Finding total unimodularity in optimization problems solved by linear programs *

Christoph Dürr Mathilde Hurand

Abstract

A popular approach in combinatorial optimization is to model problems as integer linear programs. Ideally, the relaxed linear program would have only integer solutions, which happens for instance when the constraint matrix is totally unimodular. Still, sometimes it is possible to build an integer solution with same cost from the fractional solution. Examples are two scheduling problems [3, 4] and the single disk prefetching/caching problem [2]. We show that problems such as the three previously mentioned can be separated into two subproblems: (1) finding an optimal feasible set of slots, and (2) assigning the jobs or pages to the slots. It is straigthforward to show that the latter can be solved greedily. We are able to solve the first with a totally unimodular linear program, which provides simpler (and sometimes combinatorial) algorithms with better worst case running times.

1 Introduction

We used our specific approach to give rather simple solutions to three different optimization problems. The first two are scheduling problems : the Tall-Small Jobs Problem [3] and the Equal Length Jobs Problem [4]. The last one is about Offline prefetching and caching to minimize stall time[2] . In the Tall-Small Jobs Problem, we have m machines, n unit length jobs, some of which need to execute on all the machines at the same time. In the Equal Length Jobs Problem, jobs have a given equal length $p \geq 1$ and each job executes on a single machine. In both problems jobs have given release times and deadlines in between which they need to execute. The goal is to find a feasible schedule, and moreover, for the equal length jobs problem, a feasible schedule that minimizes total completion time of the jobs. The third optimization problem, Offline prefetching and caching to minimize stall time belongs to a different field: we are given a sequence of n page requests and a cache of size k. We can evict a page from the cache and fetch a new page to replace it. This operation cannot be done in parallel and costs F time units. When a page request is served it costs 1 time unit, unless the page is not yet in the cache, then a stall time is generated until the corresponding fetch completes. The goal is to decide when to evict and fetch pages so as to minimize the total stall time.

Though quite different, those three problems were solved in a similar manner. Unlike previous works where the authors transform the solution of a relaxed integer linear program into an integer one, we used a new technique which simplifies the linear programs, and allows us to get directly optimal integer solutions: our approach is based on the notice that only

*Laboratoire de Recherche en Informatique de l'Ecole Polytechnique (LIX), CNRS UMR 7161, Palaiseau, France. {durr,hurand}@lix.polytechnique.fr.

the structure of the solution matters in the objective function, jobs and pages don't appear namely. Therefore, we completely dissociate the resolution process into two phases. First a simplified linear program can be used to find an optimal *skeleton* for the solution, and it is only later that we need to worry about *assigning* jobs or pages to this skeleton: for scheduling problems, the skeleton is a sequence of slots, and the assignment maps jobs to slots; for the the cache problem, the skeleton is a sequence of intervals and the assignment associates to every interval a page to evict at the beginning and a page to fetch at the end. Our skeletons are such that the *assignment* phase just comes down do running a greedy algorithm. Our contribution is that this strategy, where you don't compute the assignment in the linear program, leads to linear programs with very simple constraint matrices, which not only are totally unimodular, but are (the transpose of) directed vertex adjacency matrices. This allowed us to transform our scheduling problems into the search of shortest path in directed graphs. After comparison with the former way of doing, it is interesting to notice, and we will show in details how at first presenting our method, that our linear programs are not completely novel: they are in fact relaxations of the former ones.

Our important results are in that in two cases we could solve the optimization problems with a shortest path algorithm. As a result we beat in worst case complexity the best known algorithm for the tall/small job scheduling problem :$O(n^3)$ instead of $O(n^{10})$. On the other problems, our linear programs are more simple, which, using linear program theory, gives us a $O(n^{10})$ worst case complexity (instead of $O^*(n^{18})$)for the prefetch/caching problem. All our algorithm are simple and implementations are available in the authors home-pages.

2 Scheduling equal length jobs

We will first introduce our method on a basic scheduling problem. We have n jobs, each of the same length p. Every job $j \in [1, n]$ comes with an interval $[r_j, D_j]$ consisting of a release time and a strict deadline. The goal is to find a schedule on m parallel machines, such that each job is assigned to an execution slot consisting of a particular machine and a time interval $[s_j, s_j + p) \subseteq [r_j, D_j]$. In addition, all execution slots assigned to a particular machine must be disjoint. One possible application could be frequency allocation. A network operator has a link with m optical fiber strings. Users ask for allocations of a frequency band of fixed size, inside the large frequency band that the particular user devices can handle. The goal is to find an assignment which satisfies all users. In addition we want to find the solution (if it exists) that minimizes the total completion time of the jobs. In the standard Graham notation, this problem is called $P|r_j; p_j = p; D_j|\sum C_j$.

