Design and Analysis of Algorithms CS218M

Greedy Algorithms (2)

Paritosh Pandya

Indian Institute of Technology, Bombay

Autumn, 2022

The Greedy Paradigm

- Build the solution by selecting elements (or making choices) one by one.
- A simple rule allows choice of element at each stage. Local optimality.
- Greedy choice property: The current selection cannot be removed (no backtracking/exploring alternative choices).
- The final solution must be optimal.

Sequence of locally optimalchoices gives globally optimal solution.

Examples: Picking 10 coins, Finding shortest path, Minimum Spanning Tree.

Codes and Data Compression

- Finite alphabet $S = \{a_1, \ldots, a_n\}$. Message $x_1 x_2 \ldots x_r$ is a finite sequence of symbols.
- Digital communication: Message encoded and communicated as a sequence of bits. It is decoded back on receipt.
- We encode each symbol x_i as bit sequence $\gamma(x_i)$.
- Fixed length encoding: $\lceil lg(n) \rceil$ bits are needed to encode a symbol from alphabet S of size n.
- E.g. ASCII characters into 7-bit encoding. Unicode characters into 16-bits.

Variable Length Code

$S = \{a, b, c\}$ $\gamma = \boxed{a \to 0 \quad b \to 1 \quad c \to 01}$

Frequency of a denoted F_a.

$F_a = 0.8$ $F_b = 0.1$ $F_c = 0.1$

$12 \times 0.8 \times 1 + 12 \times 0.1 \times 1 +$

$12 \times 0.1 \times 2$

9.6
$+1.2$
$+2.4$

$01 \to ab$
$\searrow c$

Variable Length Code

Prefix Property

For any $x, y \in S,\ x \neq y$ we have
$\neg\ pref(\gamma(x), \gamma(y)) \quad \wedge \quad \neg\ pref(\gamma(y), \gamma(x))$

Variable Length Code

Prefix Property

For any $x, y \in S,\ x \neq y$ we have
$\neg\ pref(\gamma(x), \gamma(y)) \quad \wedge \quad \neg\ pref(\gamma(y), \gamma(x))$

Prefix property allows us to uniquely decode the encoded bit-string.

$S = \{a, b, c, d, e\}$. The encoding γ_1 specified by

acca
1100111

$$\gamma_1(a) = 11$$
$$\gamma_1(b) = 01$$
$$\gamma_1(c) = 001$$
$$\gamma_1(d) = 10$$
$$\gamma_1(e) = 000$$

Average Bits Per Letter

Let probability of letter x in message be f_x. Let γ be a prefix code. Then,

$$ABL(\gamma) = \Sigma_{x \in S} \; f_x \cdot |\gamma(x)|$$

Prefix Code

$S = \{a, b, c, d, e\}$. The encoding γ_1 specified by

$$\gamma_1(a) = 11$$
$$\gamma_1(b) = 01$$
$$\gamma_1(c) = 001$$
$$\gamma_1(d) = 10$$
$$\gamma_1(e) = 000$$

Average Bits Per Letter

Let probability of letter x in message be f_x. Let γ be a prefix code. Then,

$$ABL(\gamma) \quad = \quad \Sigma_{x \in S} \quad f_x \cdot |\gamma(x)|$$

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25.$$

Optimal Prefix Code

$$\gamma_2(a) = 11$$
$$\gamma_2(b) = 10$$
$$\gamma_2(c) = 01$$
$$\gamma_2(d) = 001$$
$$\gamma_2(e) = 000$$

The average number of bits per letter using γ_2 is

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 2 + .18 \cdot 3 + .05 \cdot 3 = 2.23.$$

Optimal Prefix Code

$$\gamma_2(a) = 11$$
$$\gamma_2(b) = 10$$
$$\gamma_2(c) = 01$$
$$\gamma_2(d) = 001$$
$$\gamma_2(e) = 000$$

