Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Greedy Algorithms

– Algorithms that compute solutions

by *repeatedly taking* *locally optimal choices*

Example 1.

Activity-Selection Problem

Activity-Selection Problem

- You given a set of *activities* $a_1, a_2, \ldots, a_n$, where $\boldsymbol{a_i} = [\,\boldsymbol{s_i}, \boldsymbol{f_i}\,)$ and s_i, f_i denote the ***start time*** and ***finish time*** of the i^{th}-activity.

- Select a ***maximum-cardinality subset of activities*** to be scheduled in a conference room.

  – That is, a subset $A \subseteq \{1,2,\ldots,n\}$ such that

  $$a_i \cap a_j = \emptyset$$

  for all $i, j \in A$ with $i \neq j$ and $|A|$ is as large as possible.

Activity-Selection Problem

- You given a set of *activities* $a_1, a_2, \dots, a_n$, where $\boldsymbol{a_i} = [\,\boldsymbol{s_i}, \boldsymbol{f_i}\,)$ and s_i, f_i denote the ***start time*** and ***finish time*** of the i^{th}-activity.
  - For example,

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

we can select $\{a_4, a_8\}$, $\{a_1, a_4, a_9\}$, or $\{a_3, a_8, a_{11}\}$.

  - The optimal solution is $\{\,a_1, a_4, a_8, a_{11}\,\}$.

A Classic EDF-based Greedy Algorithm

- A classic solution to this problem is to schedule the activities based on the ***Earliest Deadline First (EDF) principle***.

1. Consider the activities *in sorted order of **their finish times***, and ***schedule*** the activities ***whenever possible***.
2. Output the schedule.

Is this algorithm correct? How can we prove it?

Observation

- Consider the set of activities selected by the EDF algorithm.
  - For any $i \geq 1$, let $\pi(i)$ be the index of the i^{th} activity.

- The algorithm scheduled the activity $a_{\pi(1)} = \left(s_{\pi(1)}, f_{\pi(1)}\right)$.
  - Hence, we know that ***any optimal solution*** can schedule ***at most one activity*** up to time $f_{\pi(1)}$.

If ***at least two activities*** are scheduled before time $f_{\pi(1)}$

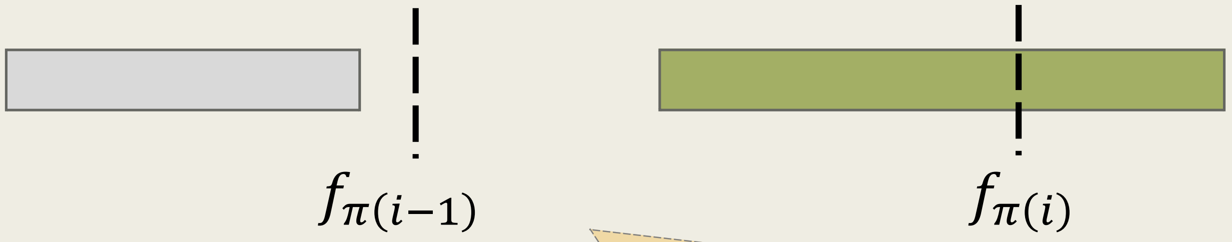
Then, this activity ***must have*** *an earlier finish time* than $a_{\pi(1)}$, a contradiction.

Observation

- The algorithm scheduled the activity $a_{\pi(1)} = \left(s_{\pi(1)}, f_{\pi(1)}\right)$.
  - Hence, we know that ***any optimal solution*** can schedule ***at most one activity*** up to time $f_{\pi(1)}$.
  - There exists an optimal solution that schedules the activity $a_{\pi(1)}$.
    - If one optimal solution doesn't do so, we can ***safely replace the activity it selects*** with the earliest finish time **with $\boldsymbol{a_{\pi(1)}}$**.
  - The same argument generalizes to $a_{\pi(i)}$ for any $i > 1$.

Observation

- For any $i > 1$, suppose that there exists an optimal solution that schedules $a_{\pi(1)}, \dots, a_{\pi(i-1)}$.
  - Since the algorithm chooses to schedule $a_{\pi(i)}$, ***any feasible schedule*** can select ***at most one activity*** between $f_{\pi(i-1)}$ and $f_{\pi(i)}$.



There **cannot be** more than one compatible activity in between.

Observation

- For any $i > 1$, suppose that there exists an optimal solution that schedules $a_{\pi(1)}, \ldots, a_{\pi(i-1)}$.
  - Since the algorithm chooses to schedule $a_{\pi(i)}$, ***any feasible schedule*** can select ***at most one activity*** between $f_{\pi(i-1)}$ and $f_{\pi(i)}$.
- Hence, **there exists an optimal solution** that schedules $a_{\pi(1)}, \ldots, a_{\pi(i)}$.
  - This holds for all $i \geq 1$. Hence the EDF algorithm is optimal.

Elements of Greedy Algorithms

When is greedy algorithms applicable in general?

Elements of Greedy Algorithms

- Problems ***that can be solved <u>by greedy algorithms</u>*** exhibits the following properties.

  1. ***<u>Optimal Substructure</u>*** – An optimal solution to the problem contains within it optimal solutions to subproblems.

  2. ***<u>Greedy-Choice Property</u>*** – A globally optimal solution can be assembled by making a sequence of locally optimal (greedy) choices.

The Correctness of a Greedy Algorithm

- In general, to prove the correctness of a greedy algorithm, you need to show that...
  - For ***the greedy choices*** made by the algorithm *up to any moment*,

    there always exists an optimal solution that takes *the same set of decisions*.

How can this be proved in general?

- In general, to prove the correctness of a greedy algorithm, you need to show that...
  - For ***the greedy choi***
    up to any moment, ***th***
    that exhibits *the same*
  - Take any optimal solution.

    Show that, ***switching to your choices*** ***is never worse***.
    - This often involves *proving by induction*,

      i.e., for any $i \geq 1$, the first i choices are always optimal.

For this step, it requires ***optimal substructure*** and ***greedy choice property*** from the problem.

Example 2.

Huffman Codes

Optimal prefix-free code used for data compression.

Data Compression – The Scenario

- We have a string $s \in \Pi^*$,
  where Π is the set of alphabets we consider.

