Priority Approximation for Batching

Wolfgang Bein *
Center for the Advanced Study of Algorithms
School of Computer Science
University of Nevada
Las Vegas, NV 89154
bein@cs.unlv.edu

John Noga
Department of Computer Science
California State University
Northridge, CA 91330
jnoga@csun.edu

Jeffrey Wiegley
Department of Computer Science
California State University
Northridge, CA 91330
jeffw@csun.edu

Abstract

We consider here the one-machine serial batching problem under weighted average completion. This problem is known to be $\mathcal{NP}$-hard and to date no approximation algorithm exists. Batching has wide application in manufacturing, decision management, and scheduling in information technology.

In this paper, we give the first approximation algorithm for the problem. The algorithm has an approximation ratio of 2 and is a priority algorithm, which batches jobs in decreasing order of priority. We also give a lower bound of $\frac{2+\sqrt{6}}{4} \approx 1.1124$ on the approximation ratio of any priority algorithm and conjecture that there is a priority algorithm which matches this bound. Adaptive algorithm experiments are used to support the conjecture. An easier problem is the list version of the problem where the order of the jobs is given. We give a new linear time algorithm for the list batching problem.

1 Motivation and Background

We consider the *batching problem* where a set of jobs $\mathcal{J} = \{J_i\}$ with processing times $p_i > 0$ and weights $w_i \geq 0$, $i = 1, \ldots, n$, must be scheduled on a single machine, and where $\mathcal{J}$ must be partitioned into *batches* $\mathcal{B}_1, \ldots, \mathcal{B}_r$. All jobs in the same batch are run jointly and each job's completion time is defined to be the completion time of its batch. We assume that when a batch is scheduled it requires a setup time $s = 1$. The goal is to find a schedule that minimizes the *sum of completion times* $\sum w_i C_i$, where C_i denotes the completion time of J_i in a given schedule. Given a sequence of jobs, a batching algorithm must assign every job J_i to a batch. More formally, a feasible solution is an assignment of each job J_i to the m_i^{th} batch, $i \in \{1, \ldots, n\}$.

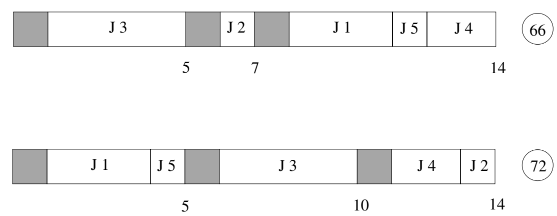

Figure 1. A Batching Example.

For example, Figure 1 shows two schedules for a 5-job problem where processing times are $p_1 = 3, p_2 = 1, p_3 = 4, p_4 = 2, p_5 = 1$ and the weights are $w_1 = w_4 = w_5 = 1$ and $w_2 = w_3 = 2$. (We note that the encircled values give the average weighted completion time of the depicted schedules.)

The problem considered in this paper has the jobs executed sequentially, thus the problem is more precisely referred to as the *s-batch* problem. We note that there is a different version of the problem not studied here, where the jobs of a batch are executed

*Research conducted while on sabbatical from the University of Nevada, Las Vegas. Sabbatical support from UNLV is acknowledged.

in parallel, known as the *p-batch* problem. In that case, the length of a batch is the maximum of the processing times of its jobs. The s-batch is also denoted in $\alpha|\beta|\gamma$ notation as the 1|s-batch|$\sum w_i C_i$ problem. Also, we will sometimes for convenience write the job data in the form $\{(p_1, w_1), (p_2, w_2), \ldots, (p_n, w_n)\}$. Thus in the example we would have written $\{(3, 1), (1, 2), (4, 2), (2, 1), (1, 1)\}$.

Brucker and Albers [1] showed that the 1|s-batch|$\sum w_i C_i$ problem is $\mathcal{NP}$-hard in the strong sense by giving a reduction from 3-PARTITION. There is a large body of work on batching problems (see e.g. [2, 3, 6, 7, 9, 12]) and batching has wide application in manufacturing (see e.g. [8, 16, 20]), decision management (see e.g. [14]), and scheduling in information technology (see e.g. [10]). More recent work on online batching is related to related to the TCP (Transmission Control Protocol) acknowledgment problem (see [4, 11, 13]).

