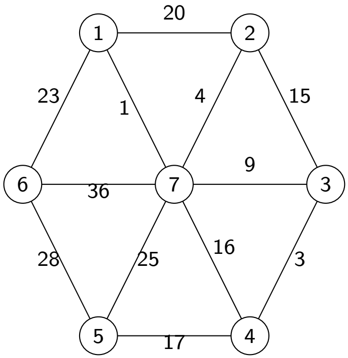
CMP_SC 3050: More greedy algorithms in graphs: minimum spanning trees

Minimum Spanning Tree

Input Connected undirected graph $G = (V, E)$ with edge costs/weights $w(i, j)$

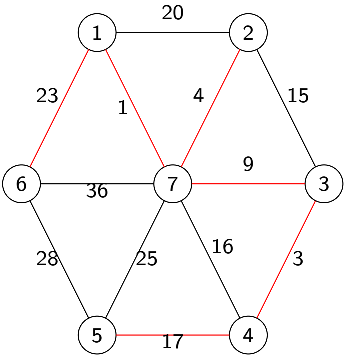
Goal Find $T \subseteq E$ such that (V, T) is connected and total cost/weight of all edges in T is smallest



Minimum Spanning Tree

Input Connected undirected graph $G = (V, E)$ with edge costs/weights $w(i, j)$

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost/weight of all edges in T is smallest

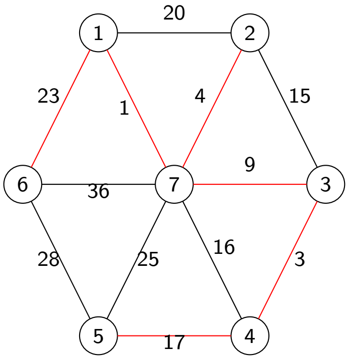


Minimum Spanning Tree

Input Connected undirected graph $G = (V, E)$ with edge costs/weights $w(i, j)$

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost/weight of all edges in T is smallest

- T is the minimum spanning tree (MST) of G



Difference between MSTs and Shortest Paths

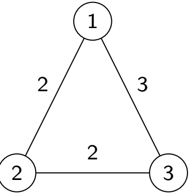


Figure: Graph G

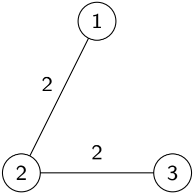


Figure: MST of G

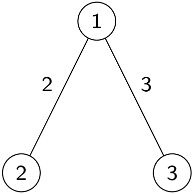


Figure: Shortest Paths (from 1)

Applications

- Network Design
  - Designing networks (roads, computer, electrical) with minimum cost but maximum connectivity
- Approximation algorithms for computationally hard problems
  - Can be used to bound the optimality of algorithms to approximate Travelling Salesman Problem, Steiner Trees, etc.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network

Greedy Template

```
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
    choose i ∈ E
    if (i satisfies condition)
        add i to T
return the set T
```

Main Task: In what order should edges be processed? When should we add edge to spanning tree?

