Resource Augmentation for Buffer Management with Bounded Delay

Jan Jeżabek

Theoretical Computer Science Department
Jagiellonian University, Cracow

June 5th 2009

Introduction

We study a problem known as *packet switching*, *buffer management with bounded delay*:

- Input: non-empty set of jobs with:
  - release time, deadline (integers)
  - weight (also called value)
- Execution of any job takes one unit of time
- Jobs must be executed one at a time
- Goal: to maximize the total weight of executed jobs

Introduction

We study a problem known as *packet switching*, *buffer management with bounded delay*:

- Input: non-empty set of jobs with:
  - release time, deadline (integers)
  - weight (also called value)
- Execution of any job takes one unit of time
- Jobs must be executed one at a time
- Goal: to maximize the total weight of executed jobs

This is the *off-line* version of the problem – the complete input is made available to the algorithm immediately.
In this version the optimal solution can be found easily (polynomial time).

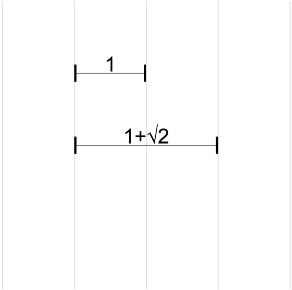
On-line version

More common scenario – there is no information about the future.
In the *on-line* version:

- At each step the algorithm makes a decision which job to execute
- The jobs become "visible" after their respective release times
- Each decision is irrevocable

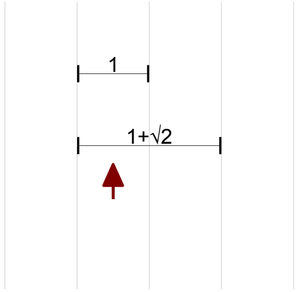
In the on-line setting the algorithm seems to have a clear disadvantage compared to the off-line setting.

Example



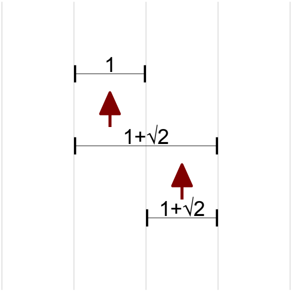
Consider the following example.

Example



Algorithm non-optimal by factor $\frac{2+\sqrt{2}}{1+\sqrt{2}} = \sqrt{2}$

Example



Algorithm non-optimal by factor $\frac{2+2\sqrt{2}}{2+\sqrt{2}} = \sqrt{2}$

Competitive ratio

How do we measure the quality of an on-line algorithm?

Competitive ratio

How do we measure the quality of an on-line algorithm?

Definition

Let A be an on-line algorithm. The *competitive ratio* of A is defined as follows

$$R_A = \sup_I \frac{w(OPT_1(I))}{w(A(I))}$$

Competitive ratio

How do we measure the quality of an on-line algorithm?

Definition

Let A be an on-line algorithm. The *competitive ratio* of A is defined as follows

$$R_A = \sup_I \frac{w(OPT_1(I))}{w(A(I))}$$

We already know, that no on-line algorithm has a competitive ratio lower than $\sqrt{2} \approx 1.414$.

Competitive ratio

How do we measure the quality of an on-line algorithm?

Definition

Let A be an on-line algorithm. The *competitive ratio* of A is defined as follows

$$R_A = \sup_I \frac{w(OPT_1(I))}{w(A(I))}$$

We already know, that no on-line algorithm has a competitive ratio lower than $\sqrt{2} \approx 1.414$.
But there is a better lower bound.

Lower bound

Theorem (Hajek 2001)

Every on-line algorithm has a competitive ratio at least equal to $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$.

Lower bound

Theorem (Hajek 2001)

Every on-line algorithm has a competitive ratio at least equal to $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$.

The proof uses a remarkably simple class of jobs – with lengths at most 2.
Consequently this lower bound holds also for many restricted versions of the problem.

Upper bounds

Progress in recent years:

- 2 (Kesselman et al. 2001, Hajek 2001)
- $\frac{64}{33} \approx 1.939$ (Chrobak et al. 2004)
- 1.852... (Li et al. 2007)
- $2\sqrt{2} - 1 \approx 1.828$ (Englert and Westermann 2007)

Agreeable deadlines

An interesting restriction of the problem: *agreeable deadlines*.

Definition

We say that the jobs forming the set S have agreeable deadlines if and only if

$$\forall i, j \in S : r_i < r_j \Rightarrow d_i \leqslant d_j$$

In other words – the availability interval of one job is not contained in the interior of the availability interval of another job.

Agreeable deadlines

The construction of the lower bound of ϕ works even with the restriction to instances with agreeable deadlines.
What about the upper bound?

