CSE 431/531: Algorithm Analysis and Design (Spring 2018)

Greedy Algorithms

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Main Goal of Algorithm Design

- Design fast algorithms to solve problems
- Design more efficient algorithms to solve problems

Def. The goal of an optimization problem is to find a valid solution with the minimum (or maximum) cost (or value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time $=$ efficient

Common Paradigms for Algorithm Design

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

Analysis of Greedy Algorithm

- Prove that the reasonable strategy is "safe" (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually trivial)

Outline

Toy Problem 1: Bill Changing

Input: Integer $A \geq 0$

Currency denominations: $\$1, \$2, \$5, \$10, \$20$

Output: A way to pay A dollars using fewest number of bills

Example:

- Input: 48
- Output: 5 bills, $\$48 = \$20 \times 2 + \$5 + \$2 + \$1$

Cashier's Algorithm

1. while $A \geq 0$ do
2. $\quad a \leftarrow \max\{t \in \{1, 2, 5, 10, 20\} : t \leq A\}$
3. $\quad$ pay a \$$a$ bill
4. $\quad A \leftarrow A - a$

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

- strategy: choose the largest bill that does not exceed A
- the strategy is "reasonable": choosing a larger bill help us in minimizing the number of bills
- The decision is irrevocable : once we choose a \$$a$ bill, we let $A \leftarrow A - a$ and proceed to the next

Analysis of Greedy Algorithm

- Prove that the reasonable strategy is "safe"
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

- $n_1, n_2, n_5, n_{10}, n_{20}$: number of \$1, \$2, \$5, \$10, \$20 bills paid
- minimize $n_1 + n_2 + n_5 + n_{10} + n_{20}$ subject to $n_1 + 2n_2 + 5n_5 + 10n_{10} + 20n_{20} = A$

Obs.

- $n_1 < 2$ — $2 \leq A < 5$: pay a \$2 bill
- $n_1 + 2n_2 < 5$ — $5 \leq A < 10$: pay a \$5 bill
- $n_1 + 2n_2 + 5n_5 < 10$ — $10 \leq A < 20$: pay a \$10 bill
- $n_1 + 2n_2 + 5n_5 + 10n_{10} < 20$ — $20 \leq A < \infty$: pay a \$20 bill

Analysis of Greedy Algorithm

- Prove that the reasonable strategy is "safe"
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

- Trivial: in residual problem, we need to pay $A' = A - a$ dollars, using the fewest number of bills

Toy Example 2: Box Packing

Box Packing

Input: n boxes of capacities $c_1, c_2, \cdots, c_n$

m items of sizes $s_1, s_2, \cdots, s_m$

Can put at most 1 item in a box

Item j can be put into box i if $s_j \leq c_i$

Output: A way to put as many items as possible in the boxes.

Example:

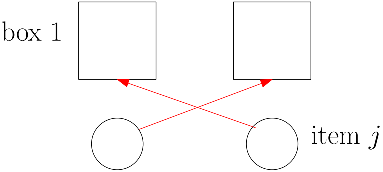
- Box capacities: 60, 40, 25, 15, 12
- Item sizes: 45, 42, 20, 19, 16
- Can put 3 items in boxes: $45 \to 60, 20 \to 40, 19 \to 25$

Box Packing: Design a Safe Strategy

Q: Take box 1 (with capacity c_1). Which item should we put in box 1?

A: The item of the largest size that can be put into the box.

- putting the item gives us the easiest residual problem.
- formal proof via exchanging argument: $j =$ largest item that can be put into box 1.



- Residual task: solve the instance obtained by removing box 1 and item j

Greedy Algorithm for Box Packing

$T \leftarrow \{1, 2, 3, \cdots, m\}$
for $i \leftarrow 1$ to n do
    if some item in T can be put into box i, then
        $j \leftarrow$ the largest item in T that can be put into box i
        print("put item j in box i")
        $T \leftarrow T \setminus \{j\}$

Steps of Designing A Greedy Algorithm

- Design a "reasonable" strategy
- Prove that the reasonable strategy is "safe" (key, usually done by "exchanging argument")
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually trivial)

Def. A choice is "safe" if there is an optimum solution that is "consistent" with the choice

Exchanging argument: let S be an arbitrary optimum solution. If S is consistent with the greedy choice, we are done. Otherwise, modify it to another optimum solution S' such that S' is consistent with the greedy choice.

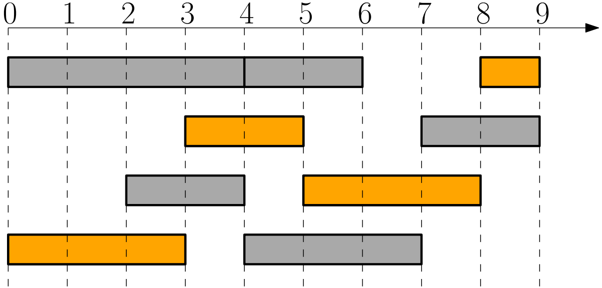
Outline

Interval Scheduling

Input: n jobs, job i with start time s_i and finish time f_i

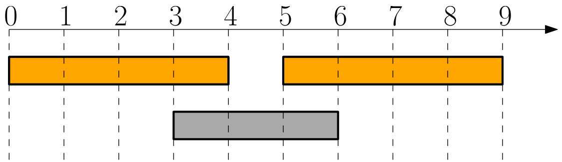
i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: A maximum-size subset of mutually compatible jobs



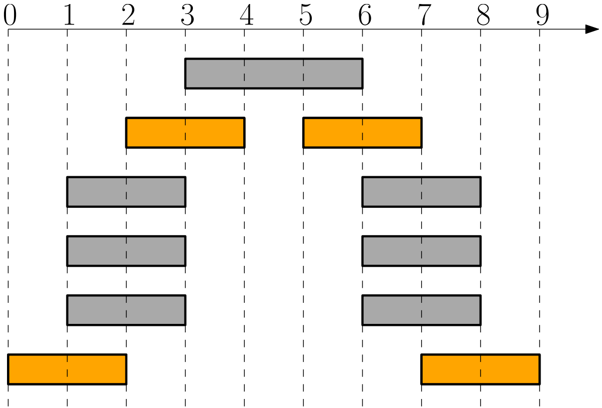
Greedy Algorithm for Interval Scheduling

- Which of the following decisions are safe?
- Schedule the job with the smallest size? No!