KA PA

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

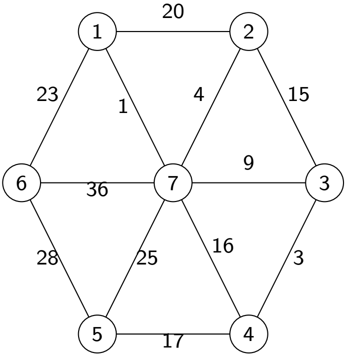


Figure: Graph G

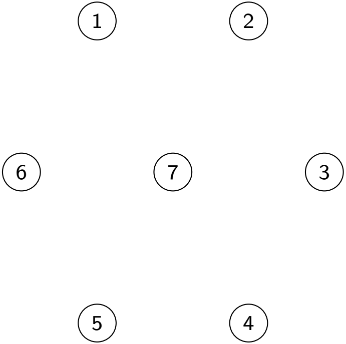


Figure: MST of G

Back

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

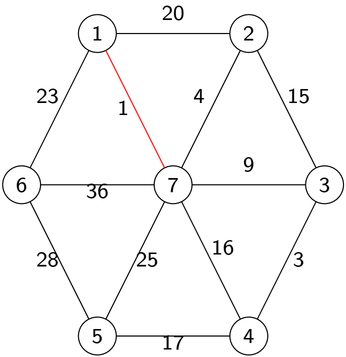


Figure: Graph G

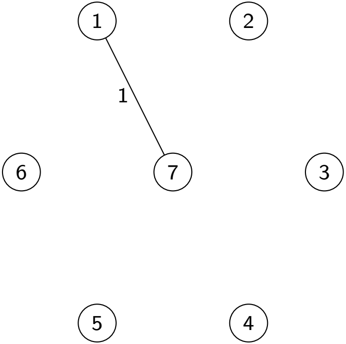


Figure: MST of G

Back

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

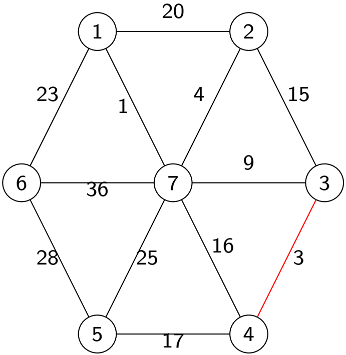


Figure: Graph G

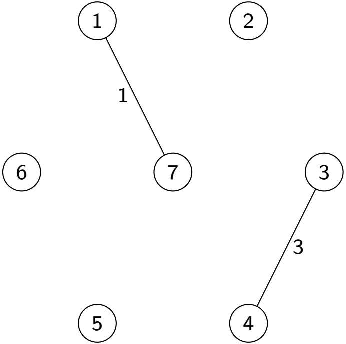


Figure: MST of G

Back

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

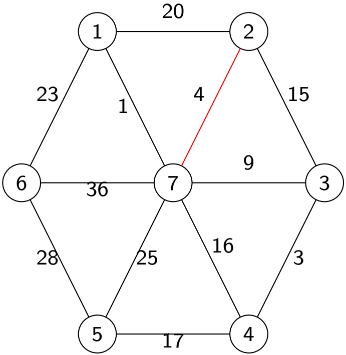


Figure: Graph G

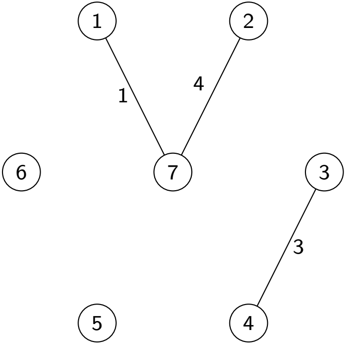


Figure: MST of G

Back

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

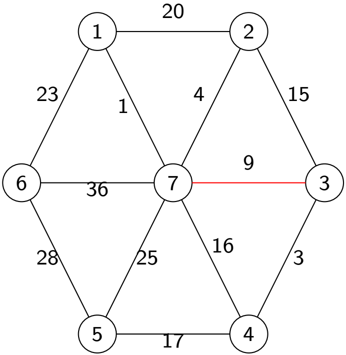


Figure: Graph G

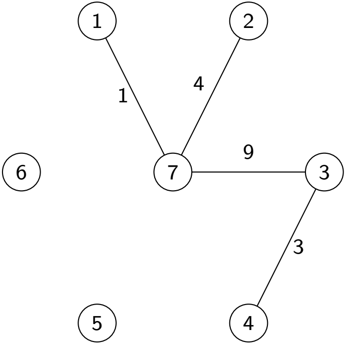


Figure: MST of G

Back

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

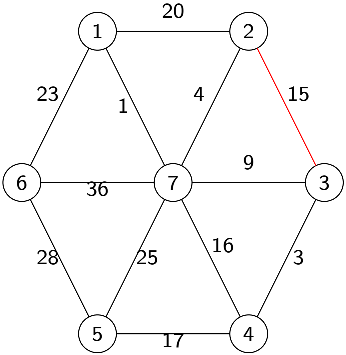


Figure: Graph G

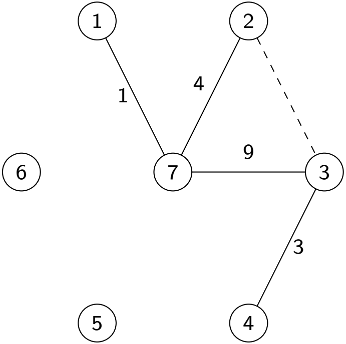


Figure: MST of G

Back

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

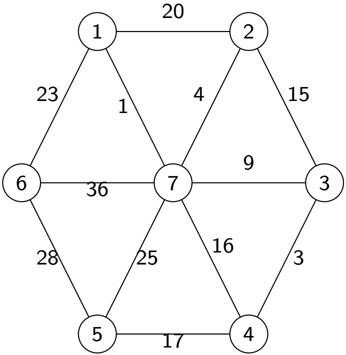


Figure: Graph G

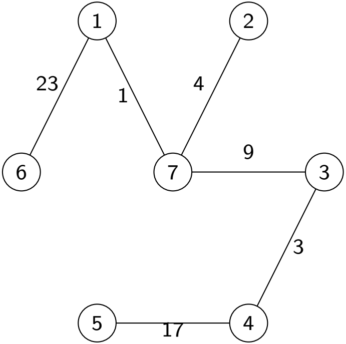


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

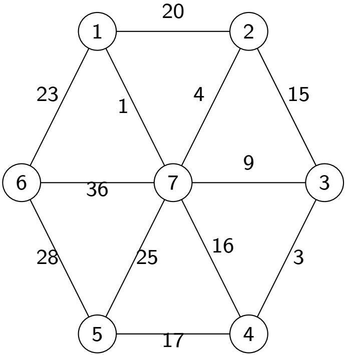


Figure: Graph G

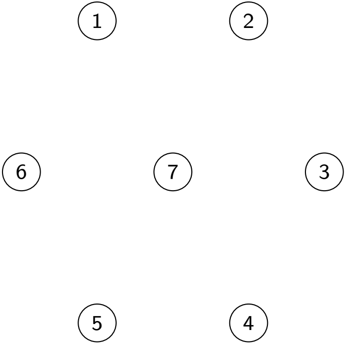


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

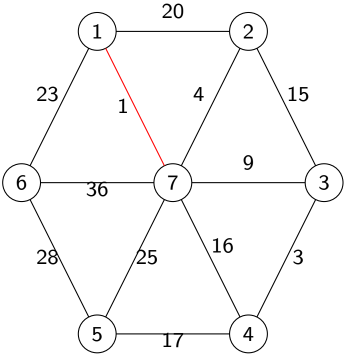


Figure: Graph G

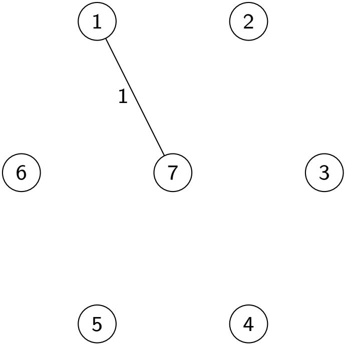


