Approximation Complexity

Optimisation problems can be grouped into *approximation complexity classes* depending on the quality of the approximation algorithms that they have.

We already mentioned NPO.

The class APX contains all NPO problems which admit a polynomial time k-approximation algorithm for *some* fixed constant $k > 1$.

The class PTAS contains all NPO problems which admit a polynomial time k-approximation algorithm for *any* constant $k > 1$.

Approximation schemes

The class PTAS takes its name and is characterised in terms of a particular family of approximation algorithms.

A *polynomial time approximation scheme* (or PTAS) for an NPO problem P is an algorithm A which takes as input an $x \in \mathcal{I}$ and an *error bound* ϵ and has a performance ratio

$$R_\epsilon(x, A(x)) \leq 1 + \epsilon$$

The algorithm A runs in time polynomial in the input order.

The PTAS is a "Fully"-PTAS (FPTAS) if its run time is polynomial in ϵ^{-1} as well.

Approximation scheme for scheduling

Recall the multiprocessor scheduling problem: n jobs have run times $p_1, \ldots, p_n$.

They are to be scheduled on m machines/processors so as to minimize the finish time.

We have already seen some approximation algorithms with bounded ratios for this problem.

We now present a PTAS for this problem due to Graham.

Assume that $n > m$ (m should be a small constant), and that the run-times are arranged in non-increasing order (i.e. $i < j$ implies that $p_i \geq p_j$). Note that the latter assumption can be easily fulfilled by sorting the jobs based on their run-times.

Parametrised algorithm

Consider now the algorithm A_k which is defined for each integer $k \in [0, n]$.

Input: $p_1, \ldots, p_n$, with $p_i \geq p_{i+1}$ for each $i \in \{1, \ldots, n-1\}$ and processor count m.

Output: A feasible schedule.

1. Schedule the first k jobs optimally.
2. Starting with the partial schedule obtained in the previous step, schedule the remaining jobs greedily using the LPT rule.

Recall that the LPT rule picks the next largest unscheduled job and schedules it on a processor which has the least load currently.

Analysis

This algorithm clearly runs in polynomial time ... but careful!

A better pseudo-code:

SCHEDULE$_k$ ($\vec{p}$, m)
    let n be the number of jobs;
    repeat
        allocate $p_1, \ldots, p_k$ in the "next possible way"
        run LPT on $p_{k+1}, \ldots, p_n$
    until ("tried them all");

If $k \leq m$ then the first few possible ways are

Proc 1	Proc 2	...	Proc $k-1$	Proc k	Proc $k+1$	Proc $k+2$	...
p_1	p_2	$\cdots$	p_{k-1}	p_k	empty	empty	
p_1	p_2	$\cdots$	p_{k-1}	empty	p_k	empty	
p_1	p_2	$\cdots$	empty	p_{k-1}	p_k	empty	
p_1	p_2	$\cdots$	p_{k-2}	p_{k-1}	p_k	empty	
empty	p_1	$\cdots$	p_{k-2}	p_{k-1}	p_k	empty	
empty	p_1	$\cdots$	p_{k-2}	p_{k-1}	empty	p_k	

Similar (slightly more complicate) enumeration scheme needed if $k > m$.

The algorithm above has approximation ratio $1 + \frac{m-1}{m+k-1}$.

Let K denote the finish time of the schedule found in Step 1.

Clearly, if $c(x, A_k(x)) = K$ then this algorithm has found an optimal schedule.

Assume now that the finish time of the total schedule is strictly greater than K (in symbols, assume that $c(x, A_k(x)) > K$).

Then it must be the case that there is some job J_j with $j > k$ that finishes at time $c(x, A_k(x))$.

This implies that all processors are busy during the time interval $[0, c(x, A_k(x)) - p_j]$ since otherwise the job J_j would have been scheduled earlier on. (Notice that once a processor becomes idle, it remains idle till the end of the schedule.)

