with unique edge costs. Safe edges are red, rest are unsafe. And all safe edges are in the MST in this case.

is connected), we have a connected spanning tree containing u and v and all its edges are cheaper then uv. In particular, adding uv to this tree form a cycle involving uv, where all the other edges in the cycle are cheaper. Namely, uv is unsafe, as desired. ■

Lemma 12.1.6 (Cut property.) *If e is a safe edge then every minimum spanning tree contains e.*

Proof: Suppose, for the sake of contradiction, that e is not in MST T. Then, since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S. Now, since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$.

It is natural to claim that since $c_f > c_e$, $T' = (T \setminus \{f\}) \cup \{e\}$ is a spanning tree of lower cost. However, this is not true, see Figure 12.5.)

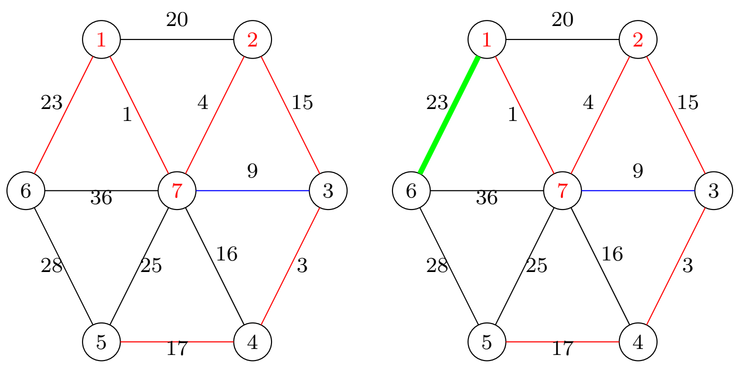
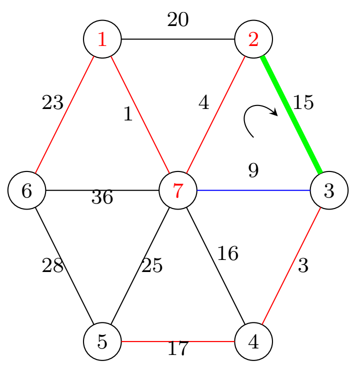


Figure 12.5: Problematic example. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. $T - f + e$ is not a spanning tree.



So, suppose that $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Without loss of generality assume that $v \in S$. Since T is spanning tree, there is a unique path P from v to w in T

Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$. Now, $T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree of lower cost, by the following observation. ■

Observation 12.1.7 *$T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree.*

Proof: Observe that T' is connected. Indeed, we removed $e' = (v', w')$ from T, but v' and w' are connected by the path $P - f + e$ in T'. Hence T' is connected if T is.

Also, T' is a tree. Indeed, T' is connected and has $n - 1$ edges (since T had $n - 1$ edges) and hence T' is a tree ■

Lemma 12.1.8 (Safe Edges form a Tree.) *Let G be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.*

Proof: Indeed, assume this is false. Then, let S be a connected component in the graph induced by the safe edges. Now, consider the edges crossing S, there must be a safe edge among them since edge costs are distinct and the graph is connected. But then, this edge has one endpoint in S and the other one w in $V \setminus S$. But then $S \cup \{w\}$ is a connected component in G. A contradiction to the choice of S. ■

Corollary 12.1.9 (Safe Edges form an MST.) *Let G be a connected graph with distinct edge costs, then the set of safe edges form the* unique *MST of G.*

Consequence: Every correct MST algorithm when G has unique edge costs includes exactly the safe edges.

Lemma 12.1.10 (Cycle Property.) *If e is an unsafe edge then no MST of G contains e.*

Proof: By the above, for an edge to be in the MST it has to be safe. However, by Lemma 12.1.4, e can not be safe. ■

Note: Cut and Cycle properties hold even when edge costs are not distinct. Safe and unsafe definitions do not rely on distinct cost assumption.

12.1.3 Correctness of Prim's Algorithm

Prim's Algorithm works by repeatedly picking an edge with minimum attachment cost to current tree, and adding it to the current tree.

