Application of the Cross-Entropy Method to the Buffer Allocation Problem in a Simulation-Based Environment

G. Allon, D.P. Kroese†, T. Raviv, R.Y. Rubinstein

Faculty of Industrial Engineering and Management, Technion, Haifa, Israel.

†*(Author to which proofs should be sent.) Department of Mathematics, University of Queensland, Brisbane 4072, Australia. Email: kroese@maths.uq.edu.au. Tel.: +61 733653287*

Abstract

The *buffer allocation problem* (BAP) is a well-known difficult problem in the design of production lines. We present a stochastic algorithm for solving the BAP, based on the *cross-entropy method*, a new paradigm for stochastic optimization. The algorithm involves the following iterative steps: (a) the generation of buffer allocations according to a certain random mechanism, followed by (b) the modification of this mechanism on the basis of cross-entropy minimization. Through various numerical experiments we demonstrate the efficiency of the proposed algorithm and show that the method can quickly generate (near-)optimal buffer allocations for fairly large production lines.

Keywords: Buffer allocation, cross-entropy method, stochastic optimization, production lines.

The BAP is a well-known problem in the design of production lines. The objective is to allocate n buffer spaces amongst the $m-1$ "niches" (storage areas) between m machines in a serial production line, so as to optimize some performance measure, such as the steady-state throughput. We will shortly give a more detailed description of the BAP, but for general references on production lines we refer to Adan and van der Wal (1989), Dallery *et al.* (1994), Glasserman and Yao (1996), Meester and Shanthikumar (1990) and Shanthikumar and Yao (1989). Buzacott and Shanthikumar Buzacott and Shanthikumar (1993) provide a good reference on stochastic modeling of manufacturing systems, while Gershwin and Schor Gershwin and Schor (2000) present a comprehensive summary paper on optimization of buffer allocation models.

There are two reasons why the BAP is a *difficult* optimization problem. The first reason is that, for a given buffer allocation, the exact value of the objective function – e.g., the steady-state throughput – is often difficult or impossible to calculate. In fact, complete knowledge of the objective function is only available for relatively small production lines in which the processing times have exponential or (simple) phase-type distributions, Gershwin and Schor (2000), Heavey *et al.* (1993). So, in a more general setting the BAP is typically a *noisy* or *simulation-based* optimization problem, i.e., an optimization problem in which the objective function needs to be estimated, e.g., via discrete-event simulation, Papadopoulos and Vouros (1997), Rubinstein and Melamed (1998).

The second reason why the BAP is difficult is that finding the optimum of the objective function, even if this function were completely known, comprises a combinatorial optimization problem over a potentially very large set with $\binom{n+m-2}{m-2}$ elements.

In this paper we present a new simulation-oriented approach to the BAP, based on the CE method. The CE method comprises a suite of techniques and algorithms for rare event simulation and combinatorial optimization, built around the notion of cross-entropy minimization. The method was first introduced in Rubinstein (1997) for the efficient estimation of rare event probabilities in stochastic networks, and was originally based on variance minimization, rather than on cross-entropy minimization. It was soon realized, Rubinstein (1999), that it could also be used for (approximately) solving complicated (e.g., NP-hard) combinatorial optimization problems (COPs). We wish to demonstrate that the CE method is well-suited for solving *noisy* optimization problems as well, and, in particular, that the CE method provides an easy and effective way to tackle the BAP.

For additional references on CE for COP see Helvik and Wittner (2001), Keith and Kroese (2002), Margolin (2002), Rubinstein (2002) - Rubinstein (2001) and the monograph Rubinstein and Kroese (2002), and for an application of simulated annealing to the BAP see Spinellis and Papadopoulos (2000). Alternative well-known heuristics for COP, capable of handling the BAP, are tabu search Glover and Laguna (1993), genetic algorithms Goldberg (1989), nested partitioning Shi and Olafsson (2000), Shi *et al.* (1999) and the Ant Colony Optimization (ACO) meta-heuristic of Dorigo and colleagues Caro and Dorigo (1998)-Dorigo

and Gambardella (1997), Gutjahr (2000b), Gutjahr (2000a).

The remainder of this paper is organized as follows. In Section 1 we give a detailed description of the BAP, providing the necessary definitions and assumptions. In Section 2 we explain the main ideas behind the CE method as a simulation-based tool for combinatorial optimization. Section 3 describes our main algorithm for the BAP. In Section 4 we present the results of various numerical experiments and in Section 5 we discuss the merits of the approach and potential directions for further research.

1 The buffer allocation problem

The basic setting of the BAP is the following. Consider a production line consisting of m machines in series, numbered $1, 2, \ldots, m$. Jobs are processed by all machines in consecutive order. The processing time at machine i has a fixed distribution with *rate* μ_i (hence the mean processing time is $1/\mu_i$), $i = 1, \ldots, m$. The machines are assumed to be unreliable, with exponential life- and repair times. Specifically, machine i has failure rate β_i and repair rate r_i, $i = 1, \ldots, m$. All life, repair and processing times are assumed to be independent of each other.

The machines are separated by $m-1$ storage areas, or *niches*, in which jobs can be stored. However, the total number of storage places, or *buffer* places, is limited to n. When a machine breaks down, this can have consequences for other machines up or down the production line. In particular, an up-stream machine could become *blocked* (when it cannot pass a processed job on to the next machine or buffer) and a down-stream machine could become *starved* (when no jobs are offered to this machine). We assume that a starved or blocked machine has the same failure rate as a "busy" machine. The first machine in the line is never starved and the last machine is never blocked.

The BAP deals with the question of how to optimally allocate the n buffer places amongst the $m-1$ niches. Here "optimally" refers to some performance measure of the flowline. Typical performance measures are the steady-state *throughput* and the expected amount of *work-in-process*. We shall only deal with the steady-state throughput.

