Greedy Algorithms

Bin Wang

School of Software
Tsinghua University

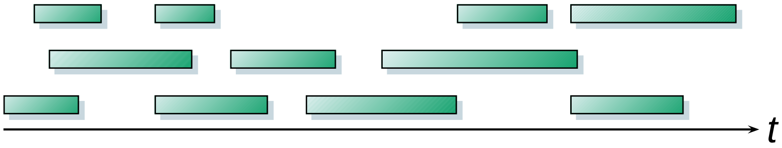
April 12, 2022

Outline

An activity-selection problem

Problem

Scheduling several competing activities with a goal of selecting a maximum-size set of mutually compatible activities.



An activity-selection problem

Mathematical illustration

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities***.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \leq s_i < f_i < \infty$.
- Activities a_i and a_j are ***compatible*** if $s_i \geq f_j$ or $s_j \geq f_i$.

An activity-selection problem

Mathematical illustration

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities***.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \leq s_i < f_i < \infty$.
- Activities a_i and a_j are ***compatible*** if $s_i \geq f_j$ or $s_j \geq f_i$.

An activity-selection problem

Mathematical illustration

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities***.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \leq s_i < f_i < \infty$.
- Activities a_i and a_j are ***compatible*** if $s_i \geq f_j$ or $s_j \geq f_i$.

An activity-selection problem

Mathematical illustration

- To select a ***maximum-size*** subset of mutually compatible 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	8	9	10	11	12	13	14

A DP solution

Step1: optimal substructure

- ***Making a choice:*** Suppose a_k is one of the activities selected.
- ***Space of subproblems:***
  1. The activities before a_k starts.
  2. The activities after a_k finishes.

A DP solution

Step1: optimal substructure

- ***Making a choice:*** Suppose a_k is one of the activities selected.
- ***Space of subproblems:***
  1. The activities before a_k starts.
  2. The activities after a_k finishes.

A DP solution

Step1: optimal substructure

- Can we combine the solution of subproblems into the solution of original problem?

  Let us start by defining sets

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\},$$

A DP solution

Step1: optimal substructure

- S_{ij} is the sub set of activities in S that start after activity a_i finishes and finish before activity a_j starts. Add $f_0 = 0$ and $s_{n+1} = \infty$, then $S = S_{0,n+1}$.
  Suppose a solution to S_{ij} includes some activity a_k, so that $f_i \le s_k < f_k \le s_j$, then we get two subproblems, S_{ik} and S_{kj}.

A DP solution

Step1: optimal substructure

- ***Optimal Substructure:*** Suppose now 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.

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

A DP solution

A recursive solution

Let $c[i,j]$ be the number of activities in a maximum size subset of mutually compatible activities in S_{ij}.

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

Converting a DP solution to a greedy one

Theorem 16.1

Consider any $S_k \neq \emptyset$, where $S_k = \{a_i \in S : s_i \geq f_k\}$. Let a_m be the activity in S_k with the ***earliest*** finish time.
Then a_m is used in some maximum-size subset of mutually compatible activities of S_k.

Converting a DP solution to a greedy one

Proof.

Suppose that A_k is a maximum-size subset of mutually compatible activities of S_k. Let a_j be the first activity in A_k.If $a_j = a_m$, we are done. If $a_j \neq a_m$, we construct the subset $A'_k = \{A_k - \{a_j\}\} \cup \{a_m\}$. A'_k is a maximum-size subset of mutually compatible activities of S_k that includes a_m. □

Converting a DP solution to a greedy one

Why is Theorem 16.1 valuable?

