The Buffer Minimization Problem for Multiprocessor Scheduling with Conflicts

Marek Chrobak[1], János Csirik[2], Csanád Imreh[2], John Noga[3], Jiří Sgall[4], and Gerhard J. Woeginger[3]

[1] Dept. of Computer Science, University of California, Riverside, CA 92521, USA, marek@cs.ucr.edu

[2] József Attila University, Department of Computer Science, Árpád tér 2, H-6720 Szeged, Hungary, csirik,cimreh@inf.u-szeged.hu

[3] TU Graz, Institut für Mathematik B, Steyrergasse 30, A-8010 Graz, Austria, gwoegi,noga@opt.math.tu-graz.ac.at

[4] Mathematical Inst., AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic, sgall@math.cas.cz

Abstract. We consider the problem of scheduling a sequence of tasks in a multi-processor system with conflicts. Conflicting processors cannot process tasks at the same time. At certain times new *tasks* arrive in the system, where each task specifies the amount of work (processing time) added to each processor's workload. Each processor stores this workload in its *input buffer*. Our objective is to schedule task execution, obeying the conflict constraints, and minimizing the maximum buffer size of all processors. In the off-line case, we prove that, unless $P = NP$, the problem does not have a polynomial-time algorithm with a polynomial approximation ratio. In the on-line case, we provide the following results: (i) a competitive algorithm for general graphs, (ii) tight bounds on the competitive ratios for cliques and complete k-partite graphs, and (iii) a $(\Delta/2 + 1)$-competitive algorithm for trees, where Δ is the diameter. We also provide some results for small graphs with up to 4 vertices.

1 Introduction

We consider the problem of scheduling a sequence of tasks in a multi-processor system with conflicts. The term "conflict" refers to a situation where two or more processors share common resources that can only be accessed by one processor at any given time, e.g. specialized human operators, equipment (say, a printer or a phone line), materials, etc. Conflicting processors cannot process tasks at the same time. In other words, at any moment in time only a non-conflicting set of processors can be run simultaneously. At certain times new *tasks* arrive in the system, where each task specifies the amount of work (processing time) added to each processor's workload. Each processor stores this workload in its *input buffer*. Our objective is to schedule task execution, obeying the conflict constraints, and minimizing the maximum buffer size of all processors.

We model the multi-processor system as an undirected graph $G = (V, E)$, where V is the set of processors and the edges in E represent conflicts. The instance of the problem consists of a task sequence and a sequence of task arrival times. The sequence of task arrival times, denoted $\bar{t} = t^1, \ldots, t^m$, is a non-decreasing sequence of real numbers. The task that arrives at time t^k is given by a *task vector* $\boldsymbol{\tau}^k = (\tau_1^k, \ldots, \tau_n^k)$ of non-negative real numbers, and the whole task sequence $\boldsymbol{\tau}^1, ..., \boldsymbol{\tau}^m$ is denoted by $\bar{\boldsymbol{\tau}}$.

The state of the system at any time is fully described by the *load vector* $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$, where z_i denotes the amount of work stored in the input buffer of processor i. Upon the arrival of task $\boldsymbol{\tau}^k$, the state changes to $\boldsymbol{z} + \boldsymbol{\tau}^k$. In-between task arrivals, the processors execute the work stored in their buffers. At any time, an independent set of vertices I can be chosen for processing. If a processor i executes for σ time units during which no task arrives, it decreases its load from z_i to $\max\{z_i - \sigma, 0\}$.

A *schedule* is described by a sequence of independent sets I_j, $0 = 1, \ldots, m-1$ and an increasing sequence of times $\theta_0 = 0, \theta_1, \ldots, \theta_m$. During the time interval $[\theta_j, \theta_{j+1})$, the set of running processors is I_j. The load vector at any time is computed according to the rules above. A schedule is feasible for a given instance if $\theta_m > t^k$ and the load vector at time θ_m is the zero vector (i.e., all requests are processed). The *buffer size* of a schedule is the maximum of any coordinate in any load vector over all times. A schedule is called a *B-schedule* if it is feasible and its buffer size is at most B.

The buffer minimization problem can be now formulated as follows: given an instance $\langle G, \bar{t}, \bar{\boldsymbol{\tau}} \rangle$, where G is the conflict graph and $\bar{t}$, $\bar{\boldsymbol{\tau}}$ are the sequences of task arrival times and task vectors, find a B-schedule for $\langle G, \bar{t}, \bar{\boldsymbol{\tau}} \rangle$ with minimum buffer size B.

On-line buffer minimization. An on-line algorithm processes the workload from the buffers without knowledge of future tasks. In round k it learns the task vector $\boldsymbol{\tau}^k$ and determines a schedule that is identical with the current one at all times up to t^k and is feasible in case no further task arrives. This schedule is executed until the next task arrives, and then the process is repeated.

We evaluate on-line algorithms by comparing their buffer size to that of an optimal off-line algorithm. An on-line algorithm is *c-competitive* for a graph G if, for any task sequence that has a B-schedule, it constructs a feasible schedule with buffer size at most cB. The *competitive ratio of* G, denoted $\texttt{buf}(G)$, is the infimum over all c for which there is a c-competitive on-line algorithm on G.

