Introduction to Algorithms

Topic 6-2 : Greedy Algorithm

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2023

Outline

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, using dynamic programming to determine the best choices is **overkill**; simpler, more efficient algorithms will do.
- A **greedy algorithm** always makes the choice that **looks the best at the current moment**. That is, it makes a *locally* optimal choice in the hope that this choice will lead to a global optimal solution.
- This chapter explores optimization problems for which greedy algorithms provide optimal solutions.

Contents

Problem Description

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed **activities** that wish to use a resource, such as a lecture hall, which can **serve only one activity at a time**.

Each activity a_i has a **start time** s_i and a **finish time** f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.

Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible iff $s_i \geq f_j$ or $s_j \geq f_i$.

In the activity-selection problem, we wish to select a **maximum-size subset of mutually compatible activities**. We assume that the activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Example

Consider the following set S of activities:

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

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a **largest** subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

Dynamic Programming Method

Define the set $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$, it is the subset of activities in S that can **start after activity a_i finishes and finish before activity a_j starts**.

Add fictitious activities a_0 and a_{n+1}, and adopt the conventions that $f_0 = 0$ and $s_{n+1} = \infty$, thus $0 \leq i, j \leq n+1$.

Therefore the subproblem is to select a maximum-size subset of mutually compatible activities from S_{ij}, for $0 \leq i < j \leq n+1$.

Dynamic Programming Method

The **optimal substructure** of this problem is as follows: Suppose that an optimal solution A_{ij} to S_{ij} includes activity a_k. Then the solutions A_{ik} to S_{ik} and A_{kj} to S_{kj} used within this optimal solution to S_{ij} must be optimal as well.

Thus, we can transform the problem of building an maximumsize subset of mutually compatible activities in S_{IJ} into finding maximum-size subsets A_{ik} and A_{kj} of mutually compatible activities in S_{ik} and S_{kj}. Then forming the solution A_{ij} as

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}.$$

Dynamic Programming Method

Let $c[i,j]$ be the number of activities in a maximum-size subset of mutually compatible activities in S_{ij} (have $c[i,j] = 0$ for $i \geq j$)

For a nonempty subset S_{ij}, if a_k is used in an optimal solution, we have the recurrence (assumed k is given)

$$c[i,j] = c[i,k] + c[k,j] + 1$$

But we dont know the value of k and need to look up it in the range from $i+1$ to $j-1$ to find the best. Thus

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing, \\ \max\limits_{i<k<j} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \varnothing. \end{cases}$$

Making the Greedy Choice

Based on the dynamic-programming process, we should write a tabular, bottom-up algorithm based on the recurrence above.

Theorem 16.1

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time, then:

- Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij};
- The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as only one subproblem that may be nonempty.

Proof

Prove the **second** part first. If S_{im} is nonempty, there is some activity such that $f_i \leq s_k < f_k \leq s_m < f_m$. Then a_k is also in S_{ij} and its finish time is earlier than a_m, which **contradicts** our choice of a_m. So S_{im} is empty;

Then prove the **first** part. Suppose that A_{ij} is a maximum-size subset of mutually compatible activities of S_{ij}. The activities in A_{ij} are ordered in monotonically increasing order of finish time and let a_k be the first activity.

- If $a_k = a_m$, we are done.
- If $a_k \neq a_m$, construct the subset $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$. The activities in A'_{ij} are disjoint and it has the same number of activities as A_{ij}. So A'_{ij} is a maximum-size subset of mutually compatible activities of S_{ij} that includes a_m.