Reduce the space of subproblems

1. Only one subproblem is used in an optimal solution.
2. We need consider only one choice when solving the subproblem.

A recursive greedy algorithm

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

```
1  m = k + 1
2  while m ≤ n and s[m] < f[k]   // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Running time

$$\Theta(n)$$

A recursive greedy algorithm

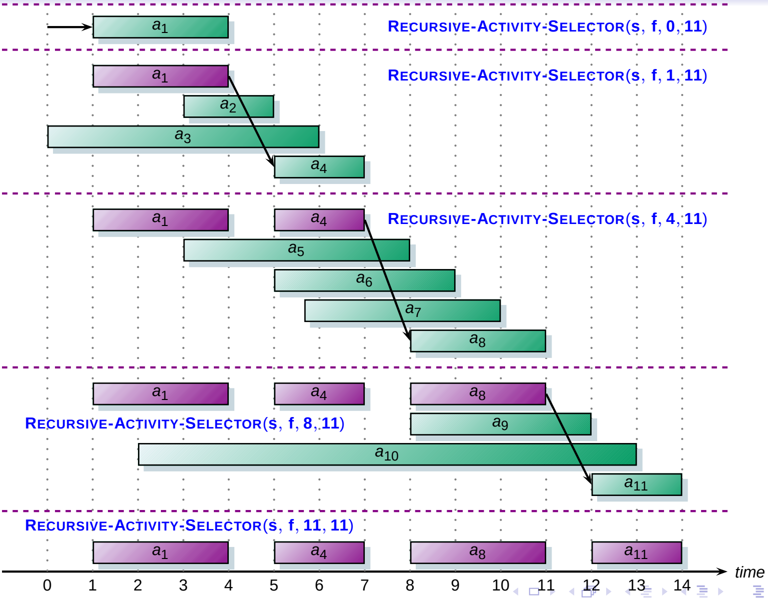
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
1  m = k + 1
2  while m ≤ n and s[m] < f[k]   // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Running time

$\Theta(n)$

RECURSIVE-ACTIVITY-SELECTOR



An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k = 1
for m = 2 to n
        if s[m] ≥ f[k]
                A = A ∪ {a_m}
                k = m
return A
```