- We want to **encode** each character $\alpha \in \Pi$ with a bit string $\{0,1\}^*$ such that
  - The ***total number of bits*** used to represent s is as small as possible.
  - The encoding of s ***can be (uniquely) decoded*** back to s.

Binary Prefix-Free Codes

- Let $\mathrm{enc} : \Pi \mapsto \{0,1\}^*$ be a function that encodes the characters in Π with a bit string.

- The encoding enc is **prefix-free** if none of the codewords is a prefix of another.

  Decoding is very simple.

  – Hence, the encoded string $\mathrm{enc}(s)$ is ***never ambiguous*** when parsing in order.

- Question: How can we compute a prefix-free coding enc for Π such that $\mathrm{enc}(s)$ has a minimum length possible.

Characterization of Binary Prefix-Free Codes

- Let $\text{enc} : \Pi \mapsto \{0,1\}^*$ be a prefix-free encoding of the characters in Π.
  - Let $|\Pi| = n$.
- Observe that, each of such functions ***corresponds to a binary tree* with n leaf nodes**, where
  - Each character in Π is stored in one leaf node, and
  - Each leaf node stores one character in Π.
- Hence, it suffices to consider binary trees with $n = |\Pi|$ leaves.

Huffman Code

- Huffman code is an optimal prefix-free coding that can be used for data compression.
  - It compresses data well – savings of 20% to 90% are typical.
  - It is **optimal** when ***prefix-free codes*** are to be used.
    - If non-prefix-free codes are allowed, better encoding is possible.

Huffman Code

- Let $s \in \Pi^*$ be the string to be compressed.
  - For each character $\alpha \in \Pi$, let p_α denote the frequency of α in s.

W.L.O.G., we may assume that $|\Pi| > 1$ and $p_\alpha > 0$.

- Goal – Compute a binary tree T with n leaves and assign each character in Π to one leaf node such that

$$\sum_{\alpha \in \Pi} p_\alpha \cdot d_T(\alpha)$$

Length of the encoding of s.

is minimized, where $d_T(\alpha)$ is the depth of α in T.

W.L.O.G., $|\Pi| > 1$, $p_\alpha > 0$.

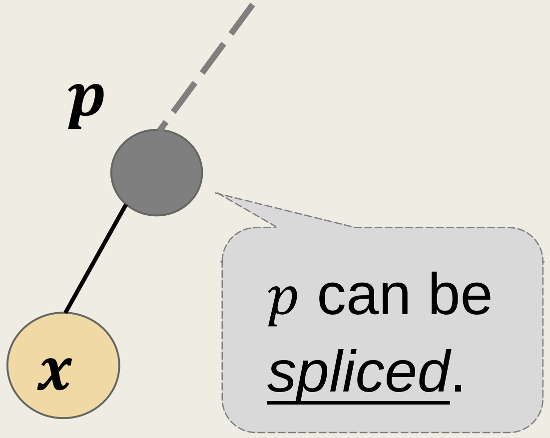
Observing the Optimal Solutions

- Let T be an optimal binary tree for (Π, s).

Observation 1.

Let x be a leaf node with the maximum depth, and p be the parent of x. Then p must have two children nodes.

  - If not, the depth of x can be decreased by 1, and the quality of T can be strictly improved.
  - A contradiction to the optimality of T.



W.L.O.G., $|\Pi| > 1$, $p_\alpha > 0$.

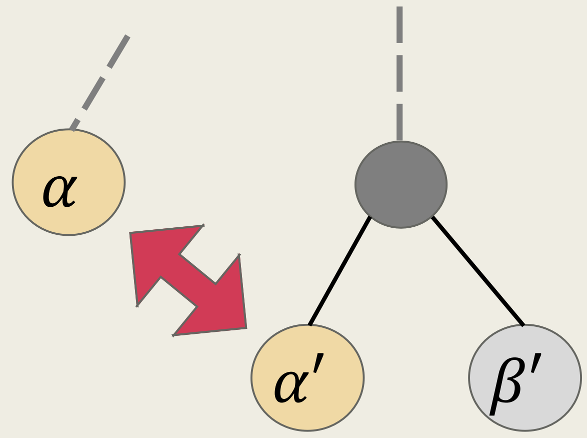
- Let T be an optimal binary tree for (Π, s).
  - Let u and v be two sibling leaf nodes with maximum depth.
  - Let $\alpha, \beta \in \Pi$ be two ***characters with the lowest frequencies***.

Observation 2.

If α, β are not stored at u and v,
swapping them there never worsens the quality of the tree.

By the setting, we have

$$d_T(\alpha) \le d_T(\alpha') \text{ and } p_\alpha \le p_{\alpha'}.$$

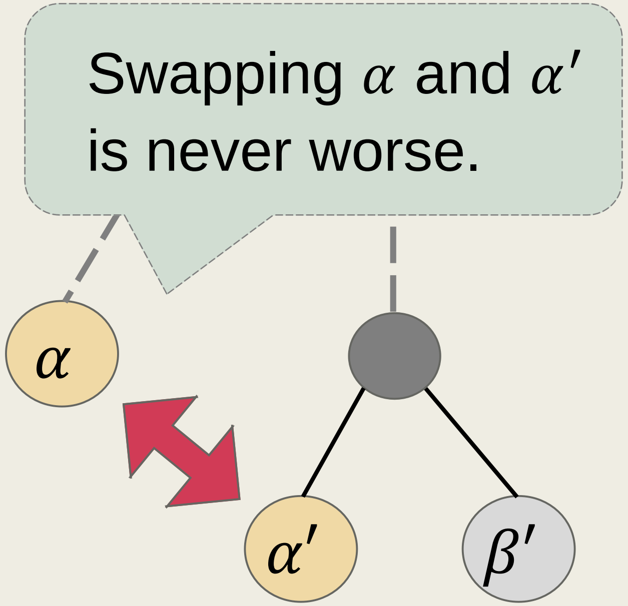


Observation 2.

If α, β are not stored at u and v,
swapping them there never worsens the quality of the tree.