Making the Greedy Choice

Theorem 16.1 shows some benefits of the solution:

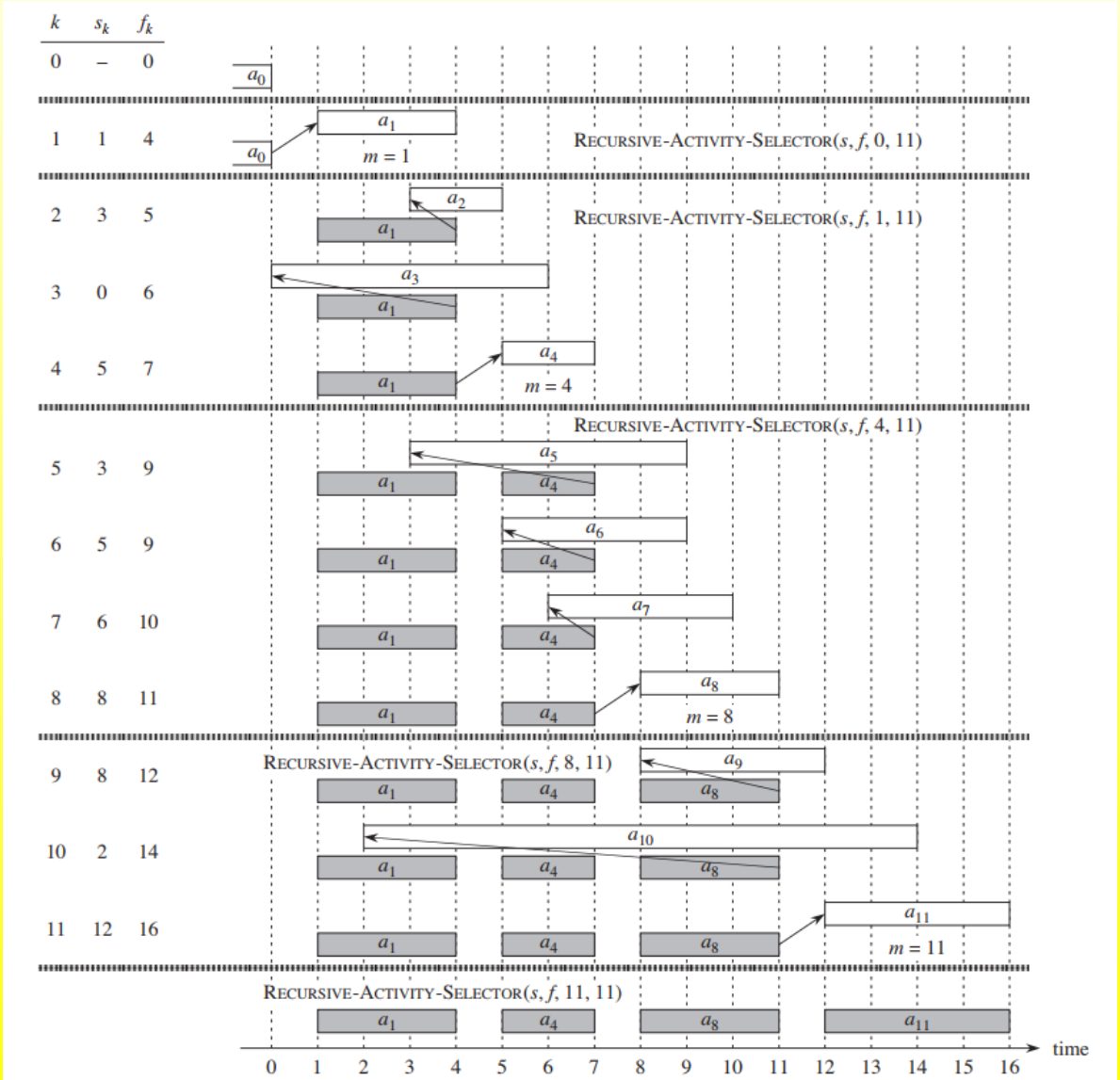
- It reduces the number of subproblems and the number of choices compared with dynamic-programming solution;
- It can solve each subproblem in a **top-down fashion**, rather than the bottom-up manner typically used in dynamic programming;
- Each subproblem consists of the last activities to finish, and the number of such activities are different each other;
- The finish times of the activities chosen over all subproblems are strictly increasing over time;
- The activity a_m is a greedy choice which maximizes the amount of unscheduled time remaining.

