Introduction to Algorithms
Lecture 6 Greedy Method

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

Outline

Overview

Greedy method — a powerful algorithm design technique

1. Very efficient
2. Simple to implement
3. Widely used in practice

Overview

Greedy method — a powerful algorithm design technique

1. Very efficient
2. Simple to implement
3. Widely used in practice

Two ways to understand it

Greedy Algorithms

Find OPT solution efficiently
- DP++
- strong structure on problems

Find approximately good sol
- broadly used
- What is app. good??

Short Intro

For optimization problems, a greedy algorithm makes the choice that is the best at this moment — called local optimal

Short Intro

For optimization problems, a greedy algorithm makes the choice that is the best at this moment — called local optimal

1. For many problems, locally optimal choices lead to a globally optimal solution
2. For more problems, it does not — but one could show the solution is not bad

Short Intro

For optimization problems, a greedy algorithm makes the choice that is the best at this moment — called local optimal

1. For many problems, locally optimal choices lead to a globally optimal solution
2. For more problems, it does not — but one could show the solution is not bad

Main Focus

Proving the correctness of greedy algorithms is highly non-trivial.

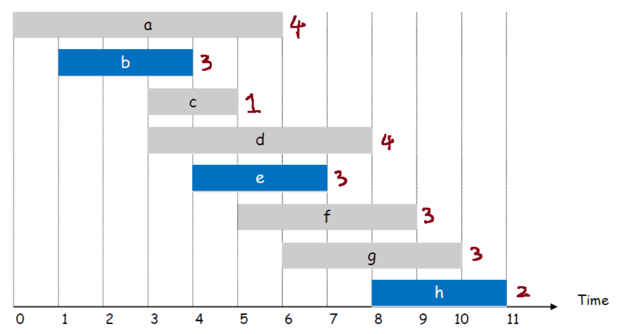
Outline

Interval Scheduling Problem

Recall: DP solves weighted scheduling

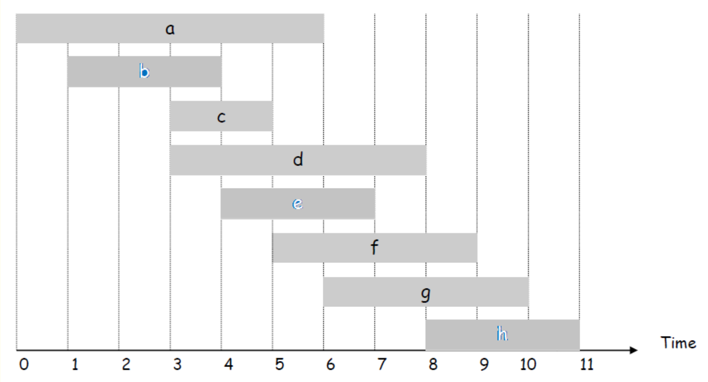
Description

1. n jobs in $[0, T]$
2. Job j starts from $s(j)$ and finishes at $f(j)$ with weight w_j
3. Two jobs are compatible if they don't overlap.
4. Goal: find a max-weight subset with compatible jobs.



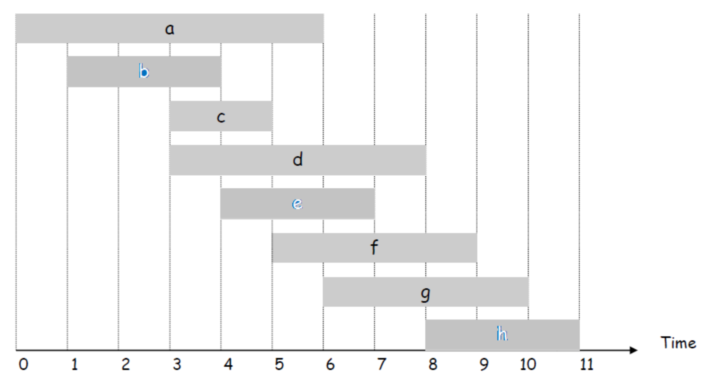
Special Case: Unit Weights

What if all weights are 1?