Simons [8] give a complicated *greedy-backtrack* algorithm running in time $O(n^3 \log\log n)$, and later improved to $O(mn^2)$ [9]. Recently Brucker and Kravchenko [4] gave another algorithm for it, using a completely different approach. While their algorithm has worse complexity it is interesting because of a generalization which permits to solve an open problem, namely minimizing the weighted total completion time, where jobs are given priority weights.

Variants of this problem have been studied extensively. If we allow job lengths 1 or p, then the problem becomes NP-complete [9]. If however the release times are multiple of p, the problem can be solved in time $O(n \log n)$ [6], and if there are no deadlines, the greedy algorithm solves the problem. For a single machine, the problem can be solved in time $O(n \log n)$ by a tricky algorithm of Garey, Simons, and Tarjan [7].

A generalization of the feasibility problem is to find a maximal set of jobs, which can all

be scheduled between their release times and deadlines. This problem is still open. Even the more general problem, when jobs come with a weight, and the goal is to find a maximal weighted feasible job set, is not known to be NP-hard.

2.1 Previous work

First we observe that without loss of generality we can restrict ourselves to schedules where each execution slot starts at some release time plus a multiple of p, simply by shifting each slot as much to the beginning as possible. Let $\mathcal{T} = \{r_i + (a-1)p : 1 \leq i, a \leq n\}$ be this set of time points. And finally for a fixed schedule, if we number the execution slots from left to right, we can always reassign the j-th slot to the machine $(j \bmod m) + 1$. This way we don't need to take care of which machines the slots are assigned to, as long as there are at most m slots starting in every time interval of size p, which ensures that slots don't overlap on a particular machine. The linear program of [4] has a variable x_{jt} for each job j and time $t \in \mathcal{T}$, with the meaning that $x_{jt} = 1$ if job j is executed in the slot $[t, t+p)$. Then the program is to minimize $\sum_{jt}(t+p)x_{jt}$ subject to

$$\forall j \in [1,n] \quad : \quad \sum_{t \in \mathcal{T}} x_{jt} = 1 \quad \text{(every job completes)} \tag{1}$$

$$\forall j \in [1,n], \forall t \in \mathcal{T} \backslash [r_j, D_j - p] \quad : \quad x_{jt} = 0 \quad \text{(allowed interval)} \tag{2}$$

$$\forall s \in \mathcal{T} \quad : \quad \sum_{s \leq t < s+p} \sum_{j \in [1,n]} x_{jt} \leq m \quad \text{(no overlapping)} \tag{3}$$

It is quite clear that there is an integer solution to this linear program if and only if there is a feasible schedule. While this linear program is not totally unimodular, the authors of [4] were still able to round the fractional solution into an integer solution of the same cost.

2.2 Relaxing the linear program

The linear program above computes not only the time slots of the schedule, but also the assignment of jobs to slots. However once we are given the *skeleton* of a schedule, meaning a set of time slots, it is always possible to assign the jobs greedily in EDD fashion: assign to every slot the job with smallest deadline among the available jobs. We release the linear program from the job assignment, in order to obtain a simpler linear program which only computes a feasible skeleton.

We proceed in several steps. First we weaken equation (**??**) into the inequality $\sum_{t \in \mathcal{T}} x_{jt} \geq 1$. Then combining this new constraint with (**??**) leads to

$$\forall j \in [1,n] : \sum_{t \in [r_j, D_j - p]} : x_{jt} \geq 1. \tag{4}$$

Now for every pair $s, t \in \mathcal{T}, s \leq t$ we sum (4) over all jobs j that have $[r_j, D_j - p] \subseteq [s,t]$, upper-bounding the left hand side we obtain

$$\forall s, t \in \mathcal{T}, s \leq t : \sum_{s' \in [s,t]} \sum_{j} x_{js'} \geq |\{i : [r_i, D_i - p] \subseteq [s,t]\}|. \tag{5}$$

The constraints are clearly necessary, and we will show later they are also sufficient to get the optimal solutions. We then group $\sum_j x_{jt}$ into a single variable: we set $y_t := \sum_{s \leq t} \sum_j x_{jt}$.