Running time

$$\Theta(n)$$

An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k = 1
for m = 2 to n
        if s[m] ≥ f[k]
                A = A ∪ {a_m}
                k = m
return A
```

Running time

$\Theta(n)$

Steps of algorithm

Steps of activity-selection problem

1. Determine the optimal substructue of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.

Steps of algorithm

Steps of activity-selection problem

1. Determine the optimal substructue of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.

Steps of algorithm

Steps of activity-selection problem

1. Determine the optimal substructue of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.

Steps of algorithm

Steps of activity-selection problem

4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Steps of algorithm

Steps of activity-selection problem

4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Steps of algorithm

Steps of activity-selection problem

4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Steps of algorithm

Steps of greedy algorithms

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***.

Steps of algorithm

Steps of greedy algorithms

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***.

Steps of algorithm

Steps of greedy algorithms

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 subpoblem with the greedy choice we have made, we arrive at ***an optimal solution to the original problem***.

Greedy vs Dynamic programming

Greedy-choice property

- A globally optimal solution can be arrived at by making a locally **optimal (greedy)** choice.
- In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Greedy vs Dynamic programming

Greedy vs Dynamic programming

- Dynamic programming solves the subproblems bottom up, a greedy strategy progresses in a top-down fashion.
- Greedy is more efficient.
- Dynamic programming is more powerful, greedy is a special case of Dynamic programming.

Knapsack problem

0-1 knapsack problem

Given n items, the ith item is worth v_i dollars and weights w_i pounds, where v_i and w_i are integers. Give a knapsack with capacity W pounds, how to get a load with most valuable items?

Fractional knapsack problem

The setup is the same as 0-1 knapsack problem, but we can take fractions of items.

Knapsack problem

0-1 knapsack problem

Given n items, the ith item is worth v_i dollars and weights w_i pounds, where v_i and w_i are integers. Give a knapsack with capacity W pounds, how to get a load with most valuable items?

Fractional knapsack problem

The setup is the same as 0-1 knapsack problem, but we can take fractions of items.