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

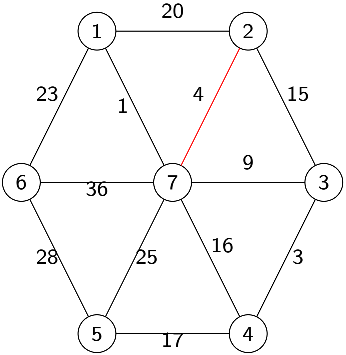


Figure: Graph G

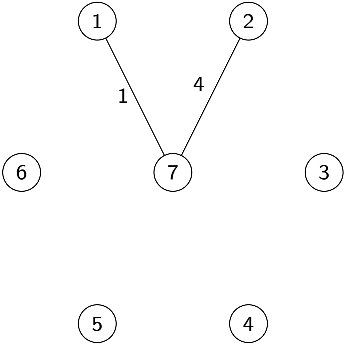


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

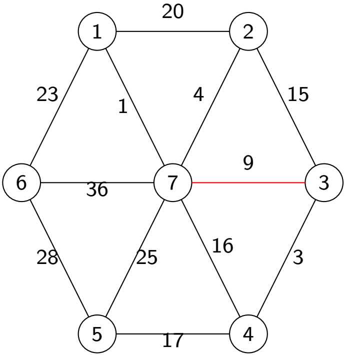


Figure: Graph G

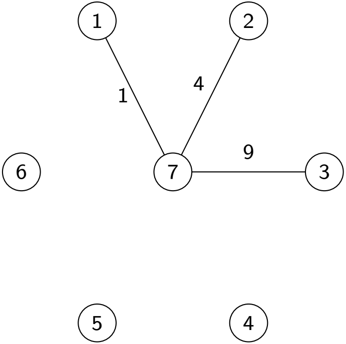


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

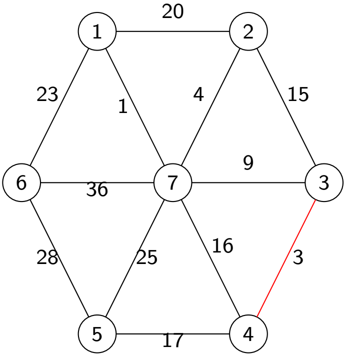


Figure: Graph G

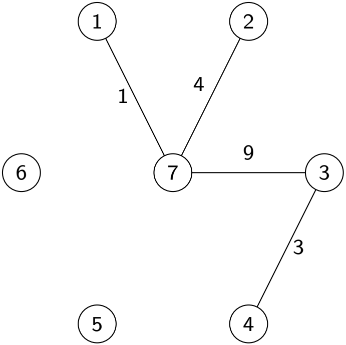


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

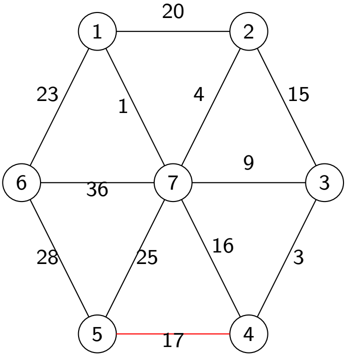


Figure: Graph G

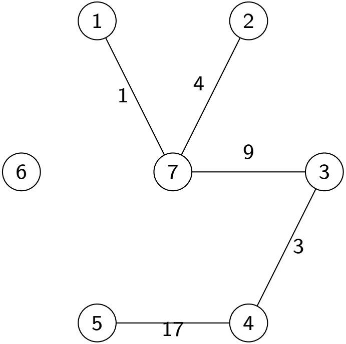


Figure: MST of G

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Can start with any vertex. In each iteration, pick edge with least attachment cost to T.
In other words, consider all the edges that have exactly one endpoint in T and choose the one with the least weight amongst these edges.

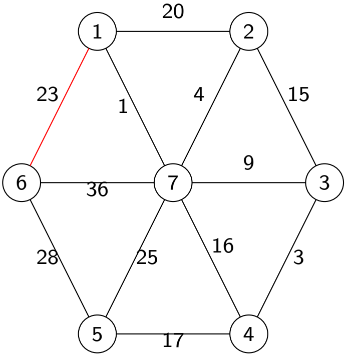


Figure: Graph G

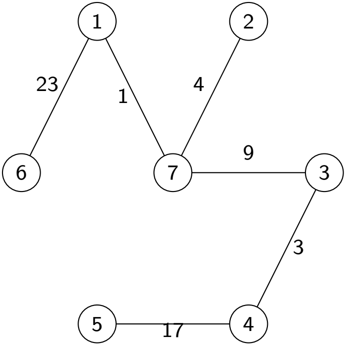


Figure: MST of G

Back

And for now . . .

No two edge costs are equal.

Key Observation: Cut Property

Let S be a proper non-empty subset of V. Let $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V \setminus S$. Then every minimum spanning tree contains e

Key Observation: Cut Property

Let S be a proper non-empty subset of V. Let $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V \setminus S$. Then every minimum spanning tree contains e

- Suppose (for contradiction) e is not in MST T.