In this paper, we give the first approximation algorithm for the 1|s-batch|$\sum w_i C_i$ problem. In fact, we give two algorithms, one called PseudoBatch, and the other one called CanonicalBest. For approximation algorithms, it is natural to consider the jobs according to the order of *priorities* $q_i = \frac{w_i}{p_i}$. If the jobs are renumbered such that $\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \ldots \geq \frac{w_n}{p_n}$, we say that the jobs are in *canonical order*. An algorithm that schedules the jobs in this order is called a *priority algorithm*. Both of our approximation algorithms are priority algorithms.

We recall that the quality of an approximation is measure in terms of its approximation ratio ρ: Given an optimization problem $\mathcal{P}$ we say that algorithm $\mathcal{A}_\mathcal{P}$ has approximation ratio ρ if for every instance $\pi \in \mathcal{P}$,

$$\rho \leq \frac{\text{cost of the solution given by } \mathcal{A}_\mathcal{P} \text{ for instance } \pi}{\text{cost of O}\text{PTIMUM for instance } \pi},$$

where Optimum is value of an optimal solution. We show that PseudoBatch and CanonicalBest both have approximation ratio $\rho = 2$. We also give a lower bound of $\frac{2+\sqrt{6}}{4} \approx 1.1124$ on the approximation ratio of any priority algorithm and conjecture that CanonicalBest matches this bound. Adaptive algorithm experiments are used to support the conjecture.

A much easier version of the problem is the *list* version of the problem where the order of the jobs is given, i.e., $m_i \leq m_j$ if $i < j$. For example, Figure 2 shows three schedules for a 5-job problem $\{(3, 1)(1, 1)(4, 1)(2, 1)(1, 1)\}$. The encircled values give the average weighted completion time of the schedules to the left.

Brucker and Albers [1] gave a linear time algorithm for the list batching problem. (Thus, to solve the

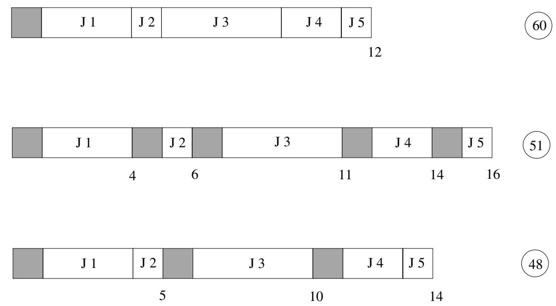

Figure 2. List Batching.

1|s-batch|$\sum w_i C_i$ problem, it is sufficient to know the order of jobs in the optimal solution.) We give an alternative algorithm in this paper. Our algorithm exploits the fact that the problem can be reduced to a shortest path problem, where the underlying cost matrix is a totally monotone matrix and thus can use the matrix searching algorithm of Larmore and Schieber [15] as a subroutine. A matrix A is called *totally monotone* if for all $i < i'$ and $j < j'$, $A[i, j] > A[i, j']$ implies $A[i', j] > A[i', j']$; matrix A is called *Monge* if $A[i, j] + A[i', j'] \leq A[i', j] + A[i, j']$. Clearly, every Monge matrix is totally monotone. We note that the linear time list batching algorithm is used to implement CanonicalBest in run time $O(n \log n)$.

Our paper is organized as follows: In Section 2 we give our priority approximation algorithms. Section 3 gives our alternate linear time algorithm for the list batching problem. Section 4 presents the lower bound on the approximation ratio of any priority algorithm. Section 5 describes genetic algorithm experiments. Specifically, we give an adaptive algorithm experiment which supports the conjecture that the approximation ratio of CanonicalBest matches the lower bound. This section also contains the description of a genetic algorithm for the 1|s-batch|$\sum w_i C_i$ problem implemented under GAlib, the object-oriented library of Matthew Wall [19] developed at MIT. We conclude with open problems in Section 6.

2 Approximation Algorithms

We give the following technical lemma, which is also known as the "Smith Rule" in the area of scheduling.