- By the setting, we have $d_T(\alpha) \leq d_T(\alpha')$ and $p_\alpha \leq p_{\alpha'}$.
- Let T' be the tree obtained by swapping α and α'.

$$
\begin{aligned}
& \mathrm{len}(T) - \mathrm{len}(T') \\
= \; & p_\alpha \cdot \left(d_T(\alpha) - d_T(\alpha')\right) + p_{\alpha'} \cdot \left(d_T(\alpha') - d_T(\alpha)\right) \\
= \; & \left(d_T(\alpha') - d_T(\alpha)\right) \cdot \left(p_{\alpha'} - p_\alpha\right) \geq 0 .
\end{aligned}
$$



W.L.O.G., $|\Pi| > 1$, $p_\alpha > 0$.

Observing the Optimal Solutions

- Let $\alpha, \beta \in \Pi$ be two characters with the lowest frequencies in s.
- From Observation 1 and Observation 2,
  we know that
  - ***There exists an optimal tree T*** that places α and β as two sibling leaf nodes.
  - Hence, it is equivalent to replace α and β with a new character z with frequency $p_z \coloneqq p_\alpha + p_\beta$.
  - Then we can ***repeat this argument*** until $|\Pi| = 1$.

- The Huffman code is constructed by the following greedy algorithm.

- Huffman(Π, p) – Π is the alphabets with frequency p.

A. Let Q be a min-heap for (Π, p).

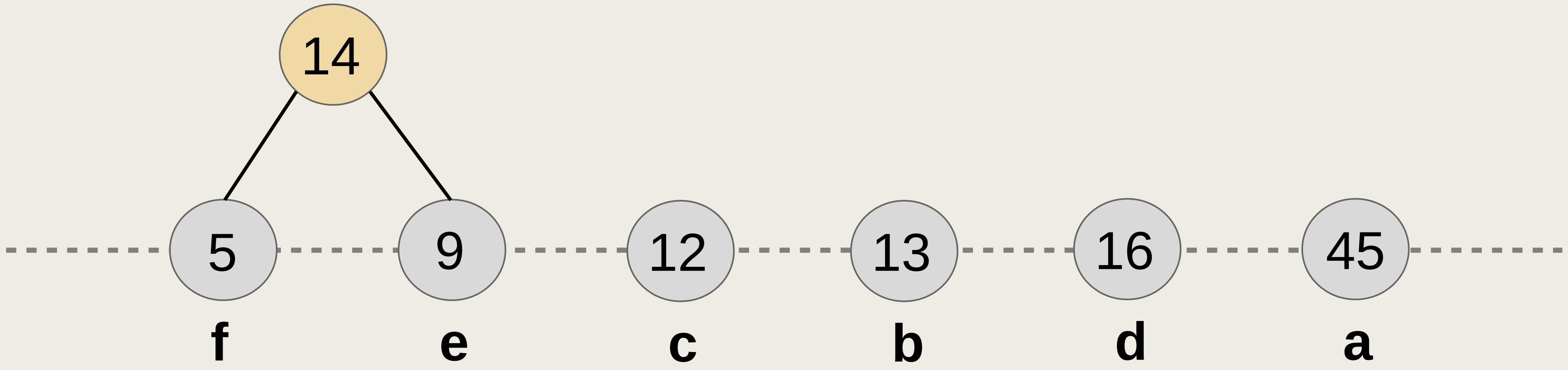
B. While $|Q| > 1$, repeat the following.

   1. Let $x \leftarrow$ Extract-Min(Q) and $y \leftarrow$ Extract-Min(Q).
   2. Create a new node z

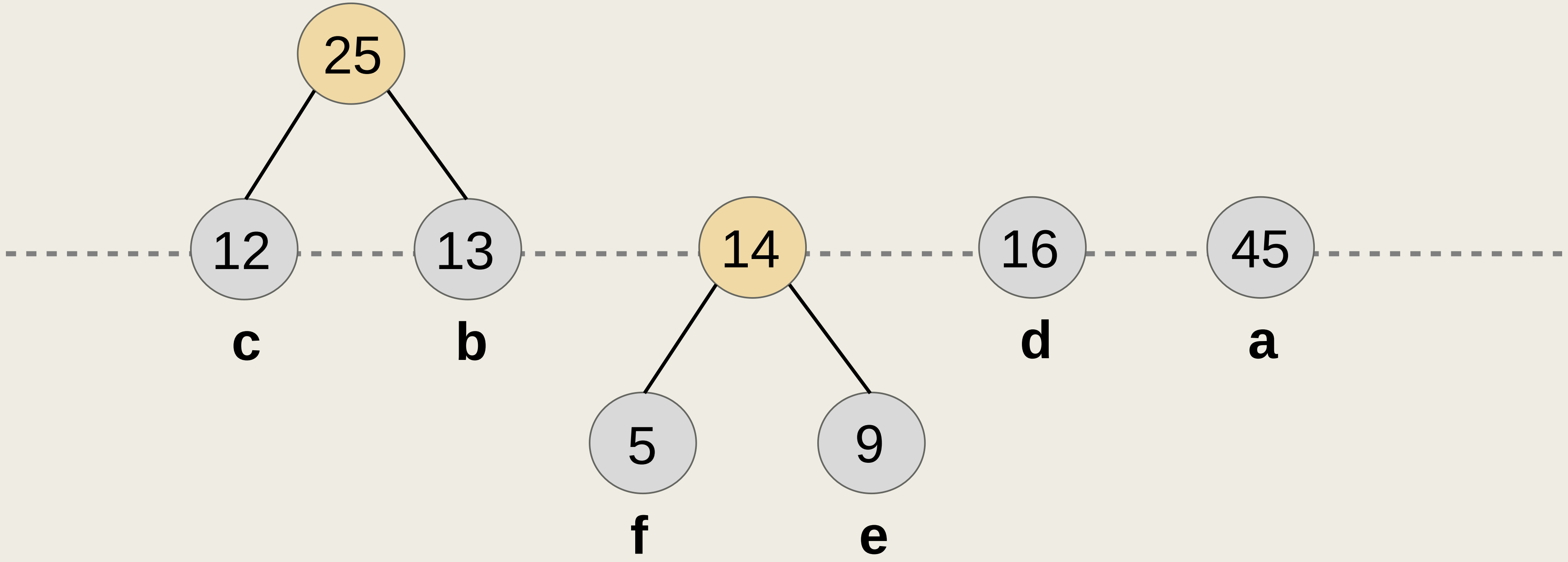
      with left-child x, right-child y, and $p_z \coloneqq p_x + p_y$.
   3. Insert (z, p_z) into Q.