We also consider a slightly weaker definition, where the optimal size of buffers is fixed in advance. Without loss of generality, $B = 1$. We say that an on-line algorithm is *weakly c-competitive* for G if, for any task sequence that has a 1-schedule, it constructs a feasible schedule with buffer size at most c. The *weak competitive ratio of* G, denoted $\texttt{buf}^-(G)$, is the infimum of all c for which there is a weak c-competitive on-line algorithm on G. Using the doubling technique to estimate the buffer size, it is quite easy to show that, for any graph G, the competitive ratio $\texttt{buf}(G)$ is at most 4 times the weak competitive ratio $\texttt{buf}^-(G)$.

Some algorithms are easier to describe if we allow them to process a convex combination of independent sets rather than a single set. For example, instead of rapidly switching between two conflicting processors, we can run each of them at half speed. This generalization does not change the competitive ratio.

Our results. We introduce the buffer minimization problem as a model for studying various scheduling problems with conflicts. We believe that our model faithfully captures essential properties of scheduling multiprocessing systems over long periods of time. Other objectives in multiprocessor scheduling include fairness, load balancing and makespan minimization. Schedules that minimize the buffer size also typically perform well with respect to these other measures.

Off-line buffer minimization is closely related to fractional graph coloring. In fact, in Section 2, we show that it is even harder, namely that it has no polynomial time approximation algorithm with worst case guarantee that is polynomially bounded in the number n of processors, unless $P = NP$.

For the on-line case, we provide the following results:

(i) For every graph G, its competitive ratio $\texttt{buf}(G)$ is finite (Section 3).

(ii) The clique K_n has competitive ratio $\texttt{buf}(K_n) = \texttt{buf}^-(K_n) = H_n$ where $H_n = \sum_{i=1}^{n} 1/i$ is the nth harmonic number. For any complete bipartite graph $K_{m,n}$, $\texttt{buf}(K_{m,n}) = \texttt{buf}^-(K_{m,n}) = 2$. Further for complete k-partite graphs the competitive ratio is between H_k and $H_{k-1} + 1$ (Section 4). These upper bounds are achieved by a simple greedy algorithm.

(iii) For trees we show that their competitive ratio is at most $\Delta/2 + 1$, where Δ is the tree diameter (Section 5).

(iv) Finally, we provide bounds on the competitive ratios for graphs with up to four vertices (Section 6). All these ratios are between 1.5 and 2.5.

Previous work. The general concept of conflicts is not new in scheduling; in fact, one could argue that the whole area of scheduling is about resolving various types of conflicts for access to limited resources. In the classical literature on scheduling, conflicts are modeled by precedence relations between jobs and machine environments.

A model similar to ours was studied by Irani and Leung [3, 2]. They also introduce a conflict graph, but in their work this graph represents conflicts between individual jobs (not processors), and the objective is to minimize makespan. They show that even for paths the competitive ratio is $\Omega(n)$. They also provide algorithms that are competitive for general graphs under some assumptions on the job arrival probabilities.

Similar problems to ours were also studied in relation to resource allocation in distributed systems, where the conflict graphs is often referred to as the *resource graph* [9]. For example, Bar-Noy et al [4] investigate resource allocation problems with the objective to minimize the average response time, and they provide some hardness and approximation results. The problems studied in [4] (see also [8] and the references in [4]) differ from ours in that they are *one-shot* resource allocation problems, while in our scenario we have a stream of tasks arriving over time. Our objective function is different as well. Finally, unlike in [4], where the allocation problems reduce to various color sum problems for graphs, the buffer minimization turns out to be closely related to fractional graph coloring.

Notation. States, or load vectors, will be denoted $\boldsymbol{z}$, $\boldsymbol{y}$, etc, possibly with superscripts. States of an on-line algorithm will be typically denoted by $\boldsymbol{a}$. $G = (V, E)$ denotes an undirected graph with n vertices. We will use the convention that $V = \{1, 2, \ldots, n\}$. By $N(v)$ we denote the set of neighbors of a vertex v in G. Given any non-negative vector $\boldsymbol{w} = (w_1, \ldots, w_n)$, and a vertex set $X \subseteq V$, we introduce the following notation:

$$\Sigma_X \boldsymbol{w} \;=\; \sum_{i \in X} w_i \quad \text{and} \quad \max\nolimits_X \boldsymbol{w} \;=\; \max_{i \in X} w_i$$

For any real number x, we define $[x]^+ = \max\{0, x\}$, and for a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ let $[\boldsymbol{x}]^+ = ([x_1]^+, \ldots, [x_n]^+)$. For two n-dimensional vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, we write $\boldsymbol{x} \leq \boldsymbol{y}$ to denote that $\boldsymbol{x}$ is component-wise less than or equal to $\boldsymbol{y}$.

2 Buffer Minimization and Fractional Chromatic Number

We now exploit the relationship between the fractional chromatic number and the buffer size to show that the minimum buffer size is hard to approximate.

As usual, the chromatic number of a graph G is denoted by $\chi(G)$, and its clique number by $\omega(G)$. Consider a *weight* vector $\boldsymbol{w}$ indexed by the vertices of V, with $w_j \geq 0$ denoting the weight of vertex $j \in V$. The *weighted fractional chromatic number* $\chi_f(G, \boldsymbol{w})$ is the optimal objective value of the following linear program, where we use I to denote independent sets in G:

$$\begin{aligned} \chi_f(G, \boldsymbol{w}) \; &= \min \textstyle\sum_I x_I \\ &\quad \text{s.t.} \; \textstyle\sum_{I \ni j} x_I \;=\; w_j \quad \text{for } j \in V \\ &\quad\qquad x_I \geq 0 \qquad\quad\; \text{for each } I \end{aligned} \tag{1}$$

If $\boldsymbol{w} = \mathbf{1}$ (the vector of all 1's) and if the variables x_I are restricted to integral values, then the optimal solution of the resulting integer program is just the ordinary chromatic number $\chi(G)$: Every independent set I with $x_I = 1$ forms a color class, the n equations in (1) enforce that every vertex is contained in some color class, and the objective is to use the minimum number of color classes.