Lemma 1 *Given* $p_1, \ldots, p_n > 0$, $w_1, \ldots, w_n \geq 0$ *with* $\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \ldots \geq \frac{w_n}{p_n}$ *and permutation* π. *For* $i = 1, \ldots, n$ *let* $P_i^\pi = \sum_{j=1}^{i} p_{\pi(j)}$. *Then* $f_\pi = \sum P_i^\pi w_{\pi(i)}$ *is minimized when* π *is the identity.*

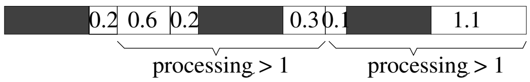

Figure 3. PSEUDOBATCH **for** $p_1 = 0.2, p_2 = 0.6, p_3 = 0.2, p_4 = 0.3, p_5 = 0.1, p_6 = 1.1$**.**

Proof: Consider permutation τ, which is not the identity. Then τ has an inversion $j > i$ with i immediately before j in τ. Let τ' be the permutation with i and j interchanged. We have

$$\begin{aligned} f_{\tau'} - f_\tau &= p_i w_i + (p_i + p_j) w_j - p_j w_j - (p_i + p_j) w_i \\ &= p_i w_j - p_j w_i \\ &\leq 0, \end{aligned}$$

since $\frac{w_i}{p_i} \geq \frac{w_j}{p_j}$. It follows that $f_{\tau'} \leq f_\tau$ and we are done. □

Lemma 2 *Let C_i be the completion times of an optimal schedule for the $1|s\text{-batch}|\sum w_i C_i$ problem. Then we have*

$$\sum_{i=1}^{n} w_i C_i \geq \sum_{i=1}^{n} w_i (P_i + 1)$$

Proof: Let permutation σ be the order of the optimal schedule. Then

$$P_i^\sigma + 1 \leq C_i.$$

Due to Lemma 1 we have

$$\sum_{i=1}^{n} w_i C_i \geq \sum_{i=1}^{n} w_i (P_i^\sigma + 1) \geq \sum_{i=1}^{n} w_i (P_i + 1).$$

□

We now present a simple, parameterized algorithm, PSEUDOBATCH, for the $1|\text{s-batch}|\sum w_i C_i$ problem. PSEUDOBATCH first reorders the jobs so that they are in canonical order. Then jobs are assigned to batches in that order. After receiving J_i, our algorithm has only two choices, namely whether to assign J_i to the same batch as J_{i-1} or not. We use the phrase "$\mathcal{A}$ batches at step i" to mean that algorithm $\mathcal{A}$ decides that J_i is the first job of a new batch, i.e. $m_i = m_{i-1} + 1$. We use the phrase "current batch" to denote the batch to which the last job was assigned. Then, when J_i is received, $\mathcal{A}$ must decide whether to add J_i to the current batch, or "close" the current batch and assign J_i to a new batch. PSEUDOBATCH maintains a variable P which will be the sum of the processing times of a set of recent jobs: we call this set the *current pseudobatch.* When J_1 is received, P is set to 0. After receiving each subsequent J_i, PSEUDOBATCH first adds p_i to P. If $P > 1$, PSEUDOBATCH batches and also sets P to zero. Thus, the i^{th} pseudo-batch contains all but the first member of the i^{th} batch, together with the first member of the $(i+1)^{st}$ batch, unless $i = r$. Every job except J_1 belongs to just one pseudo-batch.

Figure 3 gives an example for $p_1 = 0.2, p_2 = 0.6, p_3 = 0.2, p_4 = 0.3, p_5 = 0.1$ and $p_6 = 1.1$.

Theorem 1 PSEUDOBATCH *has an approximation ratio of* 2.

Proof: As before let C_i be the completion times of an optimal schedule for the $1|\text{s-batch}|\sum w_i C_i$ problem. Let $\hat{C}_i$ denote the completion times of the jobs when algorithm PSEUDOBATCH is run on the instance and let m be the number of batches created by the algorithm. Clearly we have

$$\hat{C}_i \leq P_i + m + 1$$

and

$$(m - 1) \leq P_i.$$

Thus,

$$\hat{C}_i \leq 2P_i + 2.$$

By Lemma 2 we have

$$\begin{aligned} \sum_{i=1}^{n} w_i \hat{C}_i &\leq 2 \sum_{i=1}^{n} w_i (P_i^\sigma + 1) \\ &\leq 2 \sum_{i=1}^{n} w_i C_i. \end{aligned}$$

□

Let CANONICALBEST be the algorithm which puts the jobs in canonical order and the linear time algorithm of the next section (or the algorithm of [1]) to get the optimal list batching schedule under the canonical order. Clearly we have:

Theorem 2 CANONICALBEST *has an approximation ratio of* 2.