Agreeable deadlines

The construction of the lower bound of ϕ works even with the restriction to instances with agreeable deadlines.
What about the upper bound?

Theorem (Li et al. 2005)

There exists an algorithm having a competitive ratio exactly $\phi \approx 1.618$ *in the agreeable deadlines setting.*

Resource augmentation

Resource augmentation – a different approach for analyzing the hardness of an on-line scheduling problem.

Resource augmentation

Resource augmentation – a different approach for analyzing the hardness of an on-line scheduling problem.

The modification

The on-line algorithm may now execute more than one job per time slot, given by the parameter k.

The "quick" on-line algorithm is compared to the "slow" off-line algorithm using the competitive ratio.
Our task is to find some lower and upper bounds for this ratio (depending on k).

Upper bound

Natural first choice: greedy algorithm.

Fact

The competitive ratio of the greedy algorithm is equal to $1 + \frac{1}{k}$.

But we can do better than that.

Upper bound

A better algorithm $EG(k)$ is presented below. Let h denote the heaviest available job (note that it may change during the step). In each time slot the algorithm executes:

- The most urgent available job with weight at least $2^{-k} w_h$
- The most urgent available job with weight at least $2^{-k+1} w_h$
- $\ldots$
- The most urgent available job with weight at least $2^{-1} w_h$

"Most urgent" means the job whose deadline will be reached next. Ties can be broken in an arbitrary way.

Upper bound

Theorem (J. 2009)

The competitive ratio of $EG(k)$ is $1 + \frac{1}{2^k - 1}$.

Upper bound

Theorem (J. 2009)

The competitive ratio of $EG(k)$ is $1 + \frac{1}{2^k - 1}$.

The proof goes by a charging scheme.
For a given instance I we first take an optimal off-line schedule and reorder it so that the sequence of executed jobs is similar to the sequence generated by $EG(k)$.

Theorem (J. 2009)

The competitive ratio of $EG(k)$ is $1 + \frac{1}{2^k - 1}$.

The proof goes by a charging scheme.
For a given instance I we first take an optimal off-line schedule and reorder it so that the sequence of executed jobs is similar to the sequence generated by $EG(k)$.
We define a charging function $c : OPT_1(I) \to \mathbb{Z}$ such that

$$c(j) = min(t_{OPT_1}(j), t_{EG_k}(j))$$

Upper bound

For every time slot t such that $w(c^{-1}(t)) > 0$ we prove that

$$w(c^{-1}(t)) < \left(1 + \frac{1}{2^k - 1}\right) w(t_{EG_k}^{-1}(t))$$

Upper bound

For every time slot t such that $w(c^{-1}(t)) > 0$ we prove that

$$w(c^{-1}(t)) < \left(1 + \frac{1}{2^k - 1}\right) w(t_{EG_k}^{-1}(t))$$

Thus

$$w(OPT_1(I)) < \left(1 + \frac{1}{2^k - 1}\right) w(EG_k(I))$$

This means that $EG(k)$ is $\left(1 + \frac{1}{2^k-1}\right)$-competitive. It can be shown easily that $EG(k)$ is not competitive for any lower ratio.

Lower bound

Question

Is there any k and an on-line algorithm executing k jobs per time slot with competitive ratio equal to 1?

Lower bound

Question

Is there any k and an on-line algorithm executing k jobs per time slot with competitive ratio equal to 1?

Theorem (J. 2009)

Every k-speed on-line algorithm has a competitive ratio higher than $1 + \varepsilon_k$.

In fact this remains true if we strengthen the algorithm by allowing it to conserve its processing power for the future – we call such an algorithm *cumulative*.

Lower bound – proof outline

We view the task as a game between Algorithm and Adversary.
Adversary creates new jobs that are presented to Algorithm.
We define a strategy $\mathcal{S}_k$ for Adversary recursively.

Lower bound – proof outline

We view the task as a game between Algorithm and Adversary.
Adversary creates new jobs that are presented to Algorithm.
We define a strategy $\mathcal{S}_k$ for Adversary recursively.

Modification

Algorithm can execute as many jobs as he wants.

Lower bound – proof outline

We view the task as a game between Algorithm and Adversary.
Adversary creates new jobs that are presented to Algorithm.
We define a strategy $\mathcal{S}_k$ for Adversary recursively.

Modification

Algorithm can execute as many jobs as he wants.

Goal

Algorithm playing against strategy $\mathcal{S}_k$ will either execute more than k jobs per step, or has lower throughput than OPT_1 on the same instance.

