Online Algorithms	2/11/2011

Lecture Notes 1: Introduction

Professor: Yossi Azar *Scribe:Shahar Yifrah*

1 Introduction

In the 50's and 60's, exact algorithms were the main concern.
An optimization problem: given problem input I, find optimal solution $A(I)$ such that its value $A(I) = OPT(I)$. Note that we abuse notation and use $A(I)$ both for the solution as well its value.
Usually the goal is to minimize the **time** (as well as space) complexity of the algorithm.
An *efficient* algorithm has typically polynomial time complexity. Specifically its running time is $O(|I|^c)$ for some constant, c.

Examples:

- Minimum Spanning Tree (MST)
- Shortest Paths
- Flow Problems

At the end of the 60's, it was discovered that some families of problems cannot be solved within these requirements.
These are the NP and NP-Complete problems.
E.g., the traveling salesman problem (TSP).

For $\alpha \geq 1$ define the following:

Definition 1. *Algorithm A (for a minimization problem) is an* α**-approximation** *if* $\forall I$: $A(I) \leq \alpha \cdot OPT(I)$
(The time complexity is usually polynomial.)

The goal is to achieve small α and thus the approximation is considered better. Note that for maximization problem, the relation is inverted, so $A(I) \geq \alpha \cdot OPT(I)$ for some $\alpha \leq 1$.

Then came another category - Distributed Algorithms. In distributed algorithms not all information is given at the same location.

Then came along a newer one: **Online Algorithms**. Here the input is revealed over time i.e., not all the information is given beforehand.

There are other types of algorithms: dynamic algorithm, streaming algorithms, sub-linear algorithms, incentive compatible algorithms, quantum algorithms, machine learning algorithms.

2 Examples

2.1 Ski Rental

At any time we can

- Buy for $M
- Rent for another day for $1

Although we have to decide each day, we shall look at it as a Continuous process. This problem is analogous to the **context switch problem**, as we have to decide at every point if to test for I/O or block until it occurs. We don't know beforehand for how long we need the resource.
The **offline version** is trivial: "If we need it for more than M time, buy".
To elaborate:
Say we need it for t time-units.
If $t < M$, rent, and $OPT = t$.
If $t \geq M$, buy, and $OPT = M$.

2.2 Path Search

We are on the beach at night, with a flash-light. we can walk in one of two directions along the shore, in order to find a treasure. Only once we are over the treasure, we see it and win. The goal is to reach the treasure in a minimal length path, and the solution is the path description itself.
The offline version is trivial: "walk in the right direction up-to the treasure".

2.3 Paging

Lets assume two levels of memory:

- Main memory, which is made of k pages. Transfer rate: 300 million pages/second.
- Hard disk - external memory, made of infinite number of pages. Transfer rate: 600 pages/second (million times slower than main)

The input is a series of requests, e.g., 7,3,1,2,3,7,4,... Whenever a page is brought, an existing one is thrown out of the main memory. The goal is to minimize the number of page-faults.

In the **offline version** we know the request sequence in advance. The solution: "throw the page we will need in the most distant future". In the online version, operating systems use the LRU (Least Recently Used) algorithm, which is a "time-wise" reflection of the offline solution. That is, the past and future are interchanged.

2.4 Scheduling

We have m identical machines, and a series of tasks with specified load, $w_1, w_2, w_3, ...$ As a task arrives, we have to assign it to a machine. So we have an assignment algorithm, A(i) = j, where i is the task index and j is the machine index.
The **load of a machine** is the sum of the tasks loads: $l_j = \sum_{i|A(i)=j} w_i$
And the general load indicator is: $\max_j l_j$
The **goal** is to minimize this indicator.

A standard solution is **Round Robin** algorithm, where every incoming task is given to the next machine in a circular fashion.

The **offline version**, even for m=2, is equivalent to the knapsack (or partition) problem, which is NP-Complete (in a "week" manner for 2 machines, and "strong" for over 2 machines). For any m the problem can be solved approximately within $1+\epsilon$ for each $\epsilon > 0$, and the time would be $n^{(1/\epsilon)^2}$, i.e., PTAS (a term from complexity theory.)

How do we asses online algorithms?

Ski Rental: If we want to minimize the maximum payment, we buy immediately.
Paging: A page-fault is possible any minute for all algorithms - it's easy to give the worst-case scenario.
Options for assessment:

- Assume some distribution of the input, then use the **expectation** for evaluating the cost. If this is possible, it's not such a bad idea.
- Test algorithm for probable inputs. E.g, for Paging, test on **historical traces**. But history based forecast isn't perfect, and some singular events may occur.
- Approximation ratio - Competitive ratio:

$$\sup_I \frac{ALG(\sigma)}{OPT(\sigma)}$$

  As used in approximation algorithm, only here the input isn't given in advance but is a series, σ. OPT is the algorithm that knows all this series beforehand.