Note that there are $\binom{n+m-2}{m-2}$ possible buffer allocations. An illustration of the definitions is given in Figure 1.

[Figure 1 about here.]

We will use the following mathematical formulation of the BAP. Each possible *buffer allocation* (BA) will be represented by a vector $\boldsymbol{x} = (x_1, \ldots, x_{m-1})$ in the set $\mathcal{X} := \{(x_1, \ldots, x_{m-1}) : x_i \in \{0, 1, \ldots, n\}, i = 1, \ldots, m-1, \sum_{i=1}^{m-1} x_i = n\}$. Here, of course, x_i represents the number of buffer spaces allocated to niche i, $i = 1, \ldots, m-1$.

For each buffer allocation $\boldsymbol{x}$ let $S(\boldsymbol{x})$ denote the steady-state throughput of the production line. Thus the BAP can be formulated as the optimization problem:

$$\text{maximize } S(\boldsymbol{x}) \text{ over } \boldsymbol{x} \in \mathcal{X}. \tag{1}$$

In case the steady-state output needs to be estimated, we have instead the *noisy* optimization problem:

$$\text{maximize } \hat{S}(\boldsymbol{x}) \text{ over } \boldsymbol{x} \in \mathcal{X}, \tag{2}$$

where $\hat{S}(\boldsymbol{x})$ is an estimate of $S(\boldsymbol{x})$.

2 Combinatorial Optimization via the CE-Method

Consider the following general maximization problem. Let $\mathcal{X}$ be a finite set of *states*, and let S be a real function on $\mathcal{X}$, the *score*. We wish to find the maximum of S over $\mathcal{X}$, and the corresponding state(s) at which this maximum is attained. For simplicity assume there is only one such state $\boldsymbol{x}^*$. Let us denote the maximum by γ^*. Thus,

$$S(\boldsymbol{x}^*) = \gamma^* = \max_{\boldsymbol{x} \in \mathcal{X}} S(\boldsymbol{x}) \; . \tag{3}$$

The starting point in the methodology of the CE method is to associate an *estimation problem* with the optimization problem (3). We thereto define a collection of functions $\{H(\cdot;\gamma)\}$ on $\mathcal{X}$, via

$$H(\boldsymbol{x};\gamma) = \begin{cases} 1 & \text{if} \quad S(\boldsymbol{x}) \geq \gamma, \\ 0 & \text{if} \quad S(\boldsymbol{x}) < \gamma \; , \end{cases}$$

for each $\boldsymbol{x} \in \mathcal{X}$ and *threshold* $\gamma \in \mathbb{R}$. Next, let $\{f(\cdot;\boldsymbol{v})\}$ be a family of probability mass functions (pmf's) on $\mathcal{X}$, parameterized by a real-valued parameter (vector) $\boldsymbol{v}$. We associate with (3) the problem of estimating the number

$$\ell_{\boldsymbol{v}}(\gamma) = \mathbb{P}_{\boldsymbol{v}}(S(\boldsymbol{X}) \geq \gamma) = \sum_{\boldsymbol{x}} H(\boldsymbol{x};\gamma) f(\boldsymbol{x};\boldsymbol{v}) = \mathbb{E}_{\boldsymbol{v}} H(\boldsymbol{X};\gamma), \tag{4}$$

where $\mathbb{P}_{\boldsymbol{v}}$ is a probability measure under which the random state $\boldsymbol{X}$ has pmf $f(\cdot;\boldsymbol{v})$; and $\mathbb{E}_{\boldsymbol{v}}$ denotes the corresponding expectation operator. We will call the estimation problem (4) the *associated stochastic problem* (ASP). To indicate how (4) is associated with (3), suppose for example that γ is equal to γ^*. In that case $\ell_{\boldsymbol{v}} = f(\boldsymbol{x}^*;\boldsymbol{v})$, which typically would be a very small number. A well-known technique for estimating such "rare-event" probabilities is *Importance Sampling* (IS), where we take a random sample $\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(N)}$ from a different pmf g on $\mathcal{X}$, and evaluate

$$\frac{1}{N} \sum_{k=1}^{N} H(\boldsymbol{X}^{(k)};\gamma) \, \frac{f(\boldsymbol{X}^{(k)};\boldsymbol{v})}{g(\boldsymbol{X}^{(k)})} \; , \tag{5}$$

which is an (unbiased) estimator of $\ell_{\boldsymbol{v}}(\gamma)$. In the special case where $\gamma = \gamma^*$ the best possible choice for g is such that g assigns *all* its probability mass to $\boldsymbol{x}^*$; the estimator then has zero variance. Similarly, if γ is *close* to the optimal γ^*, it is plausible that the optimal g should assign most of its probability mass close to $\boldsymbol{x}^*$. At this point the CE method becomes relevant, since it was specifically developed as a fast adaptive estimation method for finding the optimal IS "change of measure", i.e, g.

To explain how the CE method works for the efficient estimation of (4), consider again the estimator (5). It is well known (and not difficult to see) that the optimal (i.e., zero-variance) way to estimate $\ell_{\boldsymbol{v}}(\gamma)$ is to use the change of measure with pmf

$$g^*(\boldsymbol{x}) := \frac{H(\boldsymbol{x};\gamma) f(\boldsymbol{x};\boldsymbol{v})}{\ell_{\boldsymbol{v}}(\gamma)} \,. \tag{6}$$

The obvious difficulty is of course that this g^* depends on the unknown parameter $\ell_{\boldsymbol{v}}$. However, we can still try to choose an "optimal" pmf $f(\cdot,\tilde{\boldsymbol{v}})$ in the sense that the *distance* between this pmf and g^* is minimal. A particular convenient measure of "distance" between two pmf's g and h is the *Kullback-Leibler distance* or *cross-entropy*, which is defined as

$$\mathcal{K}(g,h) = \mathbb{E}_g \log \frac{g(\boldsymbol{X})}{h(\boldsymbol{X})} = \sum_{\boldsymbol{x}} g(\boldsymbol{x}) \log g(\boldsymbol{x}) - \sum_{\boldsymbol{x}} g(\boldsymbol{x}) \log h(\boldsymbol{x}) \,.$$