RECURSIVE-ACTIVITY-SELECTOR

RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)

1: $m = k+1$
2: **while** $m \leq n$ and $s[m] < f[k]$ **do**
3: $\quad m = m+1$ // find the earliest finish compatatible job
4: **if** $m \leq n$ **then**
5: $\quad$ **return** $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
6: **else**
7: $\quad$ **return** $\varnothing$

The initial call is RECURSIVE-ACTIVITY-SELECTOR$(s,f,0,n+1)$


k
s_k
f_k
0 – 0
1 1 4
2 3 5
3 0 6
4 5 7
5 3 9
6 5 9
7 6 10
8 8 11
9 8 12
10 2 14
11 12 16
a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
a_9
a_{10}
a_{11}
$m = 1$
$m = 4$
$m = 8$
$m = 11$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Time Complexity

Over all recursive calls, each activity is examined **exactly once** in the while loop test of line 2.

Thus the running time of the call RECURSIVE-ACTIVITY-SELECTOR$(s,f,0,n+1)$ is $\Theta(n)$ (assuming that the activities have already been sorted by finish times)

An Iterative Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s,f)

1: $n = s.length$
2: $A = \{a_1\}$
3: $k = 1$
4: **for** $m = 2$ to n **do**
5: **if** $s[m] \geq f[k]$ **then**
6: $A = A \cup \{a_m\}$
7: $k = m$
8: **return** A

An Iterative Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s,f)

1: $n = s.length$
2: $A = \{a_1\}$
3: $k = 1$
4: **for** $m = 2$ to n **do**
5: **if** $s[m] \geq f[k]$ **then**
6: $A = A \cup \{a_m\}$
7: $k = m$
8: **return** A

By following the steps of GREEDY-ACTIVITY-SELECTOR, we know that its result set A is the **same** with that of RECURSIVE-ACTIVITY-SELECTOR.

Contents

Elements of the Greedy Strategy

For a greedy algorithm, each choice **seems the best** at the moment, but this heuristic strategy does not always produce an optimal solution. To develop a greedy algorithm, we went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
4. Show that all but one of the subproblems induced by having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Elements of the Greedy Strategy

More generally, we design greedy algorithms according to the following steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Elements of the Greedy Strategy

- If we can demonstrate that the problem has two key ingredients: the greedy-choice property and optimal sub-structure, a greedy algorithm can be developed for it.
- **Greedy-choice property**: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice
- **Optimal sub-structure**: A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems
- Now we discuss the two ingredients in detail respectively

Greedy-choice Property

- Greedy algorithms differ from dynamic programming in some points:
  - In dynamic programming, every choice at each step usually depends on the solutions to subproblems; but the choice made by a greedy algorithm only depends on choices so far.
  - Dynamic-programming problems are solved in a bottom-up manner; but a greedy strategy usually progresses in a top-down fashion.
- We must prove that a greedy choice at each step yields a globally optimal solution.
- The greedy-choice property often gains us some efficiency in making our choice in a subproblem.

Optimal Substructure

- This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.
- Assumed that we arrived at a subproblem by having made the greedy choice in the original problem.
- Then we need to argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem.