Key Observation: Cut Property

Let S be a proper non-empty subset of V. Let $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V \setminus S$. Then every minimum spanning tree contains e

- Suppose (for contradiction) e is not in MST T.
- Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$

Key Observation: Cut Property

Let S be a proper non-empty subset of V. Let $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V \setminus S$. Then every minimum spanning tree contains e

- Suppose (for contradiction) e is not in MST T.
- Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
- Since weight of e is smaller than weight of f, then we can remove f from T and add e to get a spanning tree T' of lower cost

Key Observation: Cut Property

Let S be a proper non-empty subset of V. Let $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V \setminus S$. Then every minimum spanning tree contains e

- Suppose (for contradiction) e is not in MST T.
- Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
- Since weight of e is smaller than weight of f, then we can remove f from T and add e to get a spanning tree T' of lower cost

Key Observation: Cut Property

Let S be a proper non-empty subset of V. Let $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V \setminus S$. Then every minimum spanning tree contains e

- Suppose (for contradiction) e is not in MST T.
- Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
- Since weight of e is smaller than weight of f, then we can remove f from T and add e to get a spanning tree T' of lower cost
  - T' may not be a spanning tree!!

Error in the argument: Example

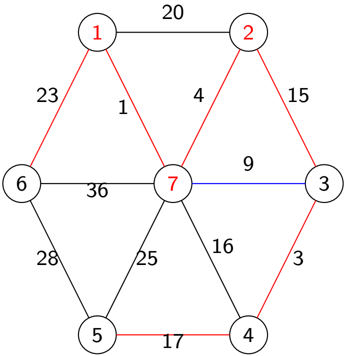


Figure: Problematic example: The Spanning Tree is shown in red. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$.

Error in the argument: Example

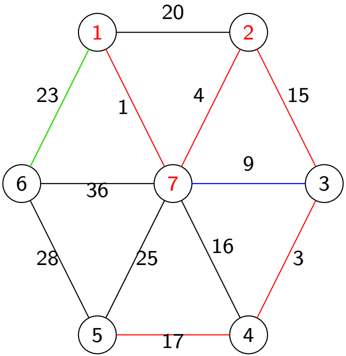
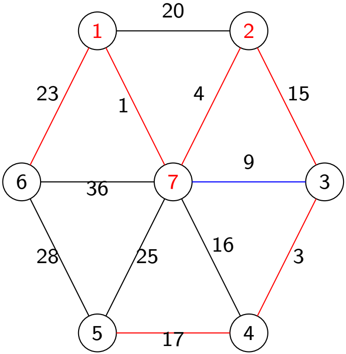


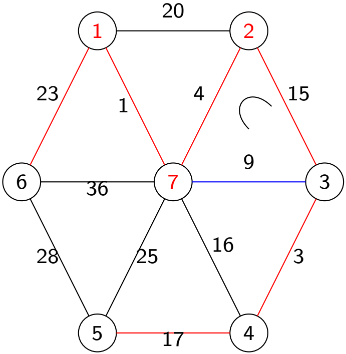
Figure: Problematic example: The Spanning Tree is shown in red. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. Replacing f by e yields a cycle and does not cover 6.

Why the Cut Property holds?



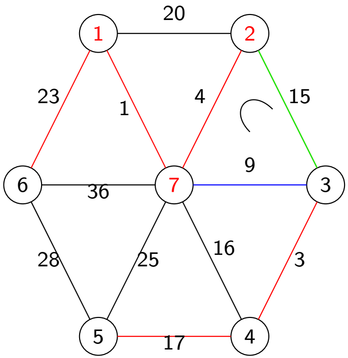
- Suppose minimum $(S, V \setminus S)$-cut edge $e = (v, w)$ is not in MST T.

Why the Cut Property holds?



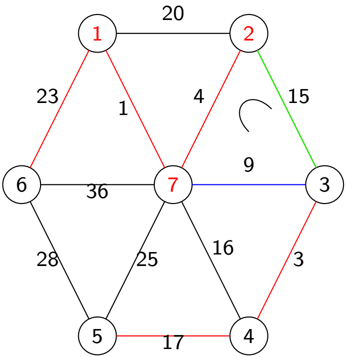
- Suppose minimum $(S, V \setminus S)$-cut edge $e = (v, w)$ is not in MST T.
- Since T is connected, there is some path (say P) from v to w in T

Why the Cut Property holds?



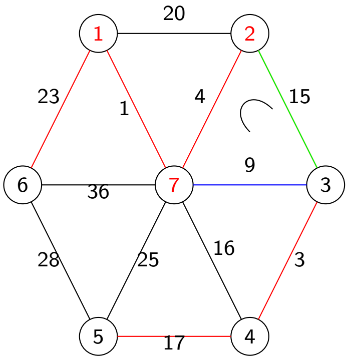
- Suppose minimum $(S, V \setminus S)$-cut edge $e = (v, w)$ is not in MST T.
- Since T is connected, there is some path (say P) from v to w in T
- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$.

Why the Cut Property holds?



- Suppose minimum $(S, V \setminus S)$-cut edge $e = (v, w)$ is not in MST T.
- Since T is connected, there is some path (say 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 $e' = (v', w')$.
- In the example on the left, e' is the edge $(2, 3)$

Why the Cut Property holds?



- Suppose minimum $(S, V \setminus S)$-cut edge $e = (v, w)$ is not in MST T.
- Since T is connected, there is some path (say 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 $e' = (v', w')$.
- In the example on the left, e' is the edge $(2, 3)$
- $T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree of lower cost

Why Cut Property holds (contd)?

Observation

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

T' is connected.