Lemma 12.1.11 (Correctness of Prim's algorithm.) *Prim's algorithm outputs the MST.*

Proof: If an edge e is added to tree, then e is safe and belongs to every MST. Indeed, let S be the set vertices connected by edges in T when e is added. The edge e is the edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.

Observe that the set of edges output is a spanning tree. To this end, observe that the set of edges output forms a connected graph, as an easy induction shows. Specifically, the set S of vertices maintained by Prim's algorithm is connected in the subtree constructed so far. And eventually $S = V$.

Note, that the algorithm added only safe edges, and safe edges can not form a cycle by themselves (because such a cycle would contain an unsafe edge). ■

12.1.4 Correctness of Kruskal's Algorithm

Kruskal's Algorithm works by repeatedly picking edge of lowest cost and add if it does not form a cycle with existing edges.

Lemma 12.1.12 (Correctness of Kruskal's algorithm.) *Kruskal's algorithm outputs a MST.*

Proof: Observe that if $e = (u, v)$ is added to tree, then e is safe. Indeed, when the algorithm adds e let S and S' be the connected components containing u and v, respectively. If e is the lowest cost edge crossing S (and also S') and it is thus safe.

If there is an edge e' crossing S that has lower cost than e, then e' would come before e in the sorted order and would be added by the algorithm to T before e. A contradiction.

Now, by induction, one can show that the current forest maintained by Kruskal algorithm corresponds to the exact same connected components in the graph having all the edges considered by Kruskal's algorithm. In particular, since the input graph is connected, it follows that the output of Kruskal algorithm is connected. And since it has no cycles, and it is made of safe edges, we conclude that the output is the MST. ■

12.1.4.1 Correctness of Reverse Delete Algorithm

The reverse delete algorithm, consider the edges of the graph in decreasing cost and remove an edge if it does not disconnect the graph. By arguing that only unsafe edges are removed, it is easy to show that this algorithm indeed generates the MST.

12.1.4.2 When edge costs are not distinct

In the above we assumed that the prices of all edges are distinct. Our arguments technically do not hold if this fails. One easy (heuristic argument) to overcome this, is to make edge costs distinct by adding a tiny different costs to each edge, thus making them distinct.

A more Formal argument, and a rather elegant one, is to order edges lexicographically to break ties. Formally, we define a strict ordering $\prec$ over the edges, as follows:

$$e_i \prec e_j \qquad \text{if and only if} \qquad \Big(c(e_i) < c(e_j) \quad \text{or} \quad \big(c(e_i) = c(e_j) \text{ and } i < j \big) \Big).$$

Clearly, under this order all the edges are distinct, and furthermore, the MST has the same weight as the MST of the original graph.

Interestingly, lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or $(c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A)$ In particular, using this, one can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST under this ordering.

Prim's, Kruskal, Borøuvka and Reverse Delete Algorithms all output the optimal (unique) MST when used wit the to lexicographic ordering.

12.1.4.3 Edge Costs: Positive and Negative

The above algorithms and proofs do not assume that edge costs are non-negative! As such, all these MST algorithms work for arbitrary edge costs. Another way to see this, is to make edge costs non-negative by adding to each edge a large enough positive number. (Note, that this does not for shortest paths, why?)

We can also compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

12.2 Data Structures for MST algorithms

12.2.1 Implementing Prim's Algorithm

The pseudo-code of Prim's algorithm is show in Figure 12.6. Observe that Prim's algorithm performs $O(n)$ iterations, where n is number of vertices. Naively, picking e takes $O(m)$ time at each iteration, where m is the number of edges of G. As such, the total running time of **PrimBasic** is $O(nm)$.