Greedy versus Dynamic Programming

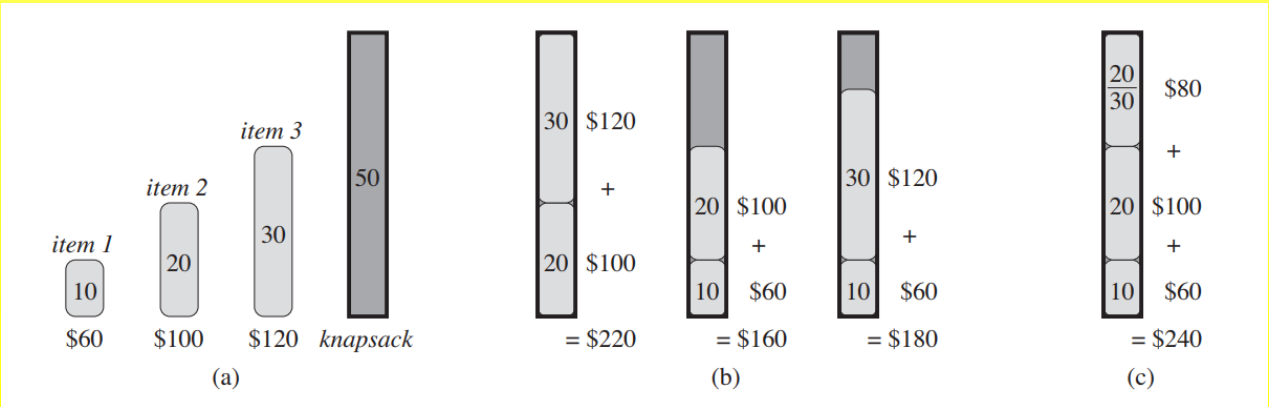
- To avoid confusion of the greedy and dynamic-programming strategies when deciding which one to choose for situations, we should illustrate the subtleties between the two techniques.
- First, investigate two variants of a classical optimization problem: $0-1$ knapsack problem and fractional knapsack problem.
- $0-1$ **knapsack problem**: A thief robbing a store finds n items; the ith item is worth v_i dollars and weighs w_i pounds (v_i, w_i: integer); he can carry at most W (W: integer) pounds. Which items should he take in order to take as valuable a load as possible?
- **fractional knapsack problem**: The setup is the same, but the thief can take fractions of items, rather than having to make a binary $(0-1)$ choice for each item.

Greedy versus Dynamic Programming

- To solve the fractional problem, we first compute the **value per pound** v_i/w_i for each item;
- Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound;
- Then he takes the next greatest valuable item, and so forth until he fills the knapsack;
- Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time.
- The fractional knapsack problem has the **greedy-choice property**.

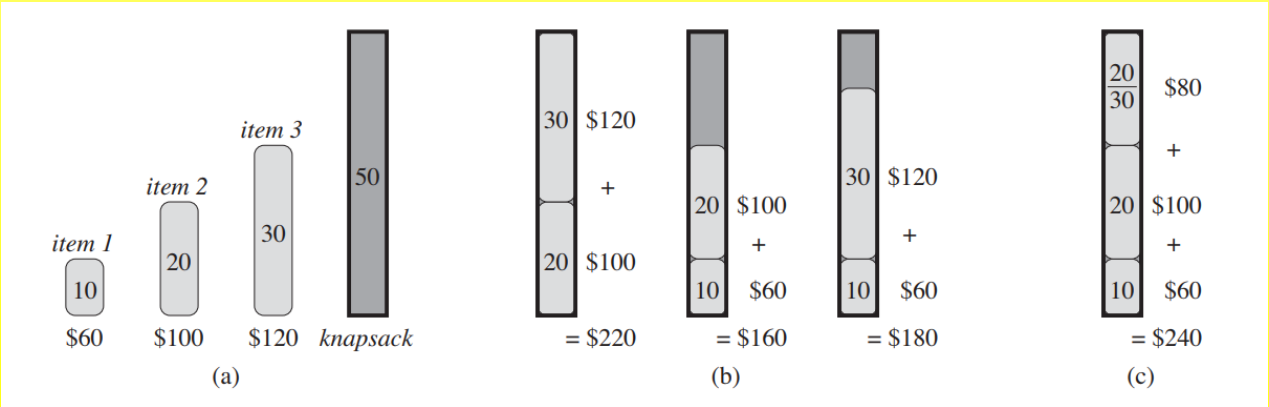
Greedy versus Dynamic Programming

- But this greedy strategy **does not work** for the $0-1$ knapsack problem. To see the reason, consider the problem instance illustrated in Figure 16.2(a).
- The value per pound of item 1 is 6 dollars per pound, which is greater than that of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound).
- However, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.



Greedy versus Dynamic Programming

- The reason is that taking item 1 the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load.
- But for the comparable fractional problem, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 16.2(c).