T' is acyclic

Why Cut Property holds (contd)?

Observation

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

T' is connected.

- If path uses $e' = (v', w')$, then go from v' to v, use edge (v, w) and then go from w to w' in T'

T' is acyclic

Why Cut Property holds (contd)?

Observation

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

T' is connected.

> If path uses $e' = (v', w')$, then go from v' to v, use edge (v, w) and then go from w to w' in T'

T' is acyclic

> Only one cycle in $T' \cup \{e'\}$, namely, one involving e and e', which is not present in T'

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree T, and add to current tree T

- If e is added to tree, then e belongs to every MST.
- Set of edges output is a spanning tree.

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree T, and add to current tree T

- If e is added to tree, then e belongs to every MST.
  - Let S be the vertices connected by edges in T just before e is added.
- Set of edges output is a spanning tree.

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree T, and add to current tree T

- If e is added to tree, then e belongs to every MST.
  - Let S be the vertices connected by edges in T just before e is added.
  - e is edge of lowest cost with one end in S and the other in $V \setminus S$.
- Set of edges output is a spanning tree.

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree T, and add to current tree T

- If e is added to tree, then e belongs to every MST.
  - Let S be the vertices connected by edges in T just before e is added.
  - e is edge of lowest cost with one end in S and the other in $V \setminus S$.
- Set of edges output is a spanning tree.
  - There are no cycles in T because we always add an edge from an "unattached vertex"

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree T, and add to current tree T

- If e is added to tree, then e belongs to every MST.
  - Let S be the vertices connected by edges in T just before e is added.
  - e is edge of lowest cost with one end in S and the other in $V \setminus S$.
- Set of edges output is a spanning tree.
  - There are no cycles in T because we always add an edge from an "unattached vertex"
  - Algorithm stops when all vertices are connected

Implementing Prim's algorithm

- Start from an arbitrary root vertex r

Implementing Prim's algorithm

- Start from an arbitrary root vertex r
- Maintain a priority queue Q, where Q is the set of vertices in $V \setminus T$

Implementing Prim's algorithm

- Start from an arbitrary root vertex r
- Maintain a priority queue Q, where Q is the set of vertices in $V \setminus T$
- Key of a vertex u in Q is the minimum weight of any edge (v, u) where v is in T. We also maintain the predecessor $u.\pi$ which gives this minimum edge

Implementing Prim's algorithm

- Start from an arbitrary root vertex r
- Maintain a priority queue Q, where Q is the set of vertices in $V \setminus T$
- Key of a vertex u in Q is the minimum weight of any edge (v, u) where v is in T. We also maintain the predecessor $u.\pi$ which gives this minimum edge
- In each iteration, extract the vertex u from Q having the smallest key. The edge $(u.\pi, u)$ is the edge with minimum cost to T

Implementing Prim's algorithm

- Start from an arbitrary root vertex r
- Maintain a priority queue Q, where Q is the set of vertices in $V \setminus T$
- Key of a vertex u in Q is the minimum weight of any edge (v, u) where v is in T. We also maintain the predecessor $u.\pi$ which gives this minimum edge
- In each iteration, extract the vertex u from Q having the smallest key. The edge $(u.\pi, u)$ is the edge with minimum cost to T
  - Thus, we get a new tree T' by adding this edge to T

Implementing Prim's algorithm

- Start from an arbitrary root vertex r
- Maintain a priority queue Q, where Q is the set of vertices in $V \setminus T$
- Key of a vertex u in Q is the minimum weight of any edge (v, u) where v is in T. We also maintain the predecessor $u.\pi$ which gives this minimum edge
- In each iteration, extract the vertex u from Q having the smallest key. The edge $(u.\pi, u)$ is the edge with minimum cost to T
  - Thus, we get a new tree T' by adding this edge to T
  - We can remove u from Q and update the keys of vertices adjacent to u

Pseudocode and analysis of Prim's Algorithm

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Pseudocode and analysis of Prim's Algorithm

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Analysis

- Initialize Q and first for loop: $O(|V| \log |V|)$

Pseudocode and analysis of Prim's Algorithm

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)      // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Analysis

- Initialize Q and first for loop: $O(|V| \log |V|)$
- Decrease key of r : $O(\log |V|)$

Pseudocode and analysis of Prim's Algorithm

PRIM(G, w, r)

```
Q = ∅
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
    INSERT(Q, u)
DECREASE-KEY(Q, r, 0)        // r.key = 0
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            DECREASE-KEY(Q, v, w(u, v))
```

Analysis

- Initialize Q and first for loop: $O(|V| \log |V|)$
- Decrease key of r : $O(\log |V|)$
- While loop:
  - $|V|$ EXTRACT-MIN calls $O(|V| \log |V|)$
  - $\leq |E|$ DECREASE-KEY calls $O(|E| \log |V|)$

Pseudocode and analysis of Prim's Algorithm

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Analysis

- Initialize Q and first for loop: $O(|V| \log |V|)$
- Decrease key of r : $O(\log |V|)$
- While loop:
  - $|V|$ EXTRACT-MIN calls $O(|V| \log |V|)$
  - $\leq |E|$ DECREASE-KEY calls $O(|E| \log |V|)$
- Total time: $O(|E| \log |V|)$

Implementing Kruskal's Algorithm

```
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
    choose e ∈ E of minimum cost
    if (T U {e} does not have cycles)
        add e to T
return the set T
```

