6331 - Algorithms, Spring 2014, CSE, OSU
Lecture 7: Greedy algorithms

Instructor: Anastasios Sidiropoulos

Activity-selection problem

Activity-selection problem
Set of *activities* $S = \{a_1, \ldots, a_n\}$.
Activity a_i has *start time* s_i, and *finish time* f_i, where

$$0 \leq s_i < f_i$$

Activities a_i and a_j are *compatible* if

$$[s_i, f_i) \cap [s_j, f_j) = \emptyset$$

We will assume

$$f_1 \leq f_2 \leq \ldots \leq f_n$$

Goal: Fine a maximum-size set of mutually compatible activities.

Example

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

$\{a_3, a_9, a_{11}\}$ is a valid solution.

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

Structure of an optimal solution

Let S_{ij} be the set of activities that start after a_i finishes, and finish before a_j starts, i.e.

Structure of an optimal solution

Let S_{ij} be the set of activities that start after a_i finishes, and finish before a_j starts, i.e.

$$S_{ij} = \{a_r : s_r \geq f_i \text{ and } f_r < s_j\}.$$

Let A_{ij} be an optimal solution for S_{ij}.

Structure of an optimal solution

Let S_{ij} be the set of activities that start after a_i finishes, and finish before a_j starts, i.e.

$$S_{ij} = \{a_r : s_r \geq f_i \text{ and } f_r < s_j\}.$$

Let A_{ij} be an optimal solution for S_{ij}.
Suppose $a_k \in A_{ij}$.

Structure of an optimal solution

Let S_{ij} be the set of activities that start after a_i finishes, and finish before a_j starts, i.e.

$$S_{ij} = \{a_r : s_r \geq f_i \text{ and } f_r < s_j\}.$$

Let A_{ij} be an optimal solution for S_{ij}.
Suppose $a_k \in A_{ij}$. Let

$$A_{ik} = A_{ij} \cap S_{ik} \qquad A_{kj} = A_{ij} \cap S_{kj}$$

Structure of an optimal solution

Let S_{ij} be the set of activities that start after a_i finishes, and finish before a_j starts, i.e.

$$S_{ij} = \{a_r : s_r \geq f_i \text{ and } f_r < s_j\}.$$

Let A_{ij} be an optimal solution for S_{ij}.
Suppose $a_k \in A_{ij}$. Let

$$A_{ik} = A_{ij} \cap S_{ik} \qquad A_{kj} = A_{ij} \cap S_{kj}$$

Then

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

Structure of an optimal solution

Let S_{ij} be the set of activities that start after a_i finishes, and finish before a_j starts, i.e.

$$S_{ij} = \{a_r : s_r \geq f_i \text{ and } f_r < s_j\}.$$

Let A_{ij} be an optimal solution for S_{ij}.
Suppose $a_k \in A_{ij}$. Let

$$A_{ik} = A_{ij} \cap S_{ik} \qquad A_{kj} = A_{ij} \cap S_{kj}$$

Then

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

So

$$|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$$

Structure of an optimal solution

Let $c[i, j]$ be the size of an optimal solution for A_{ij}.

Structure of an optimal solution

Let $c[i, j]$ be the size of an optimal solution for A_{ij}.
Then, assuming $a_k \in A_{ij}$, we have

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

Structure of an optimal solution

Let $c[i,j]$ be the size of an optimal solution for A_{ij}.
Then, assuming $a_k \in A_{ij}$, we have

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

So,

$$c[i,j] = \begin{cases} 0 & \text{, if } S_{ij} \neq \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{, if } S_{ij} \neq \emptyset \end{cases}$$

Structure of an optimal solution

Let $c[i,j]$ be the size of an optimal solution for A_{ij}.
Then, assuming $a_k \in A_{ij}$, we have

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

So,

$$c[i,j] = \begin{cases} 0 & \text{, if } S_{ij} \neq \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{, if } S_{ij} \neq \emptyset \end{cases}$$

This can be used to obtain a recursive algorithm.

Structure of an optimal solution

Let $c[i,j]$ be the size of an optimal solution for A_{ij}.
Then, assuming $a_k \in A_{ij}$, we have

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

So,

$$c[i,j] = \begin{cases} 0 & \text{, if } S_{ij} \neq \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{, if } S_{ij} \neq \emptyset \end{cases}$$

This can be used to obtain a recursive algorithm.
Also, a dynamic programming algorithm.

Structure of an optimal solution

Let $c[i, j]$ be the size of an optimal solution for A_{ij}.
Then, assuming $a_k \in A_{ij}$, we have

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

So,

$$c[i, j] = \begin{cases} 0 & \text{, if } S_{ij} \neq \emptyset \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{, if } S_{ij} \neq \emptyset \end{cases}$$

This can be used to obtain a recursive algorithm.
Also, a dynamic programming algorithm.
There is a simpler approach.

The greedy approach

Lemma

Let $S_k \neq \emptyset$ be a subproblem. Let a_m be an activity in S_k with earliest finish time. Then, a_m is included in some optimal solution for S_k.