C. Return Extract-Min(Q).

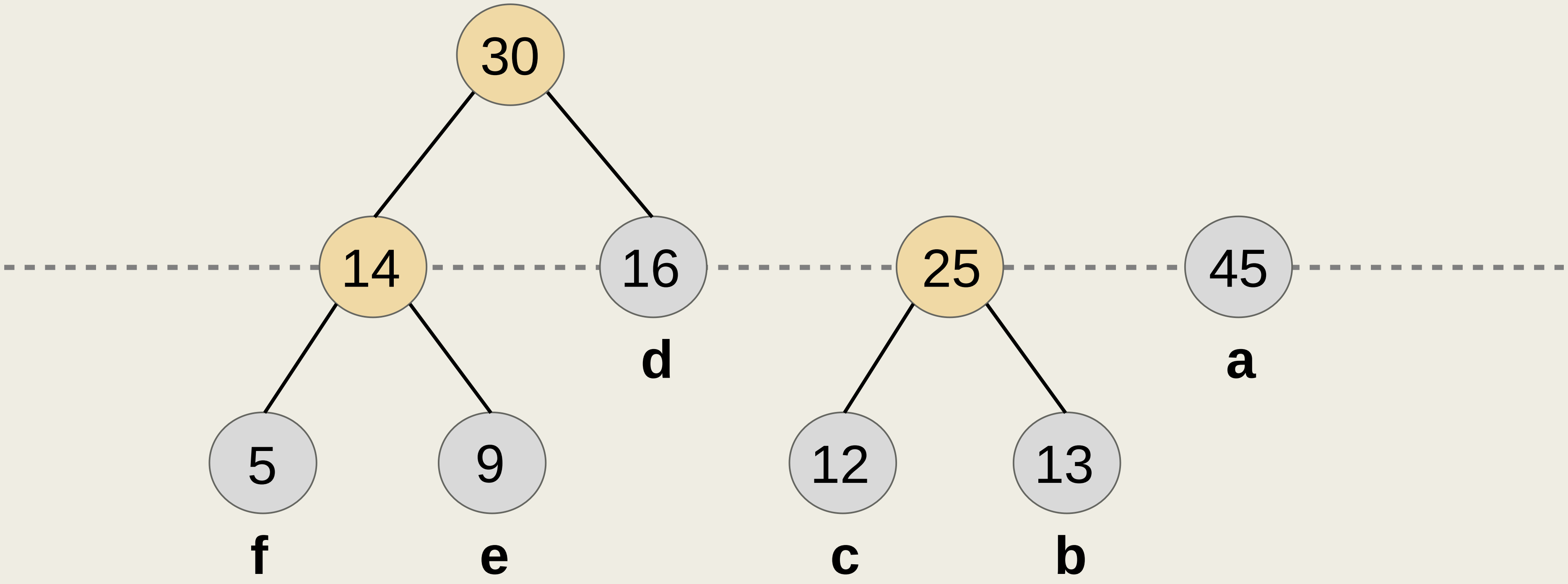
Huffman Codes



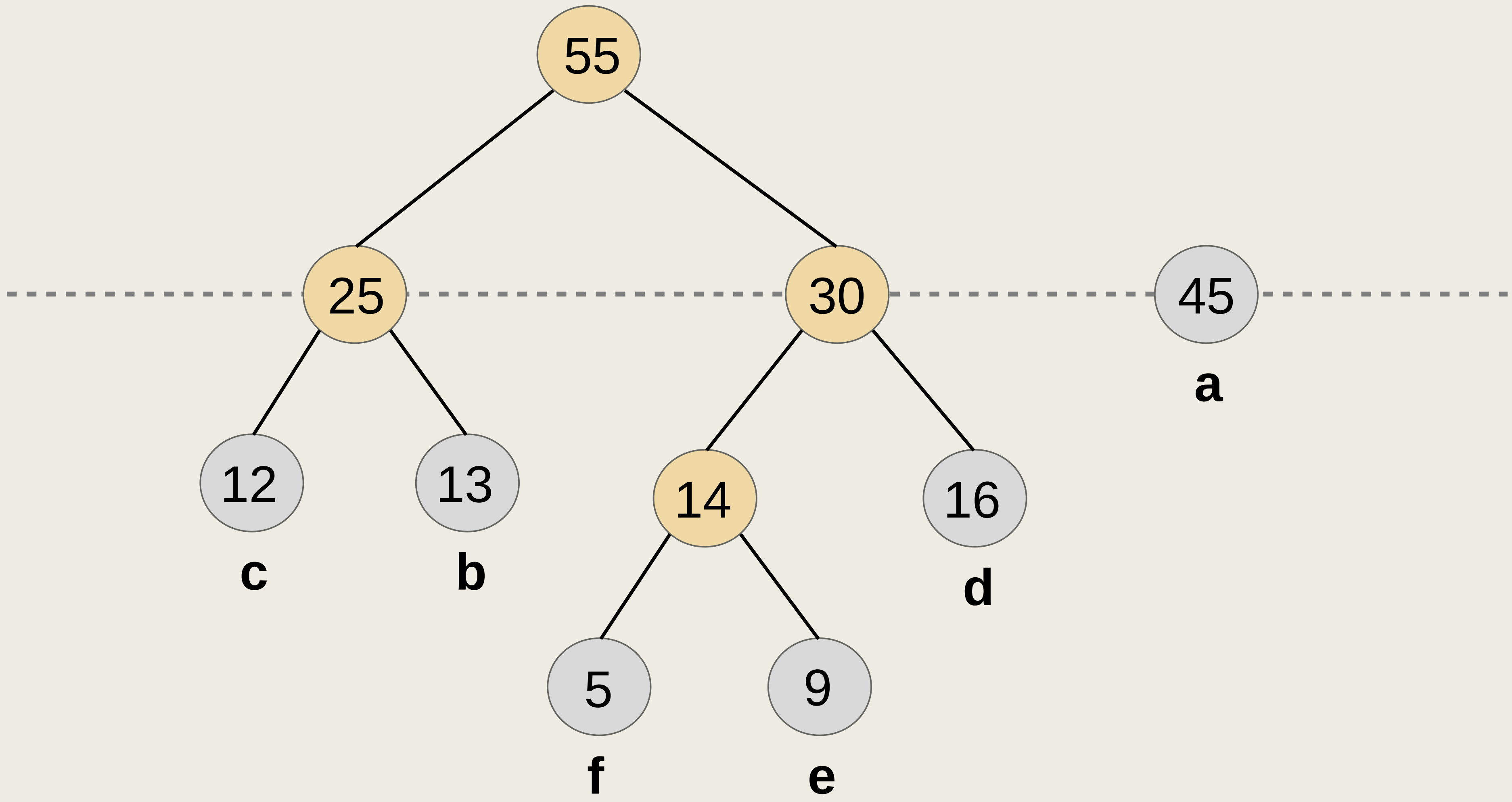
Huffman Codes



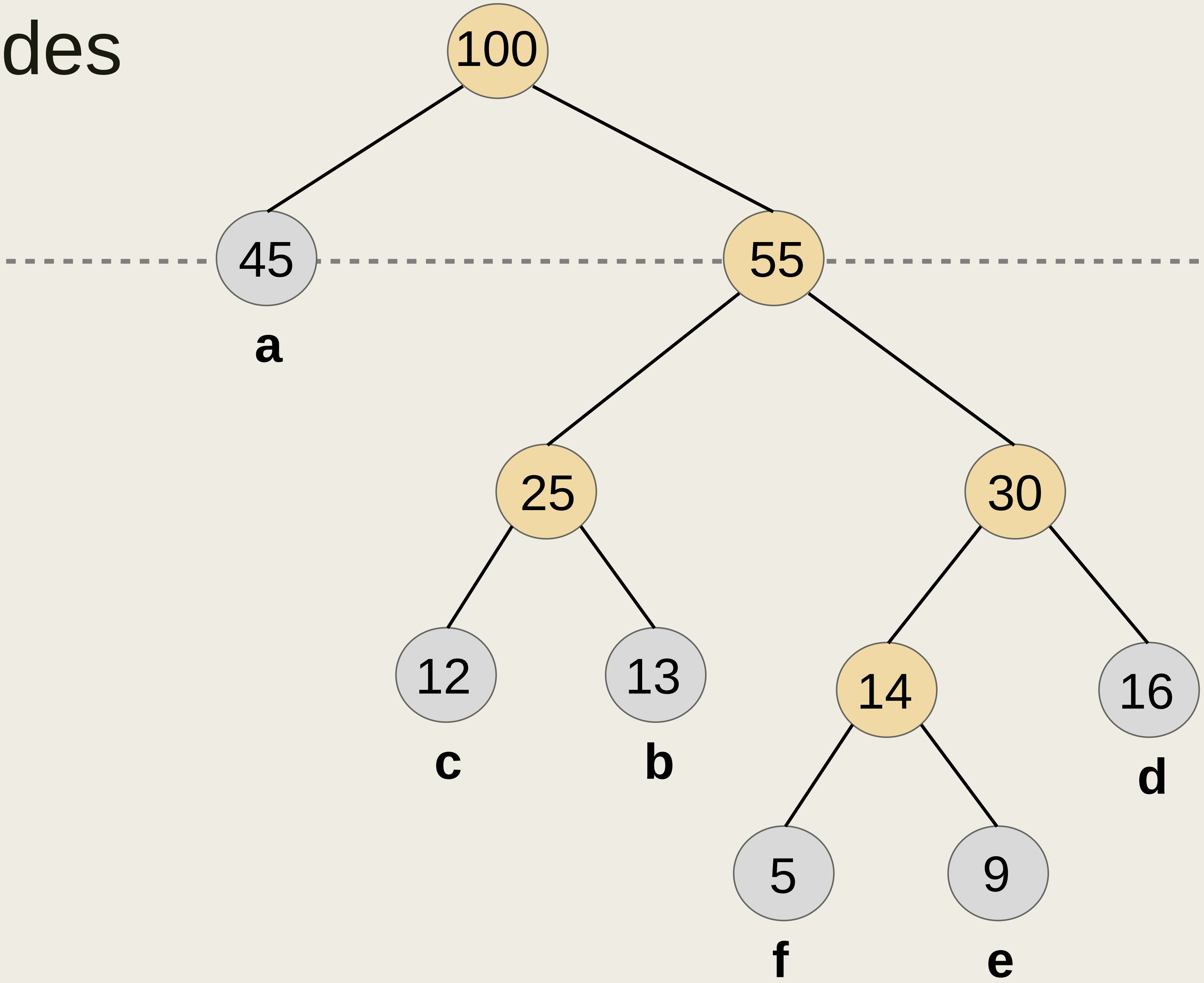
Huffman Codes



Huffman Codes

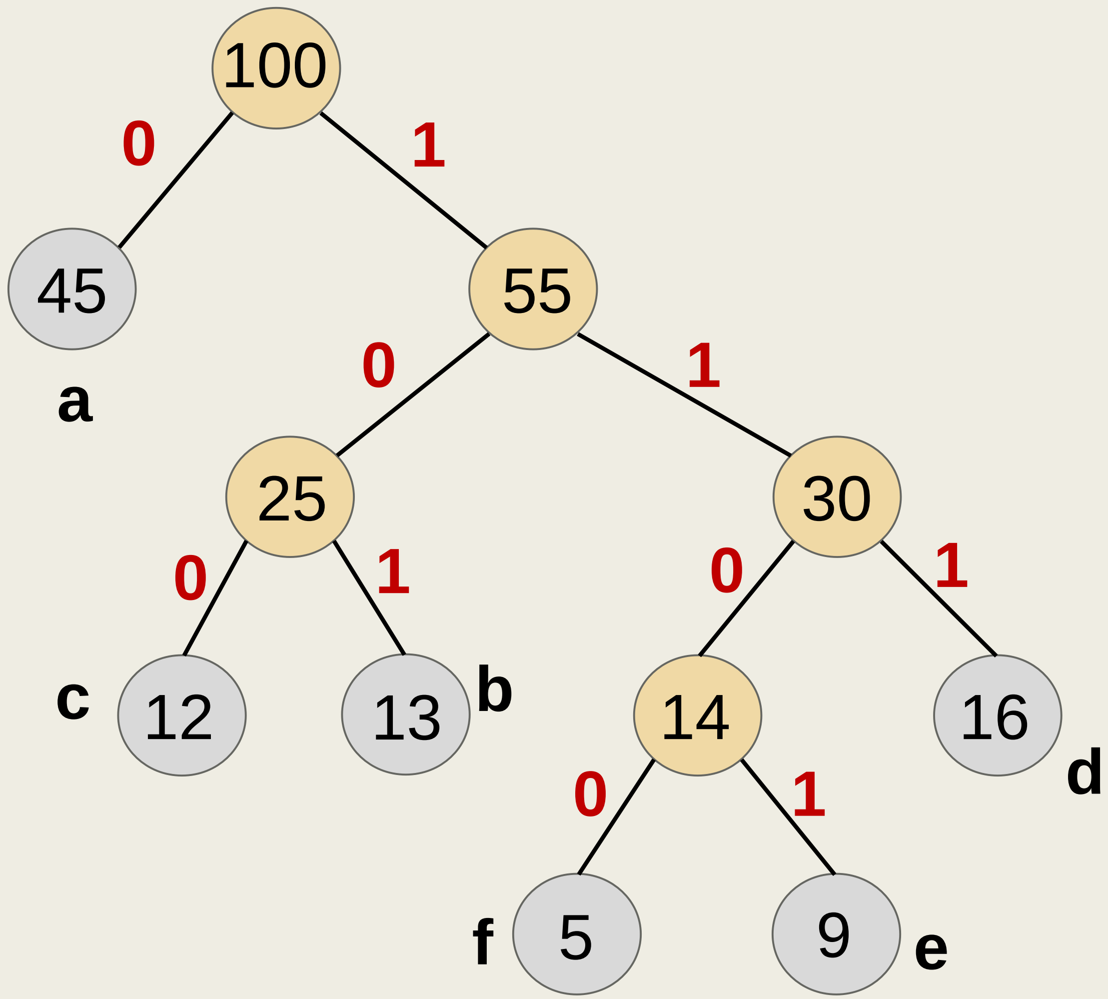


Huffman Codes



Huffman Codes

- The resulting codes
  - a : 0
  - b : 101
  - c : 100
  - d : 111
  - e : 1101
  - f : 1100



- Note that, the labeling of 0,1 on the edges is not important and can be arbitrary.

Matroids and Greedy Algorithms

Matroid

- **Matroid** is a combinatorial structure that abstracts and generalizes *the notion of* ***linear independence*** in vector spaces.
  - There are many equivalent ways to define a matroid.
    - Independence of elements – Independent sets
    - Bases – Maximal independent sets
    - Circuits – Minimal dependent sets
  - On graphs with ***acyclic being independent***, the above concepts correspond to *Forests*, *Spanning Trees*, and *Cycles*, respectively.

Definition 1

Imagine that
E is the set of vectors in a (finite) vector space, and
I is the collection of all independent vector subsets.

- A (finite) matroid M is a pair (E, I), where E is the (finite) ground set of elements, $I \subseteq 2^E$ is a collection of subsets of E such that

1. $\emptyset \in I$, i.e., the empty set is independent.
2. If $A \in I$ and $B \subseteq A$, then $B \in I$,
   i.e., subsets of independent sets are also independent.
