Announcements

- HW1 grading will be out around this weekend
  - Will also post the reference solution

CS6161: Design and Analysis of Algorithms (Fall 2020)

Minimum Spanning Trees

Instructor: Haifeng Xu

Outline

- MSTs and Accelerating Kruskal's Algorithm
- Prim's Algorithm and Its Acceleration

Minimum-Spanning Trees (MSTs)

Spanning Trees

A **spanning tree** of an undirected connected graph $G = (V, E)$ is a set of edges $T \subseteq E$ such that

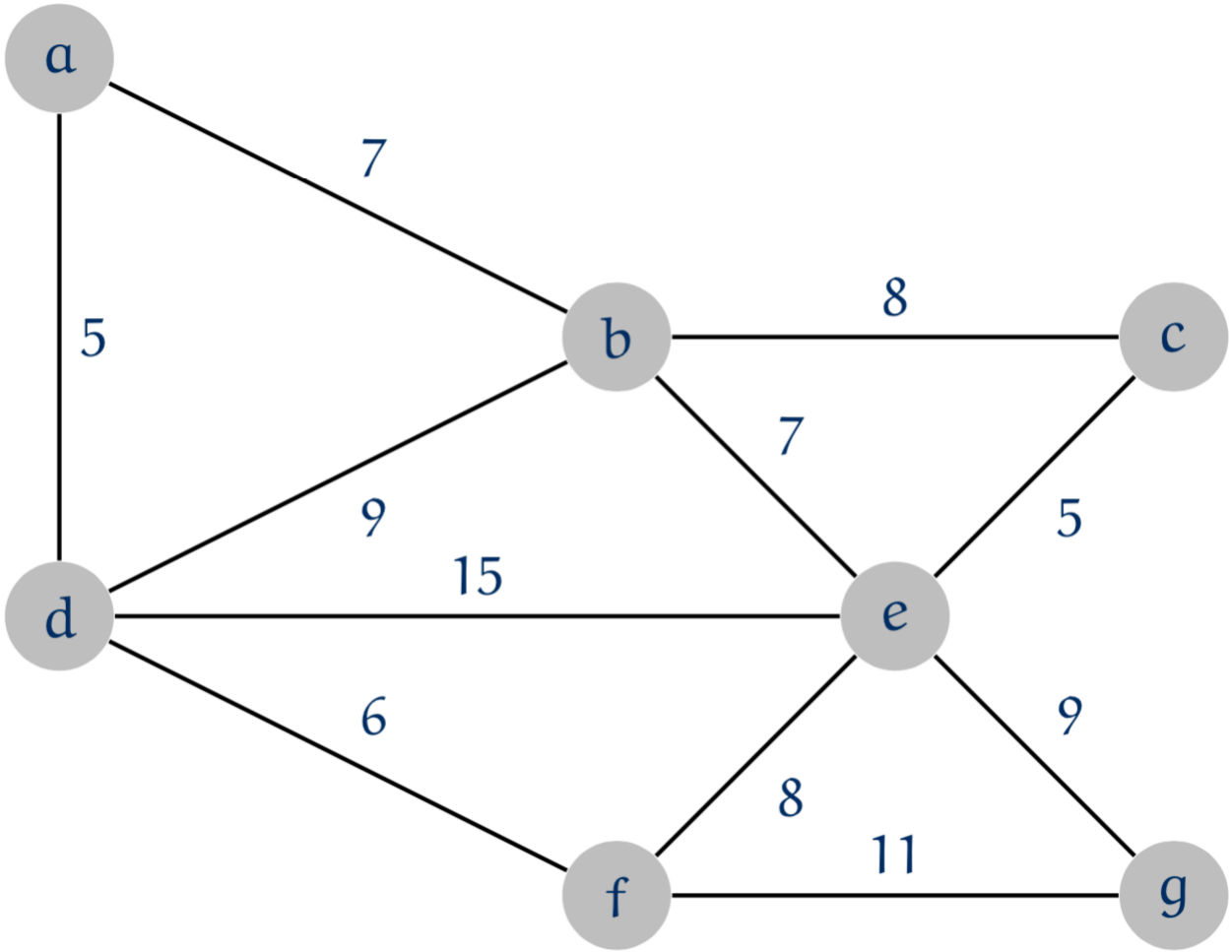
- T forms a tree, and
- (V, T) is connected.

In a weighted graph G, a **minimum spanning tree** is a spanning tree with smallest sum of edge weights

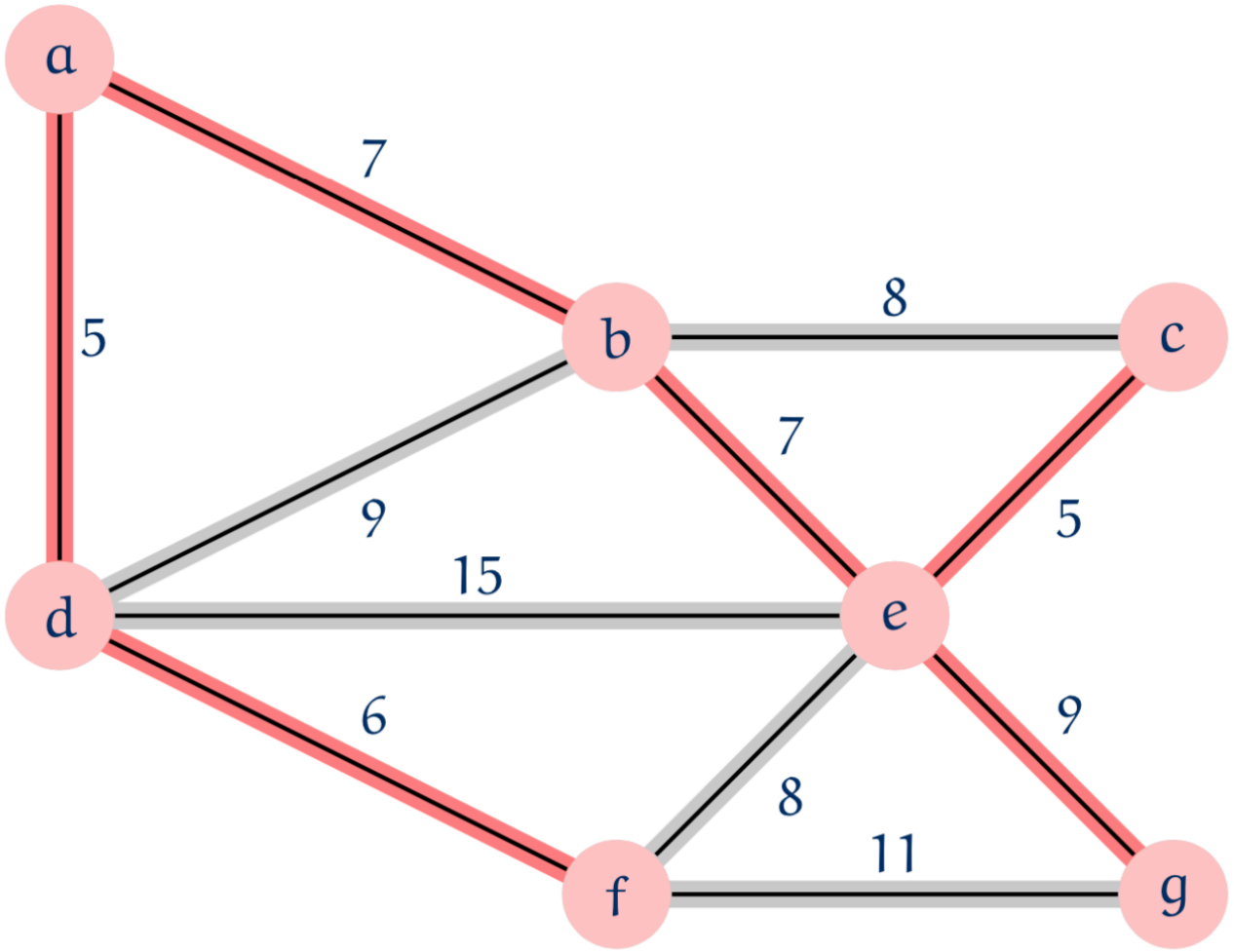
The MST problem

Find a minimum spanning tree of a weighted graph G

MST: An Example



MST: An Example



Kruskal's MST Algorithm

- Pick edges in increasing order of their costs, and add edges to T as long as they do not form a cycle

```
KruskalMST(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3    while T is not a spanning tree of G
4
5
6
7
8    return T
```