The *fractional chromatic number* $\chi_f(G)$ equals $\chi_f(G, \mathbf{1})$. The chromatic number and the fractional chromatic number are closely related; see [1]. For instance, the chromatic number $\chi(G)$ is at most $\log|V|$ times $\chi_f(G)$.

The connection between the buffer scheduling problem and the fractional chromatic number is as follows: Suppose that G is the conflict graph, and that the current load vector is $\boldsymbol{w}$. Then the minimum time needed to empty all the buffers (without any new tasks arriving) equals $\chi_f(G, \boldsymbol{w})$. The variables x_I indicate the amount of time for which I should be run in this schedule.

Using this connection and amplifying the hardness of coloring over time, we show that the minimum buffer size cannot be efficiently approximated. Thus we should not hope that simple greedy-type (on-line or off-line) algorithms will have good worst case ratios on *all* graphs.

Theorem 2.1. *For any $c > 0$, the buffer minimization problem cannot be approximated within an $O(n^c)$ factor in polynomial time, unless $P = NP$.*

Proof. We use the following theorem of Lund and Yannakakis [6]: There is a $c > 0$ for which the fractional chromatic number $\chi_f(G)$ of an n-vertex graph G cannot be approximated within an $O(n^c)$ factor in polynomial time, unless $P = NP$. (Stronger results are known for coloring, but using them would not improve the theorem.)

Suppose that there exists a polynomial-time approximation algorithm $\mathcal{A}$ for buffer minimization with worst case guarantee an^c. We can use $\mathcal{A}$ to design a polynomial-time decision procedure $\mathcal{D}$ for the following problem: Given a graph $G = (V, E)$ and a positive integer F, decide whether $F < \chi_f(G)$ or $\chi_f(G) \leq 2F$ (when $F < \chi_f(G) \leq 2F$ then both answers are correct outputs). By repeatedly calling $\mathcal{D}$ for G and the values $F_j = 2^j$, we can sandwich $\chi_f(G)$ in polynomial time between two consecutive powers of two. This would yield a polynomial time 2-approximation algorithm for computing the fractional chromatic number, and then the result of Lund and Yannakakis [6] would imply $P = NP$.

How do we design this procedure $\mathcal{D}$? We construct a special instance of the buffer minimization problem on G. At every time jF, for $j = 0, 1, 2, \ldots, an^{c+1}$, a task $\mathbf{1}$ arrives. The size of the task sequence $\bar{t}, \bar{\boldsymbol{\tau}}$ is polynomially bounded in the size of G. We feed this instance $\langle G, \bar{t}, \bar{\boldsymbol{\tau}} \rangle$ into $\mathcal{A}$. Denote by $B^{\mathcal{A}}$ the buffer size computed by $\mathcal{A}$. If $B^{\mathcal{A}} \leq an^c$, return "$\chi_f(G) \leq 2F$", else return "$\chi_f(G) > F$".

To justify the correctness of $\mathcal{D}$, suppose first $B^{\mathcal{A}} \leq an^c$. The total work assigned to each node is an^{c+1}, and right after the last task arrives at time Fan^{c+1} the workload in all buffers of $\mathcal{A}$ is at most an^c. Thus in time Fan^{c+1} each node processed work at least $an^{c+1} - an^c$. Therefore $\chi_f(G) \leq Fan^{c+1}/(an^{c+1} - an^c) \leq 2F$ (without loss of generality, we assume $n \geq 2$).

On the other hand, if $B^{\mathcal{A}} > an^c$ then the optimal buffer size is greater than 1. Thus it is not possible to process the workload $\mathbf{1}$ in time F, and thus $\chi_f(G) > F$.

3 The On-line Problem for Arbitrary Conflict Graphs

Let $G = (V, E)$ be an arbitrary but fixed conflict graph. In this section we show that $\texttt{buf}(G) < \infty$. The main idea is the following. Since an on-line algorithm does not know the current state of the off-line algorithm, it tries to choose states that would be good for *every* possible state of the off-line algorithm.

For two states $\boldsymbol{z}$ and $\boldsymbol{z}'$, define $\pi(\boldsymbol{z}, \boldsymbol{z}') = \chi_f(G, [\boldsymbol{z} - \boldsymbol{z}']^+)$, that is, $\pi(\boldsymbol{z}, \boldsymbol{z}')$ is the minimum processing time needed to reach a state $\boldsymbol{z}'' \leq \boldsymbol{z}'$ from $\boldsymbol{z}$. A state $\boldsymbol{y}$ is called *off-line feasible* at time t, if there exists an off-line 1-schedule for all the requests seen till time t which is in state $\boldsymbol{y}$ at time t. A state $\boldsymbol{z}$ is called *α-universal* at time t if, for any off-line feasible state $\boldsymbol{y}$ at time t, $\pi(\boldsymbol{z}, \boldsymbol{y}) \leq \alpha$. In other words, from an α-universal state we can reach any possible off-line feasible state within α time units. An algorithm is *α-universal* if at each point in time its state is α-universal. From the definition of α-universality we immediately get the following lemma.