For estimating (4) we choose the parameter $\tilde{\boldsymbol{v}}$ such that $\mathcal{K}(g^*, f(\cdot;\tilde{\boldsymbol{v}}))$, with g^* given in (6), is minimal. It is easy to see that $\tilde{\boldsymbol{v}}$ should be such that

$$\mathbb{E}_{\boldsymbol{v}}\, H(\boldsymbol{X};\gamma) \log f(\boldsymbol{X};\tilde{\boldsymbol{v}}) \tag{7}$$

is *maximal*. An important aspect of this approach (to find the "near-optimal" change of measure via CE minimization) is that the parameter (vector) can often be calculated *analytically*. In particular, for discrete random vectors $\boldsymbol{X}$ the components of $\tilde{\boldsymbol{v}}$ will always be of the form

$$\frac{\mathbb{E}_{\boldsymbol{v}}\, H(\boldsymbol{X};\gamma) I_{\{\boldsymbol{X}\in A\}}}{\mathbb{E}_{\boldsymbol{v}}\, H(\boldsymbol{X};\gamma) I_{\{\boldsymbol{X}\in B\}}}, \tag{8}$$

where $I_{\{\boldsymbol{X}\in A\}}$ and $I_{\{\boldsymbol{X}\in B\}}$ are indicator random variables and $A \subset B \subset \mathcal{X}$. This number typically needs to be estimated. For this we can use the estimator

$$\frac{\sum_{k=1}^N H(\boldsymbol{X}^{(k)};\gamma)\, I_{\{\boldsymbol{X}^{(k)}\in A\}}}{\sum_{k=1}^N H(\boldsymbol{X}^{(k)};\gamma)\, I_{\{\boldsymbol{X}^{(k)}\in B\}}} \,, \tag{9}$$

where $\boldsymbol{X}^{(1)},\ldots,\boldsymbol{X}^{(N)}$ is a random sample from the pmf $f(\cdot;\boldsymbol{v})$. However, it is important to note that the estimator above is only of practical use when the numerator and the denominator in (9) are non-zero. This means for example that when γ is close to γ^*, $\boldsymbol{v}$ needs to be such that $\mathbb{P}_{\boldsymbol{v}}(S(\boldsymbol{X}) \geq \gamma)$ is not too small. Thus, the choice of $\boldsymbol{v}$ and γ in (4) are closely related. On the one hand we would like to choose γ as close as possible to γ^*, and find (an estimate) of $\tilde{\boldsymbol{v}}$ via the procedure above, which assigns almost all mass to state(s)

close to the optimal state. On the other hand, we would like to keep γ relatively small in order to obtain a viable estimator for $\tilde{\boldsymbol{v}}$.

The idea is now to construct a *sequence* of parameter (vectors) $\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots$, and thresholds $\gamma_0, \gamma_1, \ldots$ such that $\{\gamma_k\}$ converges to a value γ_∞ close to the optimal γ^* and $\{\boldsymbol{v}_k\}$ converges to a parameter $\boldsymbol{v}_\infty$ such that the corresponding pmf assigns high probability mass to the collection of states that give a score.
This strategy is embodied in the following procedure, see e.g., Rubinstein (1999):

Algorithm 2.1 (CE algorithm for Combinatorial Optimization)

Start *with some* $\boldsymbol{v}_0$. *Let* $k = 0$.

Repeat

1. *Draw a random sample* $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$ *from* $f(\cdot, \boldsymbol{v}_k)$.

2. *Calculate the scores* $S(\boldsymbol{x}^{(i)})$ *for all* i, *and order them from biggest to smallest,* $s_1 \geq \ldots \geq s_N$. *Let* $[\rho N]$ *be the integer part of* ρN. *Define* $\gamma_k = s_{[\rho N]}$.

3. *Define* $\boldsymbol{v}_{k+1}$ *as the estimate of the optimal* $\tilde{\boldsymbol{v}}$ *in* (7) *with* $\boldsymbol{v} = \boldsymbol{v}_k$. *Thus, the components of* $\boldsymbol{v}_{k+1}$ *are found from* (9). *Increase* k *by 1.*

Until *convergence is reached.*

Note that the stopping criterion, the initial state $\boldsymbol{v}_0$, the sample size N and the number ρ (typically between 0.01 and 0.1) have to be specified in advance, but that for the rest the algorithm is "self-tuning".

In many applications, the sequence of pmf's $f(\cdot; \boldsymbol{v}_0), f(\cdot; \boldsymbol{v}_1), \ldots$ converges, or is numerically observed to converge, to a degenerate measure (Dirac measure), assigning all probability mass to a single state $\boldsymbol{x}_\infty$, for which, by definition, the function value is greater than or equal to γ_∞. For convergence results and proofs we refer to Lieber (1998), Margolin (2002) and Rubinstein (1999).

The above procedure can, in principle, be applied to any maximization problem. However, for each individual problem two essential ingredients need to be supplied.

1. We need to specify how the samples are generated. In other words, we need to specify the family of pmf's $\{f(\cdot; \boldsymbol{v})\}$.

2. We need to provide explicit updating rules for the parameters, based on cross-entropy minimization.

In general there are many ways to generate samples from $\mathcal{X}$, and it is not always immediately clear which way of generating the sample will yield better results or easier updating formulas.

3 Main algorithm

In this section we specify the main algorithm for the buffer allocation problem, based on the CE algorithm.