The average number of bits per letter using γ_2 is

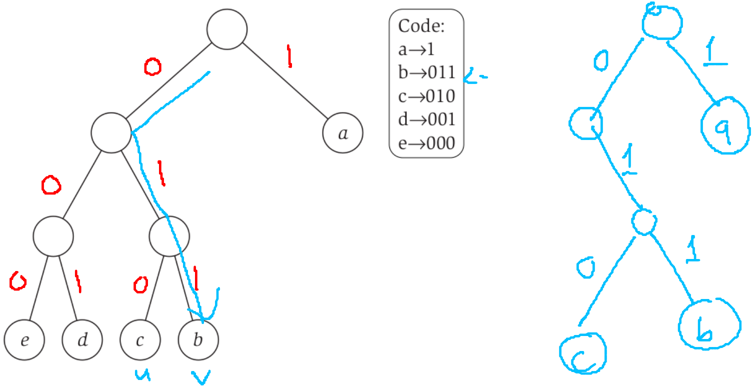
$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 2 + .18 \cdot 3 + .05 \cdot 3 = 2.23.$$

Optimal Prefix Code

For a given alphabet S and probability distribution f, a prefix code γ over S is optimal if for all prefix codes γ' over S we have

$$ABL(\gamma) \leq ABL(\gamma')$$

Labelled Binary Trees



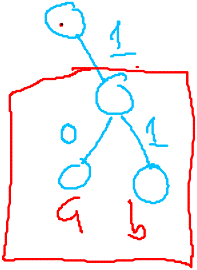
- LBT is a binary tree where leaves are labelled with letters from S.
- There is bijection between LBTs and prefix codes. They represent the same thing.
- Notation: Given LBT T and a leaf node u let $lb(u) \in S$ denote the label of u.

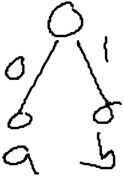
Properties of LBT

- Let $ABL(T) \quad = \quad \Sigma_{x \in S} \quad f_x \cdot depth(x)$

Properties of LBT

- Let $ABL(T) \;=\; \Sigma_{x \in S} \;\; f_x \cdot depth(x)$
- A LBT is called full if every interior node has exactly two children.





Properties of LBT

- Let $ABL(T) = \Sigma_{x \in S} \ f_x \cdot depth(x)$
- A LBT is called full if every interior node has exactly two children.
- Every LBT can be tranformed to a more efficient full LBT. Hence, every optimal LBT is full.

Properties of LBT

- Let $ABL(T) \; = \; \Sigma_{x \in S} \;\; f_x \cdot depth(x)$
- A LBT is called full if every interior node has exactly two children.
- Every LBT can be tranformed to a more efficient full LBT. Hence, every optimal LBT is full.
- In LBT T, if labels of leaves at same depth are permuted to get T' then $ABL(T') = ABL(T)$.

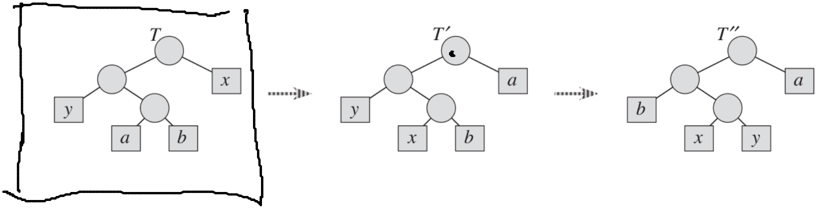
- Let $ABL(T) \;=\; \Sigma_{x \in S} \;\; f_x \cdot depth(x)$
- A LBT is called full if every interior node has exactly two children.
- Every LBT can be tranformed to a more efficient full LBT. Hence, every optimal LBT is full.
- In LBT T, if labels of leaves at same depth are permuted to get T' then $ABL(T') = ABL(T)$.
- In LBT T if $f_u = f_v$ and T' is obtained by exchanging labels of u, v then $ABL(T') = ABL(T)$.

Key Property of Optimal LBT (2)

Greedy Choice

Let $x, y \in S$ s.t. $\forall z \in S, z \neq x, y$ we have $(f_x \leq f_z \wedge f_y \leq f_z)$. Such x, y are called least frequent pair. Then, there exists a optimal LBT where x, y occur as siblings at maximum depth.

- Let a, b be sibling leaves occurring at maximum depth in T.
- Hence, $f_x \leq f_a$ and $depth(x) \leq depth(a)$.
- First exchange a and x to get T'. Next, exchange y and b.