A more efficient implementation is possible, by maintaining vertices in $V \setminus S$ in a priority queue with the key $a(v)$. Formally, the ***attachment cost*** of a vertex $v \in V \setminus S$, is the price

```
PrimBasic($G$):
    $E$ is the set of all edges in $G$
    $S = \{1\}$
    $T = \emptyset$ (* $T$ store edges of MST *)
    while $S \neq V$ do
        pick $e = (v, w) \in E$ such that
            $v \in S$ and $w \in V - S$
            $e$ has minimum cost
        $T = T \cup e$
        $S = S \cup w$
    return the set $T$
```

(A)

```
PrimMoreEfficient($G$):
    $E$ is the set of all edges in $G$
    $S = \{1\}$
    $T$ is empty (* $T$ store edges of MST *)
    for $v \notin S$, $a(v) = \min_{w \in S} c(w, v)$
    for $v \notin S$, $e(v) = w$ such that $w \in S$
            and $c(w, v)$ is minimum
    while $S \neq V$ do
        pick $v$ with minimum $a(v)$
        $T = T \cup \{(e(v), v)\}$
        $S = S \cup \{v\}$
        update arrays $a$ and $e$
    return the set $T$
```

(B)

Figure 12.6: (A) Basic implementation of Prim's algorithm, and (B) a more efficient implementation.

of the cheapest edge connecting v with a vertex in S; that is $a(v) = \min_{w \in S} c(w, v)$. To this end, we will use priority queues to maintain the attachment costs efficiently.

12.2.2 Priority Queues and Prim's Algorithm

Prim's algorithm using priority queue is depicted on the right.

```
$E$ is the set of all edges in $G$
$S = \{1\}$
$T$ is empty (* $T$ will store edges of a MST *)
for $v \notin S$, $a(v) = \min_{w \in S} c(w, v)$
for $v \notin S$, $e(v) = w$ such that $w \in S$ and $c(w, v)$ is min
while $S \neq V$ do
    pick $v$ with minimum $a(v)$
    $T = T \cup \{(e(v), v)\}$
    $S = S \cup \{v\}$
    update arrays $a$ and $e$
return the set $T$
```

Formally, a ***priority queue*** is a data structure to store a set S of n elements where each element $v \in S$ has an associated real / integer key $k(v)$.

A priority queue need to support the following operations:

(A) **makeQ**: create an empty queue
(B) **findMin**: find the minimum key in S
(C) **extractMin**: Remove $v \in S$ with smallest key and return it
(D) **add**$(v, k(v))$: Add new element v with key $k(v)$ to S
(E) **delete**(v): Remove element v from S
(F) **decreaseKey** $(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$
(G) **meld**: merge two separate priority queues into one

As such, for Prim's algorithm, maintain vertices in $V \setminus S$ in a priority queue using $a(v)$ as the key value. The algorithm requires: (i) $O(n)$ **extractMin** operations, and (ii) $O(m)$ **decreaseKey** operations.

```
KruskalBasic($G$):
    $E \leftarrow$ set of all edges in $G$
    $T \leftarrow \emptyset$ (* = set of edges of MST *)
    while $E$ is not empty do
        choose $e \in E$ of minimum cost
        if (no cycle in $T \cup \{e\}$)
            add $e$ to $T$
        $E \leftarrow E \setminus \{e\}$
    return the set $T$
```

(A)

```
KruskalEfficient($G$):
    Sort edges in $E = E(G)$ by inc' cost
    $T$ is empty (* $T$ store edges of MST *)
    each vertex $u$ is in a set by itself
    while $E$ is not empty do
        pick $e = (u, v) \in E$ of minimum cost
        if $u$ and $v$ belong to different sets
            add $e$ to $T$
            merge the sets containing $u$ and $v$
        $E \leftarrow E \setminus \{e\}$
    return the set $T$
```

(B)

Figure 12.7: (A) Basic implementation of Kruskal's algorithm, and (B) a more efficient implementation.

Using standard Heaps, the operations **extractMin** and **decreaseKey** can be implemented in $O(\log n)$ time. As such, the total running time is $O((m + n) \log n)$.

One can do better – Fibonacci Heaps take $O(\log n)$ for **extractMin** and $O(1)$ (amortized) for **decreaseKey**. As such, the total running time is $O(n \log n + m)$ in this case.