Consider the BAP (1). In order to apply the CE algorithm we need to specify (a) how to generate random buffer allocations, and (b) how to update the parameters at each iteration. The easiest way to explain how the random buffer allocations are generated and how the parameters are updated is to relate (1) to an *equivalent* maximization problem. Specifically, let $\tilde{\mathcal{X}} = \{(x_1, \ldots, x_{m-1}) : x_i \in \{0, 1, \ldots, n\}\}$, and define the function $\tilde{S}$ on $\tilde{\mathcal{X}}$ such that $\tilde{S}(\boldsymbol{x}) = S(\boldsymbol{x})$, if $\boldsymbol{x} \in \mathcal{X}$ and $\tilde{S}(\boldsymbol{x}) = -\infty$, otherwise. Then, obviously (1) is equivalent to the maximization problem

$$\text{maximize } \tilde{S}(\boldsymbol{x}) \text{ over } \boldsymbol{x} \in \tilde{\mathcal{X}}. \tag{10}$$

A simple method to generate a random vector $\boldsymbol{X} = (X_1, \ldots, X_{m-1})$ in $\tilde{\mathcal{X}}$ is to independently draw $X_1, X_2, \ldots, X_{m-1}$ according to fixed distributions $(p_{i0}, \ldots, p_{in})$, $i = 1, \ldots, m-1$. We can amalgamate the p_{ij} into the $(m-1) \times (n+1)$-matrix $P := (p_{ij})$. Note that the rows of P sum up to 1. The pmf $f(\cdot; P)$ of $\boldsymbol{X}$ is thus parameterized by the matrix P and given by

$$f(\boldsymbol{x}; P) = \prod_{i=1}^{m-1} \sum_{j=0}^{n} p_{ij}\, 1_{\{\boldsymbol{x} \in \tilde{\mathcal{X}}_{ij}\}},$$

where $\tilde{\mathcal{X}}_{ij} = \{\boldsymbol{x} \in \tilde{\mathcal{X}} : x_i = j\}$. The updating rules for this modified optimization problem follow from the maximization of (7) (where H refers to $\tilde{S}$ and not to S), under the condition that the rows of P sum up to 1. Using Lagrange multipliers $u_1, \ldots, u_{m-1}$ we obtain the maximization problem

$$\max_{\tilde{P}, u_1, \ldots, u_{m-1}} \left[\mathbb{E}_P H(\boldsymbol{X}; \gamma) \log f(\boldsymbol{X}; \tilde{P}) + \sum_{i=1}^{m-1} u_i \left(\sum_{j=0}^{n} \tilde{p}_{ij} - 1 \right) \right] .$$

Differentiating with respect to $\tilde{p}_{ij}$, yields, for all $j = 0, \ldots, n$,

$$\mathbb{E}_P \frac{H(\boldsymbol{X}; \gamma) I_{\{\boldsymbol{X} \in \tilde{\mathcal{X}}_{ij}\}}}{\tilde{p}_{ij}} + u_i = 0 .$$

Summing over $j = 0, \ldots, n$ gives $\mathbb{E}_P H(\boldsymbol{X}; \gamma) = -u_i$, so that

$$\tilde{p}_{ij} = \frac{\mathbb{E}_P H(\boldsymbol{X}; \gamma) I_{\{\boldsymbol{X} \in \tilde{\mathcal{X}}_{ij}\}}}{\mathbb{E}_P H(\boldsymbol{X}; \gamma)} .$$

This is of the form (8). The corresponding estimator, as in (9), is

$$\frac{\sum_{k=1}^{N} I_{\{\tilde{S}(\boldsymbol{X}^{(k)}) \geq \gamma\}}\, I_{\{\boldsymbol{X}^{(k)} \in \tilde{\mathcal{X}}_{ij}\}}}{\sum_{k=1}^{N} I_{\{\tilde{S}(\boldsymbol{X}^{(k)}) \geq \gamma\}}} . \tag{11}$$

This has a very simple interpretation. We simply count how many of the $\boldsymbol{X}^{(i)}$ have a function value greater than γ and of those we count how many have their ith coordinate equal to j. Dividing this last number by the former gives the updated value for p_{ij}.

This is how we could, *in principle*, carry out the sample generation and parameter updating for problem (10). We first generate X_1 from the first row of P, then independently generate X_2 from the second row of P, etcetera. And, for a sample of size N, we use updating formula (11). However, *in practice*, we would never generate the vectors in this way, since the majority of these vectors would be irrelevant (their components would not sum up to n, and therefore their $\tilde{S}$ values would be $-\infty$). In order to avoid the generation of irrelevant vectors, we proceed as follows.

Algorithm 3.1 (Generation of buffer allocations) .

Generate a random permutation $(\pi_1, \ldots, \pi_{m-1})$ *of* $\{1, \ldots, m-1\}$.
Let $k = 0$
For $i = 1, \ldots, m-1$
    Let $t = \sum_{j=0}^{n-k} p_{\pi_i, j}$
    For $j = 0, \ldots, n-k$ *let* $p_{\pi_i, j} = p_{\pi_i, j}/t$
    For $j = n-k+1, \ldots, n$ *let* $p_{\pi_i, j} = 0$
    Generate X_{π_i} *according to* $(p_{\pi_i, 0}, \ldots, p_{\pi_i, n})$.
    Let $k = k + X_{\pi_i}$
End

Algorithm 3.1 is further illustrated in Figure 2.

[Figure 2 about here.]

The updating formula remains the same, of course. But since we only generate vectors in $\mathcal{X}$, the updated value for p_{ij} can be estimated as

$$\frac{\sum_{k=1}^{N} I_{\{S(\boldsymbol{X}^{(k)}) \geq \gamma\}} \, I_{\{\boldsymbol{X}^{(k)} \in \mathcal{X}_{ij}\}}}{\sum_{k=1}^{N} I_{\{S(\boldsymbol{X}^{(k)}) \geq \gamma\}}} \,. \tag{12}$$

which has the same "natural" interpretation as discussed for (11).