Special Case: Unit Weights

What if all weights are 1?

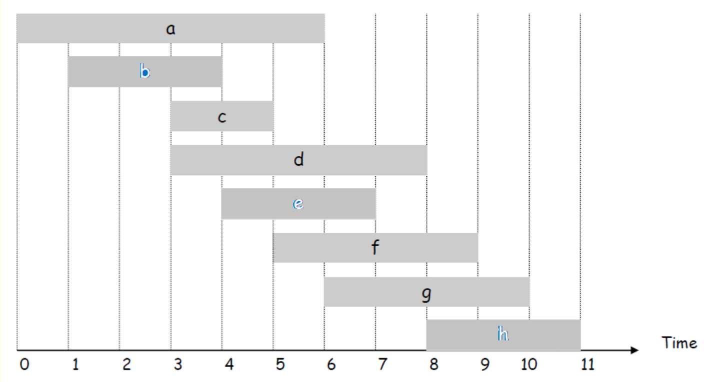


Questions

While DP still works, faster or simpler algorithms?

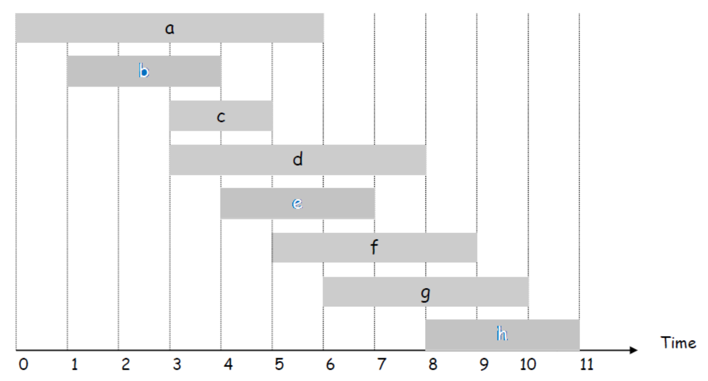
Intuition

Let us try greedy choices.



Intuition

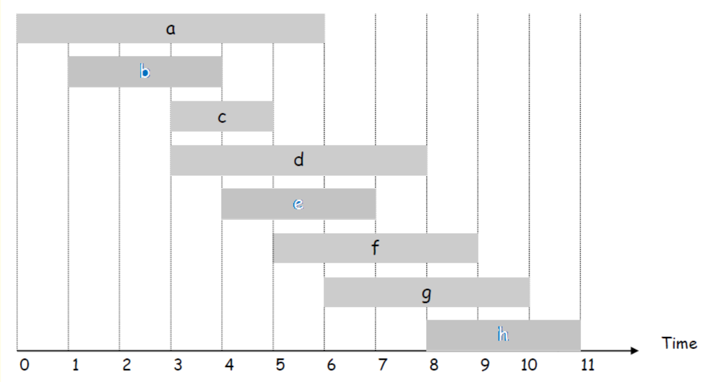
Let us try greedy choices.



1. What if we pick the job with the earliest starting time?

Intuition

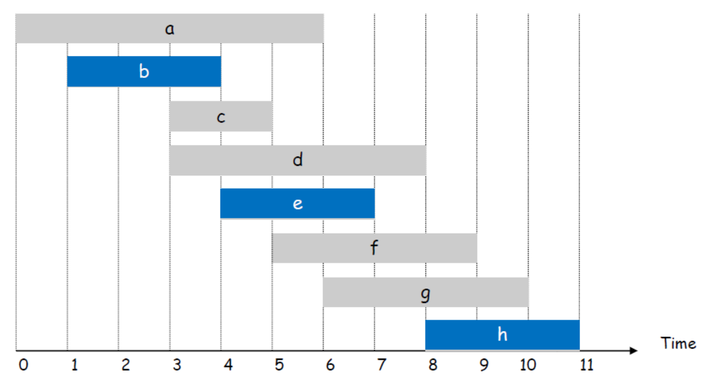
Let us try greedy choices.