Proof: Algorithm PSEUDOBATCH has approximation ratio of 2 and is a priority algorithm. Given an instance of the problem, algorithm CANONICALBEST produces a schedule with weighted average completion no worse than algorithm PSEUDOBATCH. □

3 An Alternate Linear Algorithm for List Batching

We now turn to the run time of Algorithm CANONICALBEST. As mentioned earlier, Brucker and Albers

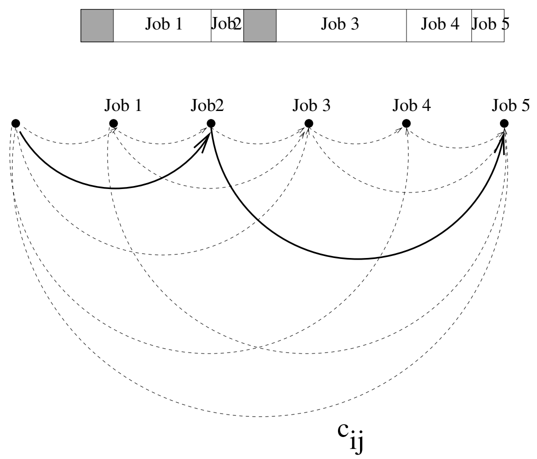

Figure 4. Reduction of the List Batching Problem to a Path Problem.

[1] gave a linear time algorithm for the list batching problem. We give here a simple alternate algorithm here. The algorithm is similar to their algorithm but it utilizes, as a subroutine, the well-known linear time matrix searching algorithm of Larmore and Schieber [15].

To this end let us assume that the jobs are $1, \ldots, n$ and are given in this order. One can then reduce the list batching problem to a shortest path problem in the following manner: Construct a weighted directed acyclic graph G with nodes $i = 1, \ldots, n$ (i.e. one node for each job) and add a dummy node 0. There is an edge (i, j) iff $i < j$. (See Figure 4 for a schematic.) Let edge costs $c_{i,j}$ for $i < j$ be defined as

$$c_{i,j} = (\sum_{\ell=i+1}^{n} w_\ell)(s + \sum_{\ell=1}^{j} p_\ell). \qquad (1)$$

We briefly note:

Lemma 3 *The matrix $C = (c_{i,j})$ defined in (1) is Monge for all choices of $p_i, w_i \geq 0$. Furthermore values can be queried in $O(1)$ time after linear preprocessing.*

Proof: Let $W_i = \sum_{\nu=1}^{i} w_\nu$ and $P_i = \sum_{\nu=1}^{i} p_\nu$ be the partial sum of the p_i and w_i values. Then we have

$$c[i, j] = c_{i,j} = (W_n - W_i)(s + P_j - P_i)$$

For $i < i'$ and $j < j'$

$$\begin{aligned} c[i, j] + c[i', j'] &- c[i', j] - c[i, j'] \\ &= (P_{j'} - P_j)(W_{i'} - W_i) \\ &\geq 0. \end{aligned}$$

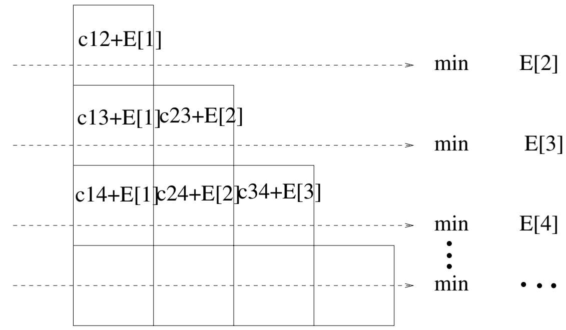

Figure 5. Dynamic Programming Tableau.

Furthermore notice that these values can be queried in $O(1)$ time after linear preprocessing by setting up arrays of partial sums for W_i and P_i by in linear time. $\square$

Returning now to the discussion of the reduction, it is easily seen (see [1] for details) that the cost of path $< 0, i_1, i_2, \ldots, i_k, n >$ gives the $\sum C_i w_i$ value of the schedule which batches at each job $i_1, i_2, \ldots, i_k$. Conversely, any batching with cost A corresponds to a path in G with with path length A.