Knapsack problem

0-1 knapsack problem

If we remove item j from this load, the remaining load must be at most valuable items weighing at most $W - w_j$ from $n - 1$ items.

Fractional knapsack problem

If we remove a weight w of one item j from the optimal load, then the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Knapsack problem

0-1 knapsack problem

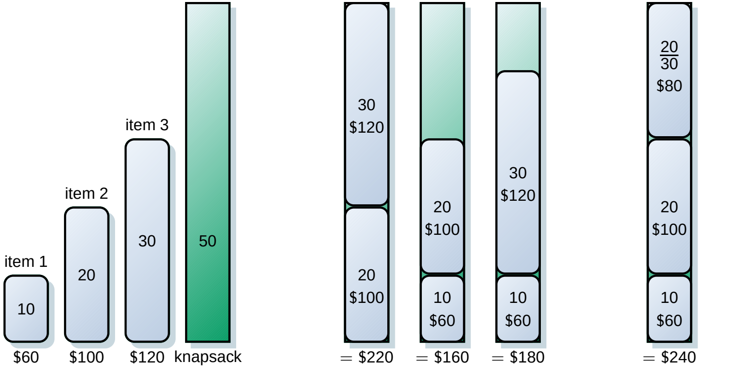
If we remove item j from this load, the remaining load must be at most valuable items weighing at most $W - w_j$ from $n - 1$ items.

Fractional knapsack problem

If we remove a weight w of one item j from the optimal load, then the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from the $n - 1$ original items plus $w_j - w$ pounds of item j.

An example

Fractional knapsack problem is solvable by a greedy strategy, whereas the 0-1 problem is not.



(a) (b) (c)

Overview

Binary character code

A **variable-length code** can do considerably better than a **fixed-length code**.

Prefix(-free) codes

The codes in which no codeword is also a prefix of some other codeword.

Overview

Binary character code

A **variable-length code** can do considerably better than a **fixed-length code**.

Prefix(-free) codes

The codes in which no codeword is also a prefix of some other codeword.

Overview

Huffman codes

- **Huffman codes** are a widely use and very effective technique for compressing data: saving of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- **Huffman codes** give frequent characters short codewords and infrequent characters long codewords.

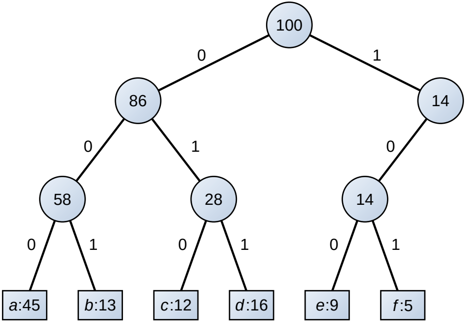
An Example

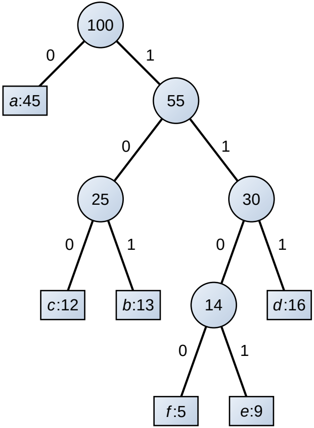
	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length	000	001	010	011	100	101
Variable-length	0	101	100	111	1101	1100

An Example

If each character is assigned a 3-bit codeword the file can be encoded in $300,000$ bits.
Using the variable-length code shown, the file can be encoded in $(45\cdot 1+13\cdot 3+12\cdot 3+16\cdot 3+9\cdot 4+5\cdot 4)\cdot 1000 = 224,000$ bits.
Example: 001011101 parses uniquely as $0{\cdot}0{\cdot}101{\cdot}1101$, which decodes to *aabe*

Trees corresponding to the example

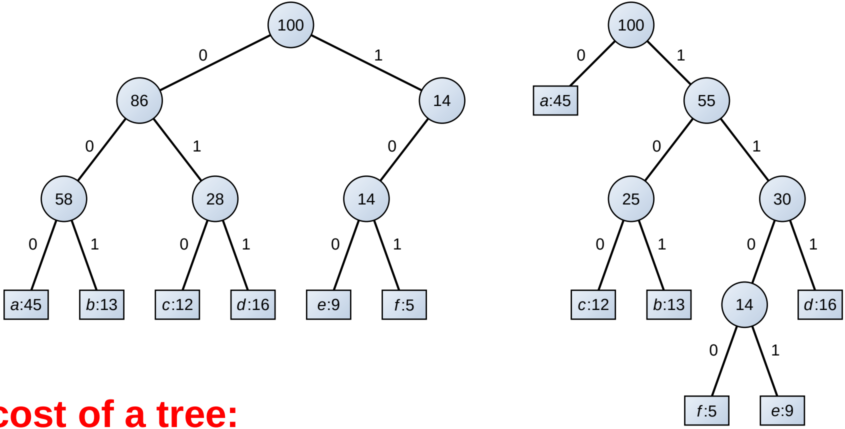




cost of a tree:

$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$, $d_T(c)$: depth.

Trees corresponding to the example



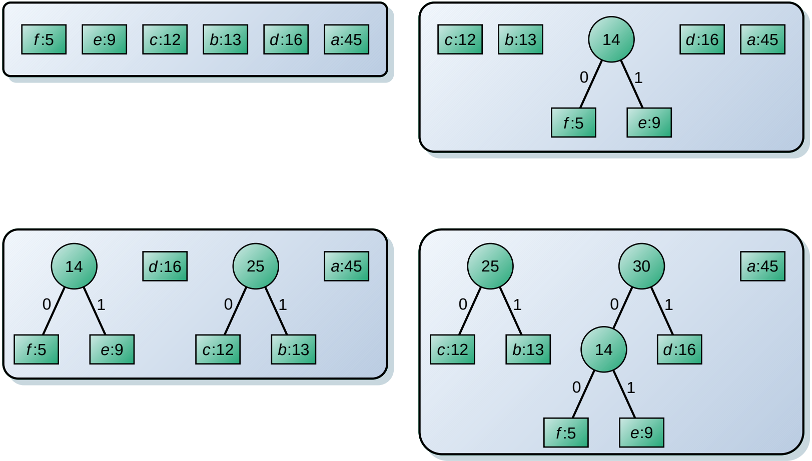
cost of a tree:

$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$, $d_T(c)$: depth.

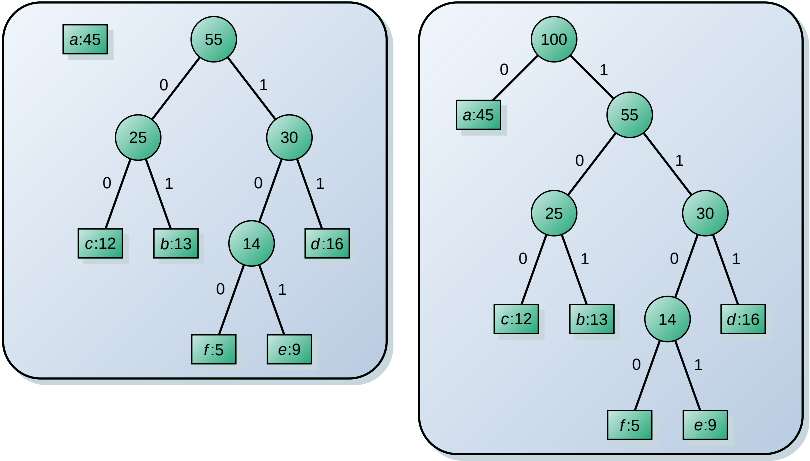
Constructing a Huffman code