Now y_t represents the total number of slots up to time t. To simplify notations we introduce an additional time point $t_0 < \min \mathcal{T}$, and set $\mathcal{T}' = \mathcal{T} \cup \{t_0\}$. For any time $t > t_0$, we define the functions $\text{round}(t) := \max\{s \in \mathcal{T}' : s \leq t\}$ and $\text{prec}(t) := \max\{s \in \mathcal{T}' : s < t\}$.

minimize $\sum_{t \in \mathcal{T}} (t+p)(y_t - y_{\text{prec}(t)})$
subject to

$$
\begin{aligned}
y_{t_0} &= 0 \\
y_{max\mathcal{T}} - y_{t_0} &\leq n \\
\forall t \in \mathcal{T}, s = \text{prec}(t) : y_s - y_t &\leq 0 \qquad \textbf{(order)} \\
\forall s \in \mathcal{T}, t = \text{round}(s+p) : y_t - y_s &\leq m \qquad \textbf{(load)} \\
\forall i, j \in [1,n], s = \text{prec}(r_i), t = \text{round}(D_j - p), s \leq t : y_t - y_s &\geq c_{ij}, \quad \textbf{(inclusion)}
\end{aligned}
$$

where $c_{ij} := |\{k : [r_k, D_k] \subseteq [r_i, D_j]\}|$ is the number of jobs which have to be executed in the interval $[r_i, D_j]$.

The two first inequalities give the conditions on y at origin time and ending time, it could be shown that they are not in fact necessary, but their presence simplify demonstrations. The *order* inequalities ensure that (y_t) is a non decreasing increasing sequence. The *load* inequalities verify that there are never more than m slots overlapping, and the *inclusion* inequalities, we will see it, are here to ensure that there is a feasible mapping from jobs to slots. The linear program has in every constraint exactly two variables, and with the respective coefficients $+1$ and -1. So the dual of the constraint matrix is the incidence matrix of a directed graph, which means the constraint matrix is totally unimodular. Therefore our linear program's constraints are in the from $Ay \leq b$ with A totally unimodular and b integer. This means its solutions are all integer.

Let (y_t) be an optimal integer solution to this linear program. It indeed defines the skeletons of a solution: at each time $t \in \mathcal{T}$ ther will be $y_t - y_{prec(t)}$ slots available for scheduling. Assigning greedily jobs to these slots means scheduling at each time on each existing slot the job with the smallest deadline among the *available* jobs, meaning jobs which are not yet scheduled and which release time, deadline intervals permits to be scheduled in that slot.

Lemma 1 *The greedy assignment produces a valid schedule.*

Proof: We can notice that according to the second condition and the *inclusion* condition on $[t_0, max\mathcal{T}]$, $y_{max\mathcal{T}} = n$. We define V to be the multiset of time slots, such that slot $[t, t+p]$ is contained $y_t - y_{prec(t)}$ times. Therefore $|V| = n$. As mentioned in the previous section, by the *load* inequality, the slots can be assigned to machines without overlapping. So, it only remains to show that, first, there exist assignments of jobs to slots, which respect release times and deadlines, and then that the greedy assignment is one of them.

Let U be the set of n jobs, and $G(U, V, E)$ a bipartite graph where E contains all edges between a job j and a slot $[t, t+p]$ if $[t, t+p] \in [r_j, D_j]$. We have to show that this graph has an injection from U to V, and will use Hall's theorem for this.

For a set of jobs S, we denote the neighboring slots ∂S, as the set of all slots t such that there is a job $j \in S$ with $(j, t) \in E$. We need to show that for every set S, $|S| \leq |\partial S|$, which by Hall's theorem, characterizes the existence of an injection. Let S be a set of jobs. Suppose S can be partitioned into $S_1 \cup S_2$ such that for any jobs $i \in S_1$ and $j \in S_2$ the intervals $[r_i, D_i - p]$ and $[r_j, D_j - p]$ are disjoint. Then clearly ∂S is the disjoint union of ∂S_1

and ∂S_2. Therefore we can without loss of generality assume that $\bigcup_{j\in S}[r_j, D_j]$ is a unique interval $[r_i, D_j]$, for $i = \text{argmin}_{i\in S} r_i$ and $j = \text{argmax}_{j\in S} D_j$. Then $|S| \leq c_{ij}$. Also the number of slots in the interval $[r_i, D_j)$ is exactly $y_t - y_s$ for $s = \text{prec}(r_i), t = \text{round}(D_j - p)$. From the *inclusion* inequality we get the required inequality and we conclude that there exist a valid assignment. Now since $|V| = |U| = n$, the injection is in fact a bijection, and there exists at least one perfect matching from jobs to slots with respect to release times and deadlines.