A shortest path can be computed in time $O(n^2)$ using the following dynamic program:

Let

$$E[\ell] = \text{ cost of the shortest path from 0 to } \ell,$$

then

$$E[\ell] = \min_{1 \leq k \leq \ell} \{E[k] + c_{k,l}\} \text{ with } E[0] = 0, \qquad (2)$$

which results in a table, in which elements can be computed row by row (see Figure 5.)

In other words, the dynamic program computes the row minima of the $(n-1) \times (n-1)$ matrix E, where

$$E[\ell, k] = \begin{cases} E[k] + c[k, \ell] \text{ if } \ell < k \\ \infty \text{ else} \end{cases} \qquad (3)$$

with $\ell = 2, \ldots n$ and $k = 0, \ldots, n-1$.

Lemma 4 *The matrix $E = (E_{\ell,k})$ defined in (3) is Monge.*

Proof: Monge is preserved under addition and taking the minimum. $\square$

As it turns out all row minima can be computed in linear time using the algorithm of Larmore and Schieber, which is also known as the Larsch algorithm [15]. To execute the Larsch algorithm we need only that the matrix E satisfy the following conditions:

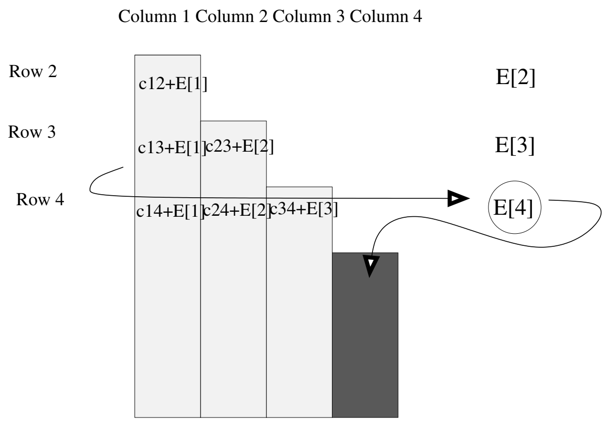

Figure 6. The "Online" Protocol of the Tableau.

1. For each row index ℓ of E, there is a column index γ_ℓ such that for $k > \gamma_\ell$, $E_{\ell,k} = \infty$. Furthermore, $\gamma_\ell \leq \gamma_{\ell+1}$.

2. If $k \leq \gamma_\ell$, then $E[\ell, k]$ can be evaluated in $O(1)$ time *provided that the row minima of the first ℓ rows are already known.*

3. E is a totally monotone matrix.

If these conditions are satisfied, the Larsch algorithm then calculates all of the row minima of E in $O(n)$ time. (See also [5]).

Condition 1 is clear from the fact that in matrix E all infinities are in the upper triangle of the matrix.

We turn to Condition 2. Condition 2 describes the online protocol underlying the computation of the dynamic programming tableau. It states that an element in column k is "knowable" once the row minimum of row k is revealed. Figure 6 illustrates this. For example, once the minimum of row 4 (that is, the value for $E[4]$) is known then the values of column 4 are available, since these values are of the form $E[4] + c[.,.]$. Furthermore, these values can be queried in $O(1)$ time due to Lemma 3.

Condition 3 follows from Lemma 4.

Since CanonicalBest requires the priorities to be sorted first, in summary we have:

Theorem 3 *Algorithm* CanonicalBest *has run time $O(n \log n)$.*

4 A Lower Bound for Priority Algorithms

We now give a lower bound of $\frac{2+\sqrt{6}}{4} \approx 1.1124$ on the approximation ratio of any priority algorithm. To show the lower bound, consider a problem which consists of exactly two jobs. There are two orders of the jobs possible. With each order there are two ways to batch the problem; either a single batch consisting of both items or two single batches, each containing one of the items. Since the order of jobs within a batch is irrelevant, two of the possibilities are identical and thus a total of three possibilities exist for Optimum, and CanonicalBest.

Given now a problem $\{(p_1, w_1), (p_2, w_2)\}$ we assuming equal priorities $\frac{w_1}{p_1} = \frac{w_2}{p_2} = 1$ it can be shown that $\{(p_1, w_1), (p_2, w_2)\} = \{(1, 1+\epsilon), (1+\sqrt{6}, 1+\sqrt{6})\}$ maximizes $C = \frac{C_{can}}{C_{opt}}$. We note the value ϵ, is used to break the tie and force CanonicalBest to choose an order different than Optimum; however, ϵ will tend to 0.