Implementing Kruskal's Algorithm

```
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
    choose e ∈ E of minimum cost
    if (T U {e} does not have cycles)
        add e to T
return the set T
```

Implementing Kruskal's Algorithm

```
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
    choose e ∈ E of minimum cost
    if (T U {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in $O(1)$ time

Implementing Kruskal's Algorithm

```
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
    choose e ∈ E of minimum cost
    if (T U {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in $O(1)$ time

Implementing Kruskal's Algorithm

```
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
    choose e ∈ E of minimum cost
    if (T U {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in $O(1)$ time
- Do breadth-first search on $T \cup \{e\}$. Takes $O(|V| + |E|)$ time

Implementing Kruskal's Algorithm

```
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
    choose e ∈ E of minimum cost
    if (T U {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in $O(1)$ time
- Do breadth-first search on $T \cup \{e\}$. Takes $O(|V| + |E|)$ time
- Total time $O(|E| \log |E|) + O(|E| \cdot (|V| + |E|))$

Implementing Kruskal's Algorithm Efficiently

Maintain the sets of vertices already connected during the running of Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

Implementing Kruskal's Algorithm Efficiently

Maintain the sets of vertices already connected during the running of Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

Implementing Kruskal's Algorithm Efficiently

Maintain the sets of vertices already connected during the running of Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

Need a new data structure to check if two elements belong to same set and to merge two sets.

Disjoint Set Data Structure

A disjoint set data structure $\mathcal{S}$ is used to store a collection of disjoint sets of elements. (Also known as Union-Find data structure.) Supports following operations.

MAKE-SET (x)	:	If x does not belong to a set in $\mathcal{S}$ then create a new set whose only member is x
FIND(x)	:	Find the set in $\mathcal{S}$ containing x
UNION(x,y)	:	Takes the two sets in $\mathcal{S}$ that contain x and y and merges them into one

Implementing Disjoint sets using arrays

Store the name of the sets in an array indexed by the elements of the set.

- For an element $x \in S$, $Set[x]$ will give the name of the set containing x. The name of a set can be the name of a unique element in the set.

Implementing Disjoint sets using arrays

Store the name of the sets in an array indexed by the elements of the set.

- For an element $x \in S$, $Set[x]$ will give the name of the set containing x. The name of a set can be the name of a unique element in the set.
- $\textsc{Find}(x)$ involves reading the entry $Set[x]$: $O(1)$

Implementing Disjoint sets using arrays

Store the name of the sets in an array indexed by the elements of the set.

- For an element $x \in S$, $Set[x]$ will give the name of the set containing x. The name of a set can be the name of a unique element in the set.
- $\textsc{Find}(x)$ involves reading the entry $Set[x]$: $O(1)$
- $\textsc{Union}(x,y)$ involves updating the entries $Set[z]$ for all elements z in $Set[x]$ and $Set[y]$: $O(n)$

Improving the Array Implementation: weighted union heuristic

For an element $x \in S$, $Set[x]$ will give the name of the set containing x.

Improving the Array Implementation: weighted union heuristic

For an element $x \in S$, $Set[x]$ will give the name of the set containing x.

- In addition, for each set, maintain a list of pointers to elements belonging to the set, along with the size of the set.

Improving the Array Implementation: weighted union heuristic

For an element $x \in S$, *Set*$[x]$ will give the name of the set containing x.

- In addition, for each set, maintain a list of pointers to elements belonging to the set, along with the size of the set.
- When performing $\text{UNION}(x,y)$, append the smaller set to the larger set and update the size of the set. Also now change the entries in the array *Set* of the smaller set to that of the larger set

Improving the Array Implementation: weighted union heuristic

For an element $x \in S$, $Set[x]$ will give the name of the set containing x.

- In addition, for each set, maintain a list of pointers to elements belonging to the set, along with the size of the set.
- When performing $\textsc{Union}(x,y)$, append the smaller set to the larger set and update the size of the set. Also now change the entries in the array *Set* of the smaller set to that of the larger set
- $\textsc{Find}(x)$ still takes $O(1)$ time.

Improving the Array Implementation: weighted union heuristic

For an element $x \in S$, $Set[x]$ will give the name of the set containing x.

- In addition, for each set, maintain a list of pointers to elements belonging to the set, along with the size of the set.
- When performing $\textsc{Union}(x,y)$, append the smaller set to the larger set and update the size of the set. Also now change the entries in the array *Set* of the smaller set to that of the larger set
- $\textsc{Find}(x)$ still takes $O(1)$ time.
- $\textsc{Union}(x,y)$ takes time $O(r)$, where r is the size of the smaller set. Worst case, still $O(n)$.

Running time

Key Observation: UNION takes $O(n)$ time for large sets. But then the union is even larger and so we cannot have too many of these bad unions.

Running time

Key Observation: UNION takes $O(n)$ time for large sets. But then the union is even larger and so we cannot have too many of these bad unions.

With weighted union, a sequence of m operations on n elements takes $O(m + n \log n)$ time.