Greedy versus Dynamic Programming

- In the $0-1$ problem, when we consider an item for inclusion in the knapsack, we must compare the two solutions to the subproblems in which the item is included and excluded before we can make the choice.
- The problem formulated in this way gives rise to many overlapping subproblems.
- Indeed, the $0-1$ problem can be solved by dynamic programming. How?

Contents

Huffman Codes

- Huffman codes are a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- Consider the data to be a sequence of characters.
- Huffman's greedy algorithm **uses a table of the occurrence frequencies of the characters** to build up an optimal way of representing each character as a binary string.

Example

- We wish to store a $100,000$-character data file compactly.
- Only six different characters appear, and the character "a" occurs $45,000$ times.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Binary Character Code

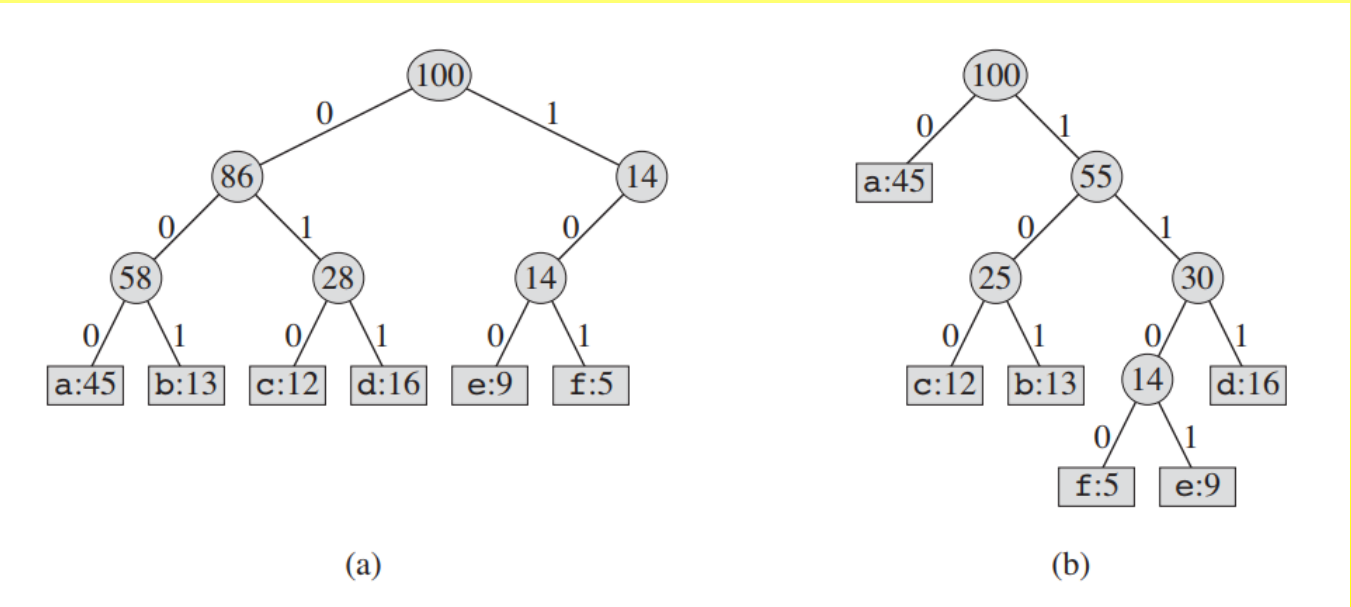
- There are many ways to represent such a file of information.
- We design a **binary character code** wherein each character is represented by a unique binary string.
- There are two coding ways:
  - **Fixed-length code**: we need 3 bits to represent six characters: a $= 000$, b $= 001$, $\ldots$, f $= 101$. This method requires $300,000$ bits to code the entire file
  - **Variable-length code**: it gives frequent characters short codewords and infrequent characters long codewords. This code requires
    $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$ bits
- So a variable-length code can do considerably better than a fixed-length code.