We use square brackets [ ] to denote a batch. Now consider:

$$\begin{aligned} & [\{(1, 1+\epsilon), (1+\sqrt{6}, 1+\sqrt{6})\}] \\ C &= (s + p_1 + p_2)(w_1 + w_2) \\ &= (1 + 1 + \epsilon + 1 + \sqrt{6})(1 + 1 + \sqrt{6}) \\ &= (3 + \sqrt{6} + \epsilon)(2 + \sqrt{6}) \\ &\approx 24.25 \end{aligned} \tag{4}$$

$$\begin{aligned} & [\{(1, 1+\epsilon)\}][\{(1+\sqrt{6}, 1+\sqrt{6})\}] \\ C &= (s + p_1)w_1 + (2s + p_1 + p_2)w_2 \\ &= (1 + 1 + \epsilon)1 + (2 + 1 + 1 + \sqrt{6})(1 + \sqrt{6}) \\ &= (2 + \epsilon) + (4 + \sqrt{6})(1 + \sqrt{6}) \\ &\approx 24.25 \end{aligned} \tag{5}$$

$$\begin{aligned} & [\{(1+\sqrt{6}, 1+\sqrt{6})\}][\{(1, 1+\epsilon)\}] \\ C &= (s + p_2)w_2 + (2s + p_2 + p_1)w_1 \\ &= (1 + 1 + \sqrt{6})(1 + \sqrt{6}) + \\ & \quad (2 + 1 + \sqrt{6} + 1)(1 + \epsilon) \\ &= (2 + \sqrt{6})(1 + \sqrt{6}) + (4 + \sqrt{6})(1 + \epsilon) \\ &\approx 21.80 \end{aligned} \tag{6}$$

Optimum chooses order (6) as it gives lowest cost possible. CanonicalBest is forced to choose between (4) or (5) due to the slight increase of the priorities of $(1, 1+\epsilon)$, caused by ϵ, ordering that job first. Both choices have the same cost. Thus:

$$C \geq \frac{C_{can}}{C_{opt}} \geq \frac{(3+\sqrt{6}+\epsilon)(2+\sqrt{6})}{(2+\sqrt{6})(1+\sqrt{6})+(4+\sqrt{6})(1+\epsilon)}$$

As ϵ tends to 0, we have

$$C \geq \frac{2+\sqrt{6}}{4} \approx 1.1124.$$

5 Adaptive Algorithm Experiments

In the previous section we have exhibited a solution which gives the worst approximation ratio for CanonicalBest when there are two jobs. The question remains whether a more difficult problem exists considering a larger number of jobs. We conjecture that this is not the case. In this section we will give results of computer experimentation that support this conjecture.

To explore problem spaces consisting of more than two jobs an evolutionary algorithm was developed. Individuals consisted of a single problem with an arbitrary number of jobs i, where $2 \leq i \leq 6$. Jobs were limited to six because the optimal solution was obtained by exhaustive search. For an individual, the number of jobs and the job data represent the genetic makeup of that individual. The fitness function $f(x)$ used to evaluate the suitability of individual x for inclusion in successive evolutions is simply the CanonicalBest competitive ratio of the individual's problem.

The algorithm is seeded with a single individual consisting of at least one job. Any individual with a single job will always have $f(x) = 1.0$; mutation is used to quickly produce individuals with two or more genes with $f(x) > 1.0$. The evolutionary environment sustains a total of $\mu = 50$ parents. Using a simple deterministic selection process, all pairs of parents were mated to produce $\lambda = \mu^2 = 2500$ offspring. Of the resulting population of $\mu + \lambda = 2550$, the strongest 50 were retained for the next generation.

In evolutionary algorithms, offspring are typically the product of mutation and crossover of their parents; however in our search we only relied on mutations. Mutations of a parent consisted of processing the parent's jobs (genes); any single job had a probability of $\phi_{mut} = 0.1$ of producing a mutation; otherwise it was simply copied to the child. Several types of mutations were applied. Mutations affected either the weight or processing time of a job.

- With probability 0.25 values could be scaled by a random amount with the intention of introducing random variability thereby landing on new neighborhoods of the landscape.
- Values could double (probability 0.25) or values could be halved (probability 0.25) in order to quickly converge on instances that relied on small or large values of a particular job in relation to others.
- With probability 0.25 values could also change by small amounts for the case that a job could be made harder by very small tweaks to the current design. (This is a bit similar to use of ϵ in Section 4).