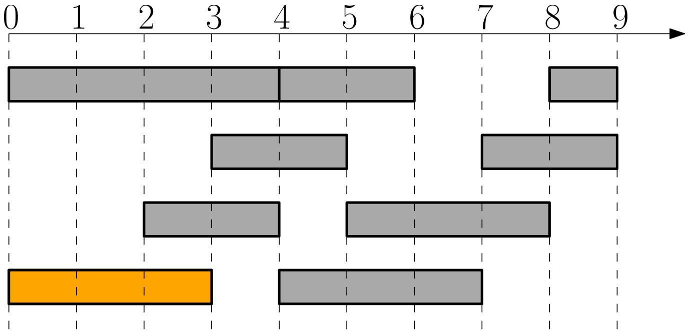
Greedy Algorithm for Interval Scheduling

- Which of the following decisions are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!



Greedy Algorithm for Interval Scheduling

- Which of the following decisions are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
- Schedule the job with the earliest finish time? Yes!

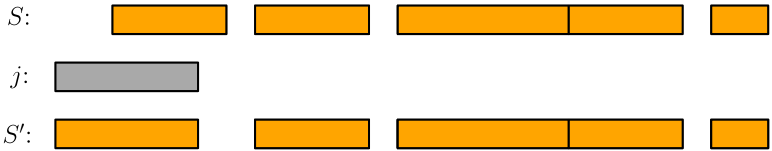


Greedy Algorithm for Interval Scheduling

Lemma It is safe to schedule the job j with the earliest finish time: there is an optimum solution where j is scheduled.

Proof.

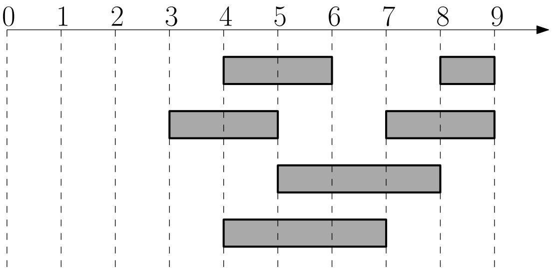
- Take an arbitrary optimum solution S
- If it contains j, done
- Otherwise, replace the first job in S with j to obtain an new optimum schedule S'. $\square$



Greedy Algorithm for Interval Scheduling

Lemma It is safe to schedule the job j with the earliest finish time: there is an optimum solution where j is scheduled.

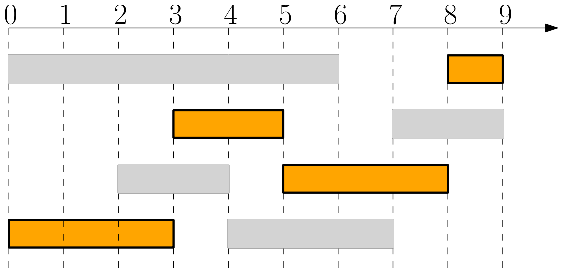
- What is the remaining task after we decided to schedule j?
- Is it another instance of interval scheduling problem? Yes!



Greedy Algorithm for Interval Scheduling

Schedule(s, f, n)

1. $A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset$
2. while $A \neq \emptyset$
3. $\quad j \leftarrow \arg\min_{j' \in A} f_{j'}$
4. $\quad S \leftarrow S \cup \{j\}$; $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5. return S



Greedy Algorithm for Interval Scheduling

Schedule(s, f, n)

1. $A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset$
2. while $A \neq \emptyset$
3. $\quad j \leftarrow \arg\min_{j' \in A} f_{j'}$
4. $\quad S \leftarrow S \cup \{j\}$; $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5. return S

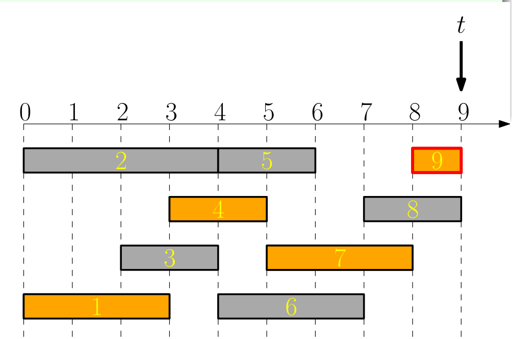
Running time of algorithm?

- Naive implementation: $O(n^2)$ time
- Clever implementation: $O(n \lg n)$ time

Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

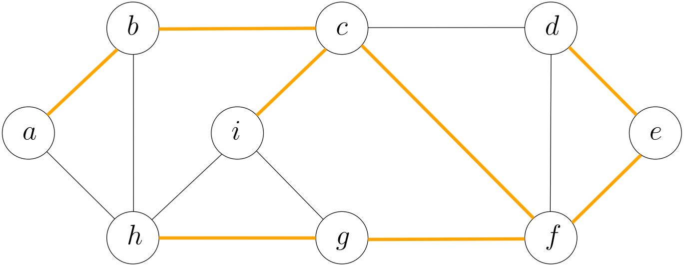
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j
    if $s_j \geq t$ then
        $S \leftarrow S \cup \{j\}$
        $t \leftarrow f_j$
return S

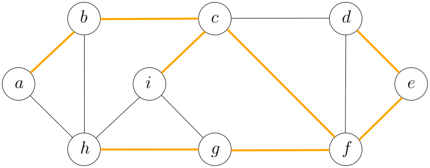


Outline

Spanning Tree

Def. Given a connected graph $G = (V, E)$, a spanning tree $T = (V, F)$ of G is a sub-graph of G that is a tree including all vertices V.





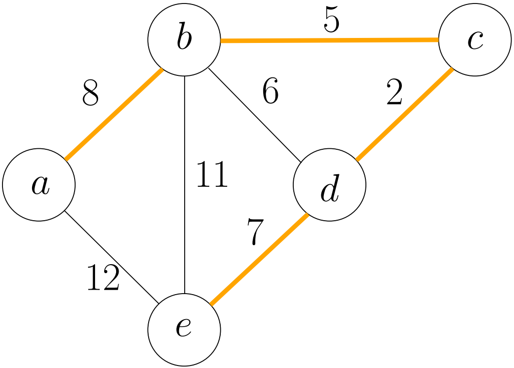
Lemma Let $T = (V, F)$ be a subgraph of $G = (V, E)$. The following statements are equivalent:

- T is a spanning tree of G;
- T is acyclic and connected;
- T is connected and has $n - 1$ edges;
- T is acyclic and has $n - 1$ edges;
- T is minimally connected: removal of any edge disconnects it;
- T is maximally acyclic: addition of any edge creates a cycle;
- T has a unique simple path between every pair of nodes.