1. What if we pick the job with the earliest starting time?
2. What if we pick the job with the earliest finish time?

Idea

Intuition: When there are multiple jobs, picking the one with the earliest finish time leaves the most space for future jobs.



Formal Proof

Theorem 16.1 in CLRS

For any problem S of job scheduling, let a_1 be the job with the earliest finish time. Then a_1 is included in some max-size compatible subset OPT of S.

Formal Proof

Theorem 16.1 in CLRS

For any problem S of job scheduling, let a_1 be the job with the earliest finish time. Then a_1 is included in some max-size compatible subset OPT of S.

1. Moreover, $OPT \setminus \{a_1\}$ is optimal for $[0, T] \setminus \big[0, f(a_1)\big]$.
2. The solution of subproblem $OPT' \cup \{a_1\}$ is optimal for S — by dynamic programming
3. Guarantee that we will reach a global OPT solution by making local OPT choice.

Algorithm

1. Recall $s[j]$ and $f[j]$ are the start and finish time separately
2. Sort all jobs according to the finish time $f[j]$.

```
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
```

Algorithm

1. Recall $s[j]$ and $f[j]$ are the start and finish time separately
2. Sort all jobs according to the finish time $f[j]$.

```
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
```

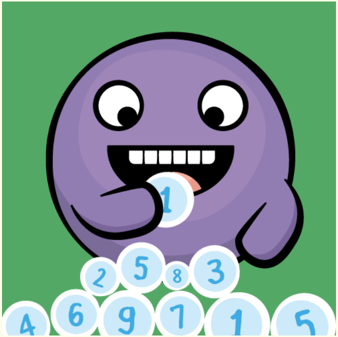
Analysis

Correctness follows from Theorem 16.1
Running time: $O(n \log n)$.

Discussion

While the running time is the same, greedy algorithm is arguably simpler than DP.

1. Easy to implement
2. However, it takes more effort to prove its correctness



Outline

Problem

Given a sequence $a_1, \ldots, a_n$ of integers, find the largest sum of a consecutive interval, i.e., $\max_{k<\ell} \left\{ \sum_{i=k}^{\ell} a_i \right\}$.

$$n=10: \ \underline{1} \quad -2 \quad \boxed{3 \ \ -1 \ \ 4 \ \ -2 \ \ 3 \ \ 1} \ -5 \quad 4$$

Problem

Given a sequence $a_1, \ldots, a_n$ of integers, find the largest sum of a consecutive interval, i.e., $\max\limits_{k<\ell}\left\{ \sum_{i=k}^{\ell} a_i \right\}$.

$n=10$: $1 \quad -2 \quad \boxed{3 \quad -1 \quad 4 \quad -2 \quad 3 \quad 1} \quad -5 \quad 4$

First idea

Enumerate all pairs k and ℓ — $O(n^2)$

1st Algorithm

```
function MAX-SUM(a)
    ans = 0
    for k = 1, ..., n do
        sum = 0
        for ℓ = k, ..., n do
            sum = sum + a[ℓ]
            ans = max{ans, sum}
    Return ans
```

Question

Can we design faster algorithms?

2nd Idea:

One standard trick: let $s[0] = 0$ and $s[i] = a_1 + a_2 + \cdots + a_i$ s.t. sum $\sum_{i=k}^{\ell} a_i = s[\ell] - s[k-1]$.

$$\text{Problem becomes } \max_{k \leqslant \ell} \left\{ s[\ell] - s[k-1] \right\}.$$

2nd Idea:

One standard trick: let $s[0] = 0$ and $s[i] = a_1 + a_2 + \cdots + a_i$ s.t. sum $\sum_{i=k}^{\ell} a_i = s[\ell] - s[k-1]$.

$$\text{Problem becomes } \max_{k \leqslant \ell} \left\{ s[\ell] - s[k-1] \right\}.$$

Dynamic Programming

Let $f[\ell]$ denote the max-sum whose endpoint is ℓ:

$$f[\ell] = s[\ell] - \min_{k \leqslant \ell} s[k-1]$$

$$ans = \max \left\{ f[1], \ldots, f[n] \right\}$$

2nd Idea:

One standard trick: let $s[0] = 0$ and $s[i] = a_1 + a_2 + \cdots + a_i$ s.t. sum $\sum_{i=k}^{\ell} a_i = s[\ell] - s[k-1]$.

$$\text{Problem becomes } \max_{k \leqslant \ell} \left\{ s[\ell] - s[k-1] \right\}.$$

Dynamic Programming

Let $f[\ell]$ denote the max-sum whose endpoint is ℓ:

$$f[\ell] = s[\ell] - \min_{k \leqslant \ell} s[k-1]$$

$$ans = \max \left\{ f[1], \ldots, f[n] \right\}$$

Time $O(n \log n)$: Use a data structure (e.g., heap or BST) to find $\min_{k \leqslant \ell} \{s[k-1]\}$ for every ℓ in time $O(\log n)$.

Improve DP via greedy method

Back to $f[\ell] = s[\ell] - \min_{k<\ell} s[k-1]$, consider ℓ and $\ell+1$:

$$f[\ell] = s[\ell] - \min_{k \leqslant \ell} s[k-1],$$

$$f[\ell+1] = s[\ell+1] - \min_{k \leqslant \ell+1} s[k-1]$$

Improve DP via greedy method

Back to $f[\ell] = s[\ell] - \min_{k<\ell} s[k-1]$, consider ℓ and $\ell+1$:

$$f[\ell] = s[\ell] - \min_{k \leqslant \ell} s[k-1],$$

$$f[\ell+1] = s[\ell+1] - \min_{k \leqslant \ell+1} s[k-1]$$

OBS on $\min_k$

$$\min_{k \leqslant \ell+1} s[k-1] = \min\left(\min_{k \leqslant \ell} s[k-1], s[\ell] \right)$$

Improve DP via greedy method

Back to $f[\ell] = s[\ell] - \min_{k<\ell} s[k-1]$, consider ℓ and $\ell+1$:

$$f[\ell] = s[\ell] - \min_{k \leqslant \ell} s[k-1],$$

$$f[\ell+1] = s[\ell+1] - \min_{k \leqslant \ell+1} s[k-1]$$

OBS on $\min_k$

$$\min_{k \leqslant \ell+1} s[k-1] = \min\left(\min_{k \leqslant \ell} s[k-1], s[\ell]\right)$$

---

function MAX-SUM(a)
- $ans = 0, k = 0$
- Compute $s[1], \ldots, s[n]$
- **for** $\ell = 1, \ldots, n$ **do**
  - $k = \arg\min\left\{s[k], s[\ell-1]\right\}$
  - $ans = \max\{ans, s[\ell] - s[k]\}$
- Return ans

Greedy Algorithm

Actually, it is a greedy algorithm while it looks like a DP.

An alternate implementation

Let sum_ℓ denote the largest sum whose endpoint is ℓ s.t. $sum_\ell = \max\{a_\ell, sum_{\ell-1} + a_\ell\}$.

---

function $\textsc{Max-Sum}(a)$

$\quad ans = 0, sum = 0$

$\quad$ **for** $\ell = 1, \ldots, n$ **do**

$\quad\quad sum = \max\{0, sum\} + a_\ell$

$\quad\quad ans = \max\{ans, sum\}$

$\quad$ Return *ans*

Summary

1. Greedy algorithms are elegant and fast
2. Need more analysis and proofs (compare to DP and divide& conquer)

Summary

1. Greedy algorithms are elegant and fast
2. Need more analysis and proofs (compare to DP and divide& conquer)
3. So far, view it as a way to improve DP
4. Next, greedy algorithms solve many problems where DP can't help with

Summary

1. Greedy algorithms are elegant and fast
2. Need more analysis and proofs (compare to DP and divide& conquer)
3. So far, view it as a way to improve DP
4. Next, greedy algorithms solve many problems where DP can't help with

Strongly recommend: Read 15.3 and 16.2 to understand DP and greedy algorithms better

Outline

Introduction

A classical problem in data compression

— solved by Huffman at 1952 in MIT

1. Consider a data set with alphabet say $\{a, b, c, \ldots, z\}$, where each character appears $f_a, \ldots, f_z$ times
2. Find a binary encoding of $\{a, b, c, \ldots, z\}$, called codewords, to minimize the total length.