Proof (exchange argument)

Let $f_x \leq f_a$. Then,

$$
\begin{aligned}
& ABL(T) - ABL(T') \\
= \quad & (f_x \cdot depth(x) + f_a \cdot depth(a)) - (f_x \cdot depth(a) + f_a \cdot depth(x)) \\
= \quad & (f_x - f_a) \cdot (depth(x) - depth(a)) \\
\geq 0 \quad & \qquad \text{(as both factors negative).}
\end{aligned}
$$

Proof (exchange argument)

Let $f_x \leq f_a$. Then,

$$
\begin{aligned}
& ABL(T) - ABL(T') \\
= \quad & (f_x \cdot depth(x) + f_a \cdot depth(a)) - (f_x \cdot depth(a) + f_a \cdot depth(x)) \\
= \quad & (f_x - f_a) \cdot (depth(x) - depth(a)) \\
\geq 0 \quad & \qquad \text{(as both factors negative)}.
\end{aligned}
$$

- If T is optimal, then so is T''.

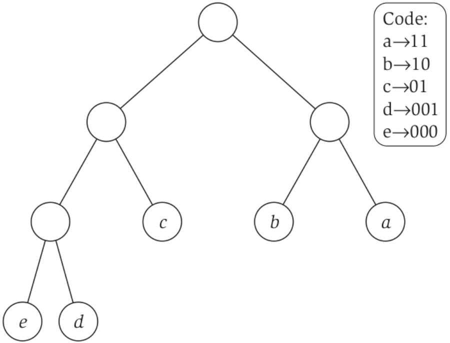
Proof (exchange argument)

Let $f_x \leq f_a$. Then,

$$
\begin{aligned}
& ABL(T) - ABL(T') \\
= \;& (f_x \cdot depth(x) + f_a \cdot depth(a)) - (f_x \cdot depth(a) + f_a \cdot depth(x)) \\
= \;& (f_x - f_a) \cdot (depth(x) - depth(a)) \\
\geq 0 \;& \qquad \text{(as both factors negative).}
\end{aligned}
$$

- If T is optimal, then so is T''.
- In optimal LBT, as depth of leaves decreases the probability must increase or remain same.

Example: Optimal LBT



$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

Huffman Algorithm

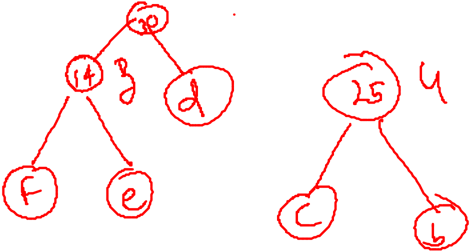
C set of letters

By repeated greedy choice of lowest-frequency pairs.

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

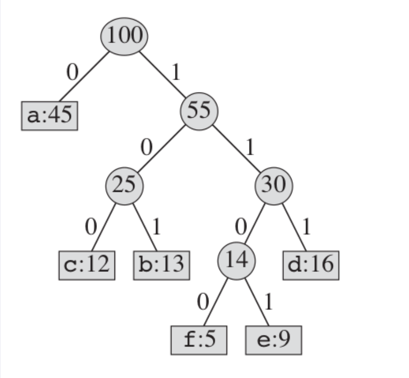
Huffman Algorithm: Example

~~f:5~~ ~~e:9~~ ~~c:12~~ ~~b:13~~ d:16 a:45



Huffman Algorithm: Example

Huffman Code

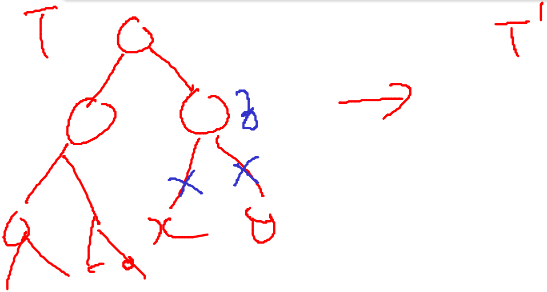


Optimality of Huffman Code

Lemma

Let LBT T of S have leaves labelled x, y as siblings. Let T' be be LBT obtained by replacing parent of x, y by leaf node labelled z. Also let $f_z = f_x + f_y$. Then,

$$ABL(T) = ABL(T') + f_x + f_y$$



Optimality of Huffman Code

Lemma

Let LBT T of S have leaves labelled x, y as siblings. Let T' be be LBT obtained by replacing parent of x, y by leaf node labelled z. Also let $f_z = f_x + f_y$. Then,

$$ABL(T) = ABL(T') + f_x + f_y$$

Theorem

Huffman LBT H for alphabet S of size n is optimal.