Definition: Algorithm ALG (for minimization problem) is C-Competitive if

$$\forall \sigma : ALG(\sigma) \leq C \cdot OPT(\sigma)$$

Where the offline algorithm OPT knows all the input in advance.

3 Analysis

3.1 Ski Rental

ALG = time of buy
ON = online algorithm cost
OPT = optimal algorithm cost

ALG	ON	OPT	Competitive Ratio
0	M	min(t,M)	∞
M/2	$\begin{cases} t & \text{if } t < M/2 \\ (3/2)M & \text{otherwise} \end{cases}$	min(t,M)	3 $(= \frac{(3/2)M}{M/2})$
M	$\begin{cases} t & \text{if } t < M \\ 2M & \text{otherwise} \end{cases}$	min(t,M)	2 $(= \frac{2M}{M})$
∞	t	min(t,M)	∞ (as $t \to \infty$)
2M	$\begin{cases} t & \text{if } t < 2M \\ 3M & \text{otherwise} \end{cases}$	min(t,M)	3
7	$\begin{cases} t & \text{if } t < 7 \\ 7+M & \text{if } t \geq 7 \end{cases}$	min(t,M)	$\frac{7+M}{7} \to M$

Note: The latter algorithm is not a homogenous case, so we can dismiss it.

Theorem 3.1. *Algorithm that buys after time M is 2-Competitive.*

Proof. Let t denote the time in the series.
If $t < M$ then $ON(\sigma) = OPT(\sigma) = t$
Otherwise, $t \geq M$, so $ON(\sigma) = 2M$ and $OPT(\sigma) = M$ and $ON(\sigma) \leq 2OPT(\sigma)$ □

Theorem 3.2. *Every (deterministic) online algorithm for the Ski-Rental problem is at least 2-competitive. (Note that 2-competitive is also 5-competitive.)*

Proof. For any given algorithm, we shall come-up with a series that it fails for. Lets take any (deterministic) algorithm, "buy after T". For $T = \infty$, the comp. ratio is surely larger than any constant, in particular from 2. (We can take a series of length, say, 4M) Otherwise, we shall take a series σ of length T (specifically chosen for this algorithm.) $ON(\sigma) = T + M$ and $OPT(\sigma) = min(T, M)$ So for every T:

$$\frac{ON(\sigma)}{OPT(\sigma)} = \frac{T+M}{min(T,M)} = \frac{T}{min(T,M)} + \frac{M}{min(T,M)} \geq 1 + 1 = 2$$

□

3.2 Path Search

Every finite move we make will be non-competitive since the treasure may be 10^7 times closer on the opposite direction.
So we have two options:

- Assume that $OPT \geq 1$
- Introduce an additive constant, β. More formally, There is a constant β such that $\forall\sigma$:

$$ALG(\sigma) \leq C \cdot OPT(\sigma) + \beta$$

And we pick the second option:

Definition 2. *Algorithm is C-Competitive if there exists such a "global constant", β, for which:*
$ON(\sigma) \leq C \cdot OPT(\sigma) + \beta$

Note: If the first move is finite there is no competitive algorithm using the original definition.

Solution 1: BFS-Like

Use the following moves: $(+1, -1, +2, -2, +3, -3, ...)$
If the treasure is in location t, the cost is:
$ON = 4(1 + 2 + 3 + ... + t - 1) + t$
and for location (-t):
$ON = 4(1 + 2 + 3 + ... + t - 1) + 3t$
Thus, $ON = O(t^2)$ and $OPT = O(t) \Rightarrow t = \infty$

Solution 2: Geometric Walk

The moves: $(+1, -1, +2, -2, +4, -4, ..., +2^i, -2^i, ...)$
And the cost: $ON = 4(1 + 2 + ... + 2^k) + 2 \cdot 2^{k+1} + 2^k + \epsilon$
(since we stop when we find the treasure)
Sum of this geometric series: $4 \cdot 2^{k+1} - 4 + 5 \cdot 2^k + \epsilon = 13 \cdot 2^k - 4 + \epsilon$
$OPT = 2^k + \epsilon \Rightarrow$ Competitive ratio is 13.