```
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)
```

The steps of Huffman's algorithm



The steps of Huffman's algorithm



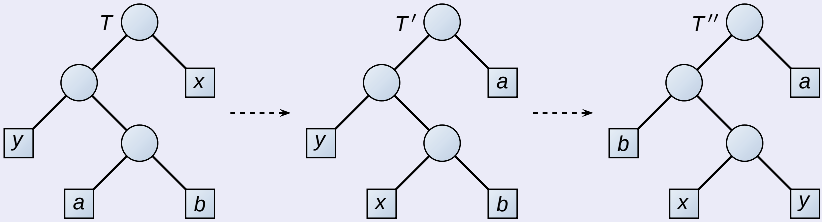
Correctness of Huffman's algorithm

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. 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

Proof



Correctness of Huffman's algorithm

Proof.

$$
\begin{aligned}
& B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) \\
&\quad - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) \\
&\quad - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0
\end{aligned}
$$

□

Correctness of Huffman's algorithm

Lemma 16.3

Let $C' = \{C - \{x, y\}\} \cup \{z\}$, where $z.freq = x.freq + y.freq$. 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.

Correctness of Huffman's algorithm

Proof

We first show that the cost $B(T)$ of tree T can be expressed in terms of the cost $B(T')$ of tree T'. For each $c \in C - \{x, y\}$, we have $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) \\ = (x.freq + y.freq)(d_{T'}(z) + 1) \\ = z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

Correctness of Huffman's algorithm

Proof

We first show that the cost $B(T)$ of tree T can be expressed in terms of the cost $B(T')$ of tree T'. For each $c \in C - \{x, y\}$, we have $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} & x.freq \cdot d_T(x) + y.freq \cdot d_T(y) \\ &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

Correctness of Huffman's algorithm

Proof

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

Suppose that T does not represent an optimal prefix code for c. Then there exists a tree T'' such that $B(T'') < B(T)$. T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frquency $z.freq = x.freq + y.freq$.

Correctness of Huffman's algorithm

Proof

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

Suppose that T does not represent an optimal prefix code for c. Then there exists a tree T'' such that $B(T'') < B(T)$. T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frquency $z.freq = x.freq + y.freq$.

Correctness of Huffman's algorithm

Proof.

Then

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \text{ **Contradiction!**} \end{aligned}$$

$\square$

Correctness of Huffman's algorithm

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

Proof.

Immediate from Lemmas 16.2 and 16.3. □

Matroids

History

- The concept of a (finite) matroid was introduced by **Hassler Whitney** in 1935 in his paper "*On the abstract properties of linear dependence*".
- In 1971, **Jack Edmonds** connected weighted matroids to greedy algorithms in his paper "*Matroids and the greedy algorithm*".

Matroids

History

- In the early of 1980's, **Korte and Lovász** generalized these ideas to objects called ***greedoids***, which allow even larger classes of problems to be solved by greedy algorithms.

Matroids

Definition

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

- S is a finite nonempty set.
- $\mathcal{I}$ is ***hereditary*** : $\mathcal{I}$ is a nonempty family of subsets of S, called the ***independent*** subsets of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. Obviously, $\emptyset$ is a member of $\mathcal{I}$.

Matroids

Definition

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

- S is a finite nonempty set.
- $\mathcal{I}$ is ***hereditary*** : $\mathcal{I}$ is a nonempty family of subsets of S, called the ***independent*** subsets of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. Obviously, $\emptyset$ is a member of $\mathcal{I}$.

Matroids

Definition

- If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the ***exchange property***.

Matroids

Graphic matroid

Given an undirected graph $G = (V, E)$, the ***graphic matroid*** is defined as follows.

1. The set S_G is defined to E, the set of edges of G.
2. If A is a subset of E, the $A \in \mathcal{I}_G$ 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.

Matroids

Theorem 16.5

If $G = (V, E)$ is an undirected graph, the $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

Proof.

- Clearly, $S_G = E$ is a finite set.
- $\mathcal{I}_G$ is *hereditary*, since a subset of a forest is a forest.

Matroids

Theorem 16.5

If $G = (V, E)$ is an undirected graph, the $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

Proof.

- Clearly, $S_G = E$ is a finite set.
- $\mathcal{I}_G$ is ***hereditary***, since a subset of a forest is a forest.

Matroids

Theorem 16.5

If $G = (V, E)$ is an undirected graph, the $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

Proof.

- Clearly, $S_G = E$ is a finite set.
- $\mathcal{I}_G$ is ***hereditary***, since a subset of a forest is a forest.

Matroids

Proof.

- ***Exchange property:***
  - Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$.