Lemma

Let LBT T of S have leaves labelled x, y as siblings. Let T' be be LBT obtained by replacing parent of x, y by leaf node labelled z. Also let $f_z = f_x + f_y$. Then,

$$ABL(T) = ABL(T') + f_x + f_y$$

Theorem

Huffman LBT H for alphabet S of size n is optimal.

Proof by Induction on n.
Base case $n = 2$ is trivially true.

Induction Step

- By Ind. Hyp. suppose that optimality holds for alphabets of size $n-1$.

Induction Step

- By Ind. Hyp. suppose that optimality holds for alphabets of size $n-1$.
- For alphabet S of size n assume that $ABL(T) < ABL(H)$ where T is optimal.

Induction Step

- By Ind. Hyp. suppose that optimality holds for alphabets of size $n-1$.
- For alphabet S of size n assume that $ABL(T) < ABL(H)$ where T is optimal.
- Wlog, assume that least-frequent x, y occur as siblings in T and H. (why?)

Induction Step

- By Ind. Hyp. suppose that optimality holds for alphabets of size $n-1$.
- For alphabet S of size n assume that $ABL(T) < ABL(H)$ where T is optimal.
- Wlog, assume that least-frequent x, y occur as siblings in T and H. (why?)
- Consider T' and H' obtained by replacing parent of x, y by z. Then,
$$\begin{aligned} ABL(T') &= ABL(T) - (f_x + f_y) < \\ & ABL(H) - (f_x + f_y) = ABL(H'). \end{aligned}$$
This contradicts induction hypothesis.

Induction Step

- By Ind. Hyp. suppose that optimality holds for alphabets of size $n-1$.
- For alphabet S of size n assume that $ABL(T) < ABL(H)$ where T is optimal.
- Wlog, assume that least-frequent x, y occur as siblings in T and H. (why?)
- Consider T' and H' obtained by replacing parent of x, y by z. Then,

  $$\begin{aligned} ABL(T') &= ABL(T) - (f_x + f_y) < \\ & ABL(H) - (f_x + f_y) = ABL(H'). \end{aligned}$$

  This contradicts induction hypothesis.

Hence, H is optimal.

Complexity Huffman Algorithm

Letters of S are stored in MIN-priority queue by their frequencies.

Let $|S| = n$.

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

- $O(n)$ to initialize the priority queue (as a MIN-heap).
- Loop is executed $O(n)$ times. Each iteration extracts 2 minimum-priority elements using $O(lg(n))$ time. Hence $O(n \cdot lg(n))$.
- Total time $O(n \cdot lg(n))$. Can be improved to $O(n \cdot lg(lg(n)))$.

Given connected and weighted undirected graph $G = (V, E, w)$ with nodes V, Edges $E \subseteq V \times V$ and $w : E \rightarrow \Re$, find $A \subseteq E$ s.t.

Minimum Spanning Tree (MST)

Given connected and weighted undirected graph $G = (V, E, w)$ with nodes V, Edges $E \subseteq V \times V$ and $w : E \to \Re$, find $A \subseteq E$ s.t.

- A is a tree spanning V.

Minimum Spanning Tree (MST)

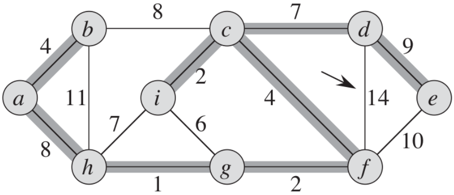
Given connected and weighted undirected graph $G = (V, E, w)$ with nodes V, Edges $E \subseteq V \times V$ and $w : E \rightarrow \Re$, find $A \subseteq E$ s.t.

- A is a tree spanning V.
- Let $wt(A) = \Sigma_{e \in A} w(e)$. Then for all $B \subseteq E$, if B is a spanning tree then $wt(B) \geq wt(A)$.

Minimum Spanning Tree (MST)

Given connected and weighted undirected graph $G = (V, E, w)$ with nodes V, Edges $E \subseteq V \times V$ and $w : E \rightarrow \Re$, find $A \subseteq E$ s.t.

- A is a tree spanning V.
- Let $wt(A) = \Sigma_{e \in A}\, w(e)$. Then for all $B \subseteq E$, if B is a spanning tree then $wt(B) \geq wt(A)$.



Grow A adding one edge at a time.

Grow A adding one edge at a time.

Kruskal

Add lowest weight edge which does not form a cycle to current A.

Grow A adding one edge at a time.

Kruskal

Add lowest weight edge which does not form a cycle to current A.