Proving that you can permute jobs in any of these matching to get the greedy matching is a quite standard scheduling procedure: let be two jobs i, j with $D_i < D_j$, and i is scheduled at some time t, while j is scheduled at some time s with $r_i \leq s < t$. Then it is possible to exchange the jobs i, j in their execution slots $[s, s+p)$ and $[t, t+p)$. By the use of a potential function, decreasing at each exchange, it is possible to transform our schedule in a so called *earliest due date schedule*. We conclude since there exists at least a valid assignment, the greedy assignment is valid as well. □

This means that an optimal integer solution can be found with a standard linear program solver. But what is interesting, is we could provide a direct combinatorial algorithm for this task. Note first that we can modify our objective function so as to transform our minimization problem in a maximization problem where all variables have non-negative coefficients since

$$\sum_{t\in\mathcal{T}}(t+p)(y_t - y_{\text{prec}(t)}) = (max\mathcal{T} + p)y_{max\mathcal{T}} - \sum_{t\in\mathcal{T}\setminus t_0}(t - \text{prec}(t))y_{prec(t)}, \tag{6}$$

where $(max\mathcal{T} + p)y_{max\mathcal{T}}$ is a constant since $y_{max\mathcal{T}} = n$.

We now show how to find the optimal solution with a standard method.

Lemma 2 (based on [11, p.558]) *Let be a linear program on variables $u_0, u_1, \ldots, u_{N-1} \geq 0$ where $u_0 = 0$ and the coefficient in the objective function is positive for the other variables. Also all the constraints are of the form $u_i - u_j \leq a$ for some variables u_i, u_j and integer a. Then the optimal solution can be found in time $O(NM)$, where N is the number of variables and M the number of constraints.*

Proof: We define a directed graph $G(V, E)$ where the vertices are the variables, and every inequality $u_i - u_j \leq a$ introduces an arc from u_j to u_i with weight a (if there are several inequalities with the same left hand part, we keep only the strongest one). Now a directed path from u_i to u_j of total weight b, corresponds to the inequality $u_i - u_j \leq b$, which is the result of summing all the inequalities associated to the edges along the path. Let $d_{i,j}$ be the shortest path from u_i to u_j and $d_i = d_{0,i}$.

Therefore if the graph G has a negative cycle, the linear program is not feasible, since it implies the inequality $u_i - u_i \leq a$ for some variable u_i on the cycle and the negative cycle length a.

If there is a vertex u_i which is not reachable from u_0, then the linear program is unbounded. Let C be the strongly connected component containing u_i, that is the ? of vertices u_j such that u_j is reachable from u_i and vice-versa. Indeed for any integer d we can set $u_i = d$, for all u_j in C, $u_j = d + d_{i,j}$, and $u_i = d_i$ for all variables outside of C, which results in a solution to the linear program (the demonstration that no constraint is then violated is the same than in next paragraph) of arbitrary high objective value.

Now suppose that G has no negative cycle, and that every vertex u_i is reachable from u_0. Setting $u_i = d_i$ will be a solution to the linear program, since if some constraint $u_i - u_j \leq a$ was violated, it would mean that $d_i > d_j + a$, which can not be the shortest path since there

is a directed edge of weight a from u_j to u_i. Also the solution is optimal since the shortest path from u_0 to u_i implies the inequality $u_i \leq d_i$ and in the objective function all coefficients are non-negative, so this solution is an upper bound on optimal solution. Therefore it is the optimal solution.

The distances can be computed with the standard Bellman-Ford shortest path algorithm in time $O(|V| \cdot |E|)$, where $|V|$ is the number of vertices and $|E|$ the number of arcs. This algorithm also detects negative cycles and vertices which are not reachable from the source u_0. □

Now we summarize the previous lemmata.

Theorem 1 *Our algorithm solves $P|r_j; p_j = p; D_j|\sum C_j$ in worst case time $O(n^4)$.*

Proof: Given the instance $m, p, r_1, \ldots, r_n, D_1, \ldots, D_n$, we construct the set $\mathcal{T}$ of $O(n^2)$ time points. Then we compute for every pair of jobs i, j the number of jobs c_{ij} which need to be scheduled in $[r_i, D_j]$. A naive algorithm does it in time $O(n^3)$, which would be enough for us. However it can be solved in time $O(n^2)$ using the following recursive formula. We assume jobs are indexed in order of release times. For convenience we set $c_{n+1,j} = 0$. Then $c_{i,j} = c_{i+1,j} + 1$ if $D_i \leq D_j$ and $c_{i,j} = c_{i+1,j}$ if $D_i > D_j$.

This permits to construct the graph G and find in time $O(n^4)$ the optimal solution to the linear program, if there is one. Finally we do an earliest due date assignment of the jobs to the slots defined by the solution to the linear program in time $O(n \log n)$ using a priority queue. □

Note that in this section we don't beat the best known algorithm for $P|r_j; p_j = p; D_j|\sum C_j$ which is $O(mn^2)$ [9]. However, it allows us to introduce our technique that will be used later on. What is more, the transformation of the problem into a shortest path problem is we think interesting, and we still have hope that another shortest path algorithm, better fitted for our specific type of graph could lower our complexity.