  - Thus, forest G_A contains $|V| - |A|$ trees which is more than $|V| - |B|$ trees of G_B.
  - So,G_B must contain some tree T whose vertices are in two different trees in forest G_A.
  - Moreover, since T is connect, there must be an edge (u, v) s.t. u and v are in different trees in G_A. $x = (u, v)$ can be added to G_A without creating a cycle.

$\square$

Matroids

Proof.

- ***Exchange property:***
  - Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$.
  - Thus, forest G_A contains $|V| - |A|$ trees which is more than $|V| - |B|$ trees of G_B.
  - So, G_B must contain some tree T whose vertices are in two different trees in forest G_A.
  - Moreover, since T is connect, there must be an edge (u, v) s.t. u and v are in different trees in G_A. $x = (u, v)$ can be added to G_A without creating a cycle.

$\square$

Matroids

Proof.

- ***Exchange property:***
  - Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$.
  - Thus, forest G_A contains $|V| - |A|$ trees which is more than $|V| - |B|$ trees of G_B.
  - So,G_B must contain some tree T whose vertices are in two different trees in forest G_A.
  - Moreover, since T is connect, there must be an edge (u, v) s.t. u and v are in different trees in G_A. $x = (u, v)$ can be added to G_A without creating a cycle.

□

Matroids

Proof.

- ***Exchange property:***
  - Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$.
  - Thus, forest G_A contains $|V| - |A|$ trees which is more than $|V| - |B|$ trees of G_B.
  - So,G_B must contain some tree T whose vertices are in two different trees in forest G_A.
  - Moreover, since T is connect, there must be an edge (u, v) s.t. u and v are in different trees in G_A. $x = (u, v)$ can be added to G_A without creating a cycle. □

Matroids

Definition

Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an ***extension*** of $A \in \mathcal{I}$ if x can be added to A while preserving independence; that is, x is an ***extension*** of A if $A \cup \{x\} \in \mathcal{I}$.

Definition

If A is an ***independent*** subset in a matroid M, we say that A is ***maximal*** if it has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M.

Matroids

Definition

Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an ***extension*** of $A \in \mathcal{I}$ if x can be added to A while preserving independence; that is, x is an ***extension*** of A if $A \cup \{x\} \in \mathcal{I}$.

Definition

If A is an ***independent*** subset in a matroid M, we say that A is ***maximal*** if it has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M.

Matroids

Theorem 16.6

All ***maximal*** independent subsets in a matroid has the **same** size.

Matroids

Proof.

- Suppose to the contrary that A is a maximal independent subset of M and there exists another ***larger*** maximal independent subset B of M.
- Then, the exchange property implies that A is extendible to a ***larger*** independent set $A \cup \{x\}$ for some $x \in B - A$, contradicting the assumption that A is ***maximal***. $\square$

Matroids

Proof.

- Suppose to the contrary that A is a maximal independent subset of M and there exists another ***larger*** maximal independent subset B of M.
- Then, the exchange property implies that A is extendible to a ***larger*** independent set $A \cup \{x\}$ for some $x \in B - A$, contradicting the assumption that A is ***maximal***. $\square$

Matroids

Example

Every maximal independent subset of M_G must be a free tree with exactly $|V| - 1$ edges that connects all the vertices of G. Such a tree is called a ***spanning tree*** of G.

Greedy algorithms on a weighted matroid

Definition

We say that a matroid $M = (S, \mathcal{I})$ 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 extends to subsets of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any $A \subseteq S$.

Greedy algorithms on a weighted matroid

Why weighted matroid?

Many problems for which a greedy approach provides optimal solutions can be formulated in terms of finding a ***maximum-weight independent subset*** in a weighted matroid.

Greedy algorithms on a weighted matroid

Example

- ***Minimum-spanning-tree problem:*** 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 are asked to find a subset of the edges that connects all of the vertices together and has minimum total length.

Greedy algorithms on a weighted matroid

Example

- ***Solution:*** 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. Each maximal independent subset A corresponds to a spanning tree, and since

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

Greedy algorithms on a weighted matroid

Example

- ***Solution(cont.):*** for any maximal independent subset A, 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.

Greedy algorithms on a weighted matroid