Minimum Spanning Tree (MST) Problem

Input: Graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$

Output: the spanning tree T of G with the minimum total weight



Recall: Steps of Designing A Greedy Algorithm

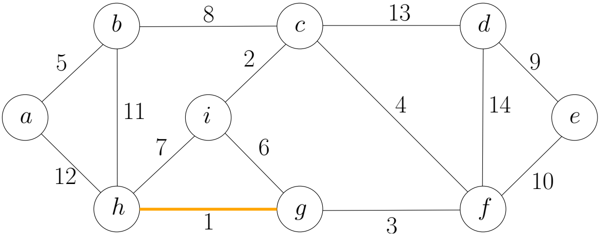
- Design a "reasonable" strategy
- Prove that the reasonable strategy is "safe" (key, usually done by "exchanging argument")
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually trivial)

Def. A choice is "safe" if there is an optimum solution that is "consistent" with the choice

Two Classic Greedy Algorithms for MST

- Kruskal's Algorithm
- Prim's Algorithm

Outline



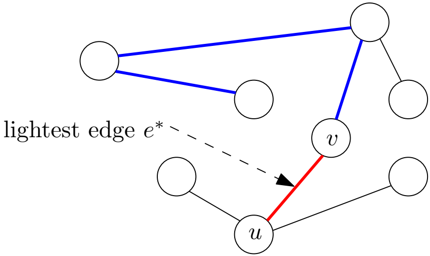
Q: Which edge can be safely included in the MST?

A: The edge with the smallest weight (lightest edge).

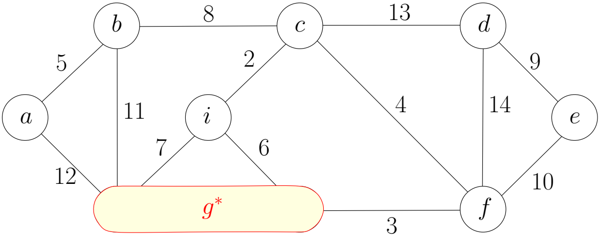
Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Proof.

- Take a minimum spanning tree T
- Assume the lightest edge e^* is not in T
- There is a unique path in T connecting u and v
- Remove any edge e in the path to obtain tree T'
- $w(e^*) \le w(e) \implies w(T') \le w(T)$: T' is also a MST $\square$

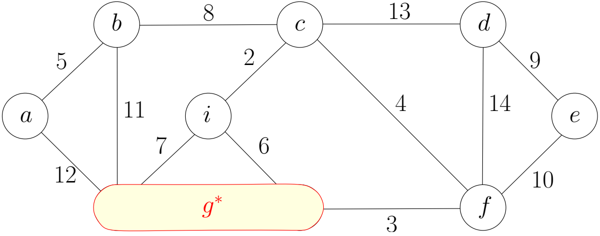


Is the Residual Problem Still a MST Problem?



- Residual problem: find the minimum spanning tree that contains edge (g, h)
- Contract the edge (g, h)
- Residual problem: find the minimum spanning tree in the contracted graph

Contraction of an Edge (u, v)



- Remove u and v from the graph, and add a new vertex u^*
- Remove all edges parallel connecting u to v from E
- For every edge $(u, w) \in E, w \neq v$, change it to (u^*, w)
- For every edge $(v, w) \in E, w \neq u$, change it to (u^*, w)
- May create parallel edges! E.g. : two edges (i, g^*)

Greedy Algorithm

Repeat the following step until G contains only one vertex:

1. Choose the lightest edge e^*, add e^* to the spanning tree
2. Contract e^* and update G be the contracted graph

Q: What edges are removed due to contractions?

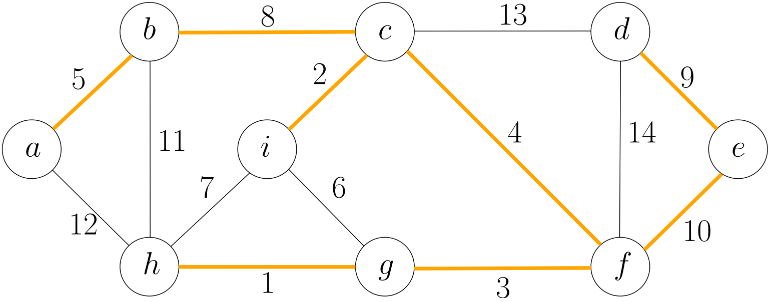
A: Edge (u, v) is removed if and only if there is a path connecting u and v formed by edges we selected

Greedy Algorithm

MST-Greedy(G, w)

```
F = ∅
sort edges in E in non-decreasing order of weights w
for each edge (u, v) in the order
    if u and v are not connected by a path of edges in F
        F = F ∪ {(u, v)}
return (V, F)
```

Kruskal's Algorithm: Example



Sets: $\{a, b, c, i, f, g, h, d, e\}$

Kruskal's Algorithm: Efficient Implementation of Greedy Algorithm

MST-Kruskal(G, w)

$F \leftarrow \emptyset$

$\mathcal{S} \leftarrow \{\{v\} : v \in V\}$

sort the edges of E in non-decreasing order of weights w

for each edge $(u, v) \in E$ in the order

  $S_u \leftarrow$ the set in $\mathcal{S}$ containing u

  $S_v \leftarrow$ the set in $\mathcal{S}$ containing v

  if $S_u \neq S_v$

    $F \leftarrow F \cup \{(u, v)\}$

    $\mathcal{S} \leftarrow \mathcal{S} \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$

return (V, F)

Running Time of Kruskal's Algorithm

MST-Kruskal(G, w)

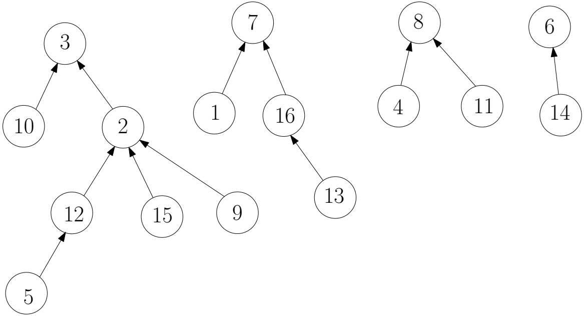
1. $F \leftarrow \emptyset$
2. $\mathcal{S} \leftarrow \{\{v\} : v \in V\}$
3. sort the edges of E in non-decreasing order of weights w
4. for each edge $(u, v) \in E$ in the order
5. $S_u \leftarrow$ the set in $\mathcal{S}$ containing u
6. $S_v \leftarrow$ the set in $\mathcal{S}$ containing v
7.   if $S_u \neq S_v$
8. $F \leftarrow F \cup \{(u, v)\}$
9. $\mathcal{S} \leftarrow \mathcal{S} \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
10. return (V, F)

Use union-find data structure to support ❷, ❺, ❻, ❼, ❾.