$a \times 85 \quad b \times 50 \quad c \times 60$

encode	a 0	a 0	a 00
	b 00	b 10	b 01
	c 1	c 11	c 1
	invalid!	length $1\times 85+2\times 50+2\times 60$	$2\times 85+2\times 50+1\times 60$

Introduction

A classical problem in data compression

— solved by Huffman at 1952 in MIT

1. Consider a data set with alphabet say $\{a, b, c, \ldots, z\}$, where each character appears $f_a, \ldots, f_z$ times
2. Find a binary encoding of $\{a, b, c, \ldots, z\}$, called codewords, to minimize the total length.
3. Extra requirement — prefix codes: no codeword is a prefix of some other codewords.

$a \times 85$ $\quad b \times 50$ $\quad c \times 60$

encode	a	0	a	0	a	00
	b	00	b	10	b	01
	c	1	c	11	c	1

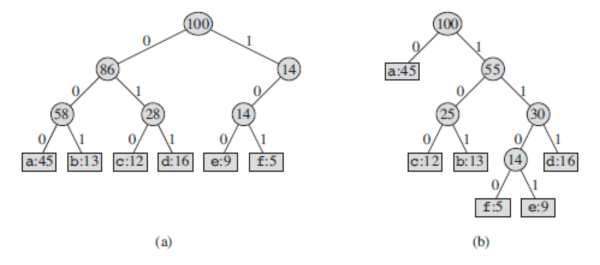
invalid!

length $1\times 85+2\times 50+2\times 60$

$2\times 85+2\times 50+1\times 60$

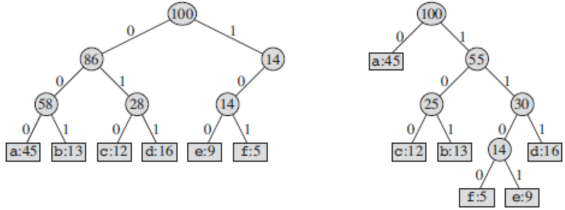
Idea: Encode them as a tree

1. Each leaf contains a character, whose codeword is the $\{0, 1\}$-path from the root
2. The number on each node denotes the sum of frequencies of all leaves in its subtree



Idea: Encode them as a tree

1. Each leaf contains a character, whose codeword is the $\{0, 1\}$-path from the root
2. The number on each node denotes the sum of frequencies of all leaves in its subtree



(a) (b)

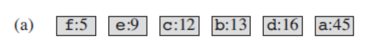
OBS

1. It satisfies the prefix codes requirement.
2. The total length is the sum of frequencies on nodes.

Greedy Algorithm

New goal: Construct a tree minimize the sum of weights

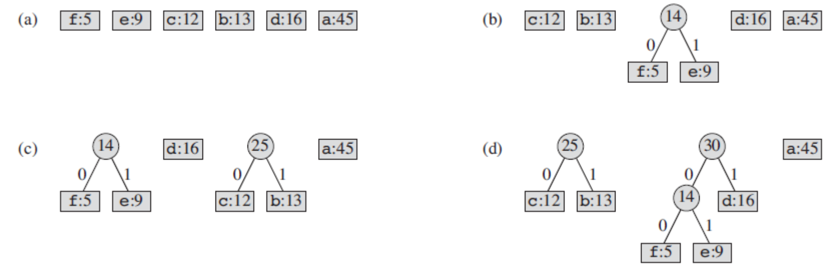
1. Let C be the set of n characters
2. For each $c \in C$, $c.freq$ is its frequency