3 Scheduling tall and small jobs

In a parallel machine environment, sometimes maintenance tasks are to be done which involve all machines at the same time. Think of business meetings or inventory. Formally we are given n jobs of unit length $p = 1$, each job j comes with an integer release time and a deadline interval $[r_j, D_j]$ in which it must be scheduled. We distinguish two kind of jobs. The first n_1 jobs are *small* jobs, in the sense that they must be scheduled on one of the m parallel machines, it does not matter which one. The $n_2 = n - n_1$ remaining jobs are *tall* jobs, in the sense that they must be scheduled on all the m machines at the same time.

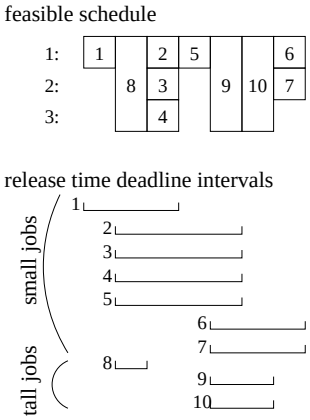

Figure 1: Example for 3 machines.

A time slot is an interval $[t, t+1[$ for an integer boundary t. The goal is to find a *feasible* schedule, where each tall job is assigned to a different time slot, and each small job is assigned to a different (machine, time slot) pair for the remaining time slots. In addition the time slot to which some job j is assigned must be included in $[r_j, D_j]$.

This problem has been solved by Baptiste and Schieber [3], with a linear program using $O(n^2)$

variables and $O(n^2)$ constraints. The linear program is not totally unimodular, however they manage to show that for the particular objective function it always has an integer solution. We provide a linear program using only $O(n)$ variables but still $O(n^2)$ constraints, but whose constraint matrix is the incidence matrix of a directed graph, and can be solved in time $O(n^3)$ with a shortest path algorithm.
Baptiste and Schieber showed that we can assume that the time interval ranges only from 1 to n, otherwise the problem could easily be divided into two disjoint subproblems.

In a similar way than before, we will denote by x_t the total number of time slots assigned to tall jobs in $[1, t+1]$. For convenience we set $x_0 = 0$. The number of small jobs that must be scheduled in $[s,t]$ is $k_{s,t} = |\{j : j \leq n_1, [r_j, D_j] \subseteq [s,t]\}|$ and the same for tall jobs is $\ell_{s,t} = |\{j : j > n_1, [r_j, D_j] \subseteq [s,t]\}|$. Let be the following linear program, which does not have an objective value.

(x_t) is a non decreasing sequence

$$\forall t \in [1,n] : x_{t-1} \leq x_t \tag{7}$$

Only one tall job can be scheduled by unit-length interval

$$\forall t \in [1,n] : x_t - x_{t-1} \leq 1 \tag{8}$$

There are enough slots for the tall jobs

$$\forall s,t \in [1,n], s \leq t : x_{t-1} - x_{s-1} \geq \ell_{s,t} \tag{9}$$

There are enough remaining slots for the small jobs

$$\forall s,t \in [1,n], s \leq t : x_{t-1} - x_{s-1} \leq t - s - \lceil k_{s,t}/m \rceil. \tag{10}$$

Once again, the transpose of the constraint matrix is the adjacency matrix of an oriented graph, and the constant vector b is integer. As previously, it has only integer solutions.

Theorem 2 *Fix an instance of the tall/small scheduling problem. There is an integer solution to this linear program if and only if there is a feasible schedule.*

Proof: It is quite obvious that fixing (x_t) according to any feasible schedule will satisfy the constraints.

For the hard direction, let (x_t) be a solution to the linear program, we know it is integer. Then $x_t - x_{t-1}$ — which can be 0 or 1 — is the number of slots for tall jobs at time t. We will again use Hall's theorem to show that there is a valid assignment of the n_2 tall jobs to these slots. Inequality (9) for $[s,t] = [1,n]$ forces $x_n \geq n_2$. Now let be $G(U,V,E)$ the bipartite graph, where U are the n_2 tall jobs, and V the x_n slots. There is an edge between job j and time slot $[t, t+1]$ if it is included in $[r_j, D_j]$. We have to show that for every subset $S \subseteq U$, the number of neighboring slots in V is at least $|S|$. Let s be the smallest release time among S and t be the largest deadline among S. Again it is sufficient to show this claim for connected sets S in the sense that $\cup_{j \in S}[r_j, D_j] = [s,t]$. Now $|S| \leq \ell_{s,t} \leq x_{t-1} - x_{s-1}$, where the last expression is the number of slots in $[s,t]$. This completes the claim that there is a valid assignment from tall jobs to the slots.