Union-Find Data Structure

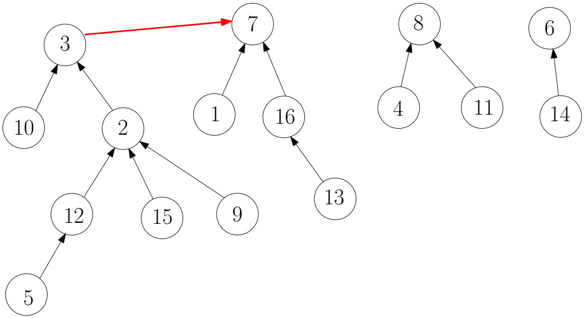
- V: ground set
- We need to maintain a partition of V and support following operations:
  - Check if u and v are in the same set of the partition
  - Merge two sets in partition

- $V = \{1, 2, 3, \cdots, 16\}$
- Partition:
  $\{2, 3, 5, 9, 10, 12, 15\}, \{1, 7, 13, 16\}, \{4, 8, 11\}, \{6, 14\}$



- $par[i]$: parent of i, ($par[i] =$ nil if i is a root).

Union-Find Data Structure



- Q: how can we check if u and v are in the same set?
- A: Check if $\text{root}(u) = \text{root}(v)$.
- $\text{root}(u)$: the root of the tree containing u
- Merge the trees with root r and r': $par[r] \leftarrow r'$.

Union-Find Data Structure

root(v)

1. if $par[v] =$ nil then
2. return v
3. else
4. return root($par[v]$)

root(v)

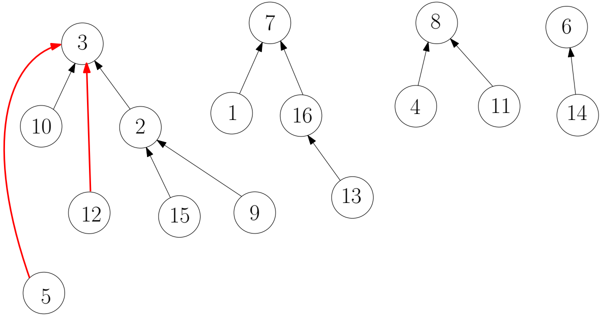
1. if par$[v] =$ nil then
2. return v
3. else
4. $par[v] \leftarrow$ root($par[v]$)
5. return $par[v]$

- Problem: the tree might too deep; running time might be large
- Improvement: all vertices in the path directly point to the root, saving time in the future.

Union-Find Data Structure

root(v)

if $par[v] = \text{nil}$ then
    return v
else
    $par[v] \leftarrow \text{root}(par[v])$
    return $par[v]$



MST-Kruskal(G, w)

$F \leftarrow \emptyset$

$\mathcal{S} \leftarrow \{\{v\} : v \in V\}$

sort the edges of E in non-decreasing order of weights w

for each edge $(u, v) \in E$ in the order

    $S_u \leftarrow$ the set in $\mathcal{S}$ containing u

    $S_v \leftarrow$ the set in $\mathcal{S}$ containing v

    if $S_u \neq S_v$

        $F \leftarrow F \cup \{(u, v)\}$

        $\mathcal{S} \leftarrow \mathcal{S} \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$

return (V, F)

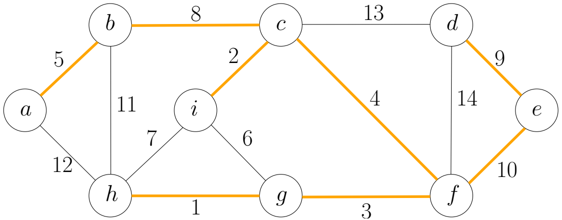
MST-Kruskal(G, w)

1. $F \leftarrow \emptyset$
2. for every $v \in V$: let $par[v] \leftarrow$ nil
3. sort the edges of E in non-decreasing order of weights w
4. for each edge $(u, v) \in E$ in the order
5. $u' \leftarrow \text{root}(u)$
6. $v' \leftarrow \text{root}(v)$
7.   if $u' \neq v'$
8. $F \leftarrow F \cup \{(u, v)\}$
9. $par[u'] \leftarrow v'$
10. return (V, F)

- 2,5,6,7,9 takes time $O(m\alpha(n))$
- $\alpha(n)$ is very slow-growing: $\alpha(n) \leq 4$ for $n \leq 10^{80}$.
- Running time $=$ time for 3 $= O(m \lg n)$.

Assumption Assume all edge weights are different.

Lemma An edge $e \in E$ is not in the MST, if and only if there is cycle C in G in which e is the heaviest edge.

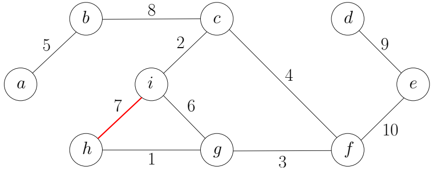


- (i, g) is not in the MST because of cycle (i, c, f, g)
- (e, f) is in the MST because no such cycle exists

Outline

Two Methods to Build a MST

1. Start from $F \leftarrow \emptyset$, and add edges to F one by one until we obtain a spanning tree
2. Start from $F \leftarrow E$, and remove edges from F one by one until we obtain a spanning tree



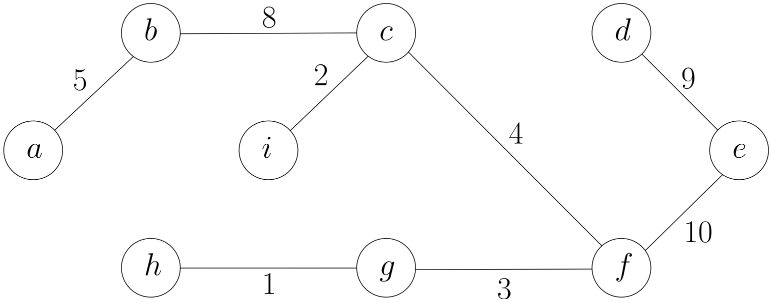
Lemma It is safe to exclude the heaviest non-bridge edge: there is a MST that does not contain the heaviest non-bridge edge.