Prefix Codes

- The codes in which no codeword is also a prefix of some other codeword are called **prefix codes**.
- There is no loss of generality in restricting attention to prefix codes.
- We concatenate the codewords representing each character of the file for the purpose of encoding. For example, with the variable-length prefix code of Figure 16.3, "abc" is coded as $0 \cdot 101 \cdot 100 = 0101100$.
- Prefix codes simplify decoding and the codeword that begins an encoded file is unambiguous. For example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to "aabe".

Binary Tree for Prefix Code

- A binary tree whose leaves are the given characters provides a convenient representation for the prefix code
- The binary codeword for a character is the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child".



The Cost of Prefix Code

- If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes;
- Given a tree T corresponding to a prefix code, we want to compute the number of bits required to encode a file;
- Let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree (i.e. the length of the codeword for character c, wherein c is one character in the alphabet C)
- Define the cost of the tree T as $B(T) = \sum_{c \in C} f(c) d_T(c)$, which is the number of bits required to encode a file.

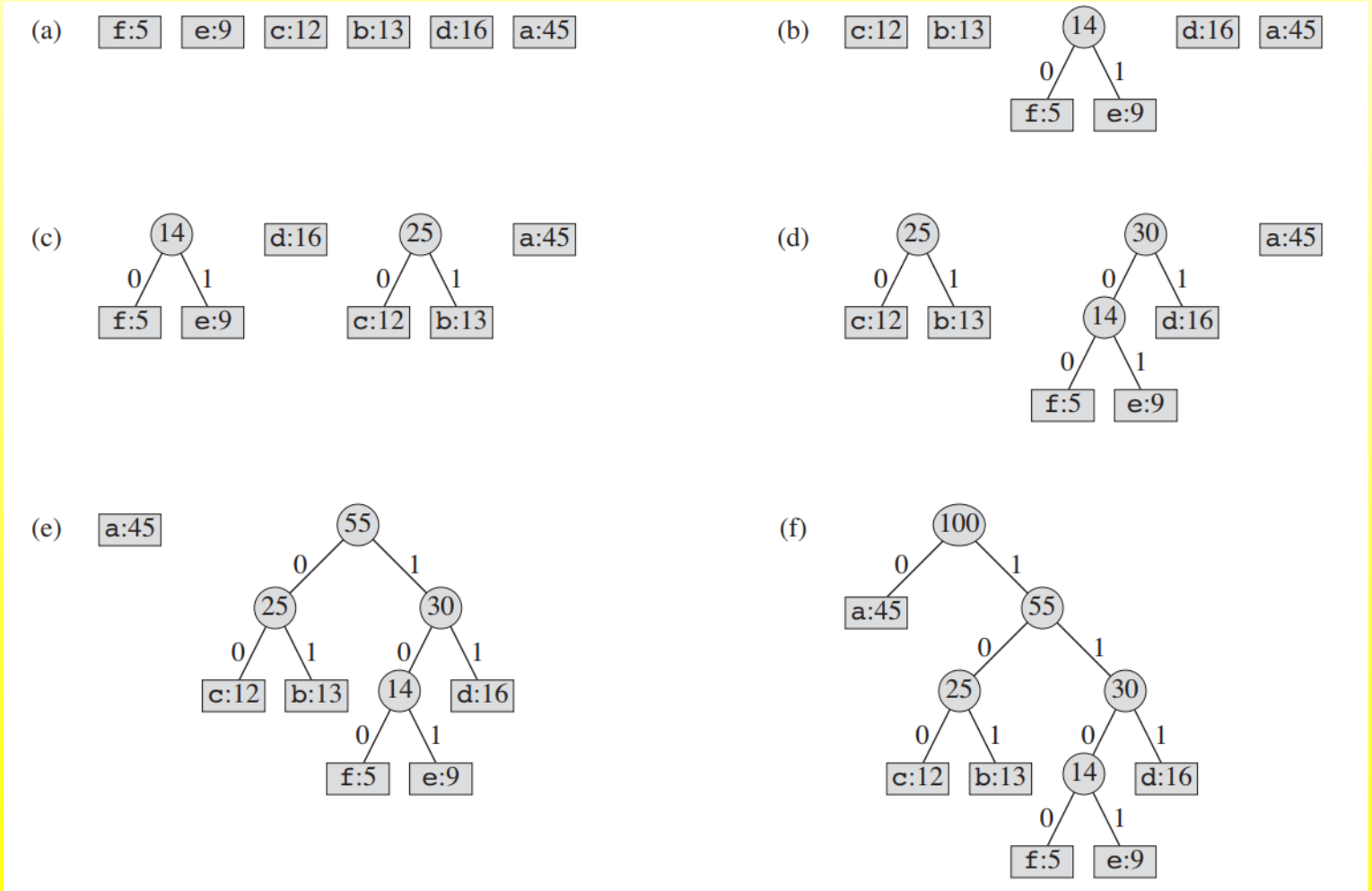
Constructing a Huffman Code

- Huffman invented a greedy algorithm that constructs Huffman code, which is an optimal prefix code;
- To clarify how the algorithm makes greedy choices, we present the pseudocode first;
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.

HUFFMAN

HUFFMAN(C)
1: $n = |C|$
2: $Q = C$
3: **for** $i = 1$ to $n-1$ **do**
4: allocate a new node z
5: $z.left = x =$ EXTRACT-MIN(Q)
6: $z.right = y =$ EXTRACT-MIN(Q)
7: $z.freq = x.freq + y.freq$
8: INSERT(Q, z)
9: **return** EXTRACT-MIN(Q)

Example for Huffman Codes



Correctness of Huffman's Algorithm

First show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

Lemma 16.2: Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Correctness of Huffman's Algorithm

Lemma 16.3: Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x,y\} \cup \{z\}$. Define f for C' as for C, except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Theorem 16.4: Procedure HUFFMAN produces an optimal prefix code.

Contents

Matroid

A matroid is an ordered pair $M = (S, \mathscr{I})$ satisfying the following conditions.

1. S is a finite set.