Kruskal's MST Algorithm

- Pick edges in increasing order of their costs, and add edges to T as long as they do not form a cycle

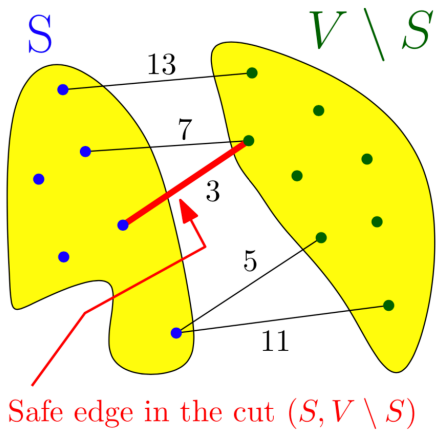
```
KruskalMST(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3    while T is not a spanning tree of G
4        choose e ∈ E of minimum cost;
5        remove e from E;
6        if T ∪ {e} does not contain cycles then
7            add e to T;
8    return T
```

Correctness

- Relies on a key property of MSTs

Theorem: Given an undirected connected graph G with distinct edge costs, the set of its safe edges form the unique MST of G.

Recall: an edge e is a safe edge if there exists a cut $(S, V \setminus S)$ such that e is the unique minimum cost edge crossing this cut.

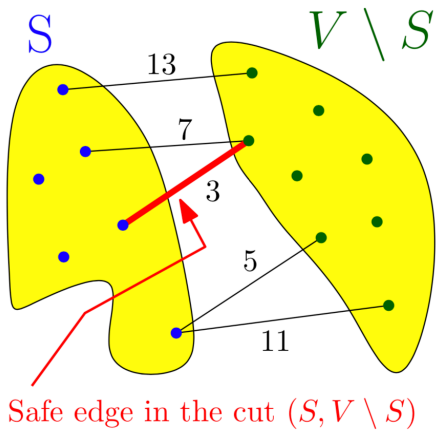


Correctness

- Relies on a key property of MSTs

Theorem: Given an undirected connected graph G with distinct edge costs, the set of its safe edges form the unique MST of G.

Proof idea: all edges added by Kruskal's algorithm are safe



Running Time

1. Sorting takes $O(m \log m)$, where $m = |E|$ times (note $m > n$)
2. While loop:
   - Executed for $m = |E|$ times
   - Step 6 takes $n = |V|$ time

In total $O(mn)$ time

```
KruskalMST(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3    while T is not a spanning tree of G
4        choose e ∈ E of minimum cost;
5        remove e from E;
6        if T ∪ {e} does not contain cycles then
7            add e to T;
8    return T
```

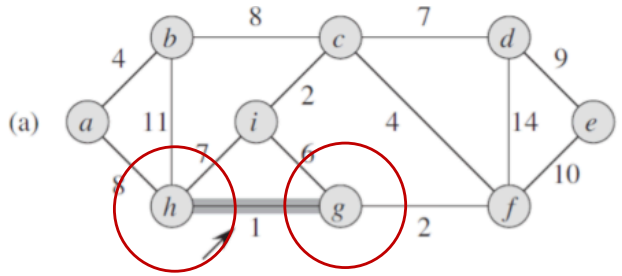
Q: Can we do better, i.e., be faster?

Yes, the same algorithm but more efficient implementation!

```
KruskalMST(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3    while T is not a spanning tree of G
4        choose e ∈ E of minimum cost;
5        remove e from E;
6        if T ∪ {e} does not contain cycles then
7            add e to T;
8    return T
```

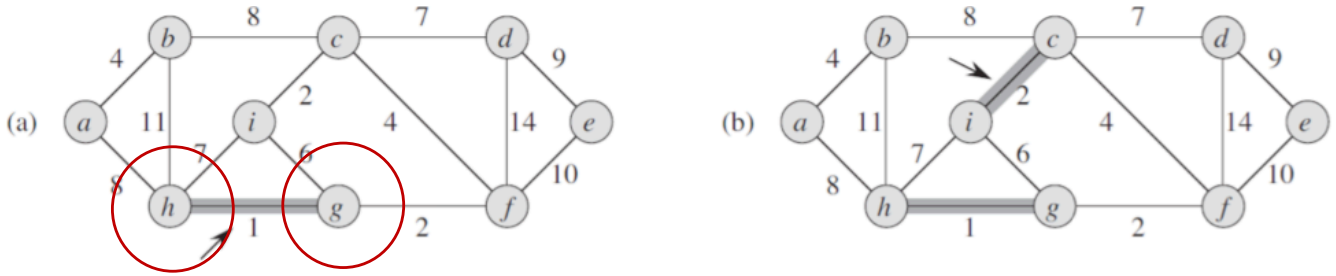
An Observation about Kruskal's Algorithm

- Each added edge "joins" two components



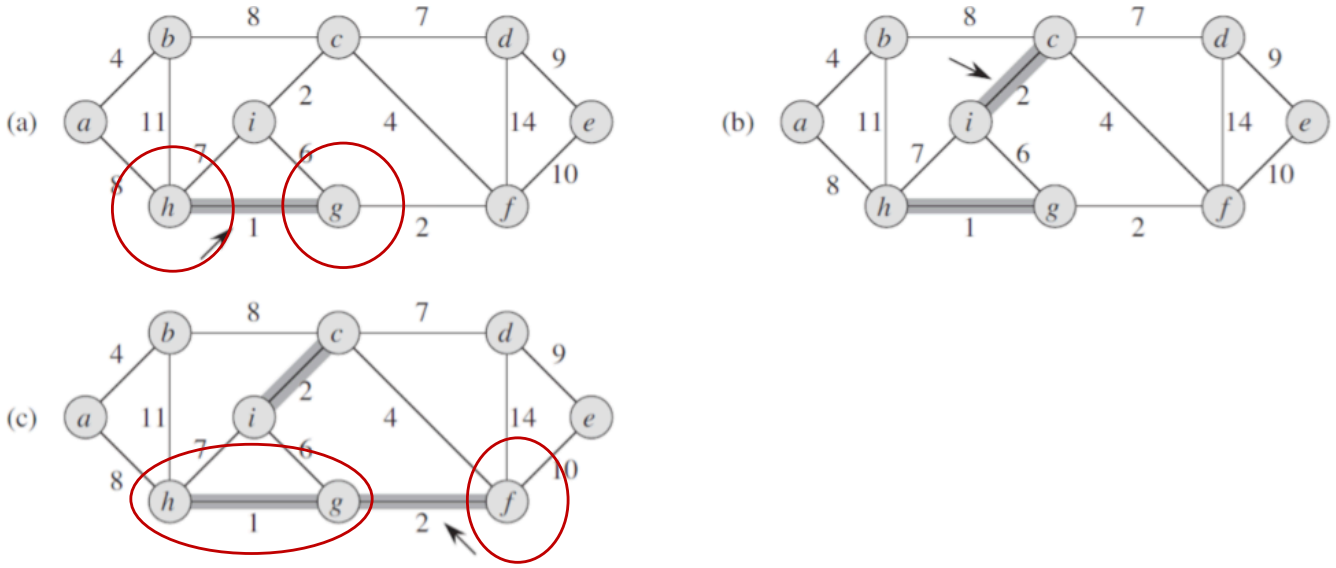
An Observation about Kruskal's Algorithm