To complete the algorithm, we need to specify the initialization and stopping conditions. For the initial matrix P_0 we simply take all elements equal to $1/(n+1)$. The stopping criterion is based on the convergence of the sequence of matrices $P_0, P_1, \ldots$, which (see also Section 2) is found to converge to a *degenerate* matrix

P_∞, i.e., a matrix in which each row has exactly one 1 and n 0's. Specifically, the algorithm is terminated if for some integer c, e.g., $c = 5$,

$$\xi_k(i) = \xi_{k-1}(i) = \cdots = \xi_{k-c}(i), \quad \text{for all } i = 1, \ldots, m-1 \,, \tag{13}$$

where $\xi_k(i)$ denotes the index of the maximal element of the ith row of P_k.

Summarizing the results above, the main algorithm can be written as follows.

Algorithm 3.2 (Main Algorithm for the BAP)

Start *with P_0 such that all elements are equal to $1/(n+1)$. Let $k = 0$.*

Repeat

1. *Draw a random sample of buffer allocations $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$ according to Algorithm 3.1, with $P = P_k$.*
2. *Calculate the throughputs $S(\boldsymbol{x}^{(i)}), i = 1, \ldots, N$, and order these from biggest to smallest, $s_1 \geq \ldots \geq s_N$. Let $[\rho N]$ be the integer part of ρN. Define $\gamma_k = s_{[\rho N]}$.*
3. *Using the same sample, calculate $P_{k+1} = (P_{k+1,ij})$ as*

$$P_{k+1,ij} = \frac{\sum_{k=1}^{N} I_{\{S(\boldsymbol{x}^{(k)}) \geq \gamma\}} \, I_{\{\boldsymbol{x}^{(k)} \in \mathcal{X}_{ij}\}}}{\sum_{k=1}^{N} I_{\{S(\boldsymbol{x}^{(k)}) \geq \gamma\}}} \,.$$

Increase k by 1.

Until $\xi_k(i) = \xi_{k-1}(i) = \cdots = \xi_{k-c}(i)$, *for all i.*

For fast generation of the buffer allocations one can use the well-known *alias method*, similar as it is used in Rubinstein (1999). For the *noisy* BAP, i.e, problem (2), the only change in Algorithm 3.2 is that item 2. is replaced by

2'. *Find the estimates of the throughputs, $\hat{S}(\boldsymbol{x}^{(i)}), i = 1, \ldots, N$, and ....*

It is intuitively clear that the noisy BAP converges in some sense to the "deterministic" BAP problem, if we decrease the relative error of the estimated throughputs to 0. For more details on these convergence aspects we refer to Rubinstein (1999).

For the parameter values ρ and N in the algorithm we choose $0.01 < \rho < 0.1$ and $N = 2mn$. The explanation for the latter being that we have to estimate the components of the $(m-1) \times (n+1)$ matrices P_k, for which the number of replications is required to be in the order of nm.

Remark 3.1 Instead of updating the matrix P_k to P_{k+1} via formula (12) we could use a *smoothing* update procedure in which

$$P_{k+1} = \alpha\, Q_{k+1} + (1-\alpha)\, P_k, \tag{14}$$

where Q_{k+1} is the matrix derived via (12). Clearly for $\alpha = 1$ we have the original updating rule in Algorithm 3.2. We found empirically that a value of α between $0.7 \leq \alpha \leq 0.9$ gives the best results. The main reason why the smoothing update procedure performs better than the non-smoothed version is that it prevents the occurrences of 0's and 1's in the matrices P_k. In the non-smoothed version, once an entry of P_k is 0 or 1, it will remain so for all $P_\ell, \ell > k$, which is not desirable, especially in the early iterations.

4 Numerical results

To evaluate the effectiveness of Algorithm 3.2 we applied it to various test problems. Specifically, we applied Algorithm 3.2 to a suite of 70 test cases in Vouros and Papadopolous Vouros and Papadopoulos (1998). In all these cases the machine processing times have exponential or Erlang_2 distributions. Since, in addition, the life- and repair times are assumed to be exponentially distributed, we can in principle calculate the exact optimal buffer allocation and corresponding steady-state throughput for these systems, using Markov Chain theory, as described in Heavey *et al.* (1993). It should be noted, however, that the solutions are *in practice* only obtainable for relatively small n and m. In addition to the 70 test cases, we applied Algorithm 3.2 to various relatively large systems for which the "solutions" were not available from Vouros and Papadopoulos (1998). In this section we summarize the results on a selection of these test problems.

In all test cases below we set $\rho = 0.1$, took $c = 5$ in our stopping rule (13). We generated at each iteration $N = 2\,m\,n$ random buffer allocations and updated the parameter matrices P_k according to the smoothed updating rule (14), with $\alpha = 0.7$. Similar results where obtained with $0.05 \leq \rho \leq 0.2$ and $0.5 \leq \alpha \leq 0.95$. The algorithm was implemented in Matlab 5.2 without compilation and ran on an Intel Pentium III 500MHz processor. For a given buffer allocation we used the *batch means* method Rubinstein and Melamed (1998) to estimate the steady-state throughput, each simulation run starting with a sufficiently long warm-up period.