Solution 3,4,...

$$(+1, -2, +4, -8, +16, ...)$$
$$(+q^0, -q^0, +q^1, -q^1, ...)$$
$$(+q^0, -q^1, +q^2, -q^3, ...)$$

And in general:

$$+x_0, -x_1, +x_2, -x_3, ...$$
$$x_0 < x_2 < x_4 < ...$$
$$x_1 < x_3 < x_5 < ...$$

If the treasure is in the positive direction, the cost:

$$2(x_0 + x_1 + x_2 + ... + x_{2k} + x_{2k+1}) + x_{2k} + \epsilon$$

letting $n = 2k$

$$= 2(x_0 + ... + x_{n+1}) + x_n + \epsilon$$

and for the negative direction ($n = 2k + 1$):

$$\begin{aligned} &2(x_0 + x_1 + x_2 + ... + x_{2k+1} + x_{2k+2}) + x_{2k+1} + \epsilon \\ =&2(x_0 + ... + x_{n+1}) + x_n + \epsilon \end{aligned}$$

Thus, the same for both directions.
(Note that we assume that the treasure is after x_0)

The competitive ratio:

$$\frac{ON}{OPT} = \frac{2(x_0 + ... + x_{n+1}) + x_n + \epsilon}{x_n + \epsilon}$$

The nominator is larger, thus the supremum is obtained where ϵ is minimal.
For $s_n = x_0 + ... + x_n$:

$$\sup_{\epsilon,n} \frac{2s_{n+1} + x_n + \epsilon}{x_n + \epsilon} = \sup_n \frac{2s_{n+1} + x_n}{x_n} = \sup_n \frac{2s_{n+1}}{x_n} + 1$$

Given a strategy, the competitive ratio is $2a + 1$ where:

$$a = \sup_n \frac{s_{n+1}}{x_n}$$

for $x_i = q^i$:

$$a = \sup_n \frac{q^{n+2} - 1}{(q-1)q^n} = \sup_n \left(\frac{q^2}{q-1} - \frac{1}{(q-1)q^n}\right) = \frac{q^2}{q-1}$$

Where in the last step the quotient with n drops since it is negligible for a large n.
Now we derive in order to find the minimum:

$$\begin{aligned} t &= \frac{q^2}{q-1} \\ t' &= \frac{2q(q-1) - q^2}{(q-1)^2} = 0 \\ q &= 2 \\ a &= \frac{2^2}{2-1} = 4 \end{aligned}$$

Hence the competitive ratio is $2a + 1 = 9$.

Theorem 3.3. *For any positive series, x_i:*

$$\forall \epsilon > 0 : sup \frac{s_{n+1}}{x_n} \geq 4 - \epsilon$$

Proof. We assume by contradiction that we get a ratio a, with $a < 4$.
Define $y_n = s_n / s_{n-1}$ and we get:

$$\begin{aligned} a \geq \frac{s_{n+1}}{x_n} &= \\ &= \frac{s_{n+1}}{s_n - s_{n-1}} \\ &= \frac{s_{n+1}/s_n}{1 - s_{n-1}/s_n} \\ &= \frac{y_{n+1}}{1 - 1/y_n} \end{aligned}$$

Lemma: $\forall \beta > 0$: $1 - 1/\beta \leq \beta/4$.
The proof is trivial: $4\beta - 4 \leq \beta^2 \Rightarrow 0 \leq (\beta - 2)^2$).
Using this lemma we get

$$y_{n+1} \leq a(1 - 1/y_n) \leq \frac{a}{4} y_n$$

We know that $\frac{a}{4} \leq 1 - \epsilon$, so for $n \to \infty$ the sandwich theorem implies that $y_n \to 0$.
Hence $y_n < 1$ starting at some n.
But $y_n = s_n / s_{n-1} > 1$ and we get a contradiction.
Hence, we cannot obtain a competitive ratio under 4. □

Scheduling

We are seeking to minimize the maximum load $l_i = \sum_{i|A(i)=j} w_i$ over all machines.

A greedy algorithm: "Always assign a task to the least loaded machine."

Theorem 3.4. *The greedy algorithm for assigning tasks to identical machines is 2-competitive.*

Proof. We only provide intuitive illustrated proof.
The following drawing of "horizon" represents the current load (y axis) for each machine (x axis).
The colored task is the task assigned most recently.

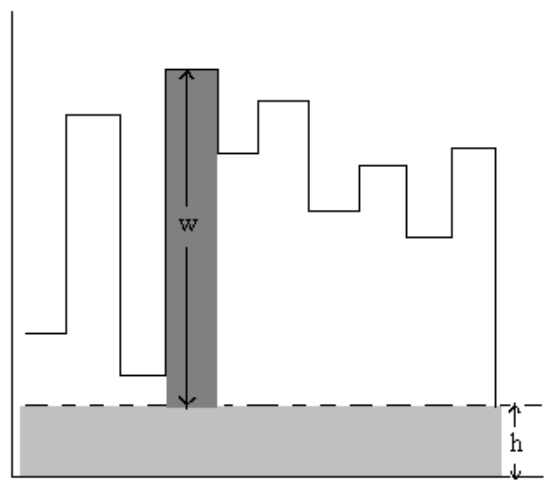

Figure 1: Load Horizon Line

Basic observation tells us that $OPT \geq w$ and $OPT \geq h$ and thus

$$ON = h + 2 \leq OPT + OPT = 2$$

□