- Each added edge "joins" two components



An Observation about Kruskal's Algorithm

- Each added edge "joins" two components



More Efficient Implementation

```
SmarterKruskal(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3
4
5
6
7
8    return T
```

More Efficient Implementation

```
SmarterKruskal(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3    Put each vertex u ∈ V into a set by itself
4
5
6
7
8    return T
```

More Efficient Implementation

```
SmarterKruskal(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;    // T will store edges of MST
3    Put each vertex u ∈ V into a set by itself
4    foreach e = (u, v) ∈ E in sorted order do
5
6
7
8    return T
```

More Efficient Implementation

```
SmarterKruskal(G)
1   Sort edges in increasing order of costs
2   Initialize T = ∅;    // T will store edges of MST
3   Put each vertex u ∈ V into a set by itself
4   foreach e = (u, v) ∈ E in sorted order do
5       if u and v belong to different sets then
6           add e to T;
7           merge the two sets containing u and v;
8   return T
```

- Need an efficient data structure to
  - Check if two elements belong to same set
  - Merge two sets

Data Structure: Union-Find

Union-Find

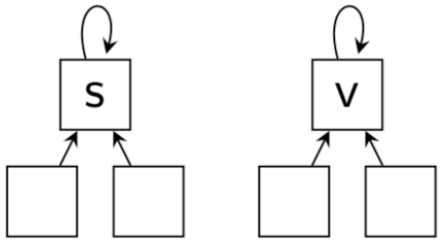
Store a set of disjoint sets with the following operations:

- Make-Set(V): generate a set $\{v\}$ for each $v \in V$. Name of set $\{v\}$ is v
- Find(u): find the name of the set containing vertex u
- Union(u, v): merge the sets named u and v. Name of the new set is either u or v.

➢ The running time of Kruskal algorithm will depend on the implementation of this data structure.

Union-Find: Implementation

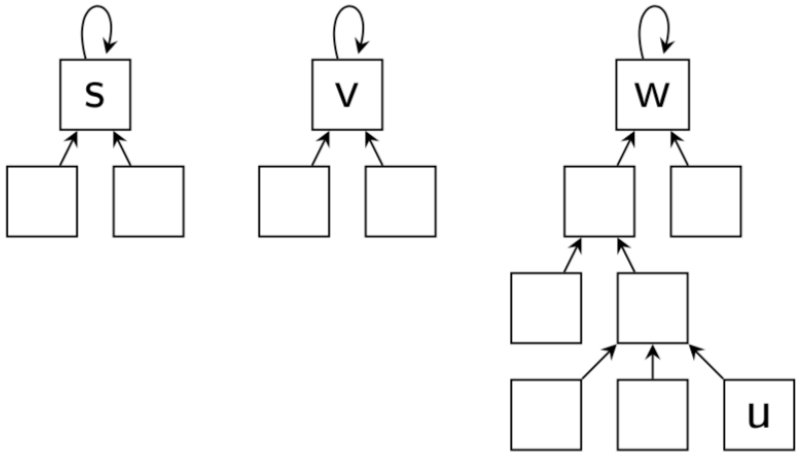
- Sets are represented as trees, by pointers towards roots. All elements in one tree belong to a set with root's name



Figure

Union-Find: Implementation

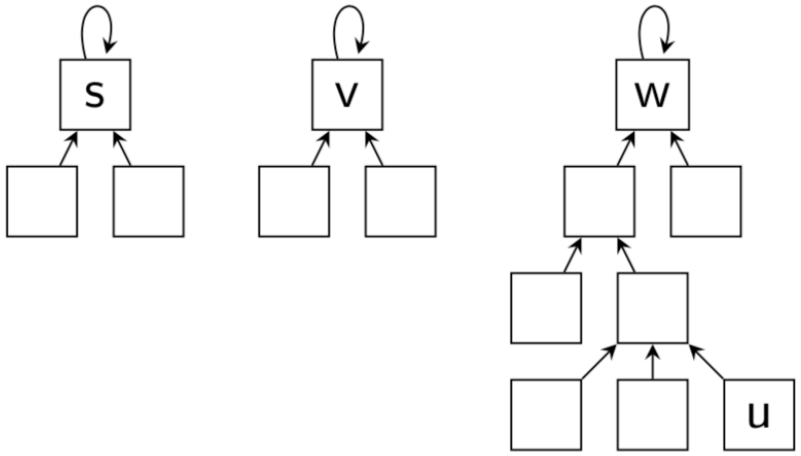
- Sets are represented as trees, by pointers towards roots. All elements in one tree belong to a set with root's name.
  - Find(u): Traverse from u to the root



Figure

Union-Find: Implementation

- Sets are represented as trees, by pointers towards roots. All elements in one tree belong to a set with root's name.
  - Find(u): Traverse from u to the root
  - Union(u, v): Make root of u (smaller set) point to root of v. Takes $O(1)$ time



Figure

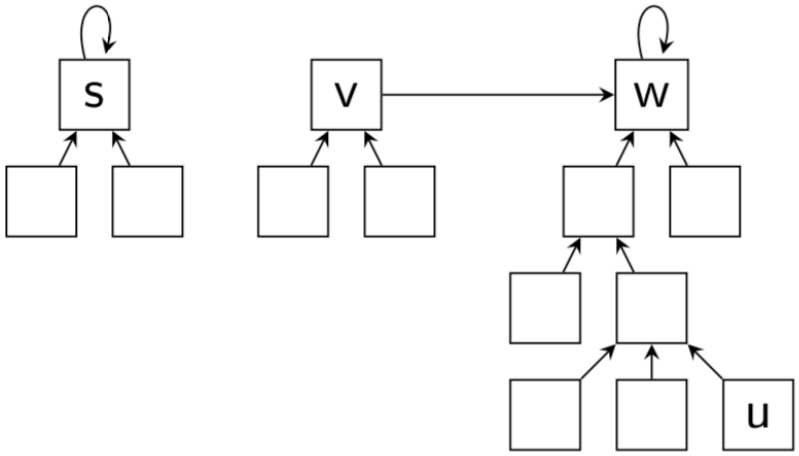
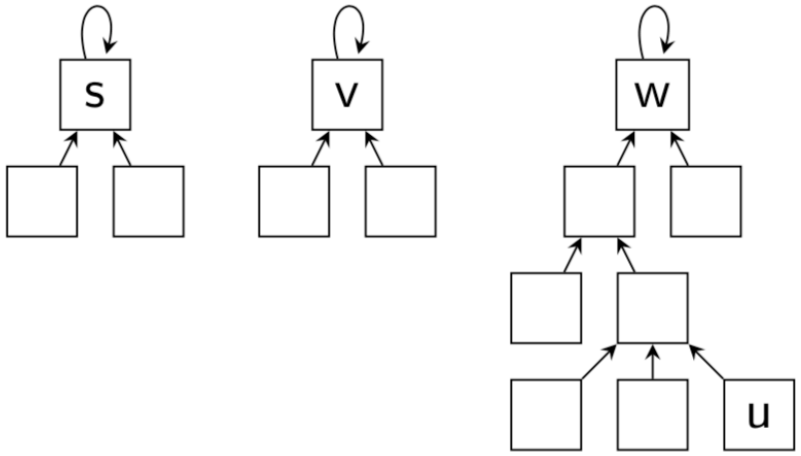


Figure: Union(Find(v), Find(u))