Let $T = \sum_{i=1}^{n} p_i$ be the total run-time of the n jobs. We now conclude that

$$T \geq m(c(x, A_k(x)) - p_j) + p_j.$$

Since the jobs are arranged in non-increasing order of run-times, we have that

$$T \geq mc(x, A_k(x)) - (m-1)p_{k+1}.$$

Observing that opt(x) is at least T/m, we have the following inequality.

$$c(x, A_k(x)) \leq \text{opt}(x) + (1 - 1/m)p_{k+1}$$

If we show that p_{k+1} is not too large in terms of opt(x), we will be done.

Consider the k largest jobs which were scheduled in Step 1. In an optimal schedule, some processor must be assigned at least $1 + \lfloor k/m \rfloor$ of these jobs. Since each of these has run-time at least as large as p_{k+1}, we conclude that

$$\text{opt}(x) \geq (1 + \lfloor k/m \rfloor)p_{k+1}$$

which, combined with previous inequalities gives us the desired result.

Arbirarily good algorithm

We can now extract the promised PTAS from the above result.

Let $A_{\epsilon\text{-good}}$, for any $\epsilon > 0$, be the algorithm A_k with k chosen such that the performance ratio is at most $1 + \epsilon$. (Exercise. Verify that this will be the case provided $k \geq \frac{1-\epsilon}{\epsilon} m$).

Initial listing

We have left out one crucial detail in the description of the algorithm A_k. How does Step 1 get implemented? It is not very hard to see that there is a brute-force algorithm which compute an optimal schedule in time $O(m^k)$, for k jobs on m processors.

The running time of this step is polynomially bounded in the length of x for sufficiently small values of m, say for constant m.

Concluding remarks and exercises

Algorithm $A_{\epsilon\text{-good}}$ is by no means a practical algorithm even for relatively small value of m.

The running time is exponential in ϵ^{-1} and so ratios arbitrarily close to 1 will not be achieved in reasonable time.

This trade-off between running-time and approximation guarantees is an important feature of any approximation scheme. In general, we would like the trade-off to be such that the running time does not increase too fast with a decrease in the performance ratio.

Exercises.

1. Exactly how large can m be without making the time of the approximation scheme super-polynomial?
2. It is instructive to compute the running time of $A_{\epsilon\text{-good}}$ for small values of ϵ. For example, what is the running time when $m = 10$ and $\epsilon = 0.1$?

An example

Assume $m = 2$, and $n = 10$, and

$p_1 = 20, p_2 = 12, p_3 = p_4 = 6, p_5 = 5, p_6 = p_7 = 3, p_8 = p_9 = 2, p_{10} = 1.$

We want to look at $A_{\frac{1}{4}\text{-good}}$, that finds a $1 + \frac{1}{4}$ solution (i.e. $\epsilon = 0.25$).

We then set $k = \frac{1-\epsilon}{\epsilon} m = 6$.

We then run through all possible ways of assigning the six heaviest jobs.

In each case we complete the assignment greedily.

Approximation scheme for KNAPSACK

Definition: A collection of items is given, each having a size s_i and a profit p_i associated with them. The goal is to pack items in a knapsack of capacity U to maximise the total profit.

Greedy algorithm ... not enough even for constant approximation.

Approximation scheme? Along similar lines as for multiprocessor scheduling.

Existence of good approximation algorithms

A poly-time approximation scheme for an optimisation problem is rightfully considered the next best thing to a poly-time *exact* algorithm.

For NP-hard optimisation problems an important question is whether such a scheme exists.

Also, even if the previous question is answered in the negative one may still be able to come up with a fixed constant approximation ratio.

Since individual questions of this sort are rather hard to answer, in the following sections we will build up on the theory of NP-completeness to prove a number of results predicated on "P$\neq$NP".

More specifically

1. We will define a suitable notion of *reducibility* (roughly, translation) between different problems. A reduction from a "hard-to-approximate" problem Π_1 to a problem Π_2 would imply that Π_2 is "hard to approximate", in the sense that a "good approximation" for Π_2 would imply P=NP.
2. We will then build a library of reductions, thus enlarging the class of problems that are hard to approximate.
3. Finally we will describe a reduction from an NP-complete problem to an optimisation problem. This reduction along with all other reductions shown earlier on imply the hardness of approximating these problems.