For the small jobs, note that $a_{s,t} := (t-s) - (x_{t-1} - x_{s-1})$ is the number of remaining slots in $[s,t]$ which are not assigned to tall jobs, and $a_{s,t} \cdot m$ small jobs can fit in that interval. Again inequality (10) implies $k_{s,t} \leq m \cdot a_{s,t}$, and Hall's theorem shows that there is a valid assignment of small jobs to the remaining slots. □

In the original paper [3] the author gave a $O(n^4)$ combinatorial algorithm. Using the transformation into shortest path allows us to improve this complexity.

Corollary 1 *The tall/small scheduling problem can be solved in time $O(n^3)$.*

Proof: As in the second section, we have a linear program with $O(n)$ variables and $O(n^2)$ constraints which can be produced in time $O(n^2)$. We just take an arbitrary objective function in which all the variable coefficients are positive, and build the associated graph as in the previous section. Then we compute the all shortest paths from the source x_0, in time $O(n^3)$. If this computation detects a negative cycle, then the problem has no solution. Otherwise, we get the skeleton of a solution to the problem that minimize the total completion time of the tall jobs. Finally if there is a solution, the standard earliest due date assignment, first of tall jobs, then of small ones, produces a valid schedule in time $O(n \log n)$. □

Here again, a direction that we are still exploring is to find another shortest path algorithm, better fitted for these specific graphes, that could improve this complexity.

4 Prefetching

Caches are used to improve the memory access times. In this context the memory unit is called a *page*, and is stored on a slow disk. The cache can store up to k pages. Now if a page request arrives, and the page is already in the cache, it can be served immediately, otherwise it must first be fetched from the disk, and that introduces a stall time of F units. In the latter case the new page replaces some other page currently in the cache. The time F represents the time to read the new page in the case of a read only memory or additionally to write the old page on the disk in the case of a read/write memory. The idea of *prefetching* is to fetch a page even before it is requested, so as to reduce the stall time: During a fetch which evicts some page y replacing it by some page z, other requests can be served for pages currently in the cache and different from y or z. In the single disk model we consider here, only a single fetch can occur at the same time. The goal is, knowing in advance the all request sequence, to come up with a prefetch schedule, which minimizes total stall time.

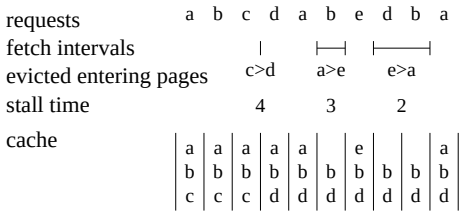

Figure 2: An optimal prefetching for a cache of size $k = 3$, and a fetch duration $F = 4$.

While the real life problem is on-line, and has been extensively studied by Cao et al. [5], the offline problem has first been solved in 1998 [2], by the use of a linear program, for which it

was shown that it always has an optimal integer solution, while not being totally unimodular. Later in 2000 [1], a polynomial time algorithm was given modeling the problem as a multi commodity flow with some postprocessing. Formally the problem can be defined as follows.

The Offline Prefetching problem

> The input is a page request sequence $x_1, \ldots, x_n$, an initial cache set C_1, and a fetch duration F. Let $k = |C_1|$ be the cache size. A fetch is a tuple (s, y, e, z), where y, z are pages and $s, t \in [1, n]$ are time points with $s \leq e \leq s + F$. The meaning is that at time s, the page y leaves the cache and at time e the page z enters the cache. Its cost, the induced stall time, is $F - (e - s)$. The goal is to come up with a sequence of fetches minimizing the total stall time, such that two fetches intersect in at most one time point, and such that every request can be served, i.e. $\forall t \in [1, n] : x_t \in C_t$, where C_t is the cache at time t obtained from C_{t-1} by evicting/fetching all the pages that had to be evicted fetched at time t. To simplify notation we assume that the request sequence contains at least k distinct pages, that C_1 consists of the first k distinct requests, and that at time 1, no page has left/entered the cache yet.

Albers, Garg and Leonardi defined a linear program with a characteristic variable for every fetch interval $[s, e]$, and two additional characteristic variables for every couple (y, $[s, e]$) indicating whether page y enters (resp leaves) the cache at the beginning (resp the end) of the fetch $[s, e]$. Finally they show that the linear program has always an integer solution for the considered objective function.