Reverse Kruskal's Algorithm

MST-Greedy(G, w)

$F \leftarrow E$
sort E in non-increasing order of weights
for every e in this order
    if $(V, F \setminus \{e\})$ is connected then
        $F \leftarrow F \setminus \{e\}$
return (V, F)

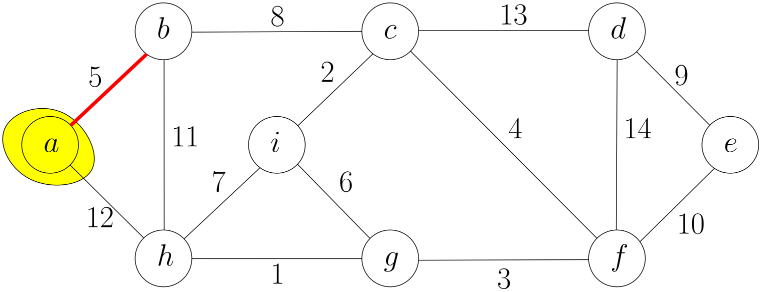
Reverse Kruskal's Algorithm: Example



Outline

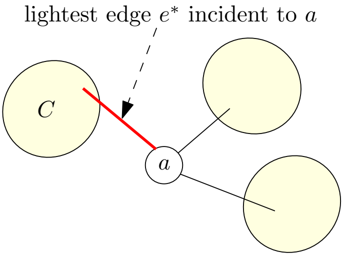
Design Greedy Strategy for MST

- Recall the greedy strategy for Kruskal's algorithm: choose the edge with the smallest weight.



- Greedy strategy for Prim's algorithm: choose the lightest edge incident to a.

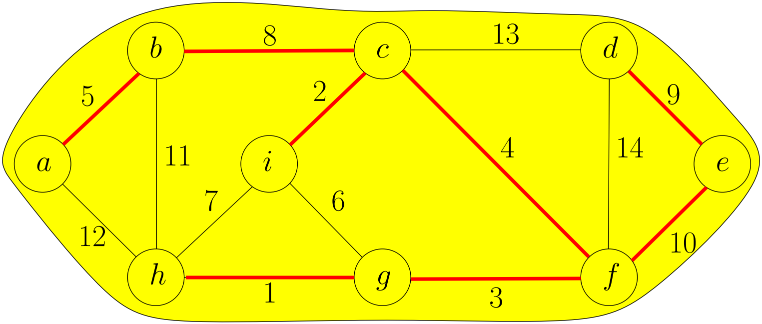
Lemma It is safe to include the lightest edge incident to a.



Proof.

- Let T be a MST
- Consider all components obtained by removing a from T
- Let e^* be the lightest edge incident to a and e^* connects a to component C
- Let e be the edge in T connecting a to C
- $T' = T \setminus e \cup \{e^*\}$ is a spanning tree with $w(T') \leq w(T)$ $\square$

Prim's Algorithm: Example



Greedy Algorithm

MST-Greedy1(G, w)

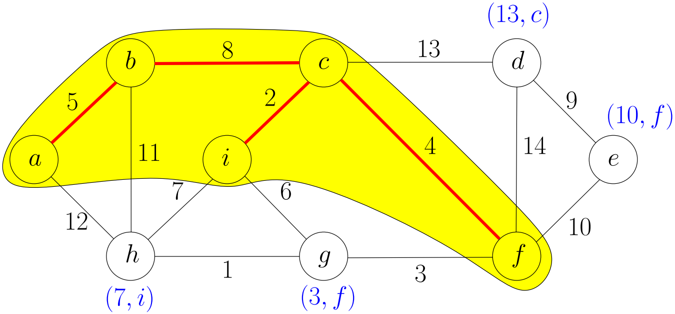
1. $S \leftarrow \{s\}$, where s is arbitrary vertex in V
2. $F \leftarrow \emptyset$
3. while $S \neq V$
4. $\quad (u, v) \leftarrow$ lightest edge between S and $V \setminus S$, where $u \in S$ and $v \in V \setminus S$
5. $\quad S \leftarrow S \cup \{v\}$
6. $\quad F \leftarrow F \cup \{(u, v)\}$
7. return (V, F)

- Running time of naive implementation: $O(nm)$

Prim's Algorithm: Efficient Implementation of Greedy Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v)\in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v)\in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S



Prim's Algorithm: Efficient Implementation of Greedy Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v) \in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v) \in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S

In every iteration

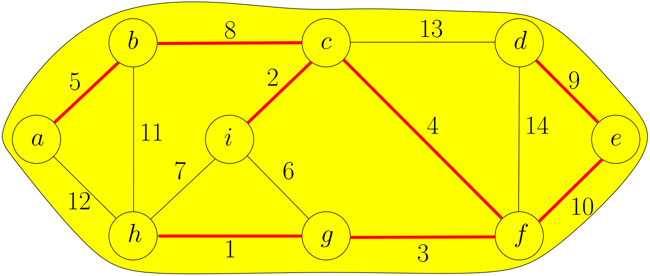
- Pick $u \in V \setminus S$ with the smallest $d(u)$ value
- Add $(\pi(u), u)$ to F
- Add u to S, update d and π values.

Prim's Algorithm

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$, do
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$
        if $w(u, v) < d(v)$ then
            $d(v) \leftarrow w(u, v)$
            $\pi(v) \leftarrow u$
return $\{(u, \pi(u)) | u \in V \setminus \{s\}\}$

Example



Prim's Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v) \in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v) \in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S

In every iteration

- Pick $u \in V \setminus S$ with the smallest $d(u)$ value   extract_min
- Add $(\pi(u), u)$ to F
- Add u to S, update d and π values.   decrease_key

Use a priority queue to support the operations

Def. A priority queue is an abstract data structure that maintains a set U of elements, each with an associated key value, and supports the following operations:

- insert(v, key_value): insert an element v, whose associated key value is key_value.
- decrease_key(v, new_key_value): decrease the key value of an element v in queue to new_key_value
- extract_min$()$: return and remove the element in queue with the smallest key value
- $\cdots$

Prim's Algorithm

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$

while $S \neq V$, do
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$
        if $w(u, v) < d(v)$ then
            $d(v) \leftarrow w(u, v)$
            $\pi(v) \leftarrow u$
return $\big\{(u, \pi(u)) | u \in V \setminus \{s\}\big\}$

Prim's Algorithm Using Priority Queue

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G

$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$

$Q \leftarrow$ empty queue, for each $v \in V$: Q.insert$(v, d(v))$

while $S \neq V$, do

    $u \leftarrow Q$.extract_min()

    $S \leftarrow S \cup \{u\}$

    for each $v \in V \setminus S$ such that $(u, v) \in E$

        if $w(u, v) < d(v)$ then

            $d(v) \leftarrow w(u, v)$, Q.decrease_key$(v, d(v))$

            $\pi(v) \leftarrow u$

return $\{(u, \pi(u)) | u \in V \setminus \{s\}\}$

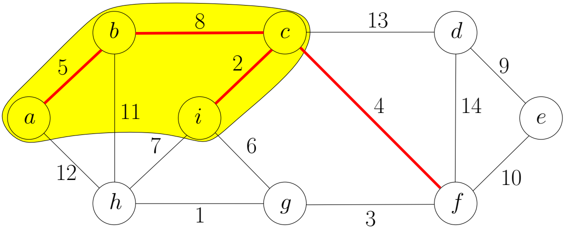
Running Time of Prim's Algorithm Using Priority Queue

$O(n)\times$ (time for extract_min) $+\ O(m)\times$ (time for decrease_key)

concrete DS	extract_min	decrease_key	overall time
heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Assumption Assume all edge weights are different.