3. If $A, B \in I$ and $|A| > |B|$,
   then there exists $x \in A - B$ such that $B \cup \{x\} \in I$,
   i.e., we can ***augment*** elements to form larger independent sets.

Example

- Let $G = (V, E)$ be an undirected graph.
  - Let $\mathcal{A}$ be the collection of all edge subsets that will induce an acyclic subgraph of G, i.e.,

$$\mathcal{A} := \{ K \subseteq E : \text{the graph } H = (V, K) \text{ is acyclic} \}.$$

  - The pair $M_1 = (E, \mathcal{A})$ satisfies all the conditions in Definition 1.
    - $M_1 = (E, \mathcal{A})$ is a matroid.

Example

An edge subset $M \subseteq E$ is a ***matching***
if none of the edges in M share a common endpoint.

- Let $G = (U, V, E)$ be a bipartite graph with partite sets U and V.
  - For any matching $M \subseteq E$,
    define $U(M)$ to be the set of endpoints of M in U.
  - Let $\mathcal{U}$ be the collection of $U(M)$ for all possible matchings of G,
    i.e., $$\mathcal{U} := \{ U(K) : K \subseteq E \text{ is a matching in } G \}.$$
  - It can be verified that the pair $M_2 = (E, \mathcal{U})$ satisfies all the conditions in Definition 1.
    - $M_2 = (E, \mathcal{U})$ is a matroid.

Definition 2

Imagine that

E is the set of vectors in a (finite) vector space, and

$\mathcal{B}$ is the collection of ***all bases*** of the space.

- A (finite) matroid M is a pair $(E, \mathcal{B})$, where E is the (finite) ground set of elements, $\mathcal{B} \subseteq 2^E$ is a collection of subsets of E such that