2. $\mathscr{I}$ is a nonempty family of subsets of S, called the **independent** subsets of S, such that if $B \in \mathscr{I}$ and $A \subseteq B$, then $A \in \mathscr{I}$. We say that I is hereditary if it satisfies this property. Note that the empty set $\varnothing$ is necessarily a member of $\mathscr{I}$.
3. If $A \in \mathscr{I}$, $B \in \mathscr{I}$, and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in \mathscr{I}$. We say that M satisfies the **exchange property**.

Example of Matroids

Graphic matroid: Given an undirected graph $G = (V, E)$, the graphic matroid $M_G = (S_G, \mathscr{I}_G)$ satisfies the following conditions:

- The set S_G is defined to be E, the set of edges of G.
- If A is a subset of E, then $A \in \mathscr{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a **forest**.

Theorem 16.5: If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, \mathscr{I}_G)$ is a matroid.

Maximal Independent Subset

- Given a matroid $M = (S, \mathscr{I})$, for any $x \notin A$, x is an **extension** of A if $A \cup \{x\} \in \mathscr{I}$;
- For example, if A is an independent set of edges in a graphic matroid M_G, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle;
- If A is an independent subset in a matroid M, we say that A is maximal if it has no extensions.

Theorem 16.6: All maximal independent subsets in a matroid have the same size.

Spanning Tree

- Spanning tree: Consider a graphic matroid M_G for a connected, undirected graph G, the spanning tree of G is a free tree with exact $|V| - 1$ edges that connects all the vertices of G
- A matroid $M = (S, \mathscr{I})$ is weighted if there is an associated weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$
- The weight function w of subset A is $w(A) = \sum_{x \in A} w(x)$ for any $A \subseteq S$.

Greedy Algorithms on a Weighted Matroid

- An optimal subset of the matroid is a subset that is independent and has the maximum possible weight;
- It is always a maximal independent subset because the weight $w(x)$ of any element $x \in S$ is positive;
- **Minimum-spanning-tree problem**: we are given a connected undirected graph $G = (V, E)$ and a length function w such that $w(e)$ is the (positive) length of edge e. We wish to find a subset of the edges that **connect all of the vertices together and have the minimum total length**.
- To view this as a problem of finding an optimal subset of a matroid, consider the weighted matroid M_G with weight function w, where $w(e) = w_0 - w(e)$ and w_0 is larger than the maximum length of any edge.