Lemma 3.1. *Any α-universal on-line algorithm is weakly $(\alpha + 1)$-competitive.*

A linear programming lemma. Let I range over the independent sets of G. We formulate a linear program with the following intended meaning of the variables: The vector $\boldsymbol{a}$ is a state of the on-line algorithm. In the proof we think of $\boldsymbol{a}$ as the vertex weights for which we seek an optimal fractional coloring. For any feasible solution, the variables x_I define a fractional coloring of $(G, \boldsymbol{a})$. The variables u_I describe a change of this coloring that gives a fractional coloring of $(G, \boldsymbol{a} - \boldsymbol{\varepsilon})$ in which the number of colors decreases by δ. Finally, d is a crucial parameter which says that if a color was used heavily in the original coloring, it is also used in the modified coloring. All variables except the u_I are non-negative.

Let W be the set of all tuples $(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta, d, \boldsymbol{u})$ satisfying the following linear constraints:

$$
\begin{aligned}
0 \leq \sum_{I \ni j} x_I \;&=\; a_j \quad \text{for each } j \in V \\
0 \leq \sum_{I \ni j} u_I \;&=\; \varepsilon_j \;\leq\; 1 \quad \text{for each } j \in V \\
0 \leq \sum_{I} u_I \;&=\; \delta \;\leq\; \chi_f(G, \mathbf{1}) \\
[u]_I^+ \leq x_I \;\text{ and }\; [u]_I^+ \;&\leq\; d \quad \text{for each } I
\end{aligned}
$$

The key step in the proof is to show that it is never necessary to use large values of d. This implies that if we have a coloring for the current weights and change the weights a little, we can also bound the change in the coloring. In the next lemma we give show that such a bound exists.

Lemma 3.2. *Define $f(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta) = \min\{d \mid (\exists \boldsymbol{u})(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta, d, \boldsymbol{u}) \in W\}$. There exists a constant D such that the function f is upper bounded by D on all points where it is defined (i.e., finite).*

Proof. The set of tuples $(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta, d)$ such that $(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta, d, \boldsymbol{u}) \in W$ for some $\boldsymbol{u}$ is a polytope, since this is simply a projection of W. Thus function f, being a value of a linear program, is piecewise linear on its domain. Moreover, its domain consists of a finite number of regions in which f is linear. Consequently, it is sufficient to verify that f is bounded on any infinite feasible ray (halfline) in variables $(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta)$.

Since feasible values of $\boldsymbol{\varepsilon}$ and δ are bounded, they are also bounded along any feasible ray. Since $\boldsymbol{a}$ and $\boldsymbol{x}$ are non-negative, they must be non-decreasing along any feasible ray.

Now take any two points $(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta)$ and $(\boldsymbol{w}', \boldsymbol{x}', \boldsymbol{\varepsilon}', \delta')$ on the same feasible ray, so that the second one is farther away from the origin of the ray. This means that $\boldsymbol{\varepsilon}' = \boldsymbol{\varepsilon}$, $\delta' = \delta$, $\boldsymbol{a}' \geq \boldsymbol{a}$, and $\boldsymbol{x}' \geq \boldsymbol{x}$. Let $(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta, d, \boldsymbol{u}) \in W$ and $(\boldsymbol{a}', \boldsymbol{x}', \boldsymbol{\varepsilon}, \delta, d', \boldsymbol{u}') \in W$ be the corresponding feasible vectors with the minimal values of d and d'. We claim that $(\boldsymbol{a}', \boldsymbol{x}', \boldsymbol{\varepsilon}, \delta, d, \boldsymbol{u}) \in W$ as well. We have $x'_I - u_I \geq$

$x_I - u_I \geq 0$ by the feasibility of the first vector. All the other constraints follow directly by the feasibility of one of the two vectors. By minimality of d', we have $d' \leq d$. Consequently, along any infinite ray, the minimal feasible value of d cannot increase.

The algorithm. Now we define an on-line algorithm as follows. Let $\alpha = qD$, where q is the number of independent sets and D is the constant from Lemma 3.2. Suppose that $\boldsymbol{a}$ is the current state of the algorithm, and let $\boldsymbol{x}$ be an optimal fractional coloring of $\boldsymbol{a}$. Process an arbitrary independent set I with $x_I > D$ for time $x_I - D$. If no such set remains and there exists a vertex with non-zero load, we process any set containing such a vertex.

Theorem 3.3. *For any graph $G = (V, E)$, $\texttt{buf}(G)$ is finite.*

Proof. We claim that the above algorithm is α-universal. By Lemma 3.1 it then follows that $\texttt{buf}^-(G) \leq \alpha + 1$, and using doubling we obtain $\texttt{buf}(G) \leq 4\alpha + 4$.

By the definition of the algorithm, if the load vector is non-zero, some non-trivial set is processed. Thus on any finite request sequence the zero load vector is eventually achieved and a feasible schedule is generated.

It remains to verify that the algorithm is in an α-universal state at any time. It is clearly true at time 0. When a task arrives, off-line and on-line loads change by the same amount. So if the algorithm was in an α-universal state right before a task arrival, it will also be in an α-universal state afterwards. Thus we only need to verify that the schedule remains in an α-universal state when processing an independent set during an interval when no task arrives.