Union-Find: Implementation

- Sets are represented as trees, by pointers towards roots. All elements in one tree belong to a set with root's name.
  - Find(u): Traverse from u to the root
  - Union(u, v): Make root of u (smaller set) point to root of v. Takes $O(1)$ time.
- Each vertex u has a pointer parent(u) to its parent



Figure

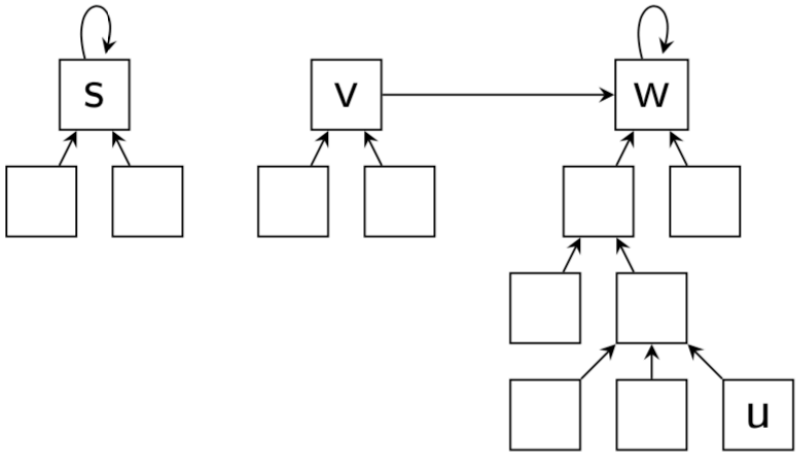


Figure: Union(Find(v), Find(u))

Union-Find: Implementation (FYI)

Algorithm: Make-Set(G):

foreach $u \in V$ **do**

parent(u) = u;

Algorithm: Find(u):

while parent(u) $\neq$ u **do**

u = parent(u);

return u

Algorithm: Union(u, v):

(* parent(u) = u & parent(v) = v *)

if |component(u)| $\leq$ |component(v)| **then**

parent(u) = v

else

parent(v) = u

set new componen size to |component(u) + component(v)|.

Union-Find: Time Analysis

- Make-Set(V): $O(n)$
- Union(u, v): $O(1)$ time
- Find(u): in $O(depth\ of\ the\ tree)$ time

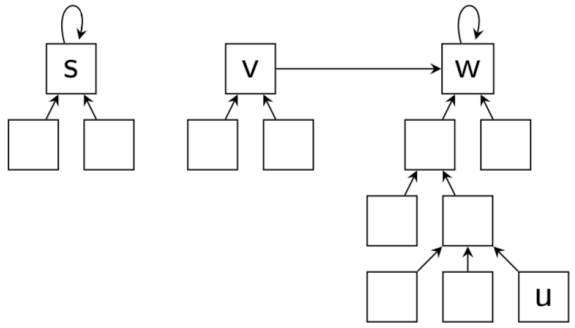


Figure: Union(Find(v), Find(u))

Fact: The maximum depth of trees in union-find is $O(\log n)$.

Proof:

- $Depth(v)$ increase by at most 1 only when the set containing v changes its name
- If $Depth(v)$ increases, then size of the set containing v at least doubles
- Maximum set size is n; so depth of any node is at most $O(\log n)$.

Union-Find: Time Analysis

- Make-Set(V): $O(n)$
- Union(u, v): $O(1)$ time
- Find(u): in $O(\log n)$ time

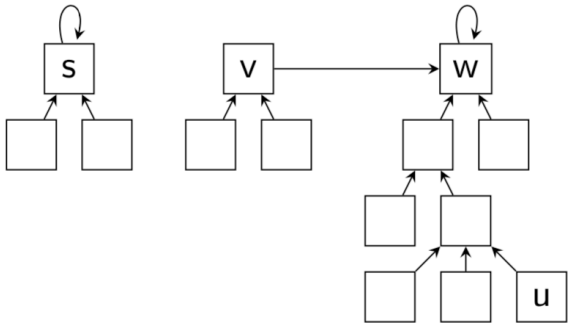


Figure: Union(Find(v), Find(u))

Fact: The maximum depth of trees in union-find is $O(\log n)$.

Proof:

- $Depth(v)$ increase by at most 1 only when the set containing v changes its name
- If $Depth(v)$ increases, then size of the set containing v at least doubles
- Maximum set size is n; so depth of any tree is at most $O(\log n)$.

Running Time Analysis: SmarterKruskal

$O(m \log m)$

SmarterKruskal(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Put each vertex $u \in V$ into a set by itself → Make-Set(V): $O(n)$

4 **foreach** $e = (u, v) \in E$ in sorted order **do** → repeat m times

5 **if** u and v belong to different sets **then**

6 add e to T;

7 merge the two sets containing u and v; → Union(u, v) $O(1)$ time

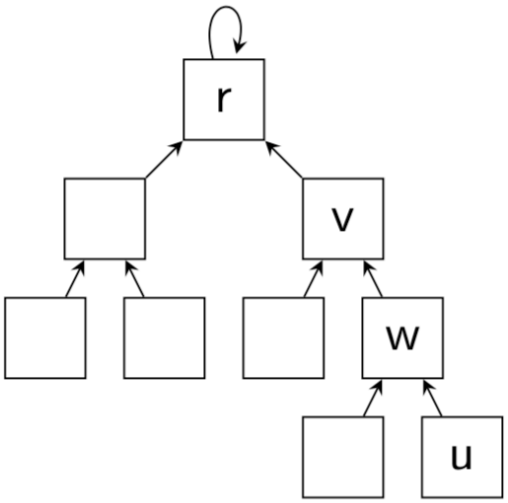
8 **return** T

Find(u), Find(v) and then compare: $O(\log n)$

In total: $O(m \log m) + m \times O(\log n) = O(m \log m)$

Can Even Further Speed up!

- Find(u) can be done in almost $O(1)$ time
- Main idea: point directly to the root
- Slightly better, but cannot lead to order-wise improvement since sorting is the bottleneck, and takes $O(m \log m)$.



Figure

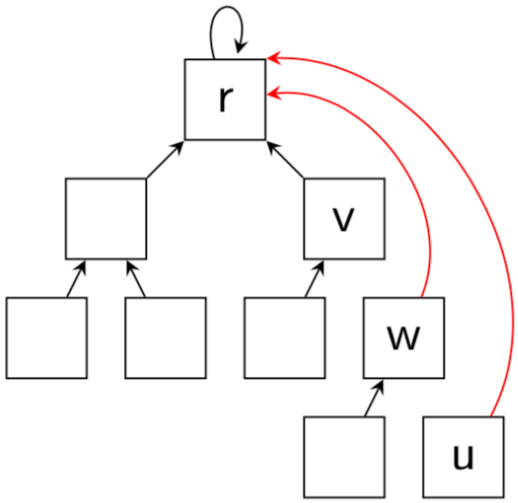


Figure: After Find(u)

Outline

- MSTs and Accelerating Kruskal's Algorithm
- Prim's Algorithm and Its Acceleration

Recall Our Greedy Framework

```
SomeGreedyMSTAlgo(G)
1   Initialize T = ∅;   // T will store edges of MST
2   while T is not a spanning tree of G
3       choose e ∈ E that "improves" solution;
4       add e to T
5   return T
```

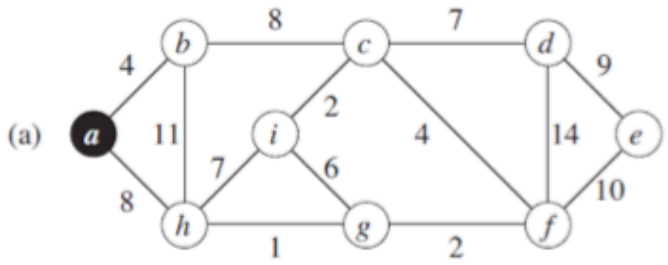
Prim's Algorithm uses a different Greedy Procedure to add edges

- T maintained by the algorithm will remain a tree
- In each iteration, pick edges with least attachment cost to T.

