Greedy Algorithms

We consider optimisation problems. Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.

> A *greedy algorithm* is a process that always makes the choice that looks best at the moment.

Greedy algorithms are natural, and in few cases solve optimally the given problem.

We will look at one simple example and we will try to understand why does it work.

We will consider more examples later.

Optimization Problem

An *optimisation problem* P is defined by four components $(\mathcal{I}, \mathcal{SOL}, c, opt)$ where:

(1) $\mathcal{I}$ is the set of the *instances* of P. Membership in $\mathcal{I}$ is decidable in polynomial time.

(2) For each $x \in \mathcal{I}$, $\mathcal{SOL}(x)$ is the set of *solutions* of x. Membership in $\mathcal{SOL}(x)$ is decidable in polynomial time and for each $y \in \mathcal{SOL}(x)$, the length of y is polynomial in the length of x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{SOL}(x)$, $c(x, y)$ is an integer, non-negative function, called the *objective* (or *cost*) function.

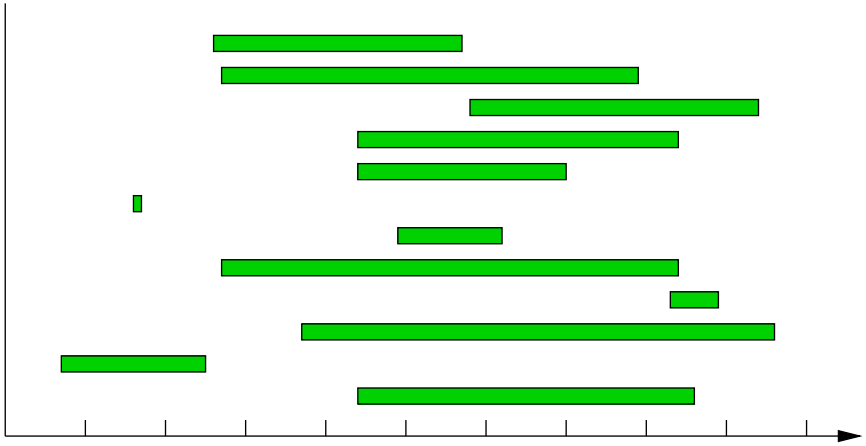
(4) $opt \in \{\max, \min\}$ is the *optimisation criterion* and tells if the problem P is a maximisation or a minimisation problem.

Activity-selection problem

We are given a set of proposed activities $\mathcal{S} = \{A_1, A_2, \ldots, A_n\}$ that wish to use a resource, which can be used by only one activity at a time. Each activity is defined by a pair consisting of a *start time* s_i and a *finish time* f_i, with $0 \leq s_i < f_i < +\infty$. If selected, activity A_i takes place during the time interval $[s_i, f_i)$. Two activities A_i and A_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$. The *activity-selection problem* is to select the maximum number of mutually compatible activities.

Example

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	44	7	37	83	27	49	16	44	44	58	27	26
f_i	86	25	96	89	84	62	17	70	84	94	79	57



Two possible solutions (the columns marked by a "*" correspond to activities picked in the particular solution).

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	44	7	37	83	27	49	16	44	44	58	27	26
f_i	86	25	96	89	84	62	17	70	84	94	79	57
	*	*										

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	44	7	37	83	27	49	16	44	44	58	27	26
f_i	86	25	96	89	84	62	17	70	84	94	79	57
		*		*							*	

Open issues

How far can we go?

Scale this up to a real life situation, where there may be few hundreds activities. We write some software that, when run on a particular example, returns 50 activities: is this the best?? The people who employ us may wonder whether it is possible to do better?

Unprofessional answer: Yep!

Better answer: My program is "certified" to produce the best possible answer (I can show you the program AND a mathematical proof that it works)!

Example of Real Life Application "Time-dependent web browsing"

The access speeds to particular sites on the World Wide Web can vary depending on the time of access.

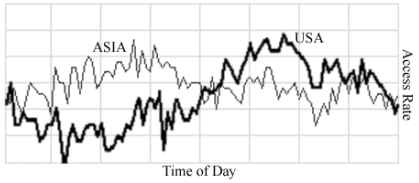


Figure 1: Access rates for USA and Asia over 24 hours. (Derived from data posted by the Anderson News Network `http://www.internettrafficreport.com/`)

