

Week 06: Tasks

FanFly

April 12, 2020

The Task Selction Problem

Today we are going to introduce another combinatorial optimzation problem, which is called the **task selection problem**.

The Task Selection Problem

Today we are going to introduce another combinatorial optimization problem, which is called the **task selection problem**.

Task Selection Problem

- Input: A collection of n tasks, each with a start time x_i and an end time y_i .
- Output: A maximum size subset of mutually compatible tasks.

The Task Selection Problem

Today we are going to introduce another combinatorial optimization problem, which is called the **task selection problem**.

Task Selection Problem

- Input: A collection of n tasks, each with a start time x_i and an end time y_i .
- Output: A maximum size subset of mutually compatible tasks.

We will use $T_i = [x_i, y_i)$ to denote the i th task.

The Task Selection Problem

Today we are going to introduce another combinatorial optimization problem, which is called the **task selection problem**.

Task Selection Problem

- Input: A collection of n tasks, each with a start time x_i and an end time y_i .
- Output: A maximum size subset of mutually compatible tasks.

We will use $T_i = [x_i, y_i)$ to denote the i th task.

Furthermore, two tasks T_i and T_j are compatible if

$$[x_i, y_i) \cap [x_j, y_j) = \emptyset.$$

Compatible Tasks

Given a set of tasks $\{T_1, \dots, T_n\}$, how can we check if they are mutually compatible?

Compatible Tasks

Given a set of tasks $\{T_1, \dots, T_n\}$, how can we check if they are mutually compatible?

- Check if all pairs of tasks are disjoint. It takes $\Theta(n^2)$ time.

Compatible Tasks

Given a set of tasks $\{T_1, \dots, T_n\}$, how can we check if they are mutually compatible?

- Check if all pairs of tasks are disjoint. It takes $\Theta(n^2)$ time.
- Sort the tasks based on start time, and then check if all consecutive tasks are disjoint. It takes $\Theta(n)$ time to check the tasks.

Compatible Tasks

Given a set of tasks $\{T_1, \dots, T_n\}$, how can we check if they are mutually compatible?

- Check if all pairs of tasks are disjoint. It takes $\Theta(n^2)$ time.
- Sort the tasks based on start time, and then check if all consecutive tasks are disjoint. It takes $\Theta(n)$ time to check the tasks.

Sorting saves lots of time!

Compatible Tasks

Given a set of tasks $\{T_1, \dots, T_n\}$, how can we check if they are mutually compatible?

- Check if all pairs of tasks are disjoint. It takes $\Theta(n^2)$ time.
- Sort the tasks based on start time, and then check if all consecutive tasks are disjoint. It takes $\Theta(n)$ time to check the tasks.

Sorting saves lots of time!

Thus, it must be a good idea to sort the tasks first.

A Brute-force Approach

Now supposed that the tasks are sorted based on start time.

A Brute-force Approach

Now supposed that the tasks are sorted based on start time.

- For each possible subset of tasks, check if it contains tasks that are not disjoint.

A Brute-force Approach

Now supposed that the tasks are sorted based on start time.

- For each possible subset of tasks, check if it contains tasks that are not disjoint.
- There are $\Theta(2^n)$ possible subsets.

A Brute-force Approach

Now supposed that the tasks are sorted based on start time.

- For each possible subset of tasks, check if it contains tasks that are not disjoint.
- There are $\Theta(2^n)$ possible subsets.
- It takes $O(n)$ time for each subset.

A Brute-force Approach

Now supposed that the tasks are sorted based on start time.

- For each possible subset of tasks, check if it contains tasks that are not disjoint.
- There are $\Theta(2^n)$ possible subsets.
- It takes $O(n)$ time for each subset.
- Thus, we can solve the problem in $O(2^n n)$ time.

A Brute-force Approach

By the way, how to enumerate all subsets of a set?

A Brute-force Approach

By the way, how to enumerate all subsets of a set?

```
def subsets(n):  
    """Returns all subsets of [0, 1, ..., n-1]."""  
    subsets = []  
    for m in range(2 ** n):  
        subset = []  
        for i in range(n):  
            if (m // (2 ** i)) % 2 == 1:  
                subset.append(i)  
        subsets.append(subset)  
    return subsets
```

A Brute-force Approach

By the way, how to enumerate all subsets of a set?

```
def subsets(n):  
    """Returns all subsets of [0, 1, ..., n-1]."""  
    subsets = []  
    for m in range(2 ** n):  
        subset = []  
        for i in range(n):  
            if (m // (2 ** i)) % 2 == 1:  
                subset.append(i)  
        subsets.append(subset)  
    return subsets
```

```
>>> subsets(3)
```

A Brute-force Approach

By the way, how to enumerate all subsets of a set?

```
def subsets(n):  
    """Returns all subsets of [0, 1, ..., n-1]."""  
    subsets = []  
    for m in range(2 ** n):  
        subset = []  
        for i in range(n):  
            if (m // (2 ** i)) % 2 == 1:  
                subset.append(i)  
        subsets.append(subset)  
    return subsets
```

```
>>> subsets(3)  
[[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2]]
```

A Simple Observation

The following theorem is a simple result, but it can help us develop a faster algorithm.

A Simple Observation

The following theorem is a simple result, but it can help us develop a faster algorithm.

Theorem

Let $S = \{T_1, \dots, T_n\}$ be a collection of tasks.

A Simple Observation

The following theorem is a simple result, but it can help us develop a faster algorithm.