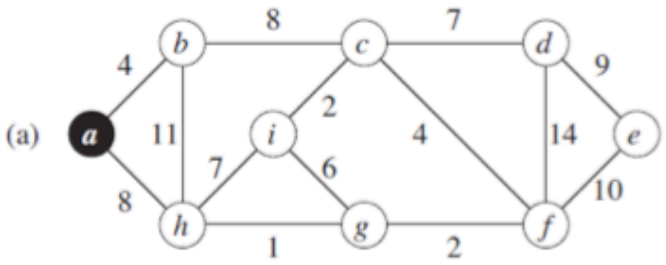
Prim's Algorithm

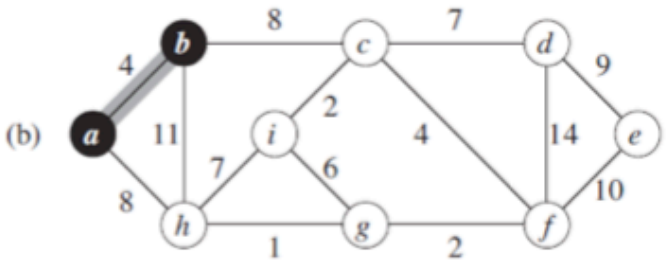
```
PrimMST(G)
1    Sort edges in increasing order of costs
2    Initialize T = ∅;          // T stores edges of MST
3    Initialize S = {1};        // S stores nodes of MST
4    while |S| < n              // i.e., T is not a ST
5        choose e = (u, v) ∈ E of minimum cost, such
         that u ∈ S and v ∈ V \ S;
6        T = T ∪ {e};
7        S = S ∪ {v};
8    return T
```