Lemma (u,v) is in MST, if and only if there exists a cut $(U, V \setminus U)$, such that (u,v) is the lightest edge between U and $V \setminus U$.



- (c,f) is in MST because of cut $\big(\{a,b,c,i\}, V \setminus \{a,b,c,i\}\big)$
- (i,g) is not in MST because no such cut exists

"Evidence" for $e \in$ MST or $e \notin$ MST

Assumption Assume all edge weights are different.

- $e \in$ MST $\leftrightarrow$ there is a cut in which e is the lightest edge
- $e \notin$ MST $\leftrightarrow$ there is a cycle in which e is the heaviest edge

Exactly one of the following is true:

- There is a cut in which e is the lightest edge
- There is a cycle in which e is the heaviest edge

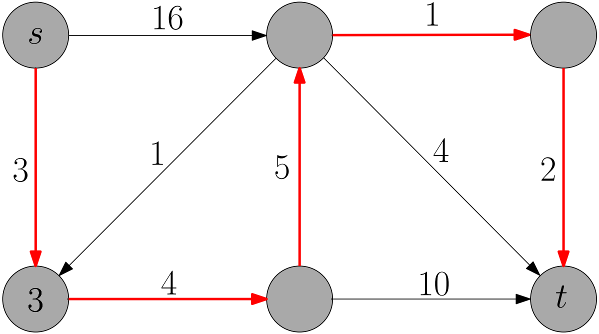
Thus, the minimum spanning tree is unique with assumption.

Outline

s-t Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s, t \in V$
$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest path from s to t



Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

Reason for Considering Single Source Shortest Paths Problem

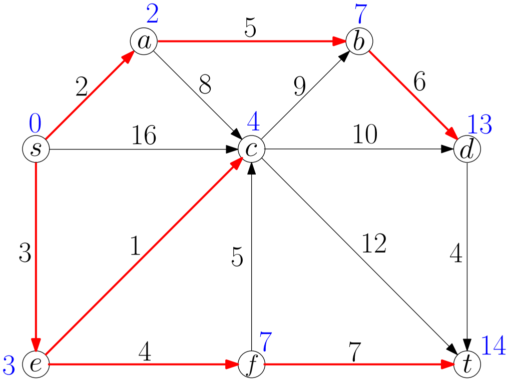
- We do not know how to solve s-t shortest path problem more efficiently than solving single source shortest path problem

- Shortest paths in directed graphs is more general than in undirected graphs: we can replace every undirected edge with two anti-parallel edges of the same weight

- Shortest path from s to v may contain $\Omega(n)$ edges
- There are $\Omega(n)$ different vertices v
- Thus, printing out all shortest paths may take time $\Omega(n^2)$
- Not acceptable if graph is sparse

Shortest Path Tree

- $O(n)$-size data structure to represent all shortest paths
- For every vertex v, we only need to remember the parent of v: second-to-last vertex in the shortest path from s to v (why?)



Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

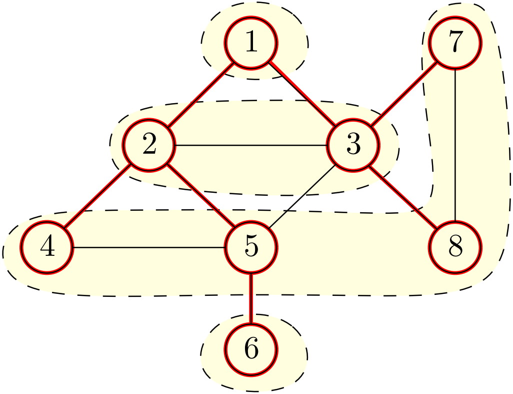
$w : E \to \mathbb{R}_{\geq 0}$

Output: $\pi(v), v \in V \setminus s$: the parent of v

$d(v), v \in V \setminus s$: the length of shortest path from s to v

Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s



Assumption Weights $w(u,v)$ are integers (w.l.o.g).

- An edge of weight $w(u,v)$ is equivalent to a pah of $w(u,v)$ unit-weight edges

Shortest Path Algorithm by Running BFS

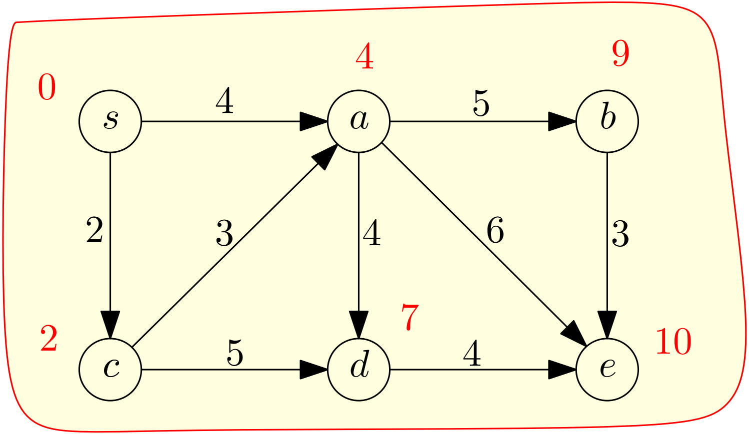
1. replace (u,v) of length $w(u,v)$ with a path of $w(u,v)$ unit-weight edges, for every $(u,v) \in E$
2. run BFS virtually
3. $\pi(v) =$ vertex from which v is visited
4. $d(v) =$ index of the level containing v

- Problem: $w(u,v)$ may be too large!