1. $\mathcal{B} \neq \emptyset$.

2. If $A, B \in \mathcal{B}$, $A \neq B$, and $a \in A - B$,
   then there exists $b \in B - A$ such that $(A - \{a\}) \cup \{b\} \in \mathcal{B}$,
   i.e., we can ***exchange elements*** from two distinct bases
   to form a new base.

Example

- Let $G = (V, E)$ be an undirected graph.
  - Let $\mathcal{T}$ be the collection of all edge subsets that will induce a spanning tree (maximal acyclic subgraph) of G, i.e.,

$$\mathcal{T} := \{ K \subseteq E : \text{the graph } H = (V, K) \text{ is a spanning tree of } G \}.$$

  - The pair $M_3 = (E, \mathcal{T})$ satisfies all the conditions in Definition 2.
    - $M_3 = (E, \mathcal{T})$ is a matroid.

Let’s prove this.

Theorem 1. (Exchange Property of Spanning Trees)

Let $G = (V, E)$ be an undirected graph and $T_1, T_2 \subseteq E$ be two spanning trees of G with $T_1 \neq T_2$.

For any $e_1 \in T_1 - T_2$, there exists $e_2 \in T_2 - T_1$ such that $(T_1 - \{e_1\}) \cup \{e_2\}$ is a spanning tree of G.

- For T_1, removing $e_1 = (u, v)$ creates two components C_1, C_2, where $u \in C_1$, $v \in C_2$.
- For T_2, adding e_1 creates a unique cycle C that contains u and v.
  - Hence, traversing the edges of C crosses C_1 and C_2 at least twice.
  - Other than e_1, some edge in C has to connect C_1 and C_2. Pick one of such edges to be e_2.