Greedy Algorithm

New goal: Construct a tree minimize the sum of weights

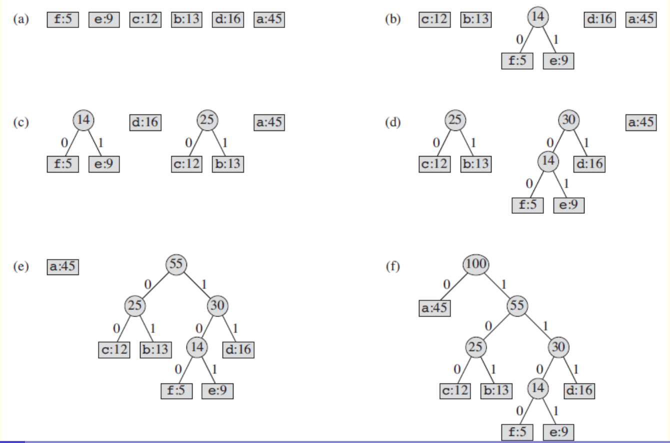
1. Let C be the set of n characters
2. For each $c \in C$, $c.freq$ is its frequency
3. Reduce to subproblem: Merge two nodes with the lowest frequencies into one



Greedy Algorithm

New goal: Construct a tree minimize the sum of weights

1. Let C be the set of n characters
2. For each $c \in C$, $c.freq$ is its frequency
3. Reduce to subproblem: Merge two nodes with the lowest frequencies into one



Algorithm Description

Implement it via a min-heap

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

Algorithm Description

Implement it via a min-heap

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

Analysis

1. Running Time: $O(n \log n)$ by heap
2. Correctness: Next 2 slides

Correctness

2 Steps: Consider one merge operation

1. $\exists$ a optimal solution T with the merged pattern
2. For the subproblem after merging, its optimal solution $T' \Rightarrow$ an optimal solution T of original problem

Correctness

2 Steps: Consider one merge operation

1. $\exists$ a optimal solution T with the merged pattern
2. For the subproblem after merging, its optimal solution $T' \Rightarrow$ an optimal solution T of original problem

Step 1: Lemma 16.2 in CLRS

Let $C = \{c_1, \ldots, c_n\}$ where $c_1.freq \geqslant c_2.freq \geqslant \cdots \geqslant c_n.freq$. Then $\exists$ an optimal prefix code whose binary tree T has a node with children c_{n-1} and c_n as two of the deepest leaves.

Correctness cont.

Step 2: Lemma 16.3 in CLRS

Let $C' = \{c_0, c_1, \ldots, c_{n-2}\}$ where $c_0.freq = c_{n-1}.freq + c_n.freq$. Let T' denote the binary tree of any optimal encoding of C'. Then the binary tree T of C, obtained by splitting the leaf c_0 into c_{n-1} and c_n, is optimal.

Correctness cont.

Step 2: Lemma 16.3 in CLRS

Let $C' = \{c_0, c_1, \ldots, c_{n-2}\}$ where $c_0.freq = c_{n-1}.freq + c_n.freq$. Let T' denote the binary tree of any optimal encoding of C'. Then the binary tree T of C, obtained by splitting the leaf c_0 into c_{n-1} and c_n, is optimal.

1. Lemma 16.2 says it is OK to combine c_{n-1} and c_n since there exists an optimal solution doing that
2. Lemma 16.3 says any optimal solution of the subproblem is good

Summary

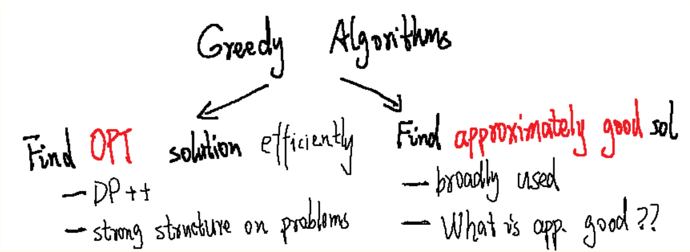
1. Greedy algorithms are elegant and fast
2. Proving Correctness is involved