Shortest Path Algorithm by Running BFS Virtually

1. $S \leftarrow \{s\}, d(s) \leftarrow 0$
2. while $|S| \leq n$
3. find a $v \notin S$ that minimizes $\displaystyle\min_{u \in S:(u,v) \in E} \{d(u) + w(u,v)\}$
4. $S \leftarrow S \cup \{v\}$
5. $d(v) \leftarrow \min_{u \in S:(u,v) \in E} \{d(u) + w(u,v)\}$

Virtual BFS: Example



Time 10

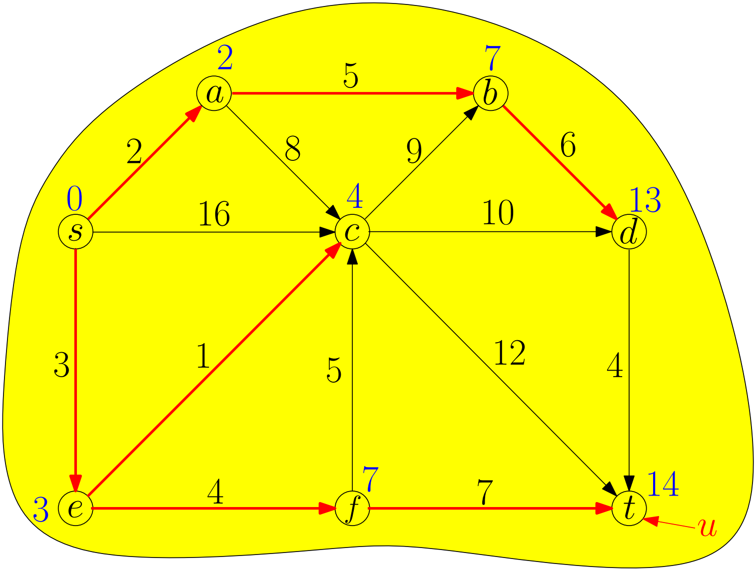
Outline

Dijkstra's Algorithm

Dijkstra(G, w, s)

$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ do
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    add u to S
    for each $v \in V \setminus S$ such that $(u, v) \in E$
        if $d(u) + w(u, v) < d(v)$ then
            $d(v) \leftarrow d(u) + w(u, v)$
            $\pi(v) \leftarrow u$
return (d, π)

- Running time $= O(n^2)$


0
s
2
a
7
b
4
c
13
d
3
e
7
f
14
t
u
5
2
8
9
6
16
10
3
1
5
12
4
4
7

Improved Running Time using Priority Queue

Dijkstra(G, w, s)

$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$

$Q \leftarrow$ empty queue, for each $v \in V$: Q.insert$(v, d(v))$

while $S \neq V$, do

    $u \leftarrow Q$.extract_min()

    $S \leftarrow S \cup \{u\}$

    for each $v \in V \setminus S$ such that $(u, v) \in E$

        if $d(u) + w(u, v) < d(v)$ then

            $d(v) \leftarrow d(u) + w(u, v)$, Q.decrease_key$(v, d(v))$

            $\pi(v) \leftarrow u$

return (π, d)

Recall: Prim's Algorithm for MST

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G

$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$

$Q \leftarrow$ empty queue, for each $v \in V$: Q.insert$(v, d(v))$

while $S \neq V$, do

    $u \leftarrow Q$.extract_min()

    $S \leftarrow S \cup \{u\}$

    for each $v \in V \setminus S$ such that $(u, v) \in E$

        if $w(u, v) < d(v)$ then

            $d(v) \leftarrow w(u, v)$, Q.decrease_key$(v, d(v))$

            $\pi(v) \leftarrow u$

return $\big\{(u, \pi(u)) | u \in V \setminus \{s\}\big\}$

Improved Running Time

Running time:
$O(n) \times$ (time for extract_min) $+ O(m) \times$ (time for decrease_key)

Priority-Queue	extract_min	decrease_key	Time
Heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci Heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Outline

Encoding Symbols Using Bits

- assume: 8 symbols a, b, c, d, e, f, g, h in a language
- need to encode a message using bits
- idea: use 3 bits per symbol

a	b	c	d	e	f	g	h
000	001	010	011	100	101	110	111

$$deacfg \rightarrow 011100000010101110$$

Q: Can we have a better encoding scheme?

- Seems unlikely: must use 3 bits per symbol

Q: What if some symbols appear more frequently than the others in expectation?

Q: If some symbols appear more frequently than the others in expectation, can we have a better encoding scheme?

A: Maybe. Using variable-length encoding scheme.

Idea

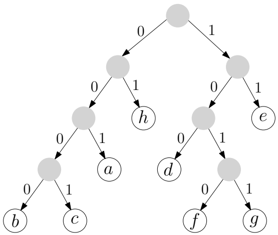
- using fewer bits for symbols that are more frequently used, and more bits for symbols that are less frequently used.

Need to use prefix codes to guarantee a unique decoding.

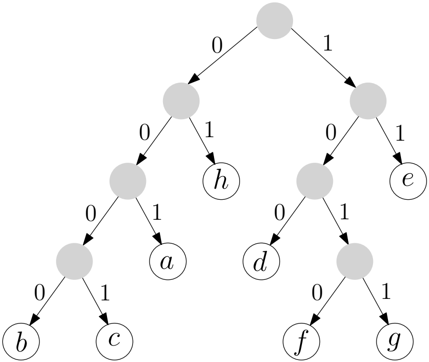
Prefix Codes

Def. A prefix code for a set S of symbols is a function $\gamma : S \to \{0,1\}^*$ such that for two distinct $x, y \in S$, $\gamma(x)$ is not a prefix of $\gamma(y)$.

a	b	c	d
001	0000	0001	100
e	f	g	h
11	1010	1011	01



- 0001/001/100/0000/01/01/11/1010/0001/001/
- cadbhhefca



Properties of Encoding Tree

- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some symbol
- If coding scheme is not wasteful: a non-leaf has exactly two children

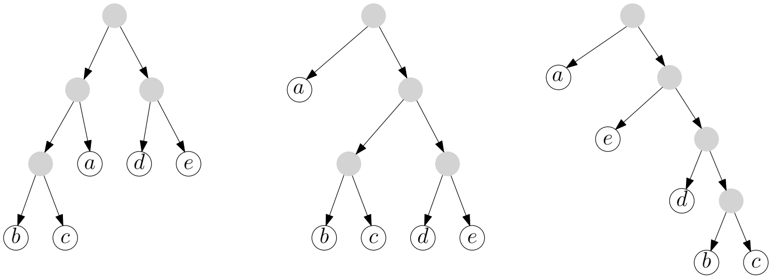
Best Prefix Codes

Input: frequencies of letters in a message

Output: prefix coding scheme giving the shortest encoding for the message

example

symbols	a	b	c	d	e	
frequencies	18	3	4	6	10	
scheme 1 length	2	3	3	2	2	total $=$ 89
scheme 2 length	1	3	3	3	3	total $=$ 87
scheme 3 length	1	4	4	3	2	total $=$ 84



scheme 1 scheme 2 scheme 3

- Example Input: (a: 18, b: 3, c: 4, d: 6, e: 10)

Q: What types of decisions should we make?