Definition 3

Imagine that E is the set of vectors in a (finite) vector space, and $\mathcal{C}$ is the collection of ***minimal dependent vector sets*** of the space.

- A (finite) matroid M is a pair $(E, \mathcal{C})$, where E is the (finite) ground set of elements, $\mathcal{C} \subseteq 2^E$ is a collection of subsets of E such that

  1. If $A, B \in \mathcal{C}$ with $A \subseteq B$, then $A = B$,
     i.e., each circuit in $\mathcal{C}$ is minimal in size.

  2. If $A, B \in \mathcal{C}$, $A \neq B$, and $e \in A \cap B$,
     then there exists $C \in \mathcal{C}$ such that $C \subseteq (A \cup B) - \{e\}$,
     i.e., $(A \cup B) - \{e\}$ contains another circuit in $\mathcal{C}$.

Example

- Let $G = (V, E)$ be an undirected graph.
  - Let $\mathcal{C}$ be the collection of all simple cycles of G, i.e.,

$$\mathcal{C} \coloneqq \{\ K \subseteq E\ :\ K \text{ forms a simple cycle in } G\ \}.$$

  - The pair $M_4 = (E, \mathcal{C})$ satisfies all the conditions in Definition 3.
    - $M_4 = (E, \mathcal{C})$ is a matroid.

Matroid

- The structure of a matroid is characterized completely by its ***independent sets***, its ***bases***, or its ***circuits***.
  - It can be shown that the three definitions lead to one another.

- Why matroids?
  - It provides an abstraction of a wide category of problems.
  - Properties or algorithms for matroids automatically apply to all of these problems.

Rank of a Matroid

- The structure of a matroid is characterized completely by its ***independent sets***, its ***bases***, or its ***circuits***.

Lemma 2. (Size of Maximal Independent Sets)

Let $M = (E, I)$ be a matroid and $B_1, B_2 \in I$ be two distinct bases for M. Then we have $|B_1| = |B_2|$.

- This lemma holds directly from the 3^{rd} condition in the definition.
- We define the size of a base to be the rank of the matroid.

Greedy Algorithms

for Weighted Matroids

Weighted Matroid

- Let $M = (E, I)$ be a matroid with a weight function $w : E \mapsto Q^{>0}$ that assigns each element $e \in E$ a positive weight.
- The following algorithm computes a maximum-weight base B for M.

- Weighted-Matroid($M = (E, I),\ w$) $-$ $E = \{1,2, \dots n\}$ the set of elements.

A. Relabel the elements such that $w_1 \geq w_2 \geq \cdots \geq w_n$.
B. Let $B \leftarrow \emptyset$.
C. For $i \leftarrow 1$ to n, do the following.
   - Add i to B if $B \cup \{i\} \in I$.
D. Return B.

Theorem 3. (Maximum-Weight Base for Weighted Matroid)

The algorithm Weighted-Matroid computes a maximum-weight subset in I ***if and only if*** $M = (E, I)$ is a matroid.

- Weighted-Matroid($M = (E, I),\ w$) $-$ $E = \{1,2, \dots n\}$ the set of elements.

A. Relabel the elements such that $w_1 \geq w_2 \geq \cdots \geq w_n$.

B. Let $B \leftarrow \emptyset$.

C. For $i \leftarrow 1$ to n, do the following.

   - Add i to B if $B \cup \{i\} \in I$.

D. Return B.

Theorem 3. (Maximum-Weight Base for Weighted Matroid)

The algorithm Weighted-Matroid computes a maximum-weight subset in I ***if and only if*** $M = (E, I)$ is a matroid.

Proof.

- Suppose that $M = (E, I)$ is a matroid.
  - Let $B = \{\pi_1 \leq \pi_2 \leq \cdots \leq \pi_k\}$ be the set returned by the algorithm.
  - First, we prove that

There exists an optimal subset O^* that contains the element π_1.

Claim. (Greedy-choice Property)

Let π be the element that is added by the algorithm to B first.

Then there exists an optimal subset $O^* \in I$ such that $\pi \in O^*$.

- Let $O \in I$ be a ***maximum-weight base*** for M.
  - If $\pi \in O$, then we are done.

The largest element in O has weight at most $w(\pi)$.

- Suppose that $\pi \notin O$.
  - Since the algorithm added π first, $\{ j \} \notin I$ for all $j < \pi$. This implies that any superset of $\{ j \}$ is not independent.
  - Hence $w(\pi) \geq w(\pi')$ for all $\pi' \in O$.

- Let $O \in I$ be a ***maximum-weight base*** for M.
  - If $\pi \in O$, then we are done.
- Suppose that $\pi \notin O$.

  Then $w(\pi) \geq w(\pi')$ for all $\pi' \in O$.

$\{\pi\} \in I$ by the hereditary property.

- Repeatedly apply the augment property for matroids in Definition 1 on O and $\{ \pi \}$, we obtain an independent set O' such that

$$O' = (O - \{\pi'\}) \cup \{\pi\}$$

for some $\pi' \in O$.

- Then $w(O') = w(O) - w(\pi') + w(\pi) \geq w(O)$, and O' is an optimal independent set containing π.

Proof. (continue)

- Suppose that $M = (E, I)$ is a matroid.
  - Let $B = \{\pi_1 \leq \pi_2 \leq \cdots \leq \pi_k\}$ be the set returned by the algorithm.
  - Then, there exists an optimal subset $O^* \in I$ with $\pi_1 \in O^*$.

  (*Optimal Substructure of Matroids*.)

  - Consider the collection of subsets $I' \coloneqq \{ A - \{\pi_1\} : \pi_1 \in A \in I \}$.
    - Then $M' = (E, I')$ forms a matroid (submatroid from M).
    - $O^* - \{\pi_1\} \in I'$.
    - The same argument applies on $B' \coloneqq B - \{\pi_1\}$ and M'.
- Hence, B is an optimal base.