If there is no set with weight larger than D in $\boldsymbol{x}$, we can reach $\mathbf{0}$ in time at most $qD = \alpha$, thus the current state and any following state (before arrival of the next task) is trivially α-universal.

The remaining case is when an independent set I with weight $w_I \geq D + \beta$ is processed for some time $\beta > 0$. Suppose that during this time the algorithm changes its state from $\boldsymbol{a}$ to $\boldsymbol{a}'$, while the adversary changes its state from $\boldsymbol{y}$ to $\boldsymbol{y}'$. We need to verify that $\pi(\boldsymbol{a}, \boldsymbol{y}) \leq \alpha$ implies $\pi(\boldsymbol{a}', \boldsymbol{y}') \leq \alpha$.

Trivially, $\pi(\boldsymbol{a}, \boldsymbol{y}') \leq \pi(\boldsymbol{a}, \boldsymbol{y}) + \pi(\boldsymbol{y}, \boldsymbol{y}') \leq \alpha + \beta$. To conclude the proof it is sufficient to show that there exists an optimal fractional coloring of $[\boldsymbol{a} - \boldsymbol{y}']^+$ with weight of I at least β. For if we have such a coloring and decrease the weight of I by β, we obtain a coloring of $[\boldsymbol{a}' - \boldsymbol{y}']^+$. Since $\boldsymbol{a}'$ is obtained from $\boldsymbol{a}$ by processing I for time β, this coloring of $[\boldsymbol{a}' - \boldsymbol{y}']^+$ has weight at most α.

Let $\boldsymbol{\varepsilon} = \boldsymbol{a} - [\boldsymbol{a} - \boldsymbol{y}']^+$, that is $\varepsilon_i = \min\{a_i, y_i'\}$ for each i. Choose $\boldsymbol{u}$ such that $\boldsymbol{x} - \boldsymbol{u}$ is an optimal fractional coloring of $\boldsymbol{a} - \boldsymbol{\varepsilon}$, and define $\delta = \chi_f(G, \boldsymbol{a}) - \chi_f(G, \boldsymbol{a} - \boldsymbol{\varepsilon})$. Clearly, $\mathbf{0} \leq \boldsymbol{\varepsilon} \leq \mathbf{1}$ and $\delta \leq \chi_f(G, \boldsymbol{\varepsilon}) \leq \chi_f(G, \mathbf{1})$. By inspection of the linear program, if d is sufficiently large then $(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \boldsymbol{u}, \delta, d) \in W$. Lemma 3.2 implies that there are $\boldsymbol{u}'$ and $d' \leq D$ for which $(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{\varepsilon}, \delta, d', \boldsymbol{u}') \in W$. Then $\boldsymbol{x} - \boldsymbol{u}'$ is also an optimal fractional coloring of $\boldsymbol{a} - \boldsymbol{\varepsilon}$ and $x_I - u_I' \geq x_I - d' \geq x_I - D \geq \beta$.

4 The On-line Problem on Complete k-Partite Graphs

At each step, a scheduling algorithm needs to determine an independent set I of processors that should execute their tasks. Algorithm GREEDY determines I in the most obvious way: it iteratively chooses the processor with highest load, and eliminates its neighbors. To define GREEDY formally, we need to be a bit careful, as the time is continuous and ties need to be appropriately resolved. Denote GREEDY's load vectors by $\boldsymbol{a}$. We view the computation as being divided into ϵ-steps, with $\epsilon \to 0$. At each such ϵ-step, GREEDY determines I as follows: Start with $I = \emptyset$. Iteratively pick v with maximum a_v, add v to I, and remove v and its neighbors from G. Stop when $G = \emptyset$. Then, for all $v \in I$, decrease a_v by ϵ. Consider a time interval $[t, t']$ in which no tasks are issued. Divide it into intervals of length ϵ. Determine the state of GREEDY at time t', and take its limit for $\epsilon \to 0$. For complete k-partite graphs this limit is always well-defined. For such graphs, if there are j color classes that contain a node with maximum buffer size, then GREEDY will process the buffers of all nodes in these color classes at speed $1/j$.

Throughout this section, by a *subgraph* X we mean the subgraph of G induced by X. By $\boldsymbol{z}$ we denote an off-line state, and by B the optimal buffer size. Since GREEDY does not depend on B, we can assume that $B = 1$.

Smooth subgraphs. If X is a vertex set, then denote $N(X) = \bigcup_{v \in X} N(v) - X$. We say that X is *smooth* if all vertices in X have the same neighbors outside X that is, $N(v) - X = N(X)$ for $v \in X$.

GREEDY's behavior on smooth subgraphs is easy to characterize. Suppose X is a smooth subgraph with neighborhood $L = N(X)$, and assume that there are no tasks issued between the current time t and some time $t' > t$. If $\max_X \boldsymbol{a} > \max_L \boldsymbol{a}$ then, when GREEDY chooses its independent set I, it will always pick at least one vertex from X, namely one that realizes the maximum $\max_X \boldsymbol{a}$, and it will not include any vertices from L. So $\max_X \boldsymbol{a}$ will keep decreasing while $\max_L \boldsymbol{a}$ will stay the same. On the other hand, if $\max_X \boldsymbol{a} \le \max_L \boldsymbol{a}$, then this inequality will remain true until time t', since, by the previous statement, for any choice of ϵ-steps, $\max_X \boldsymbol{a}$ cannot exceed $\max_L \boldsymbol{a}$ by more than ϵ.