Question 12.2.1 *Prim's algorithm and Dijkstra's algorithms are very similar. Where is the difference?*

12.2.3 Implementing Kruskal's Algorithm

The basic implementation of Kruskal's algorithm is show in Figure 12.7. Its efficiency can be improved by presorting the edges based on cost. Then, at each iteration, choosing minimum can be done in $O(1)$ time. At least naively, to check if adding an edge create a cycle, requires us to do **BFS**/**DFS** on $T \cup \{e\}$, and this takes $O(n)$ time. As such, the overall running time is $O(m \log m) + O(mn) = O(mn)$.

Fortunately, by using some data-structure magic, we can do better. Specifically, we need a data structure to check if two elements belong to same set, and be able to merge two sets. Such a data-structure is known as a ***Union-Find*** data-structure, and is described next.

12.2.4 The Union-Find Data Structure

The ***Union-Find*** data-structure Stores disjoint sets of elements, and supports the following operations.

(A) **makeUnionFind**(S) returns a data structure where each element of S is in a separate set.

(B) **find**(u) returns the *name* of set containing element u. Thus, u and v belong to the same set iff **find**(u) = **find**(v).

(C) **union**(A, B) merges two sets A and B. Here A and B are the names of the sets. Typically the name of a set is some element in the set.

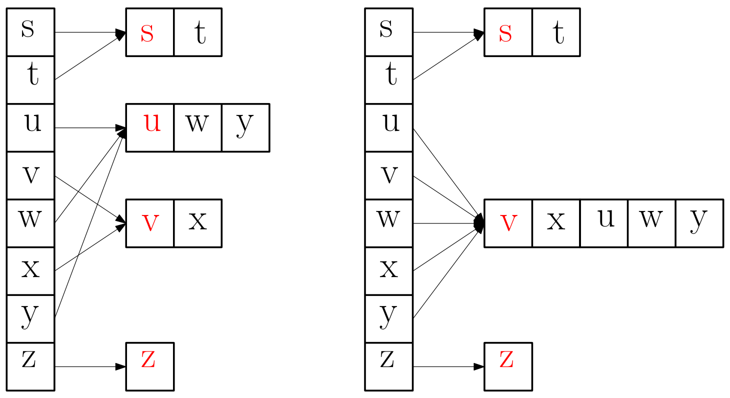


Figure 12.8: An example of the union-find data-structure, and the resulting data-structure after **union**(**find**(u), **find**(v)).

12.2.4.1 Implementing Union-Find using Arrays and Lists

Each set is stored in a linked list with a name associated with the list. (The name is, for example, a pointer to the first element in the list.) An example of such data-structure is depicted in Figure 12.8.

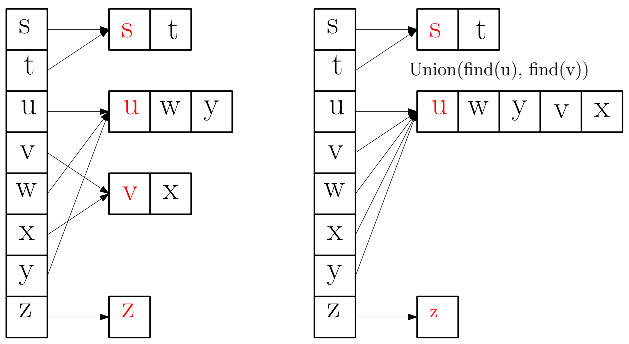
For each element $u \in S$, the data-structure maintains a pointer to the its set. Array for pointers: `set`$[u]$ is pointer for u.

The operation **makeUnionFind** (S) takes $O(n)$ time and space. We just create a list with a single element for each member of S, and set `set`$[u]$ to point to u's list, for every $u \in S$.

The operation **find**(u) works by reading the entry `set`$[u]$ and returning it, and it takes $O(1)$ time.

The **union**(A,B) operation, involves updating the entries `set`$[u]$ for all elements u in A and B, and it takes $O(|A| + |B|)$ which is $O(n)$ time. And example of such an operation is depicted in Figure 12.8.