```
GREEDY(M, w)
  A = ∅
  sort M.S into monotonically
          decreasing order by weight w
  for each x ∈ M.S, taken in monotonically
          decreasing order by weight w(x)
      if A ∪ {x} ∈ M.I
          A = A ∪ {x}
  return A
```

Running time

$$O(n \lg n + nf(n))$$

Greedy algorithms on a weighted matroid

GREEDY(M, w)

1 $A = \emptyset$
2 sort $M.S$ into monotonically decreasing order by weight w
3 **for** each $x \in M.S$, taken in monotonically decreasing order by weight $w(x)$
4 **if** $A \cup \{x\} \in M.\mathcal{I}$
5 $A = A \cup \{x\}$
6 **return** A

Running time

$$O(n \lg n + nf(n))$$

Greedy algorithms on a weighted matroid

Lemma 16.7 (Matroids exhibit the greedy-choice property)

Suppose that $M = (S, \mathcal{I})$ 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.

Greedy algorithms on a weighted matroid

Proof.

- If no such x exists, then the only independent subset is the empty set and the lemma is vacuously true.
- Otherwise, let B be any nonempty optimal subset. Assume that $x \notin B$; otherwise, letting $A = B$ gives an optimal subset of S that contains x.

Greedy algorithms on a weighted matroid

Proof.

- $\forall y \in B$, our choice of x ensures that $w(x) \geq w(y)$.
- Construct A as follows.
  - Begin with $A = \{x\}$. By the choice of x, A is independent.
  - Using the ***exchange property***, repeatedly find a new element of B that can be added to A until $|A| = |B|$ while preserving the independence of A.

Greedy algorithms on a weighted matroid

Proof.

- $\forall y \in B$, our choice of x ensures that $w(x) \geq w(y)$.
- Construct A as follows.
  - Begin with $A = \{x\}$. By the choice of x, A is independent.
  - Using the ***exchange property***, repeatedly find a new element of B that can be added to A until $|A| = |B|$ while preserving the independence of A.

Greedy algorithms on a weighted matroid

Proof.

- Construct A as follows(Cont.).
  - Then, $A = \{B - \{y\}\} \cup \{x\}$ for some $y \in B$, and so

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B) \end{aligned}$$

Because B is optimal, A must also be optimal, and because $x \in A$, the lemma is proven. □

Greedy algorithms on a weighted matroid

Lemma 16.8

Let $M = (S, \mathcal{I})$ 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 $\emptyset$.

Proof.

Since x is an extension of A, we have that $A \cup \{x\}$ is independent. Since $\mathcal{I}$ is hereditary, $\{x\}$ must be independent. Thus, x is an extension of $\emptyset$. $\square$

Greedy algorithms on a weighted matroid

Lemma 16.8

Let $M = (S, \mathcal{I})$ 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 $\emptyset$.

Proof.

Since x is an extension of A, we have that $A \cup \{x\}$ is independent. Since $\mathcal{I}$ is hereditary, $\{x\}$ must be independent. Thus, x is an extension of $\emptyset$. □

Greedy algorithms on a weighted matroid

Corollary 16.9

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S such that x is ***not*** an extension of $\emptyset$, then x is ***not*** an extension of any independent subset A of S.

Greedy algorithms on a weighted matroid

Lemma 16.10 (Matroids exhibit the optimal-substructure property)

Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$,

Greedy algorithms on a weighted matroid

Lemma 16.10 (Matroids exhibit the optimal-substructure property)

where

$$S' = \{y \in S : \{x, y\} \in \mathcal{I}\},$$
$$\mathcal{I}' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\},$$

and the weight function for M' is the weight function for M, restricted to S'. (We call M' the ***contraction*** of M by the element x.)

Greedy algorithms on a weighted matroid

Proof.

- If A is any maximum-weight independent subset of M containing x, then $A' = A - \{x\}$ is an independent subset of M'.
- Conversely, any independent subset A' of M' yields an independent subset $A = A' \cup \{x\}$ of M.

Greedy algorithms on a weighted matroid

Proof.

- If A is any maximum-weight independent subset of M containing x, then $A' = A - \{x\}$ is an independent subset of M'.
- Conversely, any independent subset A' of M' yields an independent subset $A = A' \cup \{x\}$ of M.

Greedy algorithms on a weighted matroid

Proof.

- Since we have in both cases that $w(A) = w(A') + w(x)$, a maximum-weight solution in M containing x yields a maximum-weight solution in M', and vice versa.

□

Greedy algorithms on a weighted matroid

Theorem 16.11 (Correctness of the greedy algorithm on matroids)

If $M = (S, \mathcal{I})$ is a weighted matroid with weight function w, then $\textsc{Greedy}(M, w)$ returns an optimal subset.

Greedy algorithms on a weighted matroid

Proof.

- By Corollary 16.9, any elements that are passed over initially because they are not extensions of $\emptyset$ can be forgotten about, since they can never be useful.
  Once the first element x is selected, Lemma 16.7 implies that GREEDY ***does not err*** by adding x to A, since there exists an optimal subset containing x.

Greedy algorithms on a weighted matroid

Proof.

- Finally, Lemma 16.10 implies that the remaining problem is one of finding an optimal subset in the matroid M' that is the contraction of M by x.
  B is independent in $M' \iff B \cup \{x\}$ is independent in M.

Greedy algorithms on a weighted matroid

Proof.

- Thus, the subsequent operation of GREEDY will find a maximum-weight independent subset for M', and the overall operation of GREEDY will find a maximum-weight independent subset for M.