Lemma 4.1. *Suppose $K \subseteq V$ is a smooth clique in G with $N(K) = L$. Then*

$$\Sigma_K \boldsymbol{a} \le \Sigma_K \boldsymbol{z} + |K| \max\nolimits_L \boldsymbol{a}. \tag{2}$$

Proof. That (2) is preserved when tasks are issued is obvious. So consider task execution. If $\max_K \boldsymbol{a} \le \max_L \boldsymbol{a}$ then $\Sigma_K \boldsymbol{a} \le |K| \max_K \boldsymbol{a} \le |K| \max_L \boldsymbol{a}$, so (2) holds. Further, if $\max_K \boldsymbol{a} = \max_L \boldsymbol{a}$, then this equality will be preserved throughout until the next task arrives. If $\max_K \boldsymbol{a} > \max_L \boldsymbol{a}$ then, at the next infinitesimal ϵ-step, the independent set I used by GREEDY contains exactly one node from K and is disjoint with L. Then the left-hand side of (2) decreases by ϵ and the right-hand side cannot decrease by more than ϵ.

Lemma 4.2. *Suppose that $X \subseteq V$ is a smooth, complete k-partite subgraph of G with color classes $J_1, \ldots, J_k$ and $N(X) = L$. Then*

$$\sum_{i=1}^{k} \max_{J_i}[\boldsymbol{a} - \boldsymbol{z}]^+ \le \max\{\, k \max_L \boldsymbol{a} \,,\, k-1 \,\} \tag{3}$$

Proof. Inequality (3) is preserved when tasks are issued, so it is sufficient to consider task execution. If $\max_X \boldsymbol{a} \le \max_L \boldsymbol{a}$, then the left-hand side is at most $\sum_{i=1}^{k} \max_{J_i} \boldsymbol{a} \le k \max_X \boldsymbol{a} \le k \max_L \boldsymbol{a}$. Further, if $\max_X \boldsymbol{a} = \max_L \boldsymbol{a}$ then this equality will remain true throughout task execution. Suppose now $\max_X \boldsymbol{a} > \max_L \boldsymbol{a}$. In the next ϵ-step GREEDY will use an independent set I such that $I \cap (X \cup L) = J_j$ for some J_j that maximizes $\max_{J_j} \boldsymbol{a}$. We have two sub-cases. If $a_v \le z_v$ for all $v \in J_j$ then $a_v \le 1$ for all $v \in X$. So the jth term on the left-hand side is 0 and the other terms are at most 1. Overall, the left-hand side is at most $k-1$. Finally, suppose that $a_v > z_v$ for some $v \in J_j$. All positive $a_v \in J_j$ will decrease by ϵ, decreasing the left-hand side by ϵ. There can only be one j' for which some $z_v \in J_{j'}$ decreases, increasing the left-hand side by at most ϵ. So the left-hand side cannot increase and the right-hand side does not change.

Theorem 4.3. *For the complete graph K_n, $\texttt{buf}(K_n) = \texttt{buf}^-(K_n) = H_n$.*

Proof. (Lower bound) The adversary strategy consists of phases. Before phase p starts, the following invariant holds: there is a set X of $n-p+1$ processors such that $z_i = 0$ for $i \in X$ and $\Sigma_X \boldsymbol{a} \ge (n-p+1)(H_n - H_{n-p+1})$. The adversary creates task 1 for processors $i \in X$ and waits for time $n-p$. The new buffers satisfy $\Sigma_X \boldsymbol{a}' \ge \Sigma_X \boldsymbol{a} + (n-p+1) - (n-p) = \Sigma_X \boldsymbol{a} + 1$. Pick j for which a'_j is minimum and let $X' = X - \{j\}$. Then $\Sigma_{X'} \boldsymbol{a}' \ge \frac{n-p}{n-p+1}(\Sigma_X \boldsymbol{a} + 1) \ge (n-p)(H_n - H_{n-p})$. The adversary can zero all z_i for $i \in I'$. Thus the invariant is preserved.

In phase n, the only processor in X will have workload at least $a_i = H_n - 1$ in the buffer, so after adding 1 to processor i, the workload will reach H_n.
(Upper bound) We prove that GREEDY is H_n-competitive. Order the processors so that $a_1 \ge a_2 \ge \ldots \ge a_n$. By (2), for each j we have $\sum_{i \le j} a_i \le j + j a_{j+1}$, where for $j = n$ we assume that $a_{n+1} = 0$. Multiply the jth inequality, for $j < n$, by $1/j(j+1)$, multiply the nth inequality by $1/n$, and then add all the inequalities together. We get $a_1 \le H_n$, and the upper bound follows.

Theorem 4.4. *If G is a complete k-partite graph then $\texttt{buf}(G) \le H_{k-1} + 1$.*

Proof. We prove that GREEDY is $(H_{k-1}+1)$-competitive. Let the color classes of G be $J_1, J_2, \ldots, J_k$. Let $A_i = \max_{J_i} \boldsymbol{a}$, for all i. Reorder the color classes so that $A_1 \ge A_2 \ge \ldots \ge A_k$. Then, (3) implies that $\sum_{i=1}^{j} A_i \le \max\{jA_{j+1}, j-1\} + j$ for each $j = 1, \ldots, k$. Pick the smallest $l \le k$ for which $lA_{l+1} \le l-1$. For $j = 1, \ldots, l-1$, multiply the jth inequality by $1/j(j+1)$, multiply the lth inequality by $1/l$, and then add the first l inequalities together. We get

$$\sum_{j=1}^{l-1} \frac{1}{j} \sum_{i=1}^{j} A_i - \sum_{j=1}^{l-1} \frac{1}{j+1} \sum_{i=1}^{j} A_i + \frac{1}{l} \sum_{i=1}^{l} A_i \le \sum_{j=1}^{l-1} \frac{1}{j+1} A_{j+1} + H_{l-1} + 1,$$

which yields $A_1 \leq H_{l-1} + 1 \leq H_{k-1} + 1$, and the theorem follows.