Prim

Extend current set of edges A having vertices U_A with a minimum weight edge going out of U_A.

Generic MST Algorithm

- Grow A one edge at a time.
- Invariant: Current set of edges A is a subset of some MST.
- An edge which can be added to A maintaining the invariant is called a safe edge.

GENERIC-MST(G, w)

```
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

Terminology

Given connected, undirected, weighted graph $G = (V, E, w)$, and $A \subseteq E$, define

- Pair $(S, V - S)$ is a cut.

Terminology

Given connected, undirected, weighted graph $G = (V, E, w)$, and $A \subseteq E$, define

- Pair $(S, V - S)$ is a cut.
- Edge (u, v) crosses the cut $(S, V - S)$ if $u \in S$ and $v \notin S$ or vice verse.

Terminology

Given connected, undirected, weighted graph $G = (V, E, w)$, and $A \subseteq E$, define

- Pair $(S, V - S)$ is a cut.
- Edge (u, v) crosses the cut $(S, V - S)$ if $u \in S$ and $v \notin S$ or vice verse.
- Cut $(S, V - S)$ respects A if no edge of A is a crossing edge.

Terminology

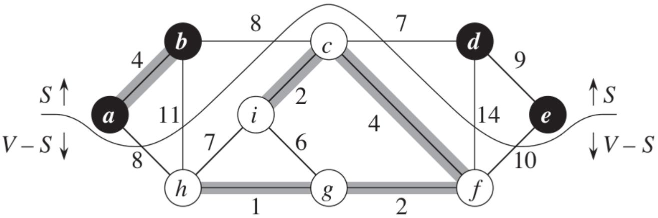
Given connected, undirected, weighted graph $G = (V, E, w)$, and $A \subseteq E$, define

- Pair $(S, V - S)$ is a cut.
- Edge (u, v) crosses the cut $(S, V - S)$ if $u \in S$ and $v \notin S$ or vice verse.
- Cut $(S, V - S)$ respects A if no edge of A is a crossing edge.
- An edge (u, v) is a light edge if it is of minimum weight amongst all edges crossing the cut.

Main Property

Theorem

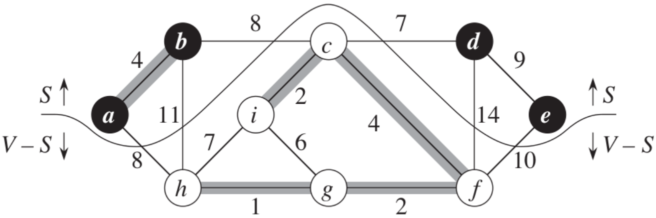
Let A subset of some MST. Let cut $(S, V - S)$ respect A and let (u, v) be a light edge. Then, (u, v) is a safe edge.



Main Property

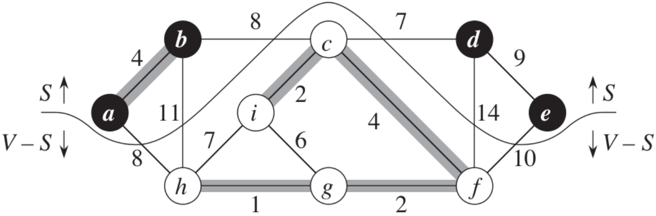
Theorem

Let A subset of some MST. Let cut $(S, V-S)$ respect A and let (u, v) be a light edge. Then, (u, v) is a safe edge.



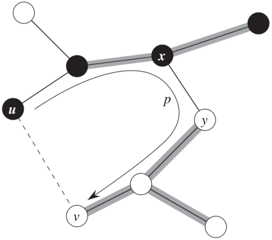
Theorem

Let A subset of some MST. Let cut $(S, V - S)$ respect A and let (u, v) be a light edge. Then, (u, v) is a safe edge.



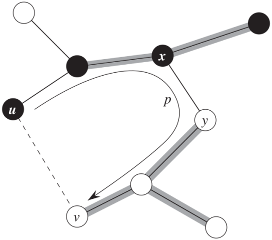
Proof

- A gray edges, MST T. Let $(u, v) \notin T$ be a light-edge.



Proof

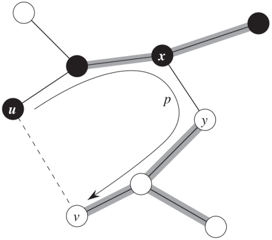
- A gray edges, MST T. Let $(u, v) \notin T$ be a light-edge.



- Let $(x, y) \in T$ be crossing edge. (Must exist).

Proof

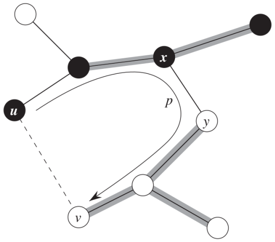
- A gray edges, MST T. Let $(u, v) \notin T$ be a light-edge.



- Let $(x, y) \in T$ be crossing edge. (Must exist).
- Hence $w(u, v) \leq w(x, y)$. (why?)

Proof

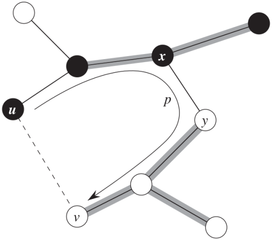
- A gray edges, MST T. Let $(u, v) \notin T$ be a light-edge.