$\square$

A task-scheduling problem

Definition

- A ***unit-time task*** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete.
- Given a finite set S of unit-time tasks, a ***schedule*** for S is a permutation of S specifying the order in which these tasks are to be performed.

A task-scheduling problem

Definition

- The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

A task-scheduling problem

Definition

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 task-scheduling problem

Definition

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 task-scheduling problem

Definition

- 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.

A task-scheduling problem

Definition

- 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.

A task-scheduling problem

Definition

- A task is ***late*** in this schedule if it finishes after its deadline. Otherwise, the task is ***early*** in the schedule.
- An arbitrary schedule can always be put into ***early-first form***, in which the **early** tasks precede the **late** tasks.

A task-scheduling problem

Definition

- A task is ***late*** in this schedule if it finishes after its deadline. Otherwise, the task is ***early*** in the schedule.
- An arbitrary schedule can always be put into ***early-first form***, in which the **early** tasks precede the **late** tasks.

A task-scheduling problem

Definition

- An arbitrary schedule can always be put into ***canonical form***, in which the **early** tasks precede the **late** tasks and the early tasks are scheduled in order of monotonically increasing deadlines.

A task-scheduling problem

Strategy of scheduling

1. Find a set A of tasks that are to be early in the optimal schedule.
2. Once A is determined, we can create the actual schedule by listing the elements of A in order of monotonically increasing deadline.
3. List the late tasks (*i.e.*, $S - A$) in any order, producing a canonical ordering of the optimal schedule.

A task-scheduling problem

Strategy of scheduling

1. Find a set A of tasks that are to be early in the optimal schedule.
2. Once A is determined, we can create the actual schedule by listing the elements of A in order of monotonically increasing deadline.
3. List the late tasks (*i.e.*, $S - A$) in any order, producing a canonical ordering of the optimal schedule.

A task-scheduling problem

Strategy of scheduling

1. Find a set A of tasks that are to be early in the optimal schedule.
2. Once A is determined, we can create the actual schedule by listing the elements of A in order of monotonically increasing deadline.
3. List the late tasks ($i.e., S - A$) in any order, producing a canonical ordering of the optimal schedule.

A task-scheduling problem

Definition

- 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 $\mathcal{I}$ denote the set of all independent sets of tasks.
- For $t = 0, 1, 2, \ldots, n$, let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier. Note that $N_0(A) = 0$ for any set A.

A task-scheduling problem

Lemma 16.12

For any set of tasks A, the following statements are equivalent.

1. The set A is independent.
2. For $t = 0, 1, 2, \ldots, n$, we have $N_t(A) \leq t$.
3. If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.

Proof.

Trivial! $\square$

A task-scheduling problem

Theorem 16.13

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

Proof.

- ***Hereditary:*** Every subset of an independent set of tasks is certainly independent.

A task-scheduling problem

Theorem 16.13

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

Proof.

- ***Hereditary:*** Every subset of an independent set of tasks is certainly independent.

A task-scheduling problem

Proof.

- ***Exchange property:*** Suppose B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) \leq N_t(A)$. (Such a value of t exists, since $N_0(A) = N_0(B) = 0$.)

A task-scheduling problem

Proof.

- ***Exchange property(Cont.):***
  Since $N_n(B) = |B|$, $N_n(A) = |A|$ and $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all j in the range $k + 1 \leq j \leq n$. Therefore, B contains more tasks with deadline $k + 1$ than A does. Let a_i be a task in $B - A$ with deadline $k + 1$. Let $A' = A \cup \{a_i\}$.

A task-scheduling problem

Proof.

- ***Exchange property(Cont.):***
  For $0 \leq t \leq k$, we have $N_t(A') = N_t(A) \leq t$, since A is independent. For $k < t \leq n$, we have $N_t(A') \leq N_t(B) \leq t$, since B is independent. Therefore, A' is independent, completing our proof that $(S, \mathcal{I})$ is a matroid.

$\square$

A task-scheduling problem

Example

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

The final optimal schedule is $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$, which has a total penalty incurred of $w_5 + w_6 = 50$.

Running time

$$O(n^2)$$

A task-scheduling problem

Example

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

The final optimal schedule is $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$, which has a total penalty incurred of $w_5 + w_6 = 50$.

Running time

$$O(n^2)$$

A task-scheduling problem

Example

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

The final optimal schedule is $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$, which has a total penalty incurred of $w_5 + w_6 = 50$.

Running time

$$O(n^2)$$