Gen	$f(x)$	*Problem*
49	1.109481	$\{(1.01, 0.40), (3.20, 1.26)\}$
144	1.109964	$\{(1.01, 0.40), (3.20, 1.26)\}$
284	1.111070	$\{(0.99, 0.39), (3.20, 1.26)\}$
1969	1.111199	$\{(0.99, 0.39), (3.20, 1.26)\}$
5705	1.111338	$\{(0.99, 0.39), (3.20, 1.26)\}$
9283	1.111842	$\{(1.00, 0.40), (3.20, 1.26)\}$
10487	1.111890	$\{(1.00, 0.40), (3.20, 1.26)\}$
10971	1.111920	$\{(1.00, 0.40), (3.20, 1.26)\}$
17910	1.111956	$\{(1.00, 0.40), (3.20, 1.26)\}$
26522	1.111957	$\{(1.00, 0.40), (3.20, 1.26)\}$

Table 1. Improvements During Random Seeded Case.

Finally, with probability 0.1 we applied an additional change to the child, where the number of jobs was changed with probability 0.5 to increase and with probability 0.5 to decrease by one job. Decreases were performed by the deletion of a random job with equal probability. Addition of a job was introduced by either adding a random job, or by duplicating one of the already present jobs with equal probability. Change was avoided if it would produce job sizes outside of the allowable range.

When the algorithm is seeded with $\{(1.0, 1.0)\}$ the resulting evolution was uninteresting. At generation 5 an individual was arrived at consisting of $\{(1.00, 1.00), (4.00, 4.00)\}$, where $f(x) = 1.1111$. From this time up until at least generation 18813, μ consisted of 50 clones of this individual and appeared to remain stable despite the existence of a known, more difficult problem (Section 4). Similarly, when seeded with that problem, $\{(3.45, 3.45), (1.00, 1.00)\}$, μ immediately converged on 50 clones of this individual, with no variation at all, up until at least generation 20553. No other, more difficult, problem is found.

When seeded with a random job of $\{(0.33, 0.54)\}$ things were a bit more interesting. The early genera-

Parent 1	$184\underbrace{637}|25$	random slice 637
Parent 2	$3\underline{52}7\underline{18}|6\underline{4}$	add underlined
Child	$218\underline{637}|45$	child is a valid permutation

Figure 7. Example of an Ordered Crossover.

tions showed a more diverse μ consisting of six unique individuals; all with $f(x) = 1.0$. In generation 49 the individual of $\{(1.01, 0.40), (3.20, 1.26)\}$ was found with $f(x) = 1.109481$ and μ consisted of 50 copies. Later generations showed further small improvements. A sampling of that evolution is illustrated in Table 1.

All the improvements appear to be the result of the small tweaks mutations, other mutations failing to produce competitive individuals. No further improvement was found as of generation 72233.

We now turn to an unrelated implementation under GAlib, the object-oriented library of Matthew Wall [19] developed at MIT. Because it is possible to give an optimal schedule in linear time if the order is fixed (as described in Section 3) in the $1|\text{s-batch}|\sum w_iC_i$, it is natural to consider a genetic algorithm where the search space is the set of permutations. Then each individual's fitness – its weighted average completion – can be evaluated in linear time.

We have to define mutation and crossover. A mutation simply swaps two arbitrary elements of the permutation. For the crossover it is important to devise a mechanism that retains some features of the original two individuals in such a meaningful way that results in two new permutations. In the ordered crossover first used by Prins (see [17]), one takes a random subsequence of the first parent's permutation and insert it directly into the child. As described in Figure 7, the child is then completed by taking material from the second second parent's permutation, where elements are inserted into the child in the order they occur in that parent, starting after the second cut location, and ignoring elements already inserted from the first parent.

All experiments had the following parameters in common:

- Population Size: 1000
- Number of Generations: 5000
- Number of jobs: 100
- Crossover probability: 0.85
- Mutation probability: 0.005

The results were compared with the (conservative) lower bound of Lemma 2 and our results consistently gave solutions within a ratio of $\rho = 1.59$. The detailed results of these experiments are in [18].