Theorem 3. (Maximum-Weight Base for Weighted Matroid)

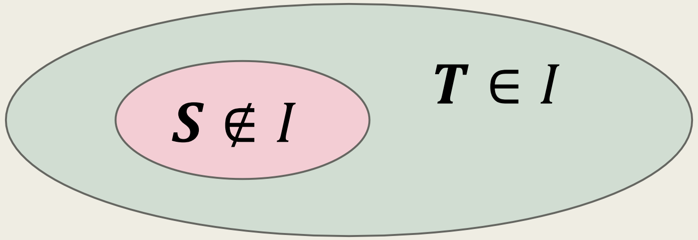
The algorithm Weighted-Matroid computes a maximum-weight subset in I ***if and only if*** $M = (E, I)$ is a matroid.

Proof.

- Suppose that $M = (E, I)$ does not satisfy the matroid property.
  - We show that, *for some weight functions*, the greedy algorithm fails to compute a maximum-weight set from the set family I.

- Suppose that $M = (E, I)$ does not satisfy the matroid property.
  - If the hereditary property is not satisfied, then there exists $S, T \subseteq E$ with $S \subset T$ such that $T \in I$, $S \notin I$.
  - For any $e \in E$, define the weight

$$w_e := \begin{cases} 2, & \text{if } e \in S, \\ 1, & \text{if } e \in T - S, \\ 0, & \text{otherwise.} \end{cases}$$



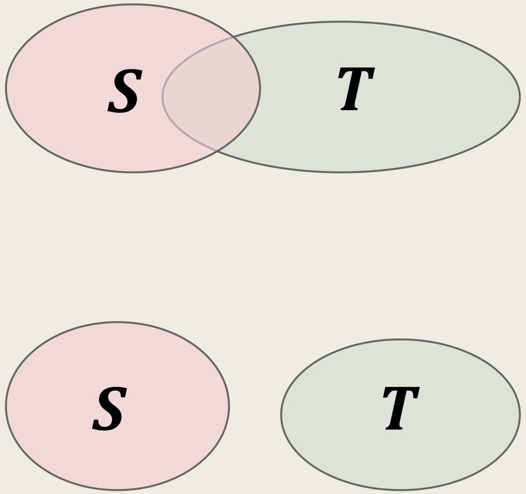
  - The optimal set is T.
  - The greedy algorithm first considers the elements in S and will skip some of the elements in S since $S \notin I$.

- If the augmentation (extension) property is not satisfied, then there exists $S, T \subseteq E$ with $|S| < |T|$ such that

$$S \cup \{e\} \notin I \quad \text{for all } \; e \in T - S \, .$$

- For any $e \in E$, define the weight

$$w_e \; := \begin{cases} 1 + \dfrac{1}{2|S|}, & \text{if } e \in S, \\ 1, & \text{if } e \in T - S, \\ 0, & \text{otherwise.} \end{cases}$$



- $w(T) \; \geq \; |T| \; \geq \; |S| + 1.$
- The algorithm cannot augment any $e \in T - S$.

  Hence $w(B) \leq w(S) = |S| + 1/2 < w(T)$.

Example 3.

Scheduling Unit-sized Jobs

with Deadlines and Penalties

The Scenario

- We have a set of n unit-sized jobs $J = \{a_1, a_2, \dots, a_n\}$, where each job a_i has a deadline d_i and a penalty p_i (to be paid) if a_i fails to finish its execution in time.

- We want to schedule the jobs on one machine so as to minimize the total penalties due to deadline misses.
  - Define I to be the collection of all subsets of J that can be scheduled on the machine.
  - Then $M = (J, I)$ is a matroid.

Apply the greedy algorithm and we are done.

Example 3.

Minimum Spanning Tree

Minimum / Maximum Spanning Tree

- Let $\mathcal{T}$ be the collection of all edge subsets that will induce a spanning tree (maximal acyclic subgraph) of G, i.e.,

$$\mathcal{T} := \{ K \subseteq E : \text{the graph } H = (V, K) \text{ is a spanning tree of } G \}.$$

- The pair $M_3 = (E, \mathcal{T})$ forms a matroid.
  - Hence, we an apply the greedy algorithm to compute a spanning tree with minimum / maximum weights.

This is also known as the Kruskal's algorithm for minimum spanning trees.

Disjoint-set Data Structure &

Implementation of Kruskal’s Algorithm

Disjoint Set

- Suppose that we want to maintain a ***partition*** (as disjoint sets) for a given set of elements, so as to support the following operations.
  - **Make-set(x)** – to create a set of a new element x.
  - **Union(x, y)** – to union the set containing x and that containing y.
  - **Find-Set(x)** – to return a representative for the set containing x.

Disjoint Set

- We introduce a data structure that supports a sequence of m operations in $O\big(m \cdot \alpha(n)\big)$ time, where
  - n is the number of elements (calls to the Make-set operation), and
  - $\alpha(n)$ is the inverse Ackerman's function,
    which is an *extraordinarily slow growing* function.
    - $\alpha(n) \leq 4$, for any number that can be written-down
      in the physical universe.

Disjoint Set

- The idea is to use a rooted tree for each disjoint set.
- In each node, we store the following information.
  - x - The element stored in the node.
  - p - The pointer to its parent node.
  - r - The rank of the node, which is *the maximum height ever attained* for the subtree rooted at that node.
- We will use "*union-by-rank*" and "*path-compression*" techniques to achieve the claimed complexity.

The Procedures

- Make-Set(x) – x is the new element to be considered.

A. Set $p[x] \leftarrow x$ and $r[x] \leftarrow 0$.

- Find-Set(x) – Return the representative of the set containing x.

A. If $x \neq p[x]$, then $p[x] \leftarrow$Find-Set($p[x]$).

B. Return $p[x]$.

- Union(x, y) – Union the sets containing x and y.

A. Link(Find-Set(x), Find-Set(y).

- Link(x, y) – Link the two subtrees by rank.

A. If $r[x] > r[y]$, then
  - Set $p[y] \leftarrow x$.

B. Else,
  - Set $p[x]$ ←y.
  - Increase $r[y]$ by 1 if $r[x] = r[y]$.

The Kruskal's Algorithm for MST

- Kruskal-MST(G, w) – graph $G = (V, E)$ with edge-weight function w.

A. $A \leftarrow \emptyset$.

B. Relabel the edges so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$.

C. For $i = 1,2, \ldots, m$, do the following

  - Let $e_i = (u, v)$.
  - If Find-Set(u) $\neq$ Find-Set(v),
    - Add e_i to A and call Union(u, v).

D. Return A.