For each test case we generated 10 independent solutions via Algorithm 3.2, say $\gamma_\infty^{(i)}$, $i = 1, \ldots, 10$. These were compared with either the optimal solution (steady-state output) γ^*, or with the best known solution $\gamma^\dagger$. In the tables below, we use the following notation. The *percentage average relative error* of the 10 solutions is defined either as

$$\bar{\varepsilon} = \frac{1}{10}\sum_{i=1}^{10} \frac{\gamma^* - \gamma_\infty^{(i)}}{\gamma^*} \times 100\%, \qquad \text{or as} \qquad \bar{\varepsilon} = \frac{1}{10}\sum_{i=1}^{10} \frac{\gamma^\dagger - \gamma_\infty^{(i)}}{\gamma^\dagger} \times 100\%, \tag{15}$$

depending on whether the true optimal solution is known or not. Also, the term $\bar{\gamma}_\infty$ (which appears below) denotes the average of the 10 generated solutions, and ε_* and ε^* denote the worst and the best percentage relative error among the 10 generated solutions. Here, we take again $\gamma^\dagger$ instead of γ^* when the optimal solution is not known. Finally, BA denotes the optimal buffer allocation and $\overline{\text{IT}}$ and CPU denote the average total number of iterations needed before stopping and the average CPU time in seconds, respectively.

Tables 1, 2 and 3 present the results for a number of test cases in Vouros and Papadopoulos (1998). In particular, in Tables 1 and 2 we consider systems with exponential processing times with rates $\mu_i, i = 1, \ldots, m$, and in Table 3 we consider systems with Erlang$_2$ processing times, with rates $\mu_i, i = 1, \ldots, m$; thus, for each machine i the processing time consists of two exponential phases with rates $2\mu_i$. We recall that the machine life and repair times are assumed to be exponential with rates $\beta_i, i = 1, \ldots, m$ and $r_i, i = 1, \ldots, m$, respectively. We see that the allocations found by the CE method are very close to the exact optimal ones (γ^*) of Vouros and Papadopolous Vouros and Papadopoulos (1998).

[Table 1 about here.]

[Table 2 about here.]

[Table 3 about here.]

Tables 4 and 5 present the performance of Algorithm 3.2 for $m = 6$ and $m = 10$, respectively, with exponential processing times and different values of n. We could not compare the results of Tables 4 and 5 with any alternatives since to the best of our knowledge no case studies are available yet for such *relatively large systems*. We argue, however, that our results are accurate and reliable and could serve as case studies to compare different algorithms. Note also that $\gamma^\dagger$ in Tables 4 and 5 corresponds to our best solution obtained (on the basis of 10 different runs) for each fixed n.

We obtained similar accuracies for different processing time distributions (i.e., exponential, normal, Erlang, uniform and deterministic), provided $0.05 \leq \rho \leq 0.2$ and $0.5 \leq \alpha \leq 0.95$.

[Table 4 about here.]

[Table 5 about here.]

Dynamics

We illustrate the dynamics of the matrices P_k for a benchmark problem with 4 niches, 10 buffer spaces, normally distributed processing times with $\mu = 6, \sigma = 2$ and $N = 80$.

$$P_0 = \begin{pmatrix} 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 \\ 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 \\ 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 \\ 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 & 0.0909 \end{pmatrix}$$

$$\vdots$$

$$P_4 = \begin{pmatrix} 0.0002 & 0.0013 & 0.0139 & 0.4484 & 0.5349 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 \\ 0.0002 & 0.0002 & 0.0303 & 0.8226 & 0.1432 & 0.0014 & 0.0013 & 0.0002 & 0.0002 & 0.0002 & 0.0002 \\ 0.0002 & 0.0483 & 0.9410 & 0.0089 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 \\ 0.0014 & 0.6007 & 0.3913 & 0.0051 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 & 0.0002 \end{pmatrix}$$

$$\vdots$$

$$P_9 = \begin{pmatrix} 0.0000 & 0.0000 & 0.0038 & 0.0179 & 0.9783 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0001 & 0.9996 & 0.0003 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0001 & 0.9445 & 0.0554 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.9801 & 0.0199 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \end{pmatrix}.$$

It follows from the results above that starting from P_0 with the elements $\frac{1}{n+1} = \frac{1}{11} = 0.0909$ Algorithm 3.2 stopped after 9 iterations allocating 4, 3, 2, 1 buffer spaces to niches 1, 2, 3, 4, respectively.

5 Conclusions and directions for future research

This paper presents an application of the cross-entropy method to the buffer allocation problem. The proposed algorithm involves the generation of buffer allocations according to an auxiliary random mechanism, followed by an updating of the parameters of this mechanism, on the basis of the simulated performance of these buffer allocations. The updating mechanism, derived via cross-entropy minimization, is very simple and involves a sequence of (stochastic) matrices P_k which are (numerically) found to converge to a degenerate matrix, from which the optimal or near optimal BA is directly found.

Our numerical studies suggest that the proposed algorithm is fast and typically performs well, in the sense that in approximately 99% of the cases the relative error ε does not exceed 1%.

Further topics for investigation include (a) establishing convergence of Algorithm 3.2 for finite sampling (i.e., $N < \infty$) with emphasis on the complexity and the speed of convergence under the suggested stopping rules; (b) establishing confidence intervals (regions) for the optimal solution; (c) application of parallel optimization techniques to the proposed methodology; and (d) investigations regarding a further speed-up of the algorithm. With respect to (d), we note that initially the throughputs do not need to be estimated very accurately, since the procedure just needs a rough idea which buffer allocations are good or not. However, further-on in the procedure the accuracy needs to be increased to distinguish between competing "good" solutions. In the present test cases the same accuracy was used for all iterations, since the goal of this

paper was to show that for the "noisy" BAP high accuracy could be achieved within a reasonable time. For further studies it would be interesting to scrutize the role of and the relationship between the *optimization module*, which includes the random mechanism of generating candidate vectors (buffer allocations) and the corresponding updating rules, and the *evaluation module*, which determines the score or performance of each generated solution. In some elementary combinatorial optimization problems such as traveling salesman problem (TSP) and the min-cut problem the evaluation task is trivial; in more complex stochastic problems such as the BAP, exact evaluation can be hard to accomplish. In this paper the performance evaluation is simulation-based and is the main user of CPU time. Clearly the performances of our algorithm can be dramatically improved using more sophisticated evaluation and/or simulation mechanisms. For example, the well-known Gershwin's decomposition method, Gershwin and Schor (2000), provides a fast and, in many cases, accurate evaluation/approximation of the performances of serial production lines. Another example is to abort the simulation of systems with less promising buffer allocation once strong enough statistical evidence is gathered that the buffer allocation is sub-optimal.