12.2.4.2 Improving the List Implementation for Union



As before we use `set`$[u]$ to store the set of u. We change the **union**(A,B) as follows:

(A) With each set, we keep track of its size.
(B) If A is larger than B then we return **union**(B,A)
(C) Now $|A| \leq |B|$, and we merge the list of A into that of B: $O(1)$ time (linked lists).
(D) Update `set`$[u]$ only for elements in the (smaller) set A.

An example of the new implementation is depicted on the right.

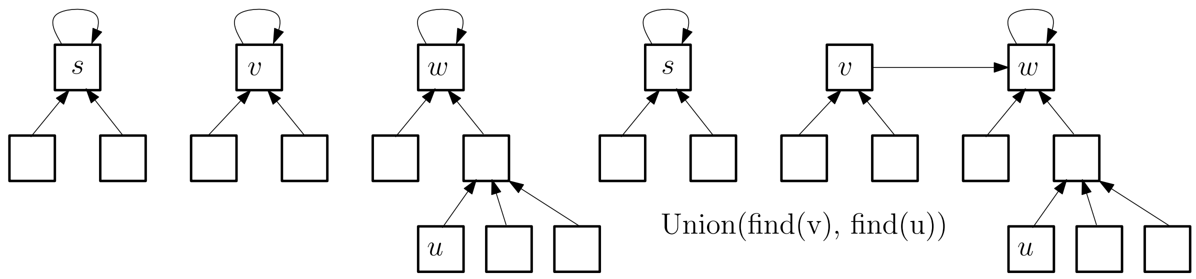


Figure 12.9: Union-find represented using a tree, and the result of a union operation.

This union operation takes $O(\min(|A|, |B|))$ time. Worst case is still $O(n)$. The **find** operation still takes $O(1)$ time

Interestingly, the above improved “heuristic” implementation is provably better. The key observation is that **union**(A,B) takes $O(|A|)$ time where $|A| \leq |B|$, and the size of the new set is $\geq 2|A|$. In particular, consider a specific element, and the sizes of the sets containing it, every time it gets merged into a bigger set. We get a sequence $1 = n_1 \leq n_2 \leq ... \leq n_k$. Observe, however, that $n_i \geq 2n_{i-1}$. And this can happen at most $O(\log n)$ times.

Theorem 12.2.2 *Any sequence of k* **union** *operations, starting from* **makeUnionFind**(S) *on set S of size n, takes at most $O(k \log k)$.*

Proof: Any union operation involves at most 2 of the original one-element sets; thus at least $n - 2k$ elements have never been involved in a union. Also, maximum size of any set (after k unions) is $2k$.

Now, **union**(A,B) takes $O(|A|)$ time where $|A| \leq |B|$. We *Charge* each element in A constant time to *pay* for the $O(|A|)$ time the union operation took.

Observe, that if `set`$[v]$ is updated, the set containing v *doubles* in size. As such, `set`$[v]$ is updated at most $\log 2k$ times. We conclude that the total number of updates to `set`$[\cdot]$ is is $2k \log 2k = O(k \log k)$, and this bounds the overall running time. ■

Corollary 12.2.3 *Kruskal's algorithm can be implemented in $O(m \log m)$ time.*

Proof: Sorting takes $O(m \log m)$ time, $O(m)$ finds take $O(m)$ time and $O(n)$ unions take $O(n \log n)$ time, by the above theorem. ■

12.2.4.3 Improving Worst Case Time

We are going to store a set as a tree. Every element points up to its parent in the tree, and the root of the tree points to itself, see Figure 12.9.

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set. The pointer to the root of such a tree is the set name. The operations are now implemented as follows:

(A) **find**(u): Traverse from u to the root.