Improving Worst Case Time: Better data structure

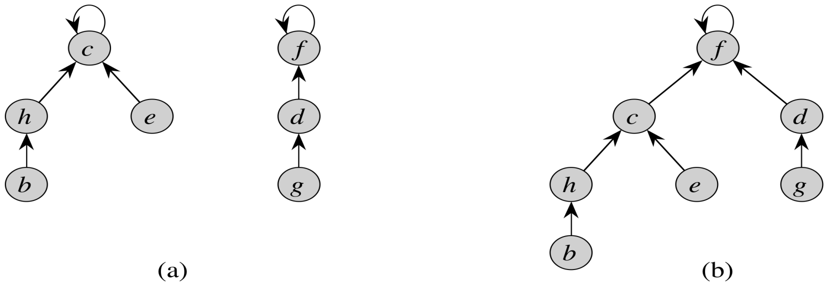


Figure: (a) Two sets $\{c, h, e, b\}$ and $\{f, d, g\}$ (b) Result of $\textsc{Union}(e,g)$

Maintain $\mathcal{S}$ in a forest; each vertex contains a single element and all elements in one tree belong to a set.

- In the tree, the child points to its parent. The root points to itself. We can use the element at the root as the name of the set.
- $\textsc{Find}(x)$: Traverse from u to the root
- $\textsc{Union}(x,y)$: Make root of x point to root of y. Takes $O(1)$ time.

Heuristics to improve worst case-behavior: union by rank

Make the root of the smaller tree (fewer nodes) a child of the root of the larger tree

- Don't actually use size

Heuristics to improve worst case-behavior: union by rank

Make the root of the smaller tree (fewer nodes) a child of the root of the larger tree

- Don't actually use size
- Use rank, which is an upper bound on height of node

Heuristics to improve worst case-behavior: union by rank

Make the root of the smaller tree (fewer nodes) a child of the root of the larger tree

- Don't actually use size
- Use rank, which is an upper bound on height of node
- Make the root with the smaller rank into a child of the root with the larger rank

More Heuristics: Path Compression

Observation: Consecutive calls of $\textsc{Find}(x)$ take $O(\log n)$ time each, but they traverse the same sequence of pointers.

More Heuristics: Path Compression

Observation: Consecutive calls of $\textsc{Find}(x)$ take $O(\log n)$ time each, but they traverse the same sequence of pointers.

Path Compression: Make all nodes encountered in the first call of $\textsc{Find}(x)$ point to root.

More Heuristics: Path Compression

Observation: Consecutive calls of $\textsc{Find}(x)$ take $O(\log n)$ time each, but they traverse the same sequence of pointers.

Path Compression: Make all nodes encountered in the first call of $\textsc{Find}(x)$ point to root.

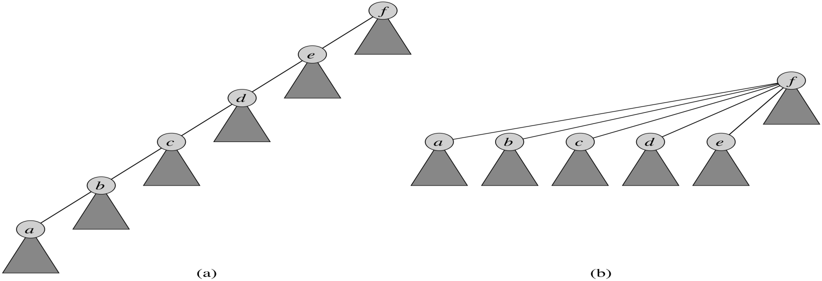


Figure: (a) Tree prior to $\textsc{Find}(a)$. Triangles represent subtrees whose roots are the vertices shown. (b) Result of $\textsc{Find}(a)$

Pseudocode for disjoint-set operations

MAKE-SET(x)
  $x.p = x$
  $x.rank = 0$

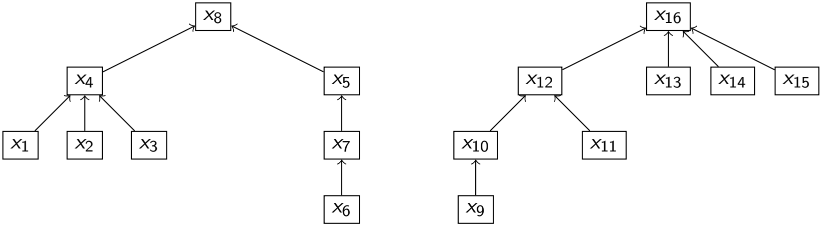
FIND-SET(x)
  if $x \neq x.p$
    $x.p =$ FIND-SET($x.p$)
  return $x.p$

UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  if $x.rank > y.rank$
    $y.p = x$
  else $x.p = y$
    // If equal ranks, choose y as parent and increment its rank.
    if $x.rank == y.rank$
      $y.rank = y.rank + 1$

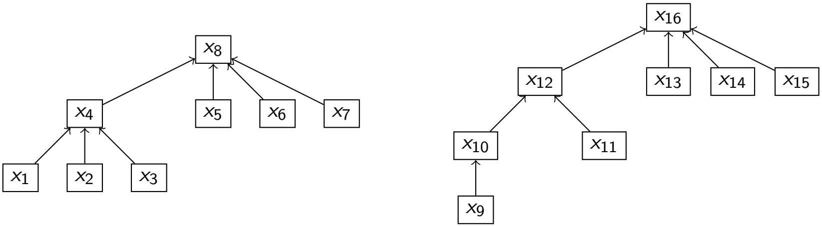
Each vertex has two attributes, p (parent) and rank

Example



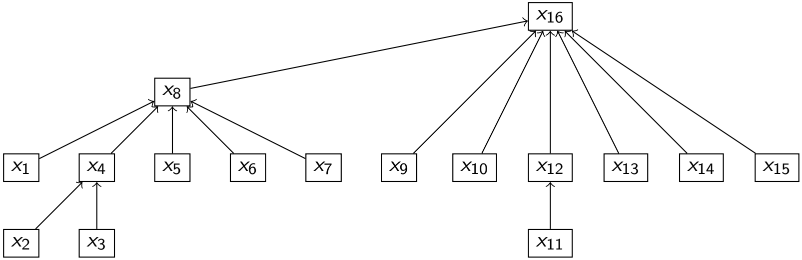
We have two sets each of rank 4.