The method presented in this paper can be adapted to non-serial production lines, such as assembly lines. For such problems the optimization mechanism remains unchanged, while the evaluation method (simulation) should be modified to reflect the new production configuration. A similar method could be used to optimize the safety stocks to be used in a supply chain or a distribution network.

Although in all of our examples a unique optimal solution exists, a general BAP could have multiple optimal solutions. Rubinstein Rubinstein (2001) addresses the issue of multiple optima for the TSP. The CE method was found to be quite reliable in finding one of the optimal solutions in a finite number of iterations, requiring a little more CPU time than in the case of a unique solution, due to "oscillation" effects, where the solution fluctuates between various optima before settling down to a particular optimum. For further reading on the cross-entropy method and noisy optimization we refer to the monograph Rubinstein and Kroese (2002).

References

Adan, I. and van der Wal, J. (1989). Monotonicity of the throughput in single server production and assembly networks with respect to the buffer sizes. In H. G. Perros and T. Altiok, editors, *Queueing Networks with Blocking*, pages 345–356. Elsevier Science.

Buzacott, J. A. and Shanthikumar, J. G. (1993). *Stochastic Models of Manufacturing Systems*. Prentice-Hall.

Caro, G. D. and Dorigo, M. (1998). AntNet: distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, **9**(317–365).

Dallery, Y., Liu, Z., and Towsley, D. (1994). Equivalence, reversibility, symmetry and concavity properties in fork/join queueing networks with blocking. *Journal of the ACM*, **41**(5), 903–942.

Dorigo, M. and Gambardella, L. M. (1997). Ant colony systems: A cooperative learning approach to the travelling salesman problem. *IEEE Transactions on Evolutionary Computation*, **1**(1), 53–66.

Gershwin, S. B. and Schor, J. E. (2000). Efficient algorithms for buffer space allocation. *Annals of Operations Research*, **93**, 117–144.

Glasserman, P. and Yao, D. D. (1996). Structured buffer-allocation problems. *Journal of Discrete Event Dynamic Systems*, **6**(9–42).

Glover, F. and Laguna, M. (1993). *Modern Heuristic Techniques for Combinatorial Optimization*, chapter Chapter 3: Tabu search. Blackwell Scientific Publications.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley.

Gutjahr, W. J. (2000a). A generalized convergence result for the graph-based ant system meta-heuristic. Technical Report 91-016, Dept. of Statistics and Decision Support Systems, University of Vienna, Austria.

Gutjahr, W. J. (2000b). A graph-based ant system and its convergence. *Future Generations Computing*, **16**, 873–888.

Heavey, C., Papadopoulos, H. Y., and Browne, J. (1993). The throughput rate of multistation unreliable production lines. *European Journal of Operation Research*, **68**, 69–89.

Helvik, B. E. and Wittner, O. (2001). Using the cross-entropy method to guide/govern mobile agent's path finding in networks. In *3rd International Workshop on Mobile Agents for Telecommunication Applications - MATA'01.*

Keith, J. and Kroese, D. P. (2002). Sequence alignment by rare event simulation. In *Proceedings of the 2002 Winter Simulation Conference*, pages 320–327, San Diego.

Lieber, D. (1998). *Rare-events estimation via cross-entropy and importance sampling.* Ph.D. thesis, William Davidson Faculty of Industrial Engineering and Management, Technion, Haifa, Israel.

Margolin, L. (2002). *Cross-Entropy Method for Combinatorial Optimization.* Master's thesis, The Technion, Israel Institute of Technology, Haifa.

Meester, L. E. and Shanthikumar, J. G. (1990). Concavity of the throughput of tandem queueing systems with finite buffer storage space. *Advances in Applied Probability*, **22**, 764–767.

Papadopoulos, H. T. and Vouros, G. A. (1997). A model management system (MMS) for the design and operation of production lines. *International Journal of production Research*, **35**(8), 2213–2236.

Rubinstein, R. Y. (1997). Optimization of computer simulation models with rare events. *European Journal of Operational Research*, **99**, 89–112.

Rubinstein, R. Y. (1999). The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability*, **2**, 127–190.

Rubinstein, R. Y. (2001). Combinatorial optimization, cross-entropy, ants and rare events. In S. Uryasev and P. M. Pardalos, editors, *Stochastic Optimization: Algorithms and Applications*, pages 304–358. Kluwer.

Rubinstein, R. Y. (2002). The cross-entropy method and rare-events for maximal cut and bipartition problems. *ACM Transactions on Modelling and Computer Simulation*, **12**(1), 27–53.

Rubinstein, R. Y. and Kroese, D. P. (2002). Lecture notes on the cross-entropy method. Manuscript.

Rubinstein, R. Y. and Melamed, B. (1998). *Modern Simulation and Modeling*. Wiley series in probability and Statistics.

Shanthikumar, J. G. and Yao, D. D. (1989). Monotonicity and concavity properties in cyclic queueing networks with finite buffers. In H. Perros and T. Altiok, editors, *Queueing Networks with Blocking*, pages 325–344. Elsevier Science.

Shi, L. and Olafsson, S. (2000). Nested partitioning method for global optimization. *Operations Research*, **48**(3), 390–407.

Shi, L., Olafsson, S., and Sun, N. (1999). New parallel randomized algorithm for traveling salesman problem. *Computers and Operations Research*, **26**, 371–394.