6 Conclusions

In this paper we have given the first approximation algorithm for the $1|\text{s-batch}|\sum w_iC_i$ problem. We have also shown that no priority algorithm can have approximation ratio less than $\frac{2+\sqrt{6}}{4}$. As we have pointed out, we conjecture that algorithm CanonicalBest matches this lower bound. However, it is an interesting open research problem to prove the correctness of this conjecture.

Note that the lower bound of $\frac{2+\sqrt{6}}{4}$ holds only for priority algorithms. It is current research to investigate if there is a lower bound even if this assumption is dropped, or to give a polynomial approximation scheme in case such a lower bound does not exist.

We note that a version of the algorithm Pseudo-Batch is useful for online batching problems and we given results for the online case in [4].

Acknowledgments

Wolfgang Bein conducted this research while visiting the University of Texas at Dallas during sabbatical leave from the University of Nevada, Las Vegas. Sabbatical support from UNLV is acknowledged. The authors thank Charles Shields of the University of Texas at Dallas for valuable comments and suggestions.

References

[1] S. Albers and P. Brucker. The complexity of one-machine batching problems. *Discrete Applied Mathematics*, 47:87–10, 1993.

[2] P. Baptiste. Batching identical jobs. *Mathematical Methods of Operation Research*, 52:355–367, 2000.

[3] P. Baptiste and A. Jouglet. On minimizing total tardiness in a serial batching problem. *Operations Research*, 35:107–115, 2001.

[4] W. Bein, L. Epstein, L. Larmore, and J. Noga. Optimally competitive list batching. In *Algorithm Theory - SWAT 2004*, volume 3111 of *LNCS*, pages 77–89. Springer, 2004.

[5] W. Bein, M. Golin, L. Larmore, and Y. Zhang. The Knuth-Yao quadrangle-inequality speedup is a consequence of total-monotonicity. *Transactions on Algorithms*. to appear.

[6] P. Brucker. *Scheduling Algorithms*. Springer Verlag, 2004.

[7] P. Brucker, A. Gladky, H. Hoogeveen, M. Kovalyov, C. Potts, T. Tautenhahn, and S. van de Velde. Scheduling a batch processing machine. *Journal of Scheduling*, 1(1):31–54, 1998.

[8] P. Brucker and J. Hurink. Solving a chemical batch scheduling problem by local search. *Annals of Operations Research*, 96:17–38, 2000.

[9] P. Brucker, M. Y. Kovalyov, Y. M. Shafransky, and F. Werner. Batch scheduling with deadline on parallel machines. *Annals of Operations Research*, 83:23–40, 1998.

[10] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proceedings of the second ACM international conference on Multimedia*, pages 15–23. ACM, 1994.

[11] D. R. Dooly, S. A. Goldman, and S. D. Scott. On-line analysis of the TCP acknowledgment delay problem. *Journal of the ACM*, 48(2):243–273, 2001.

[12] J. A. Hoogeveen and A. P. A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *Proc. 5th Conf. Integer Programming and Combinatorial Optimization (IPCO)*, pages 404–414, 1996.

[13] A. R. Karlin, C. Kenyon, and D. Randall. Dynamic TCP acknowledgment and other stories about e/(e-1). *Algorithmica*, 36(3):209–224, 2003.

[14] R. Kuik, M. Salomon, and L. N. van Wassenhove. Batching decisions: structure and models. *European journal of operational research*, 75:243–263, 1994.

[15] L. Larmore and B. Schieber. On-line dynamic programming with applications to the prediction of rna secondary structure. *Journal of Algorithms*, 12:490–515, 1991.

[16] C. N. Potts and L. N. Van Wassenhove. Integrating scheduling with batching and lot-sizing: A review of algorithms and complexity. *The Journal of the Operational Research Society*, 43:395–406, 1992.

[17] C. Prins. Competitive genetic algorithms for the open-shop scheduling problem. Technical report, Ecole des Mines de Nantes, 1999.

[18] L. Raymond. Heuristics for batching jobs under weighted average completion time. Master's Thesis, University of Nevada, Las Vegas, 2006.

[19] M. Wall. *GAlib: A C++ Library of Genetic Algorithm Components*. Cambridge, Massachusetts, version 2.4 edition, 1996. http://lancet.mit.edu/ga/.

[20] G. Zhang, X. Cai, C. Y. Lee, and F. Wong. Minimizing makespan on a single batch processing machine with nonidentical job sizes. *Naval Research Logistics*, 48:226–240, 2001.