Theorem

Let $S = \{T_1, \dots, T_n\}$ be a collection of tasks.

- If $T_i \subseteq T_j$, then there must be a maximum-size subset of mutually compatible tasks that does not contain T_j .

A Simple Observation

The following theorem is a simple result, but it can help us develop a faster algorithm.

Theorem

Let $S = \{T_1, \dots, T_n\}$ be a collection of tasks.

- If $T_i \subseteq T_j$, then there must be a maximum-size subset of mutually compatible tasks that does not contain T_j .
- The task with earliest finish time must be included in some maximum-size subset of mutually compatible tasks.

Greedy Algorithm

This theorem leads us to find the following algorithm.

Greedy Algorithm

This theorem leads us to find the following algorithm.

For each step, let $S = \{T_1, \dots, T_n\}$ be the set of tasks that we are currently dealing with, where T_1, \dots, T_n are sorted based on start time.

Greedy Algorithm

This theorem leads us to find the following algorithm.

For each step, let $S = \{T_1, \dots, T_n\}$ be the set of tasks that we are currently dealing with, where T_1, \dots, T_n are sorted based on start time.

- Note that $T_1 = [x_1, y_1)$ is the task with earliest start time.

Greedy Algorithm

This theorem leads us to find the following algorithm.

For each step, let $S = \{T_1, \dots, T_n\}$ be the set of tasks that we are currently dealing with, where T_1, \dots, T_n are sorted based on start time.

- Note that $T_1 = [x_1, y_1)$ is the task with earliest start time.
- If $y_1 \geq y_2$, then we discard T_1 .

Greedy Algorithm

This theorem leads us to find the following algorithm.

For each step, let $S = \{T_1, \dots, T_n\}$ be the set of tasks that we are currently dealing with, where T_1, \dots, T_n are sorted based on start time.

- Note that $T_1 = [x_1, y_1)$ is the task with earliest start time.
- If $y_1 \geq y_2$, then we discard T_1 .
- If $y_1 < y_2$, then we choose T_1 and discard the tasks overlapping with T_1 .

Greedy Algorithm

This theorem leads us to find the following algorithm.

For each step, let $S = \{T_1, \dots, T_n\}$ be the set of tasks that we are currently dealing with, where T_1, \dots, T_n are sorted based on start time.

- Note that $T_1 = [x_1, y_1)$ is the task with earliest start time.
- If $y_1 \geq y_2$, then we discard T_1 .
- If $y_1 < y_2$, then we choose T_1 and discard the tasks overlapping with T_1 .

Since we always make the choice that looks best at the moment, the algorithm is usually called a **greedy algorithm**.

Exercise #1

Task Selection Problem

- Input: A collection of n tasks, each with a start time x_i and an end time y_i .
- Output: A maximum size subset of mutually compatible tasks.

What is the overall time complexity of the greedy algorithm that solves the task selection problem?

- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $\Theta(2^n)$

Exercise #1

Task Selection Problem

- Input: A collection of n tasks, each with a start time x_i and an end time y_i .
- Output: A maximum size subset of mutually compatible tasks.

What is the overall time complexity of the greedy algorithm that solves the task selection problem?

- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $\Theta(2^n)$

Solution

It takes $\Theta(n \log n)$ time to sort the tasks based on start time. Then it takes $\Theta(n)$ to choose the compatible tasks. Thus, the overall time complexity is $\Theta(n \log n)$.

Exercise #2

Consider the following code, which is slightly different from the previous version.

```
def subsets(n):  
    """Returns all subsets of [0, 1, ..., n-1]."""  
    subsets = {} # use dict instead of list  
    for m in range(2 ** n):  
        subset = []  
        for i in range(n):  
            if (m // (2 ** i)) % 2 == 1:  
                subset.append(i)  
        subsets[m] = subset  
    return subsets
```


Exercise #2

Consider the following code, which is slightly different from the previous version.

```
def subsets(n):  
    """Returns all subsets of [0, 1, ..., n-1]."""  
    subsets = {} # use dict instead of list  
    for m in range(2 ** n):  
        subset = []  
        for i in range(n):  
            if (m // (2 ** i)) % 2 == 1:  
                subset.append(i)  
        subsets[m] = subset  
    return subsets
```

Then we have the following result.

```
>>> subsets(2)  
{0: [], 1: [0], 2: [1], 3: [0, 1]}
```

Exercise #2 (cont.)

Now we assume that the following Python code is executed.

```
n = 5  
P = subsets(n)
```

Please find the value of $P[18]$.

Exercise #2 (cont.)

Now we assume that the following Python code is executed.

```
n = 5  
P = subsets(n)
```

Please find the value of $P[18]$.

Solution

Since $18 = 2^1 + 2^4$, the value of $P[18]$ should be $[1, 4]$.
(That is, 18 can be represented as 10010 in binary.)

Exercise #3

Consider the task selection problem for the following set of tasks.

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
Start	0	1	2	3	3	5	5	6	8	8	12
End	6	4	14	5	9	7	9	10	11	12	16

What is the maximum size of subset of mutually compatible tasks?

Exercise #3

Consider the task selection problem for the following set of tasks.

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
Start	0	1	2	3	3	5	5	6	8	8	12
End	6	4	14	5	9	7	9	10	11	12	16

What is the maximum size of subset of mutually compatible tasks?

Solution

There can be at most 4 mutually compatible tasks, e.g., $\{T_2, T_6, T_9, T_{11}\}$.