As observed in [2] without loss of generality the page to be evicted at time t from the cache C_{t-1} is the page, who's next request is furthest in the future or which is never requested again. Also without loss of generality the page to be fetched at time t is the page who's next request starting from t is nearest in the future. Therefore all the information about the fetches is in the time intervals, and we will write a linear program which produces only the time intervals in which evictions/fetches occur. The actual pages have to be assigned in a post processing, in greedy manner just mentionned. Rather to have single variable for every interval and every page, we only count how many pages entered and how many left the cache in total since the beginning, which leaves us with $O(n)$ instead of $O(n^2 F)$ variables. We denote by I_t (resp. O_t) the total number of pages which entered (resp. left) the cache up to time t included. Our linear program becomes:

minimize $FO_n - \sum_{t=1}^{n}(O_t - I_t)$,
subject to
At time 1*, no page leaves yet*

$$O_1 = 0 \tag{11}$$

(O_t) *and* (I_t) *are non decreasing sequences*

$$\forall t \in [2, n] : O_{t-1} \leq O_t \text{ and } I_{t-1} \leq I_t \tag{12}$$

The cache cannot overflow

$$\forall t \in [1, n] : O_t \geq I_t \tag{13}$$

Two fetches don't overlap in time

$$\forall t \in [1, n] : O_t \leq I_t + 1 \tag{14}$$

A fetch length is at most F

$$\forall t \in [1,n] : \; I_{\min\{t+F,n\}} \geq O_t \tag{15}$$

There are enough fetches to serve all requests

$$\forall 1 \leq s \leq t \leq n : \; I_t - O_s \geq |\{x_s, x_{s+1}, \ldots, x_t\}| - k \tag{16}$$

The optimal solution of this linear program is always integer, since it is totally unimodular (for the same reason as in previous section: its constraint matrix the transposed incidence matrix of a directed graph.)

Theorem 3 *Let (I_t, O_t) be an optimal integer solution to the linear program. Then there is valid fetch sequence of the same cost, which can be built by greedy assignment.*

Proof: First we observe that the cost function makes sure that $O_n = I_n$, which ensures that all interval are eventually closed. The solution defines $m = O_n$ intervals as follows. For every $j = 1 \ldots m$, let s_j be the smallest time such that $O_{s_j} \geq j$ and e_j the smallest time such that $I_{e_j} \geq j$. Then by (13) and (15) we have $s_j \leq e_j \leq s_j + F$. Which means that all intervals are well defined and of length smaller or equal than F. Now by (14), $e_j \leq s_{j+1}$ (otherwise, we'd have $I_{e_j} + 1 \geq O_{e_j} > O_{s_{j+1}}$ but $I_{e_j} = j$ and $O_{s_{j+1}} = j + 1$ by definition.), and this for all $j < m$, so the intervals do not overlap (but the ending point of one might be the starting point of another). Moreover the objective value of (I_t, O_t), equals the total stall time of these intervals, for at each time t, the difference O_t - I_t is equal to 1 if an interval is currently opened and to 0 none is. It remains to prove that the greedy assignement of pages to evict/fetch to each interval is such that all requests are served, i.e. that the conditions type 16 are sufficient. We denote by C_s the cache obtained at time s, after all entrances and evictions that occur at time s. We will show that the following invariant holds in a solution of our linear program for every time $s \in [1, n]$,

$$\forall t \in [s,n] : I_t - I_s \geq |\{x_s, \ldots, x_t\} \backslash C_s|. \tag{17}$$

It means that if the number of pages requested in $[s,t]$ but not in the cache at time s is a, then at least a pages must enter the cache somewhere in $[s+1,t]$. In particular it means for $t = s$, that the page requested at time s will be the in the cache at that moment. The proof is by induction on s.

Basis case $s = 1$ Let t_0 be the greatest request time such that x_{t_0} is not in C_1. Then by the assumption that initially the cache contains the first k distinct requests, we have that for $t < t_0$, $\{x_1, \ldots, x_t\} \subseteq C_1$, so the right hand side of (17) is 0 and (17)holds by (12). For $t \geq t_0$, since the intersection of $\{x_1, \ldots, x_t\}$ and C_1 is exactly k, the invariant holds by (16) (since $O_1 = I_1 = 0$).

Induction case Assume the invariant holds for some s. Let's show that it also holds for $s+1$. Several things can happen at time $s+1$, pages can leave the cache and pages can enter the cache. We will do these operations step by step, transform slowly I_s into I_{s+1} and C_s into C_{s+1}, and show that each step preserves the invariant (17).

By induction hypothesis we have $x_s \in C_s$, so $\{x_s, \ldots, x_t\} \backslash C_s = \{x_{s+1}, \ldots, x_t\} \backslash C_s$. Therefore, if nothing happens and no page enter or leave the cache, then $C_{s+1} = C_s$, $I_s = I_{s+1}$ and the invariant is preserved for $s+1$.

Now we deal with the case when there is some page movement at time $s+1$, that is $I_{s+1} > I_s$ or $O_{s+1} > O_s$ or both. We artificially decompose this page movement in as many times as needed, so that at each time there is only one operation happening: a fetch or an eviction. The page movements at those intermediary times are set so as to alternatively evict and enter pages, among the $O_{s+1} - O_s$ pages to evict and the $I_{s+1} - I_s$ pages to enter. Of course if a fetch is pending at time s, that is $|C_s| = k-1$, then we start with entering a new page and otherwise if the cache is full, i.e. $|C_s| = k$, we start with evicting a page. Since the number of entrances and evictions can differ by at most one, it is possible to do so. Therefore, we need to do the induction case only in the case when a page is entering the cache or when one is leaving the cache but not both.