The greedy approach

Lemma

Let $S_k \neq \emptyset$ be a subproblem. Let a_m be an activity in S_k with earliest finish time. Then, a_m is included in some optimal solution for S_k.

Why?

A recursive greedy algorithm

Recursive-Activity-Selector(s, f, k, n)
$m = k + 1$
while $m \leq n$ and $s[m] < f[k]$
    $m = m + 1$
if $m \leq n$
    return $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, n)
else return $\emptyset$

Initial call: **Recursive-Activity-Selector**$(s, f, 0, n)$

A recursive greedy algorithm

Recursive-Activity-Selector(s, f, k, n)
$m = k + 1$
while $m \leq n$ and $s[m] < f[k]$
    $m = m + 1$
if $m \leq n$
    return $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, n)
else return $\emptyset$

Initial call: **Recursive-Activity-Selector**$(s, f, 0, n)$

Why does this work?

An iterative greedy algorithm

Greedy-Activity-Selector(s, f)
$A = \{a_1\}$
$k = 1$
for $m = 2$ to n
    if $s[m] \geq f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return A

An iterative greedy algorithm

Greedy-Activity-Selector(s, f)
$A = \{a_1\}$
$k = 1$
for $m = 2$ to n
    if $s[m] \geq f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return A

Why does this work?

An iterative greedy algorithm

Greedy-Activity-Selector(s, f)
$A = \{a_1\}$
$k = 1$
for $m = 2$ to n
    if $s[m] \geq f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return A

Why does this work?

Running time?

An iterative greedy algorithm

Greedy-Activity-Selector(s, f)
$A = \{a_1\}$
$k = 1$
for $m = 2$ to n
    if $s[m] \geq f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return A

Why does this work?

Running time?

What would be the running time of the dynamic programming approach?

Huffman codes

Suppose we want to construct a binary code for representing letters of the alphabet.

	a	b	c	d	e	f
Frequency/occurences	0.45	0.13	0.12	0.16	0.09	0.05
Fixed-length code-word	000	001	010	011	100	101
Variable-length code-word	0	101	100	111	1101	1100

Fixed-length code-word: 3 bits per letter.

Variable-length code-word: 2.24 bits per letter.

Prefix codes

A code is called a *prefix code* if no codeword is the prefix of any other codeword.

Prefix codes

A code is called a *prefix code* if no codeword is the prefix of any other codeword.

A prefix code can be represented by a binary tree.

- Every internal node has two children; one with a 0-labeled edges, and one with a 1-labeled edge.
- Every codeword corresponds to a root-to-leaf path.

Prefix codes

A code is called a *prefix code* if no codeword is the prefix of any other codeword.

A prefix code can be represented by a binary tree.

- Every internal node has two children; one with a 0-labeled edges, and one with a 1-labeled edge.
- Every codeword corresponds to a root-to-leaf path.

Example of a prefix code represented as a binary tree. . .

Prefix codes

A code is called a *prefix code* if no codeword is the prefix of any other codeword.

A prefix code can be represented by a binary tree.

- Every internal node has two children; one with a 0-labeled edges, and one with a 1-labeled edge.
- Every codeword corresponds to a root-to-leaf path.

Example of a prefix code represented as a binary tree...

There is always a prefix code with optimum compression rate.

A greedy algorithm for constructing a prefix code

```
Huffman(C)
n = |C|
Q = Build-Min-Heap(C)
for i = 1 to n − 1
    create 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 root
```

A greedy algorithm for constructing a prefix code

Huffman(C)
$n = |C|$
$Q =$ Build-Min-Heap(C)
for $i = 1$ to $n - 1$
    create 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 root

Example execution...

Correctness

Lemma

Let x, y be characters in C with minimum frequency. Then, there exists an optimal prefix code for C where the codewords for x and y have the same length, and differ only in the last bit.

Correctness

Lemma

Let x, y be characters in C with minimum frequency. Then, there exists an optimal prefix code for C where the codewords for x and y have the same length, and differ only in the last bit.

Proof sketch.

Find a pair of leaves a, b that are siblings, and have maximum depth.
Exchanging $\{a, b\}$ with $\{x, y\}$ gives a code of no greater cost. $\square$

Correctness

Lemma

Let x, y be characters in C with minimum frequency. Let

$$C' = C \setminus \{x, y\} \cup \{z\},$$

with $z.freq = x.freq + y.freq$.
Let T' be the optimal tree for C'.
Let T be the tree obtained from T' by replacing the leaf representing z by an internal node with children x and y.
Then, T is an optimal tree for C.

Correctness

Lemma

Let x, y be characters in C with minimum frequency. Let

$$C' = C \setminus \{x, y\} \cup \{z\},$$

with $z.freq = x.freq + y.freq$.
Let T' be the optimal tree for C'.
Let T be the tree obtained from T' by replacing the leaf representing z by an internal node with children x and y.
Then, T is an optimal tree for C.

Proof sketch.

If T is not optimal for C, then we can construct a tree T'' for C' with smaller cost than T', which is a contradiction. $\square$

Corollary

Huffman *outputs an optimal code.*