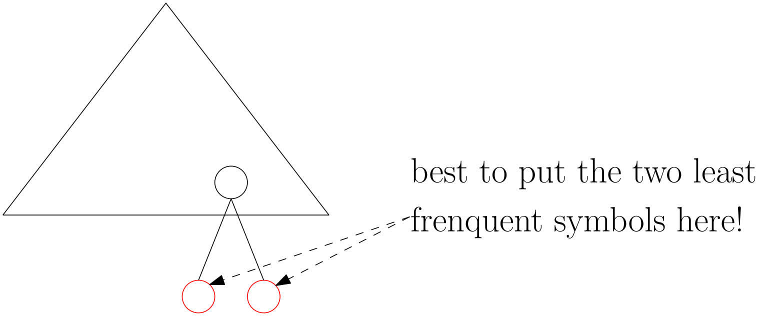
- the code for some letter?
- hard to design a strategy; residual problem is complicated.

- a partition of letters into left and right sub-trees?
- not clear how to design the greedy algorithm

A: Choose two letters and make them brothers in the tree.

Which Two symbols Can Be Safely Put Together As Brothers?

- Focus a tree structure, without leaf labeling
- There are two deepest leaves that are brothers
- It is safe to make the two least frequent symbols brothers!



- It is safe to make the two least frequent symbols brothers!

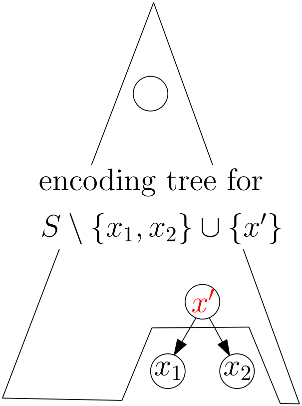
Lemma There is an optimum encoding tree, where the two least frequent symbols are brothers.

- So we can make the two least frequent symbols brothers; the decision is irrevocable.

Q: Is the residual problem an instance of the best prefix codes problem?

A: Yes, although the answer is not immediate.

- f_x: the frequency of the symbol x in the support.
- x_1 and x_2: the two symbols we decided to put together.
- d_x the depth of symbol x in our output encoding tree.



$$
\begin{aligned}
& \sum_{x \in S} f_x d_x \\
= & \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
= & \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
= & \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1) \\
= & \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\end{aligned}
$$

Def: $f_{x'} = f_{x_1} + f_{x_2}$

In order to minimize

$$\sum_{x \in S} f_x d_x,$$

we need to minimize

$$\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x,$$

subject to that d is the depth function for an encoding tree of $S \setminus \{x_1, x_2\}$.

- This is exactly the best prefix codes problem, with symbols $S \setminus \{x_1, x_2\} \cup \{x'\}$ and frequency vector f!

Huffman codes: Recursive Algorithm

Huffman(S, f)

if $|S| > 1$ **then**

  let x_1, x_2 be the two symbols with the smallest f values

  introduce a new symbol x' and let $f_{x'} = f_{x_1} + f_{x_2}$

  $S' \leftarrow S \setminus \{x_1, x_2\} \cup \{x'\}$

  call Huffman$(S', f|_{S'})$ to build an encoding tree T'

  let T be obtained from T' by adding x_1, x_2 as two children of x'

  return T

else

  let x be the symbol in S

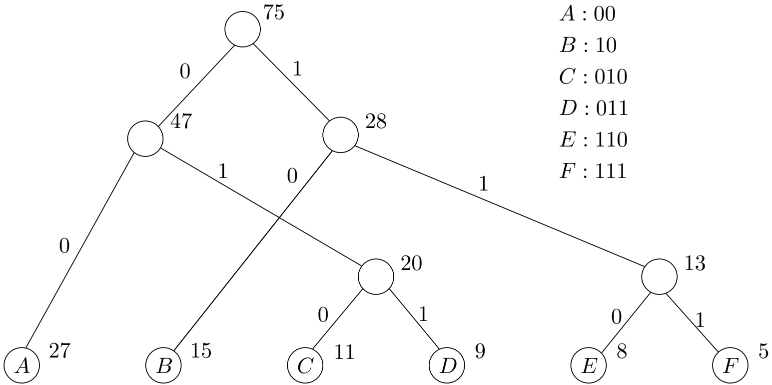
  return a tree with a single node labeled x

Huffman codes: Iterative Algorithm

Huffman(S, f)

1. **while** $|S| > 1$ **do**
2. let x_1, x_2 be the two symbols with the smallest f values
3. introduce a new symbol x' and let $f_{x'} = f_{x_1} + f_{x_2}$
4. let x_1 and x_2 be the two children of x'
5. $S \leftarrow S \setminus \{x_1, x_2\} \cup \{x'\}$
6. **return** the tree constructed

Example



$A : 00$
$B : 10$
$C : 010$
$D : 011$
$E : 110$
$F : 111$

Algorithm using Priority Queue

Huffman(S, f)

$Q \leftarrow$ build-priority-queue(S)
while $Q.\text{size} > 1$ **do**
    $x_1 \leftarrow Q.\text{extract-min}()$
    $x_2 \leftarrow Q.\text{extract-min}()$
    introduce a new symbol x' and let $f_{x'} = f_{x_1} + f_{x_2}$
    let x_1 and x_2 be the two children of x'
    $Q.\text{insert}(x')$
return the tree constructed

Outline

Summary for Greedy Algorithms

1. Design a "reasonable" strategy
   - Interval scheduling problem: schedule the job j^* with the earliest deadline
   - Kruskal's algorithm for MST: select lightest edge e^*
   - Inverse Kruskal's algorithm for MST: drop the heaviest non-bridge edge e^*
   - Prim's algorithm for MST: select the lightest edge e^* incident to a specified vertex s
   - Huffman codes: make the two least frequent symbols brothers

Summary for Greedy Algorithms

1. Design "reasonable" strategy
2. Prove that the reasonable strategy is "safe"

Def. A choice is "safe" if there is an optimum solution that is "consistent" with the choice

- Usually done by "exchange argument"
- Interval scheduling problem: exchange j^* with the first job in an optimal solution
- Kruskal's algorithm: exchange e^* with some edge e in the cycle in $T \cup \{e^*\}$
- Prim's algorithm: exchange e^* with some other edge e incident to s

Summary for Greedy Algorithms

1. Design "reasonable" strategy
2. Prove that the reasonable strategy is "safe"
3. Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
   - Interval scheduling problem: remove j^* and the jobs it conflicts with
   - Kruskal and Prim's algorithms: contracting e^*
   - Inverse Kruskal's algorithm: remove e^*
   - Huffman codes: merge two symbols into one

Summary for Greedy Algorithms

- Dijkstra's algorithm does not quite fit in the framework.
- It combines "greedy algorithm" and "dynamic programming"
- Greedy algorithm: each time select the vertex in $V \setminus S$ with the smallest d value and add it to S
- Dynamic programming: remember the d values of vertices in S for future use
- Dijkstra's algorithm is very similar to Prim's algorithm for MST