Greedy Algorithms on a Weighted Matroid

- For any maximal independent subset A,

$$w'(A) = (|V| - 1)w_0 - w(A).$$

- So an independent subset that maximizes the quantity $w(A)$ must minimize $w(A)$.
- Thus, any algorithm that can find an optimal subset A in an arbitrary matroid can solve the minimum-spanning-tree problem.
- Now we give a greedy algorithm that works for any weighted matroid.
- Its input is a weighted matroid $M = (S, \mathscr{I})$ (w: positive weight function)

Greedy Algorithms on a Weighted Matroid

GREEDY(M, w)

1: $A = \varnothing$
2: Sort $S[M]$ into monotonically decreasing order by weight w
3: **for** each $x \in S[M]$, taken in monotonically decreasing order by weight $w(x)$ **do**
4: **if** $A \cup \{x\} \in \mathscr{I}[M]$ **then**
5: $A = A \cup \{x\}$
6: **return** A

The algorithm is greedy because it considers each element $x \in S$ in turn in order of monotonically decreasing weight and immediately adds it to the set A being accumulated if $A \cup \{x\}$ is independent.

Performance of The Algorithm

Lemma 16.7: (Matroids exhibit the greedy-choice property) Suppose that $M = (S, \mathscr{I})$ is a weighted matroid with weight function w and that S is sorted into monotonically decreasing order by weight. Let x be the first element of S such that $\{x\}$ is independent if any such x exists. If x exists, then there exists an optimal subset A of S that contains x.

Lemma 16.8: Let $M = (S, \mathscr{I})$ be any matroid. If x is an element of S that is an extension of some independent subset A of S, then x is also an extension of $\varnothing$.

Corollary 16.9: Let $M = (S, \mathscr{I})$ be any matroid. If x is an element of S such that x is not an extension of $\varnothing$, then x is not an extension of any independent subset A of S.

Optimal-substructure property

Lemma 16.10: (Matroids exhibit the optimal-substructure property) Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, \mathscr{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathscr{I}')$, where $S' = \{y \in S : \{x, y\} \in \mathscr{I}\}$, $\mathscr{I}' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathscr{I}\}$, and the weight function for M' is the weight function for M, restricted to S'.

Theorem 16.11: (Correctness of the greedy algorithm on matroids) If $M = (S, \mathscr{I})$ is a weighted matroid with weight function w, then GREEDY(M, w) returns an optimal subset.

Contents

Task-scheduling problem

- Task-scheduling problem: Some unit-time tasks need to be scheduled optimally on a single processor, each task has a deadline, along with a penalty if the deadline is missed.
- A unit-time task is a job that requires exactly one unit of time to complete.
- Schedule for S is a permutation of S specifying the performing order of tasks.
- This problem can be solved using matroids

Task-scheduling problem

The problem of **scheduling unit-time tasks with deadlines and penalties for a single processor** has the following inputs:

- a set $S = \{a_1, a_2, \ldots, a_n\}$ of n unit-time tasks;
- a set of n integer deadlines $d_1, d_2, \ldots, d_n$, such that each d_i satisfies $1 \leq d_i \leq n$ and task a_i is supposed to finish by time d_i; and
- a set of n nonnegative weights or penalties $w_1, w_2, \ldots, w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task finishes by its deadline.

We are asked to find a schedule for S that **minimizes the total penalty** incurred for missed deadlines.

Task-scheduling problem

We say that a set A of tasks is **independent** if there exists a schedule for these tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms an independent set of tasks. Let $\mathscr{I}$ denote the set of all independent sets of tasks.

Theorem 16.13: If S is a set of unit-time tasks with deadlines, and $\mathscr{I}$ is the set of all independent sets of tasks, then the corresponding system $(S, \mathscr{I})$ is a matroid.