Example continued: Find(x_6)

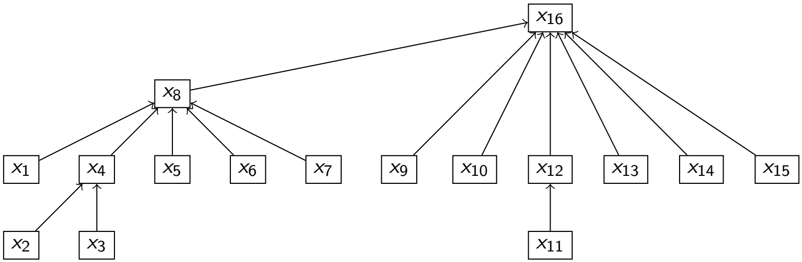


We have two sets each of rank 4. Note the rank of the first set does not change.

Example continued: Union(x_1, x_9)

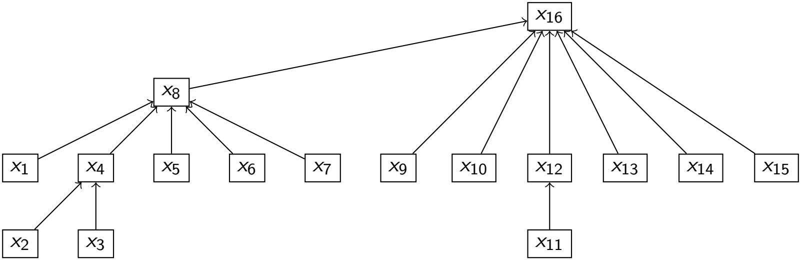


Example continued: Union(x_1, x_9)



- Notice where x_1, x_9, x_{10} point to.

Example continued: Union(x_1, x_9)



- Notice where x_1, x_9, x_{10} point to.
- The rank of the set is 5.

Running time

If use both union by rank and path compression, a sequence of m operations on n elements takes $O(m\alpha(n))$ time

- α is a very slowly growing function (slower than log) and $m\alpha$ almost linear

n	$\alpha(n)$
$0-2$	0
3	1
$4-7$	2
$8-2047$	3
$2048-A_4(1)$	4

Running time

If use both union by rank and path compression, a sequence of m operations on n elements takes $O(m\alpha(n))$ time

- α is a very slowly growing function (slower than log) and $m\alpha$ almost linear

n	$\alpha(n)$
$0-2$	0
3	1
$4-7$	2
$8-2047$	3
$2048-A_4(1)$	4

- $A_4(1)$ is greater than 10^8

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E|\log(|E|))$

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E|\log(|E|))$
  - Constructing the initial collection of sets: $|V|$ $\textsc{Make-Set}$ operations

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E|\log(|E|))$
  - Constructing the initial collection of sets: $|V|$ Make-Set operations
  - While loop: $O(|E|)$ Find and Union operations

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E| \log(|E|))$
  - Constructing the initial collection of sets: $|V|$ Make-Set operations
  - While loop: $O(|E|)$ Find and Union operations
  - Total number of elements in the set: $|V|$

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E|\log(|E|))$
  - Constructing the initial collection of sets: $|V|$ Make-Set operations
  - While loop: $O(|E|)$ Find and Union operations
  - Total number of elements in the set: $|V|$
- Total time: $O((|V|+|E|)\alpha(|V|)) + O(|E|\log|E|)$

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E| \log(|E|))$
  - Constructing the initial collection of sets: $|V|$ $\textsc{Make-Set}$ operations
  - While loop: $O(|E|)$ $\textsc{Find}$ and $\textsc{Union}$ operations
  - Total number of elements in the set: $|V|$
- Total time: $O((|V| + |E|)\alpha(|V|)) + O(|E| \log |E|)$
- $|E| > |V|$ (connected graph) and $\alpha(|V|)$ is $O(\log |V|)$

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E|\log(|E|))$
  - Constructing the initial collection of sets: $|V|$ MAKE-SET operations
  - While loop: $O(|E|)$ FIND and UNION operations
  - Total number of elements in the set: $|V|$
- Total time: $O((|V|+|E|)\alpha(|V|)) + O(|E|\log|E|)$
- $|E| > |V|$ (connected graph) and $\alpha(|V|)$ is $O(\log|V|)$
- Also as $|E| \leq |V|^2$, $\log|E| \leq 2\log|V|$

Back to Kruskal's algorithm

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

- Running time analysis
  - Sorting edges: $O(|E|\log(|E|))$
  - Constructing the initial collection of sets: $|V|$ $\textsc{Make-Set}$ operations
  - While loop: $O(|E|)$ $\textsc{Find}$ and $\textsc{Union}$ operations
  - Total number of elements in the set: $|V|$
- Total time: $O((|V|+|E|)\alpha(|V|)) + O(|E|\log|E|)$
- $|E| > |V|$ (connected graph) and $\alpha(|V|)$ is $O(\log|V|)$
- Also as $|E| \leq |V|^2$, $\log|E| \leq 2\log|V|$
- Therefore, running time $= O(|E|\log|V|)$