(B) **union**(A, B): Make root of A (smaller set) point to root of B. Takes $O(1)$ time. Each element $u \in S$ has a pointer parent(u) to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

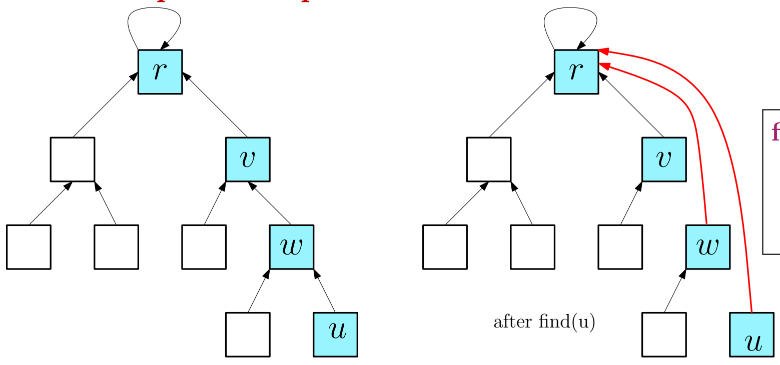
```
union(u, v):
(* parent(u) = u & parent(v) = v *)
(* if not, do u = find(u) and v = find(v) *)
    if (|set(u)| ≤ |set(v)|) then
        parent(u) = v
    else
        parent(v) = u
    update new set size to be |set(u)| + |set(v)|
```

Theorem 12.2.4 *The forest based implementation for a set of size n, has the following complexity for the various operations:* **makeUnionFind** *takes $O(n)$,* **union** *takes $O(1)$, and* **find** *takes $O(\log n)$.*

Proof: (A) **find**(u) depends on the height of tree containing u. (B) Height of u increases by at most 1 only when the set containing u changes its name. (C) If height of u increases then size of the set containing u (at least) doubles. (D) Maximum set size is n; so height of any tree is at most $O(\log n)$. ■

Observation 12.2.5 *Consecutive calls of* **find**(u) *take $O(\log n)$ time each, but they traverse the same sequence of pointers.*

Idea – ***path compression***: Make all nodes encountered in the **find**(u) point to root.



```
find(u):
    if (parent(u) ≠ u) then
        parent(u) = find(parent(u))
    return parent(u)
```

Theorem 12.2.6 *With path compression, k operations of (***find*** and/or* **union***) take overall $O(k\alpha(k, \min\{k, n\}))$ time where α is the* **inverse Ackermann function**.

The ***Ackermann function*** $A(m, n)$ is defined for $m, n \geq 0$ recursively.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

It grows amazingly fast. For example, $A(3, n) = 2^{n+3} - 3$ and $A(4, 3) = 2^{65536} - 3$. The function $\alpha(m, n)$ the ***inverse Ackermann function*** and is defined as

$$\alpha(m, n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n.\}$$

For **all practical** purposes $\alpha(m,n) \leq 5$, although it is a monotonically increasing function that goes to infinity for sufficiently larger values of its parameters.

An amazing result, due to Bob Tarjan, shows the following.

Theorem 12.2.7 *For* **Union-Find***, any data structure in the pointer model requires $O(m\alpha(m,n))$ time for m operations.*

Namely, somehow, the inverse Ackermann function is a natural phenomena.

12.2.4.4 Running time of Kruskal's Algorithm

Using Union-Find data structure:
(A) $O(m)$ **find** operations (two for each edge)
(B) $O(n)$ **union** operations (one for each edge added to T)
(C) Total time: $O(m \log m)$ for sorting plus $O(m\alpha(n))$ for union-find operations. Thus $O(m \log m)$ time despite the improved Union-Find data structure.

12.2.4.5 Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $O(n \log n + m)$.
If m is $O(n)$ then running time is $\Omega(n \log n)$.

Question: Is there a linear time ($O(m + n)$ time) algorithm for MST?
(A) $O(m \log^* m)$ time [Fredman and Tarjan '1986]
(B) $O(m + n)$ time using bit operations in RAM model [Fredman and Willard 1993]
(C) $O(m + n)$ expected time (randomized algorithm) [Karger, Klein and Tarjan '1985]
(D) $O((n + m)\alpha(m,n))$ time [Chazelle '97]
(E) Still open: is there an $O(n+m)$ time deterministic algorithm in the comparison model?