Prim's Algorithm: Example Execution

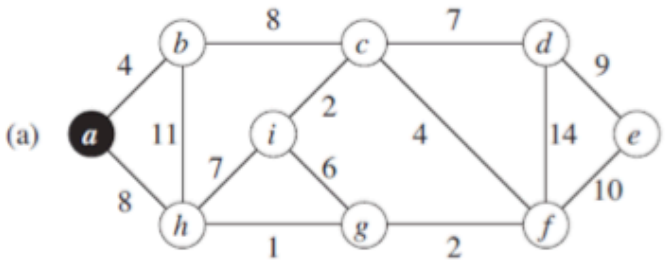


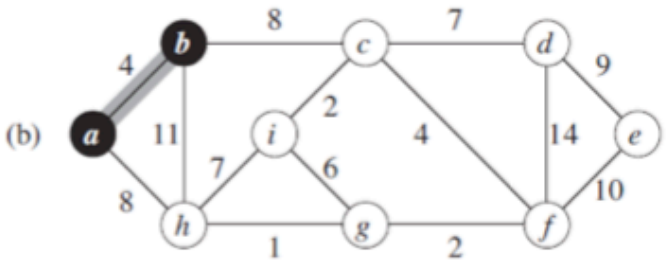
Prim's Algorithm: Example Execution

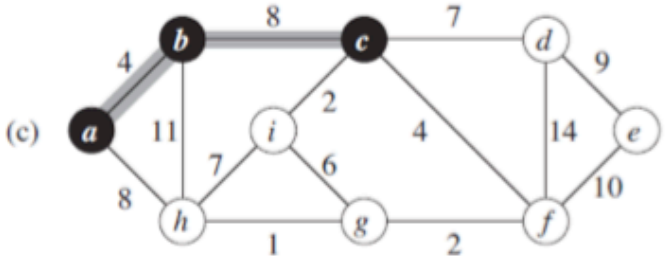




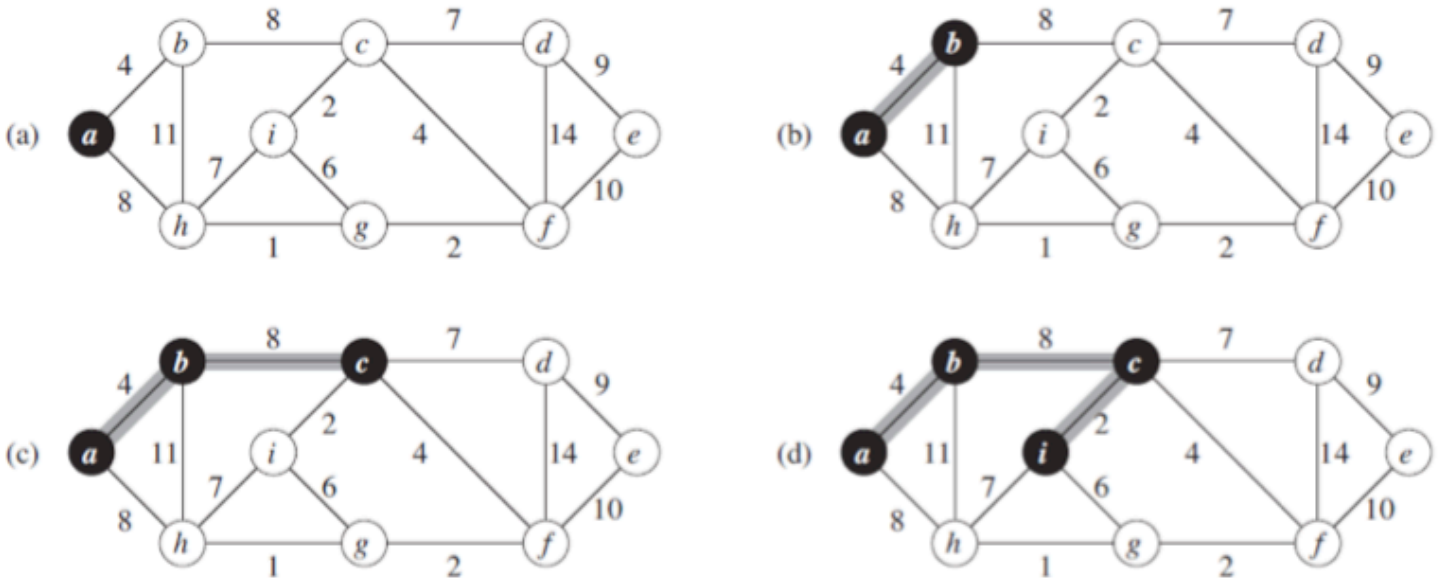
Prim's Algorithm: Example Execution



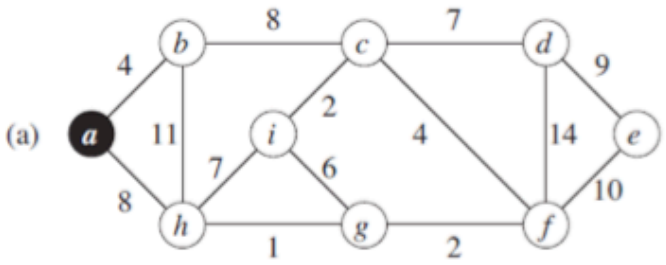


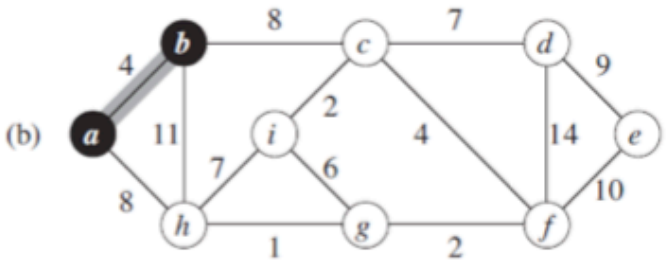


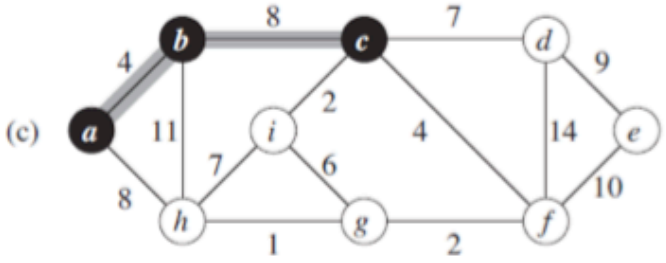
Prim's Algorithm: Example Execution

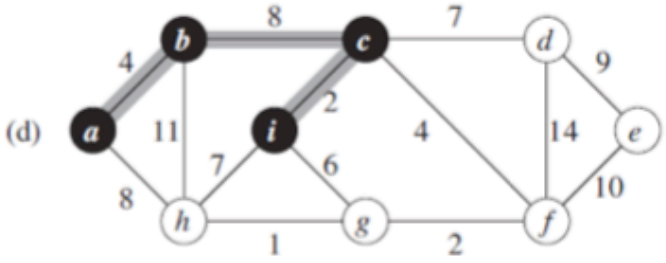


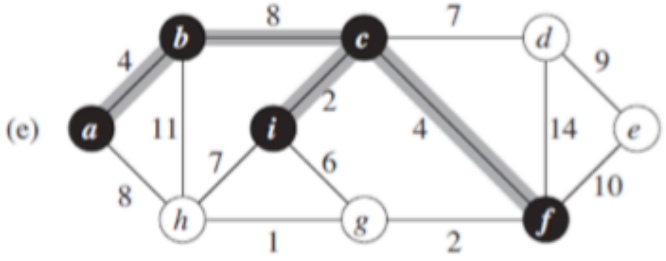
Prim's Algorithm: Example Execution



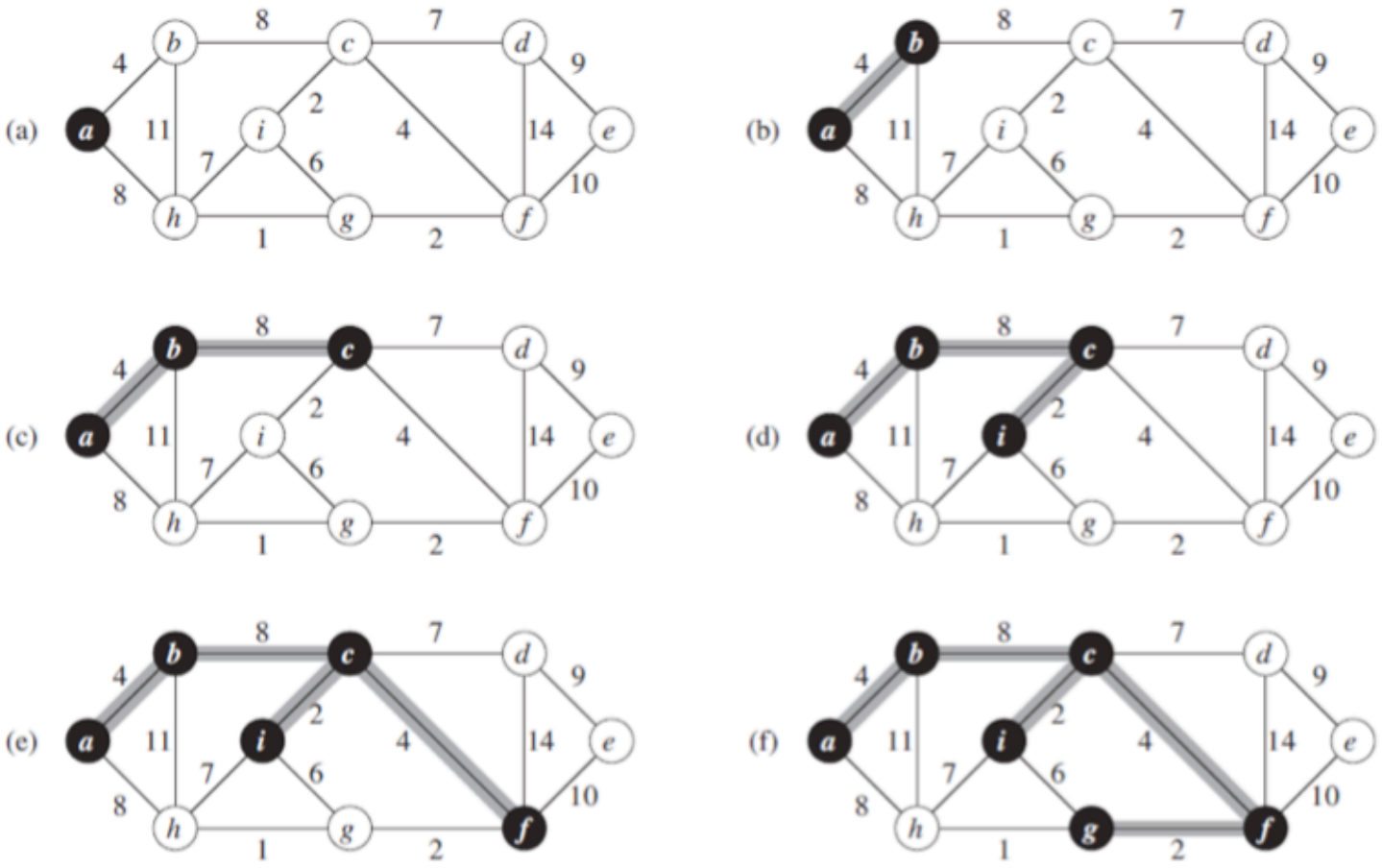






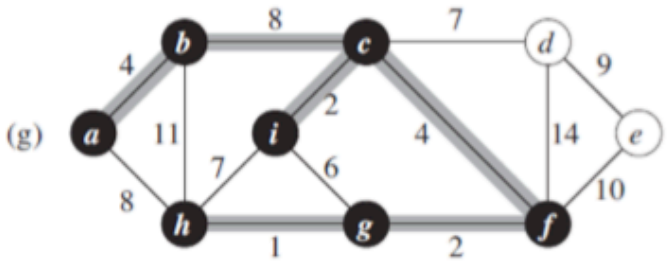


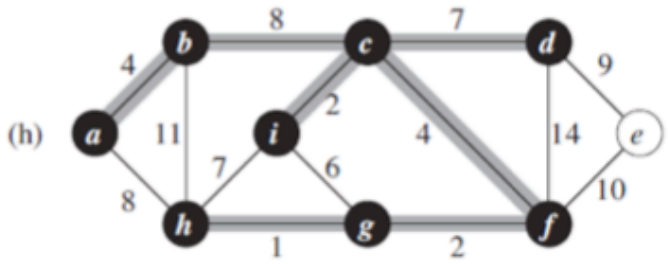
Prim's Algorithm: Example Execution

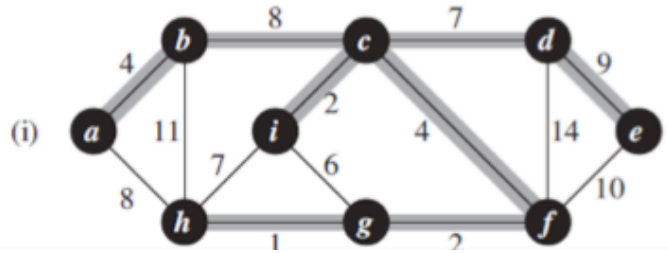


Prim's Algorithm: Example Execution

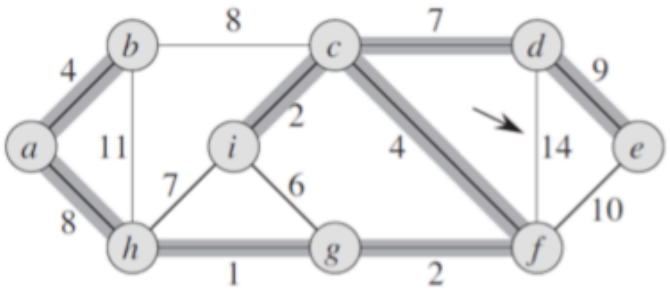
Similar remaining steps…







Compare with the MST by Kruskal's Algorithm



Correctness of Prim's Algorithm

- Want to show that all added edges are safe edges
- Again, assume edges all have different costs (not essential)

Proof of Correctness

- If e is added to the tree, then e is safe
  - Let S be the vertices connected by edges in T when e is added
  - e is the minimum cost edge crossing cut $(S, V \setminus S)$.
- S is connected in each iteration and eventually $S = V$.