Summary

1. Greedy algorithms are elegant and fast
2. Proving Correctness is involved
3. For many problems, greedy algorithms improves the running time of dynamic programming
4. Greedy algorithms provide efficient solutions for many problems where DP can not help with like Huffman code — more examples from computational hard problems

Summary

1. Greedy algorithms are elegant and fast
2. Proving Correctness is involved
3. For many problems, greedy algorithms improves the running time of dynamic programming
4. Greedy algorithms provide efficient solutions for many problems where DP can not help with like Huffman code — more examples from computational hard problems



Outline

Approximation Algorithms

1. Hard or slow to find optimal solutions in many problems — slow means a large polynomial time or super-polynomial time
2. Computer Science is happy with an efficient algorithm if its output is not bad
3. Formally, it means approximately good — say the minimum is OPT, its answer is at most $\alpha \cdot OPT$ for some $\alpha > 1$
4. $\alpha :=$ approximation ratio
5. Finding approximation solutions is a central idea in CS — streaming algorithms (heavy hitters), big data algorithms, machine learning algorithms, privacy, cryptography, ...

Set Cover

Given a ground set $[n] := \{1, 2, \ldots, n\}$ and m subsets $S_1, \ldots, S_m$, find the smallest selection of subsets S_i whose union is $[n]$.

Set Cover

Given a ground set $[n] := \{1, 2, \ldots, n\}$ and m subsets $S_1, \ldots, S_m$, find the smallest selection of subsets S_i whose union is $[n]$.

Greedy Algorithm

Repeat until $[n]$ is covered:
  Pick S_i with the largest uncovered elements.

Set Cover

Given a ground set $[n] := \{1, 2, \ldots, n\}$ and m subsets $S_1, \ldots, S_m$, find the smallest selection of subsets S_i whose union is $[n]$.

Greedy Algorithm

Repeat until $[n]$ is covered:
  Pick S_i with the largest uncovered elements.

1. Simple and easy to implement
2. Can not guarantee that the answer is optimal

Main Results

Theorem

If the optimal solution uses k subsets, the greedy algorithm uses at most $k \ln n$ subsets.

Main Results

Theorem

If the optimal solution uses k subsets, the greedy algorithm uses at most $k \ln n$ subsets.

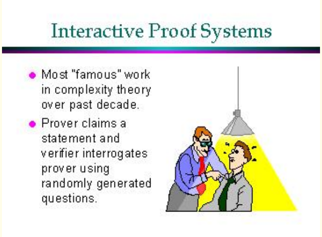
1. Greedy algorithm is not too bad ☺— approximation ratio is $\ln n$
2. Can we find optimal solution efficiently?

Main Results

Theorem

If the optimal solution uses k subsets, the greedy algorithm uses at most $k \ln n$ subsets.

1. Greedy algorithm is not too bad ☺— approximation ratio is $\ln n$
2. Can we find optimal solution efficiently?
3. PCP theorem: No poly time algorithm outputs $\leqslant 0.999 \cdot k \ln n$ subsets unless $\mathrm{P} = \mathrm{NP}$



4. Believed that $2^{n^{1-o(1)}}$-time is necessary to find the optimal

Proof

1. Let n_t be $\#$ elements still not covered after t iterations of the greedy
2. The optimal solution uses k-subsets $\Rightarrow \exists$ a subset covers n_t/k remaining elements

Proof

1. Let n_t be $\#$ elements still not covered after t iterations of the greedy
2. The optimal solution uses k-subsets $\Rightarrow \exists$ a subset covers n_t/k remaining elements
3. $n_{t+1} \leqslant n_t - n_t/k = n_t(1-1/k)$. So

$$n_t \leqslant n_0(1-1/k)^t < n_0 e^{-t/k} = n \cdot e^{-t/k},$$

   where RHS is equal to 1 for $t = k \ln n$.

Proof

1. Let n_t be $\#$ elements still not covered after t iterations of the greedy
2. The optimal solution uses k-subsets $\Rightarrow \exists$ a subset covers n_t/k remaining elements
3. $n_{t+1} \leqslant n_t - n_t/k = n_t(1 - 1/k)$. So

$$n_t \leqslant n_0(1 - 1/k)^t < n_0 e^{-t/k} = n \cdot e^{-t/k},$$

where RHS is equal to 1 for $t = k \ln n$.