When page is entering the cache, we have $I_{s+1} = I_s + 1$. Let z be the page entering the cache, and let $t_0 \geq s+1$ be the next request time of z. Then if $t < t_0$, by the choice of z, all requests of $x_{s+1}, \ldots, x_t$ must be in C_{s+1}, so the right hand side of (17) at time $s+1$ is 0, and the inequality holds by (12). Now if $t \geq t_0$, since $z \in C_{s+1}$ but $z \notin C_s$, the left hand side of (17) at time $s+1$ has decreased by 1 compared to time s, but at the same time $I_{s+1} = I_s + 1$, so both sides of the invariant decrease by 1 and by induction the inequality is preserved at time $s+1$.

Now consider the case when a page leaves the cache. Let y be the page leaving. Then $I_s = I_{s+1}$ and $O_{s+1} = O_s + 1$. Let t_0 be the next request time of y or let $t_0 = n+1$ if y is never requested again. Then if $t < t_0$, removing y from C_{s+1} does not change the right hand side of (17) when replacing s by $s+1$. The left hand side does not change either since no page enters the cache, and the inequality is preserved. For $t \geq t_0$ however by the choice of the evicted page y, we have that $C_{s+1} \subseteq \{x_{s+1}, \ldots, x_t\}$. So the left hand side of (17) at time $s+1$ is $|\{x_{s+1}, \ldots, x_t\}| - (k-1)$, and $I_{s+1} = O_{s+1} - 1$ since we have just evicted a page. Therefore, (17) holds by (16). □

It is to be noticed that the shortest path algorithm does not apply here, given that we find coefficients of both signs in the objective function.

Applying the algorithm from [10], which has worst case running time $O(\max\{n, m\}^5)$ for totally unimodular linear programs with n variables and m constraints, we get the announced running time.

Corollary 2 *The offline prefetch problem can be solved in time $O(n^{10})$.*

5 Conclusion

We demonstrated on three different optimization problems how to transform some linear programs into simpler totally unimodular linear programs, thanks to a dissociation technique where we do not compute the assignment inside the linear program. We obtained linear programs with very simple constraint structure, and algorithm easy to implement. In doing so, we also revealed very simple graph structures underlying on those optimizations problems. Further work would include trying and find other problem where our technique apply, and maybe extract from it a general framework. We are also interested in improving the combinatorial algorithms that arose from the graph structures in the scheduling problems: indeed

those graphs have, among others, this property that if you draw vertices as points on a line, the arcs from left to right have positive weights and the ones from right to left negative. One idea for instance is to try and extract from Simons and Warmuth's algorithm a shortest path algorithm suitable for our class of graphs. At last, it would be interesting to derive a combinatorial algorithm from the graph structure for the prefetching problem, even though shortest path does not work there.

We wish to thank Arthur Charguéraud, Philippe Baptiste and Miki Hermann for helpful comments.

References

[1] S. Albers and M. Büttner. Integrated prefetching and caching in single and parallel disk systems. *Information and Computation*, (198):24–39, 2005.

[2] S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. *Journal of the ACM*, (47):969–986, 2000.

[3] Philippe Baptiste and Baruch Schieber. A note on scheduling tall/small multiprocessor tasks with unit processing time to minimize maximum tardiness. *Journal of Scheduling*, 6(4):395–404, 2003.

[4] Peter Brucker and Svetlana Kravchenko. Scheduling jobs with equal processing times and time windows on identical parallel machines. Osnabrücker Schriften zur Mathematik H 257, Universität Osnabrück. Fachbereich Mathematik/Informatik, to appear in Journal of Scheduling, 2005.

[5] P. Cao, E.W. Felten, A.R. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transaction of Computer Systems*, pages 188–196, 1995.

[6] G. Frederickson. Scheduling unit-time tasks with integer release times and deadlines. *Information Processing Letters*, 16:171–173, 1983.

[7] M.R. Garey, D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal on Computing*, 10(2):256–269, 1981.

[8] B. Simons. A fast algorithm for single processor scheduling. In *Proceedings IEEE 19th Annual Symposium on Foundations of Computer Science (FOCS'78)*, pages 246–252, 1978.

[9] Barbara Simons and Manfred Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM Journal on Computing*, 18(4):690–710, 1989.

[10] E. Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34:250–256, 1986.

[11] Jan van Leeuwen. *HandBook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier, 1990.