Lower bound – proof outline

Key points of the strategy:

- The game lasts at mosts l_k steps
- The total weight of created jobs is at most M_k

Lower bound – proof outline

Key points of the strategy:

- The game lasts at mosts l_k steps
- The total weight of created jobs is at most M_k
- The game proceeds in phases, each of which lasts for l_{k-1} steps
- We have two types of jobs
    - H-jobs, which are the heaviest jobs that appear during the game
    - L-jobs are created using strategy S_{k-1} as a subroutine

Lower bound – proof outline

Key points of the strategy:

- The game lasts at mosts l_k steps
- The total weight of created jobs is at most M_k
- The game proceeds in phases, each of which lasts for l_{k-1} steps
- We have two types of jobs
  - H-jobs, which are the heaviest jobs that appear during the game
  - L-jobs are created using strategy S_{k-1} as a subroutine

The idea is that on average the algorithm executes $k-1$ L-jobs and 1 H-job per step.

What are the values of l_k, M_k and ε_k?

Lower bound – proof outline

What are the values of l_k, M_k and ε_k?

$$l_k \leqslant 2^{2^{2k}}$$

$$M_k \leqslant 2^{2^{2^{3(k-1)}}}$$

$$\varepsilon_k \geqslant 1 + \frac{1}{M_k} \geqslant 1 + \left(\frac{1}{2}\right)^{2^{2^{3(k-1)}}}$$

The gap between the lower and upper bound is quite big.

Agreeable deadlines

It looks like normal competitive analysis (without resource augmentation) is not able to make a distinction between the general case and the restriction to agreeable deadlines.

Agreeable deadlines

It looks like normal competitive analysis (without resource augmentation) is not able to make a distinction between the general case and the restriction to agreeable deadlines.
This is different using resource augmentation:

Theorem (Jeżabek 2009+)

There is a 2-speed on-line algorithm having competitive ratio 1 for inputs with agreeable deadlines.

Proof idea

We may regard a 2-speed algorithm as two algorithms with speed 1. In our case these algorithms are syntactically similar, but have very different characteristics.

Proof idea

We may regard a 2-speed algorithm as two algorithms with speed 1. In our case these algorithms are syntactically similar, but have very different characteristics.

The first algorithm:

- Looks at all jobs seen so far – even those that have expired or that have been collected

Proof idea

We may regard a 2-speed algorithm as two algorithms with speed 1. In our case these algorithms are syntactically similar, but have very different characteristics.

The first algorithm:

- Looks at all jobs seen so far – even those that have expired or that have been collected
- Pretends that all deadlines are no later than after the next time slot

Proof idea

We may regard a 2-speed algorithm as two algorithms with speed 1. In our case these algorithms are syntactically similar, but have very different characteristics.

The first algorithm:

- Looks at all jobs seen so far – even those that have expired or that have been collected
- Pretends that all deadlines are no later than after the next time slot
- Computes the optimal off-line schedule and executes any job from it that is available

Proof idea

We may regard a 2-speed algorithm as two algorithms with speed 1. In our case these algorithms are syntactically similar, but have very different characteristics.

The first algorithm:

- Looks at all jobs seen so far – even those that have expired or that have been collected
- Pretends that all deadlines are no later than after the next time slot
- Computes the optimal off-line schedule and executes any job from it that is available

Observation

This algorithm makes no mistakes – the jobs executed by it are always in the optimal off-line schedule.

Proof idea

We may regard a 2-speed algorithm as two algorithms with speed 1. In our case these algorithms are syntactically similar, but have very different characteristics.
The first algorithm:

- Looks at all jobs seen so far – even those that have expired or that have been collected
- Pretends that all deadlines are no later than after the next time slot
- Computes the optimal off-line schedule and executes any job from it that is available

Observation

This algorithm makes no mistakes – the jobs executed by it are always in the optimal off-line schedule.

The algorithm may however miss some jobs.

Proof idea

The second algorithm:

- Looks at all jobs seen so far

Proof idea

The second algorithm:

- Looks at all jobs seen so far
- Computes the optimal off-line schedule and executes the first job from it that is available

Proof idea

The second algorithm:

- Looks at all jobs seen so far
- Computes the optimal off-line schedule and executes the first job from it that is available

Observation

The second algorithm executes all jobs that are executed by OPT_1 and that have been 'missed' by the first algorithm.

Proof idea

The second algorithm:

- Looks at all jobs seen so far
- Computes the optimal off-line schedule and executes the first job from it that is available

Observation

The second algorithm executes all jobs that are executed by OPT_1 and that have been 'missed' by the first algorithm.

Consequence

The presented algorithm's throughput is always at least equal to the througput of the optimal 1-speed off-line algorithm.

Open problems

- Find an even broader class of instances where resource augmented on-line algorithms can achieve a comtetitive ratio equal 1
- Reduce the gap between the lower and upper bound in the general k-speed scenario
- Find the best possible competitive ratio for the 1-speed scenario

Thank you!