Our analysis of GREEDY in Theorem 4.4 is tight. For the lower bound, we use a complete graph with one edge missing, say $G = K_{k+1} - \{(1,n)\}$. Suppose that we have a configuration $\boldsymbol{a} = (\alpha, \beta, \beta, \ldots, \beta)$ and $\boldsymbol{z} = \mathbf{0}$, where $\alpha \leq \beta \leq 1 - 1/k$. Initially, $\alpha = \beta = 0$. Create task $(0, 1, 1, \ldots, 1)$, process for time $k-1$, then create task $(1, 0, 1, \ldots, 1)$, and process for time k. The adversary can zero all his buffers. At the end GREEDY will be in configuration $(\beta', \alpha', \beta', \ldots, \beta')$, where $\alpha' = \beta - (1 - 1/k)(1 - 1/k - \beta)$, $\beta' = \beta + (1 - 1/k - \beta)/k$. Note that $\alpha' \leq \beta' \leq 1 - 1/k$. We can now repeat the process with the nodes 1 and n switched. Thus in the limit, β will converge to $1 - 1/k$. Then we can use the strategy for K_k to increase the buffers size to $H_k + 1 - 1/k = H_{k-1} + 1$.

The next theorem shows that the upper bound achieved by GREEDY on complete k-partite graphs is tight within a small additive factor. Note that as a special case of this lower bound, we obtain that $\texttt{buf}^-(P_3) \geq 2$. The proof involves a somewhat tedious adversary argument, and is omitted in this abstract.

Theorem 4.5. *Consider a complete k-partite graph G in which μ of the independent sets in the k-partition consist of a single vertex, whereas the remaining $k - \mu$ independent sets all have at least two vertices. If $\mu = 0$, then* $\texttt{buf}^-(G) \geq H_{k-1} + 1$. *If $\mu \geq 1$, then* $\texttt{buf}^-(G) \geq H_{k-1} + (k - \mu)/(k-1)$.

5 The On-line Problem on Trees

In this section we prove that the strong competitive ratio for trees of diameter Δ is at most $1 + \Delta/2$. In particular, $\texttt{buf}(P_n) \leq (n+1)/2$.

Let $G = (V, E)$ be a graph and $\mathcal{A}$ an on-line algorithm for an induced subgraph H of G. The *greedy extension* of $\mathcal{A}$, denoted $\mathrm{GE}(\mathcal{A})$ is an algorithm for G that works like this: if $\mathcal{A}$ processes a set I at a given step, then $\mathrm{GE}(\mathcal{A})$ chooses greedily (that is, choosing nodes with largest buffers) a maximal independent set $J \subseteq V - I - N(I)$, and processes $I \cup J$.

For certain graphs G and its subgraphs H, we can estimate the relationship between the competitive ratios of $\mathcal{A}$ and $\mathrm{GE}(\mathcal{A})$.

Lemma 5.1. *Suppose that G is constructed from H by adding a number of new vertices of degree 1, each connected by a new edge to some vertex in H. Then* $\texttt{buf}(G) \leq \texttt{buf}(H) + 1$ *and* $\texttt{buf}^-(G) \leq \texttt{buf}^-(H) + 1$.

Proof. Let $\mathcal{A}$ be c-competitive on H (the proof for weak competitiveness is the same). Let $\mathcal{B}$ be $\mathrm{GE}(\mathcal{A})$. We can assume the buffer size is $B = 1$, since $\mathcal{B}$ does not use the information about the off-line buffer size, unless $\mathcal{A}$ does so.

By the definition of $\mathcal{B}$, its behavior on H is exactly the same as that of $\mathcal{A}$. In particular, the buffer size of any vertex of H is at most c at any time.

Consider $v \in G - H$, and let w be its unique neighbor in H. Let $\boldsymbol{a}$ and $\boldsymbol{z}$ be the states of $\mathcal{B}$ and the optimal off-line algorithm, respectively. We claim that the following invariant holds at all times:

$$a_v + a_w \leq z_v + z_w + c - 1. \tag{4}$$

This inequality must be true whenever $a_v = 0$, since otherwise, by putting load $1 - z_w$ at w we would contradict the c-competitiveness of $\mathcal{A}$ on G. If a new task arrives, both sides of (4) increase by the same amount, thus the inequality is preserved. If some independent set is processed for time ϵ while $a_v > 0$, then the left-hand side of (4) decreases by ϵ, by the definition of the algorithm B, and the right-hand side cannot decrease by more than ϵ. Thus (4) is preserved. Since $z_v, z_w \le 1$, we obtain $a_v \le a_v + a_w \le z_v + z_w + c - 1 \le c + 1$.