- Let $(x, y) \in T$ be crossing edge. (Must exist).
- Hence $w(u, v) \leq w(x, y)$. (why?)
- Let $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Proof

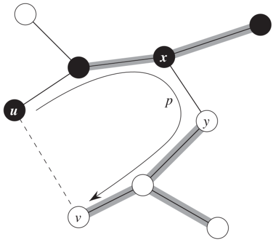
- A gray edges, MST T. Let $(u, v) \notin T$ be a light-edge.



- Let $(x, y) \in T$ be crossing edge. (Must exist).
- Hence $w(u, v) \leq w(x, y)$. (why?)
- Let $T' \; = \; T - \{(x, y)\} \;\; \cup \;\; \{(u, v)\}$.
- $wt(T') \; = \; wt(T) - w(x, y) + w(u, v)$.
  Hence, $wt(T') \leq wt(T)$.

Proof

- A gray edges, MST T. Let $(u, v) \notin T$ be a light-edge.



- Let $(x, y) \in T$ be crossing edge. (Must exist).
- Hence $w(u, v) \leq w(x, y)$. (why?)
- Let $T' = T - \{(x, y)\} \cup \{(u, v)\}$.
- $wt(T') = wt(T) - w(x, y) + w(u, v)$.
  Hence, $wt(T') \leq wt(T)$.
- Hence, T' is MST containing (u, v).

Kruskal Algorithm and Correctness

At each iteration.

- A gives rise to a set of disjoint trees.

Kruskal Algorithm and Correctness

At each iteration.

- A gives rise to a set of disjoint trees.
- Kruskal iteration extends A by minimum weight edge (u, v) connecting two trees T_1 and T_2 (and merges these).

At each iteration.

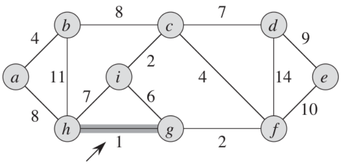
- A gives rise to a set of disjoint trees.
- Kruskal iteration extends A by minimum weight edge (u, v) connecting two trees T_1 and T_2 (and merges these).
- Choose cut respecting A as $(T_1, S - T_1)$. Clearly, (u, v) is safe edge. Theorem applies.

Kruskal Algorithm and Correctness

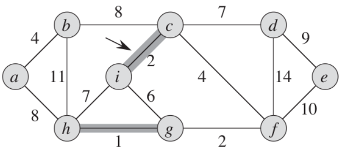
At each iteration.

- A gives rise to a set of disjoint trees.
- Kruskal iteration extends A by minimum weight edge (u, v) connecting two trees T_1 and T_2 (and merges these).
- Choose cut respecting A as $(T_1, S - T_1)$. Clearly, (u, v) is safe edge. Theorem applies.
- Adding it using UNION gives A as set of trees represented as disjoint sets.

Kruskal Algorithm: Example



(b)



Disjoint Set Data Structure

Maintain $\mathcal{S} = (S_1, \ldots, S_k)$ with $u_i \in S_i$ as unique representative.

Disjoint Set Data Structure

Maintain $\mathcal{S} = (S_1, \ldots, S_k)$ with $u_i \in S_i$ as unique representative.

- MAKESET(u)

Disjoint Set Data Structure

Maintain $\mathcal{S} = (S_1, \ldots, S_k)$ with $u_i \in S_i$ as unique representative.

- MAKESET(u)
- FINDSET(u)

Disjoint Set Data Structure

Maintain $\mathcal{S} = (S_1, \ldots, S_k)$ with $u_i \in S_i$ as unique representative.

- MAKESET(u)
- FINDSET(u)
- UNION(u,v)

Disjoint Set Data Structure

Maintain $\mathcal{S} = (S_1, \ldots, S_k)$ with $u_i \in S_i$ as unique representative.

- MAKESET(u)
- FINDSET(u)