Prim's Algorithm: Time Complexity

PrimMST(G)

1 Sort edges in increasing order of costs — $O(m \log m)$

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **while** $|S| < n$ — Repeat n times

5 choose $e = (u, v) \in E$ of minimum cost, such that $u \in S$ and $v \in V \setminus S$;

6 $T = T \cup \{e\}$;

7 $S = S \cup \{v\}$;

8 **return** T

$O(m)$ times

In total: $O(m \log m) + n \times O(m) = O(mn)$

Prim's Algorithm: Time Complexity

PrimMST(G)

1 Sort edges in increasing order of costs — $O(m \log m)$

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **while** $|S| < n$ — Repeat n times

5 choose $e = (u, v) \in E$ of minimum cost, such that $u \in S$ and $v \in V \setminus S$;

6 $T = T \cup \{e\}$;

7 $S = S \cup \{v\}$;

8 **return** T

$O(m)$ times

Q: Can we do better?

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

// $a(u)$ stores the cheapest edge from S to u

About notation "$\arg\min$"

If $x = 4$ minimizes $(x-2)^2$ subject to $x \geq 4$, we say $4 = \arg\min_{x \geq 4}(x-2)^2$

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

// $a(u)$ stores the cheapest edge from S to u

5 **while** $|S| < n$

6

7

8

9

10

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

// $a(u)$ stores the cheapest edge from S to u

5 **while** $|S| < n$

6 pick *minimum* $a(u) = (u, v)$;

7 $T = T \cup \{a(u)\}$;

8 $S = S \cup \{u\}$;

9

10

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

// $a(u)$ stores the cheapest edge from S to u

5 **while** $|S| < n$

6 pick *minimum* $a(u) = (u, v)$;

7 $T = T \cup \{a(u)\}$;

8 $S = S \cup \{u\}$;

9 update array a;

10 **return** T

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

// $a(u)$ stores the cheapest edge from S to u

5 **while** $|S| < n$

6 pick *minimum* $a(u) = (u, v)$;

7 $T = T \cup \{a(u)\}$;

8 $S = S \cup \{u\}$;

9 update array a;

10 **return** T

Running time depends on implementation of highlighted steps

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

5 **while** $|S| < n$

6 pick *minimum* $a(u) = (u, v)$;

7 $T = T \cup \{a(u)\}$;

8 $S = S \cup \{u\}$;

9 update array a;

10 **return** T

- Implementation of Step 4
  - For each u, check all edges that connect u to S
  - Each edge is checked for at most twice
  - In total, $O(m)$ time

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg \min_{e=(u,v),v \in S} c_e$

5 **while** $|S| < n$

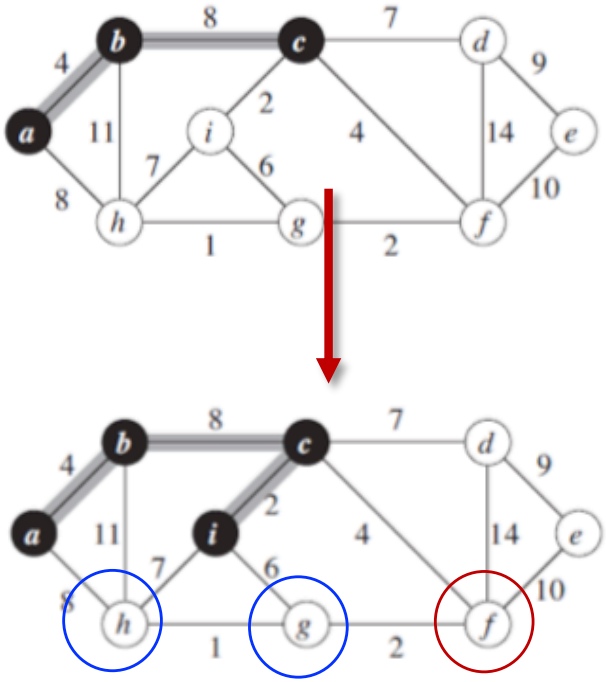
6 pick *minimum* $a(u) = (u, v)$;

7 $T = T \cup \{a(u)\}$;

8 $S = S \cup \{u\}$;

9 update array a;

10 **return** T



$a(f)$ will not change

- Implementation of Step 9: how to update $a(v)$ for any $v \notin S$?
  - First of all, only need to update the v that connects to u
  - For these v's, simply compare $c_{a(v)}$ and c_{uv}
  - During entire "while loop", each edge is considered at most once, so update $a(u)$ for at most m times

More Efficient Implementation

SmarterPrim(G)

1 Sort edges in increasing order of costs

2 Initialize $T = \emptyset$;

3 Initialize $S = \{1\}$;

4 **for** $u \notin S$, initialize $a(u) = \arg\min_{e=(u,v),v\in S} c_e$

5 **while** $|S| < n$

6 pick *minimum* $a(u) = (u, v)$;

7 $T = T \cup \{a(u)\}$;

8 $S = S \cup \{u\}$;

9 update array a;

10 **return** T

- Implementation of Step 6: which data structure is efficient for repeatedly retrieving min?
  - Priority queue!

Priority Queues

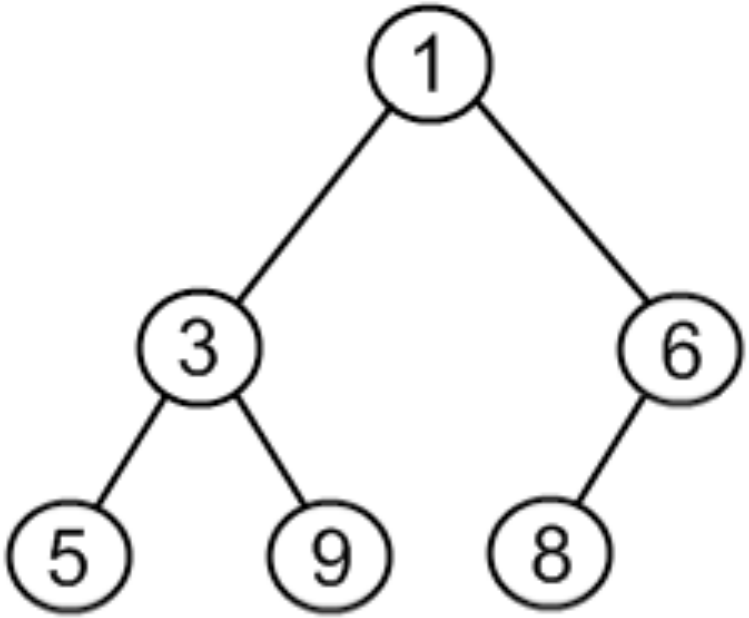
Store a set S of n elements, where each element $v \in S$ has an associated key value $k(v)$, with the following operations:

- Make-Queue: create an empty queue
- Find-Min: find the minimum key in S
- Extract-Min: remove $v \in S$ with minimum key and return it
- Decrease-Key$(v, k'(v))$: decrease key of v from $k(v)$ to $k'(v)$
- Add$(v, k(v))$: add new element v with key $k(v)$ to S

- Prim requires
  - $O(n)$ Extract-Min for pick *minimum* $a(u)$
  - $O(m)$ Decrease-Key for update array a (whenever u added to S, only update u's neighbors' key value)
- Binary heap implementation: both operations take $O(\log n)$
- Total time $O\big((m + n)\log n\big) = O(m \log m)$

An Example of Priority Queue via Binary Heap

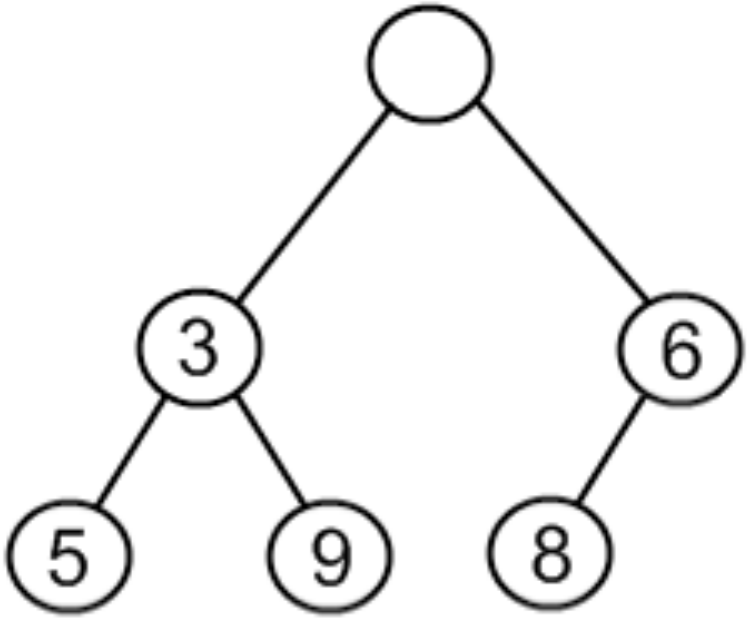
Extract-Min



Main property: any parent is smaller than both children

An Example of Priority Queue via Binary Heap

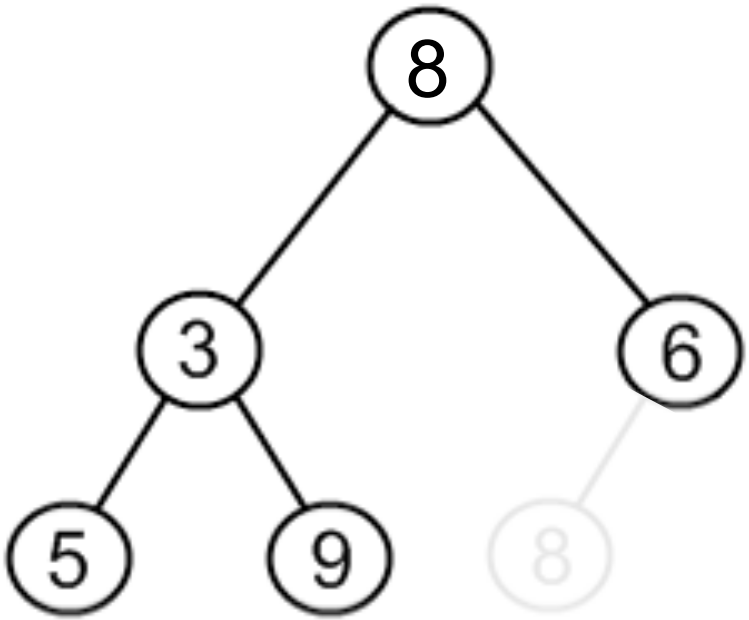
Extract-Min



Main property: any parent is smaller than both children

An Example of Priority Queue via Binary Heap

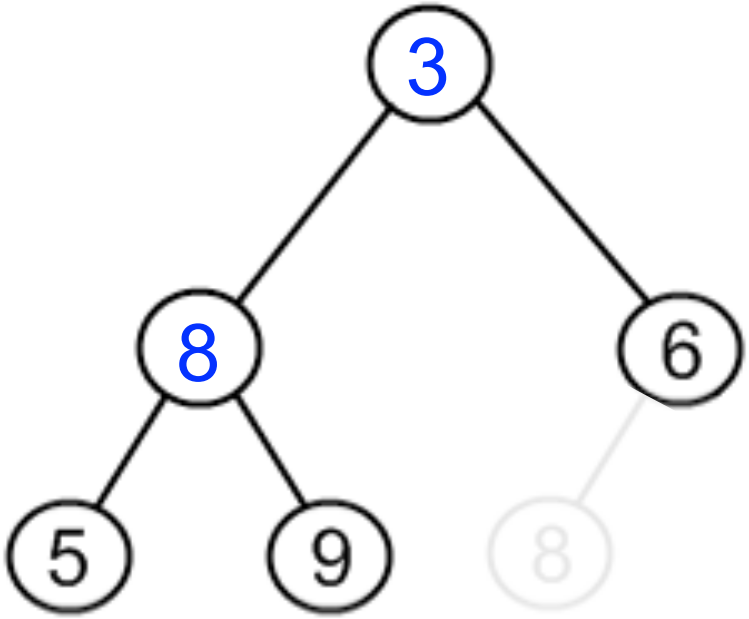
Extract-Min



Main property: any parent is smaller than both children

An Example of Priority Queue via Binary Heap

Extract-Min

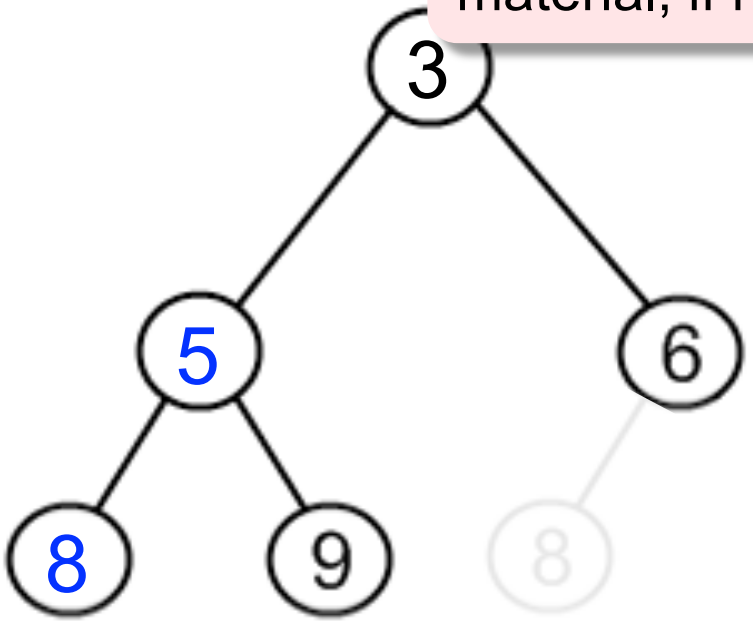


Main property: any parent is smaller than both children

An Example of Priority Queue via Binary Heap

Extract-Min

Please review data structure material, if not appear familiar to you



Main property: any parent is smaller than both children

More About MSTs

- Many different MST algorithms; all of them rely on some basic properties of MSTs
  - There is a randomized algorithm that runs in $O(m)$ expected time
- Still open: Is there an $O(m)$ time deterministic algorithm?

Lessons Learned

- One problem may have different greedy algorithms that all work
- The design of proper data structure may help to speed up algorithms
  - So, algorithm design is *not only* about inventing the algorithm *but also* about efficient implementation of the algorithm

Thank You

Haifeng Xu

University of Virginia

hx4ad@virginia.edu