Theorem 5.2. *For any tree T with diameter Δ, $\mathtt{buf}(T) \le 1 + \Delta/2$. In particular, $\mathtt{buf}(P_n) \le (n+1)/2$.*

Proof. By iteratively adding leaves, in Δ steps we can obtain any tree with diameter 2Δ from K_1, and any tree with diameter $2\Delta + 1$ from K_2. The bound follows by iterating the lemma and noting that $\mathtt{buf}(K_1) = 1$ and $\mathtt{buf}(K_2) = 1.5$.

Note that Theorem 5.2 gives another 2-competitive algorithm for P_3: If there is any load on the middle vertex, run this vertex, otherwise run the two endpoints.

6 The On-line Problem on Small Graphs

In this section we discuss competitive ratios of the ten connected graphs with up to four vertices. By Theorem 4.3, the complete graphs K_1, K_2, K_3, and K_4 have competitive ratios 1, $\frac{3}{2}$, $\frac{11}{6}$, and $\frac{25}{12}$, respectively. By Theorems 4.4 and 4.5, the complete bipartite graphs $K_{1,2}$ ($= P_3$), $K_{1,3}$, and $K_{2,2}$ ($= C_4$) have competitive ratio 2. All these bounds are attained by algorithm Greedy. The corresponding weak competitive ratios are the same. Thus for all these graphs the problem is completely solved.

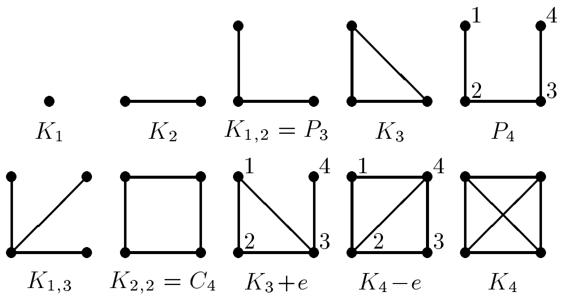

Fig. 1. The connected graphs with at most four vertices.

The three remaining graphs are the path P_4, the triangle plus an edge $K_3{+}e$, and "diamond graph" $K_4{-}e$, see Figure 1. Since all these graphs contain P_3, their weak competitive ratio is at least 2. For these graphs, we can prove the following bounds (the proofs will appear in the full version of the paper):

$$\begin{aligned} 2 \;&\le\; \texttt{buf}^-(P_4) \le \texttt{buf}(P_4) \;\le\; \tfrac{5}{2} \\ \tfrac{13}{6} \;&\le\; \texttt{buf}^-(K_3+e) \le \texttt{buf}(K_3+e) \;\le\; \tfrac{5}{2} \\ 2 \;&=\; \texttt{buf}^-(K_4-e) \le \texttt{buf}(K_4-e) \;\le\; \tfrac{5}{2} \end{aligned}$$

7 Final Comments

The main open problem is to establish tighter bounds on the competitive ratios for general graphs. The first step may be to either give a polynomial upper bound or a super-logarithmic lower bound, if any of these is possible.

For trees, we were unable to prove any lower bound better than 2. We suspect that there may be an algorithm for paths, and possibly for trees as well, with a constant competitive ratio (independent of n).

All algorithms we presented in the paper are *memoryless*, that is, they don't keep track of the past history. The behavior of such an algorithm depends only on its current buffer loads. We believe that algorithms that use information about possible adversary configurations can achieve better competitive ratios. Using the history, for any B, we can compute all possible adversary configurations that can be reached with buffer size up to B. However, the question of how to represent and use this information appears itself to be a difficult problem (in fact, we proved that maintaining this information is NP-hard). Perhaps, instead of keeping track of the whole history, it is sufficient to maintain only some lower bounds on the buffer sizes. It is quite easy to define lower bounds using complete subgraphs, for example.

A natural starting point for the above investigations would be to analyze the competitive ratios for small graphs, and for P_4 in particular. We made some progress in this direction, but many questions remain open. A complete analysis of small graphs would give good insight into the problem and may provide new ideas for the general case.

Acknowledgements. This research was partially supported by: NSF grant CR-9988360, grant A1019901 of GA AV ČR, postdoctoral grant 201/97/P038 of GA ČR, project LN00A056 of MŠMT ČR, START program Y43-MAT of the Austrian Ministry of Science, and a fellowship of the Pál Erdős Institute Budapest.

References

1. M. Grötschel, L. Lovász, and A. Schrijver [1981]. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica* 1, 169–197.
2. S. Irani and V. Leung [1997]. Probabilistic analysis for scheduling with conflicts, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 286–295.

3. S. Irani and V. Leung [1996]. Scheduling with conflicts, and applications to traffic signal control, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, 85–94.
4. A. Bar-Noy, M. Bellare, M.M. Halldórsson, H. Shachnai and T. Tamir [1998]. On chromatic sums and distributed resource allocation. *Information and Computation* 140, 183–202.
5. A. Bar-Noy and G. Kortsarz [1998]. Minimum color sum of bipartite graphs, *Journal of Algorithms* 28, 339-365.
6. C. Lund and M. Yannakakis [1994]. On the hardness of approximating minimization problems. *Journal of the ACM* 41, 960–981.
7. R. Motwani, S. Philips and E. Torng [1994]. Non-clairvoyant scheduling, *Theoretical Computer Science* 130, 17–47.
8. E. Kubicka and A.J. Schwenk [1989]. An introduction to chromatic sums, *Proc. ACM Computer Science Conference*, 39–45.
9. N.A. Lynch [1996]. Distributed Algorithms. Morgan Kauffman Publishers, San Francisco, California, 1996.