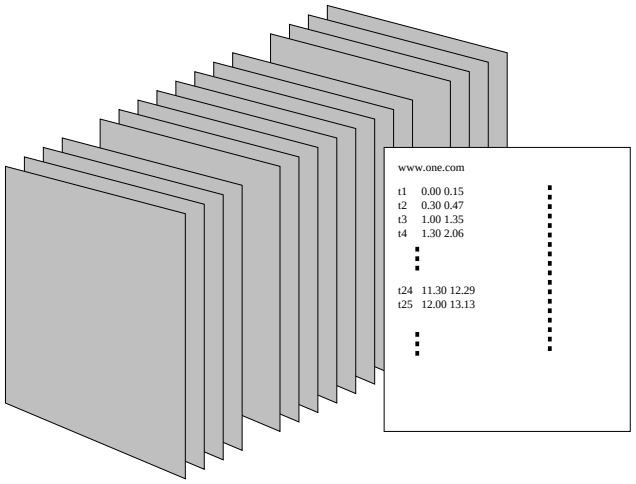
A single central computer (e.g. a search engine server) collects all the information stored in a certain number of web documents, located at various sites.

The information is gathered by scheduling a number of consecutive client/server TCP connections with the required web sites.

We assume that the loading time of any particular page from any site may be different at different times, e.g. the access to the page is much slower in peak hours than in off-peak hours.

Having a list of pages to be collected along with some information about the access time at any given instant, the goal is to download as many pages as possible.

Typical input for this problem can be a sequence of tables like the following (times are GMT), one for each remote web site



It is an optimisation problem.

The set of instances coincides with the set of all possible groups of activities (in tabular form).

A solution is a set of compatible activities.

The cost of a solution is its size (or *cardinality*), and we are seeking a maximum cardinality solution.

How do we solve it? What is a good strategy?

Greedy algorithm

GREEDY-ACTIVITY-SELECTOR ($\mathcal{S}$, n)
- $j \leftarrow$ index of the minimum f_i in $\mathcal{S}$
- $X \leftarrow \{A_j\}$
- $\mathcal{S} \leftarrow \mathcal{S} \setminus \{A_j\}$
- **for** $i \leftarrow 2$ **to** n
  - $k \leftarrow$ index of the minimum f_i in $\mathcal{S}$
  - **if** $s_k \geq f_j$
    - $X \leftarrow X \cup \{A_k\}$
    - $j \leftarrow k$
  - $\mathcal{S} \leftarrow \mathcal{S} \setminus \{A_k\}$
- **return** X

Remarks.

The algorithm takes as input a set of activities $\mathcal{S}$ (each activity stored as a pair of numbers (s_i, f_i)) and the total number of activities n.

The set X collects the selected activities. The variable j always specifies the most recent addition to X.

Notice also that the pseudo-code is using "sets" but any implementation of GREEDY-ACTIVITY-SELECTOR will have to use arrays, vectors or some other data structure present in the chosen programming language.

Examples

(1) Back to the first example we considered.

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	44	7	37	83	27	49	16	44	44	58	27	26
f_i	86	25	96	89	84	62	17	70	84	94	79	57

What are the algorithm choices? Let's simulate it!

(2) Time-dependent web browsing

site	1	2	3	4	5	6	7	8	9	10	11	12
s_i	9.15	7.42	10.00	11.54	9.17	9.47	7.34	8.16	8.36	10.45	11.53	11.05
f_i	9.35	7.49	11.34	12.52	9.57	10.19	8.51	9.23	9.25	10.56	12.30	12.16

Problem? You are not guaranteed to get information from all of them. This requirement can make the problem quite difficult (more later).

Time complexity

Claim. The time complexity of GREEDY-ACTIVITY-SELECTOR is $O(n^2)$ (assuming that set update operations only cost one step each!).

- Each iteration of the **for** loop lasts for at most $|\mathcal{S}| + 3$ steps.
- The size of $\mathcal{S}$ decreases by one at each iterations (initially $|\mathcal{S}| = n$).
- The total number of steps is therefore

$$\underbrace{(n+2) + ((n-1)+3) + ((n-2)+3) + \ldots + (1+3)}_{n \text{ times}}$$

which is at most $n \times ((n+3)+3) = n^2 + 6n$ (or, exactly, $\frac{n^2+7n}{2} - 1$, if we a just a bit more careful with the arithmetic).

The run time can be reduced to $O(n \log n)$ by an algorithm that combines a linear time greedy heuristic with a sort preprocessing step.

```
SGREEDY-ACTIVITY-SELECTOR (S, n)
    sort S in order of increasing finishing time
    j ← 1
    X ← {A_j}
    for i ← 2 to n
        if s_i ≥ f_j
            X ← X ∪ {A_i}
            j ← i
    return X
```

Here is the same example, after the execution of the first step of SGREEDY-ACTIVITY-SELECTOR.

i	9	2	12	6	7	11	5	8	1	4	10	3
s_i	16	7	26	49	44	27	27	44	44	83	58	37
f_i	17	25	57	62	70	79	84	84	86	89	94	96

The algorithm eventually selects activities 9, 12, and 4.