Discussion

For many computational hard problems, greedy algorithms provide the best solution in poly time.

Outline

Introduction

Problem Description

Given n jobs with processing load $t_1, \ldots, t_n$, schedule them into m identical machines to minimize the *makespan* (defined as the max load over all machines)

Introduction

Problem Description

Given n jobs with processing load $t_1, \ldots, t_n$, schedule them into m identical machines to minimize the *makespan* (defined as the max load over all machines)

1. Let OPT be the optimal answer
2. It is NP-hard to find OPT even for 2 machines

Introduction

Problem Description

Given n jobs with processing load $t_1, \ldots, t_n$, schedule them into m identical machines to minimize the *makespan* (defined as the max load over all machines)

1. Let OPT be the optimal answer
2. It is NP-hard to find OPT even for 2 machines
3. Our plan: Consider a simple greedy algorithm and prove its output $\leqslant 1.5 \cdot \mathrm{OPT}$
4. How to design a greedy algorithm?

A Greedy Algorithm

procedure MINMAKESPAN($t_1, \dots, t_n, m$)
 Resort all jobs s.t. $t_1 \geqslant \cdots \geqslant t_m$
 Initialize $A_j = \emptyset$ as the load of machine $j \in [m]$
 for $i = 1, \dots, n$ **do**
  Find the least load A_j
  $A_j = A_j \cup \{t_i\}$

A Greedy Algorithm

procedure MINMAKESPAN($t_1, \dots, t_n, m$)
  Resort all jobs s.t. $t_1 \geqslant \cdots \geqslant t_m$
  Initialize $A_j = \emptyset$ as the load of machine $j \in [m]$
  for $i = 1, \dots, n$ **do**
    Find the least load A_j
    $A_j = A_j \cup \{t_i\}$

Correctness

1. Running Time: $O(n \log n + n \log m)$ by heap
2. Correctness: max-load $\leqslant 1.5 \cdot \mathrm{OPT}$

Proof of Approximation

1. OBS 1: $\mathrm{OPT} \geqslant t_1$
2. OBS 2: $\mathrm{OPT} \geqslant \sum_{i=1}^{n} t_i/m$

Proof of Approximation

1. OBS 1: $\mathrm{OPT} \geqslant t_1$
2. OBS 2: $\mathrm{OPT} \geqslant \sum_{i=1}^{n} t_i/m$
3. Fact: Solution $\leqslant \sum_{i=1}^{n} t_i/m + t_1$ is a 2-approx. How to refine it?

Proof of Approximation

1. OBS 1: $\mathrm{OPT} \geqslant t_1$
2. OBS 2: $\mathrm{OPT} \geqslant \sum_{i=1}^{n} t_i/m$
3. Fact: Solution $\leqslant \sum_{i=1}^{n} t_i/m + t_1$ is a 2-approx. How to refine it?
4. Suppose the last job t_ℓ of machine A_1 has $\ell > m$ (otherwise A_1 just has 1 job)
5. Load of A_1 is $\leqslant \sum_{i=1}^{n} t_i/m + t_\ell$

Proof of Approximation

1. OBS 1: $\mathrm{OPT} \geqslant t_1$
2. OBS 2: $\mathrm{OPT} \geqslant \sum_{i=1}^{n} t_i/m$
3. Fact: Solution $\leqslant \sum_{i=1}^{n} t_i/m + t_1$ is a 2-approx. How to refine it?
4. Suppose the last job t_ℓ of machine A_1 has $\ell > m$ (otherwise A_1 just has 1 job)
5. Load of A_1 is $\leqslant \sum_{i=1}^{n} t_i/m + t_\ell$
6. But $\mathrm{OPT} \geqslant 2t_\ell$ since $\ell > m$
7. One could further refine it to $4/3$ by assuming t_n is the last *finished* job

Questions?