- UNION(u,v)
- Implemeted using union by rank and path compression (CLRS 21.3, 21.4). For m operations over n element set, $O(m \cdot \alpha(n))$ where $\alpha(n)$ is very slowly growing (almost constant!).

Kruskal Algorithm for MST

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

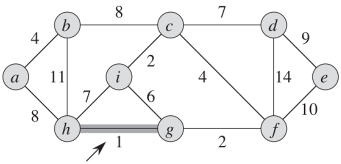
Kruskal Algorithm for MST

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

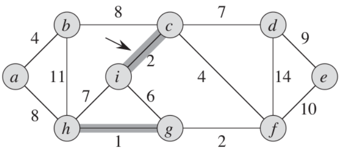
Running Time

$E \cdot lg(E)$ for sorting edges. Also, $O(V)$ of MAKE-SET and $O(E)$ of FIND-SET+UNION operations. Hence, $E \cdot lg(E) + (E + V)\alpha(V)$. Simplifies to $O(E \cdot lg(E))$.

Kruskal Algorithm: Example



(b)



Prim Algorithm for MST

- Maintain A as a single tree with set of vertices U_A. Let $Q = S - U_A$.

Prim Algorithm for MST

- Maintain A as a single tree with set of vertices U_A. Let $Q = S - U_A$.
- In each iteration, we choose edge e with minimum weight amongst $\{(u, v) \mid u \in U_A \wedge v \notin U_A\}$. Clearly, this is safe edge.

Prim Algorithm for MST

- Maintain A as a single tree with set of vertices U_A. Let $Q = S - U_A$.
- In each iteration, we choose edge e with minimum weight amongst $\{(u, v) \mid u \in U_A \wedge v \notin U_A\}$. Clearly, this is safe edge.
- For each vertex $v \in Q$, priority $v.key$ is weight of minimum weight edge between (any vertex in) T and v. If no such edge $key = \infty$.

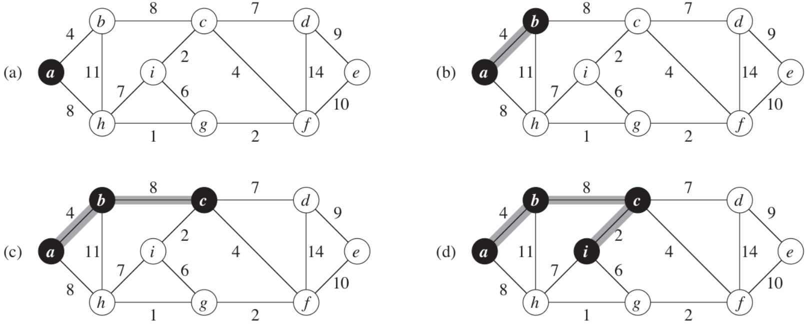
Prim Algorithm for MST

- Maintain A as a single tree with set of vertices U_A. Let $Q = S - U_A$.
- In each iteration, we choose edge e with minimum weight amongst $\{(u, v) \mid u \in U_A \wedge v \notin U_A\}$. Clearly, this is safe edge.
- For each vertex $v \in Q$, priority $v.key$ is weight of minimum weight edge between (any vertex in) T and v. If no such edge $key = \infty$.
- Maintain Q as a priority queue using the heap data structure. Choose v as EXTRACT_MIN(Q).

Prim Algorithm for MST

- Maintain A as a single tree with set of vertices U_A. Let $Q = S - U_A$.
- In each iteration, we choose edge e with minimum weight amongst $\{(u, v) \mid u \in U_A \wedge v \notin U_A\}$. Clearly, this is safe edge.
- For each vertex $v \in Q$, priority $v.key$ is weight of minimum weight edge between (any vertex in) T and v. If no such edge $key = \infty$.
- Maintain Q as a priority queue using the heap data structure. Choose v as EXTRACT_MIN(Q).
- After adding v, update key of all vertices adjecent to v which are in Q.

Prim Algorithm



Prim Algorithm

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

Prim Algorithm