Spinellis, D. D. and Papadopoulos, H. T. (2000). Production Line Buffer Allocation: Genetic Algorithms Versus Simulated Annealing. *Annals of OR*, **93**(1), 373–384.

Vouros, G. A. and Papadopoulos, H. T. (1998). Buffer allocation in unreliable production lines using a knowledge based system. *Computer & Operation Research*, **25**(12), 1055–1067.

List of Figures

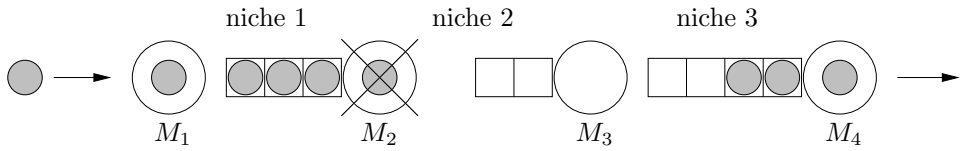

Figure 1: A production line with $m = 4$ machines. The total available buffer space is $n = 9$. The current buffer allocation is (3,2,4). Machine 1 has an infinite supply, but is currently blocked. Machine 2 has failed and is under repair. Machine 3 is starved. Machine 4 is never blocked.

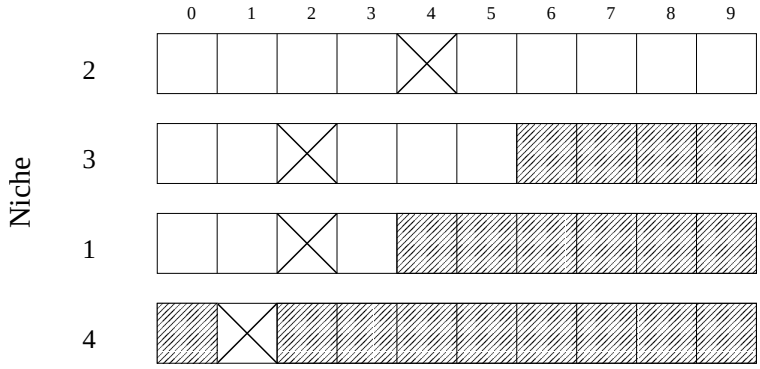

Figure 2: Generation of the BA vector $(2, 4, 2, 1)$, for the case $m = 5$, $n = 9$ and the permutation $\pi = (2, 3, 1, 4)$. For the second niche there are initially 9 possible buffer places; 4 buffer places are allocated. This reduces the number of available buffer places for the third niche to 5; 2 buffer places are allocated. Etcetera.

Table 1: Performance of Algorithm 3.2 for BAPs with $m-1=2$ niches and different values of n, exponential processing times with rates $\mu_1 = 1$, $\mu_2 = 1.2$, $\mu_3 = 1.4$, failure rates $\beta_i = 0.05$ and repair rates $r_i = 0.5, i = 1, \ldots, 3$.

n	$\overline{\text{IT}}$	BA	$\bar{\gamma}_\infty$	γ^*	$\bar{\varepsilon}$	ε_*	ε^*	CPU
1	2.0	(1,0)	.6341	.6341	0	0	0	7
2	2.0	(1,1)	.6715	.6744	0.44	0.88	0	8
3	2.6	(2,1)	.6998	.7113	1.64	5.90	0	9
4	3.5	(3,1)	.7349	.7361	0.16	0.54	0	14
5	3.8	(3,2)	.7574	.7587	0.18	0.59	0	14
6	4.3	(4,2)	.7688	.7777	0.37	2.11	0	16
7	6.2	(5,2)	.7811	.7922	0.52	1.71	0	22
8	5.1	(5,3)	.8040	.8060	0.25	0.84	0	20
9	9.1	(6,3)	.8142	.8178	0.44	1.63	0	32
10	8.3	(7,3)	.8255	.8274	0.24	0.95	0	30

Table 2: Performance of Algorithm 3.2 for $m-1=4$ niches, different values of n, exponential processing times with rates $\mu_1=1$, $\mu_2=1.1$, $\mu_3=1.2$, $\mu_4=1.3$, $\mu_5=1.5$, failure rates $\beta_i=0.05$ and repair rates $r_i=0.5, i=1,\ldots,5$.

n	$\overline{\text{IT}}$	BA	$\bar{\gamma}_\infty$	γ^*	$\bar{\varepsilon}$	ε_*	ε^*	CPU
1	2.6	(0,1,0,0)	.5213	.5213	0	0	0	12
2	4.6	(1,1,0,0)	.5479	.5514	0.60	1.10	0	32
3	3.6	(1,1,1,0)	.5824	.5824	0	0	0	39
4	6.4	(1,2,1,0)	.6015	.6027	0.20	0.85	0	67
5	9.0	(2,2,1,0)	.6202	.6213	0.18	0.32	0	103
6	5.7	(2,2,1,1)	.6420	.6422	0.03	0.31	0	89
7	7.7	(2,2,2,1)	.6572	.6585	0.20	0.87	0	116
8	7.2	(3,2,2,1)	.6731	.6744	0.20	1.20	0	132
9	9.1	(3,3,2,1)	.6885	.6894	0.13	0.67	0	166
10	10.7	(3,3,3,1)	.7004	.7005	0.02	0.03	0	197

Table 3: Performance of Algorithm 3.2 for $m-1=4$ niches, with Erlang_2 processing times with rates $\mu_1 = 1$, $\mu_2 = 1.1$, $\mu_3 = 1.2$, $\mu_4 = 1.3$, $\mu_5 = 1.5$, failure rates $\beta_i = 0.05$ and repair rates $r_i = 0.5$, $i = 1, \ldots, 5$.

n	$\overline{\text{IT}}$	BA	$\bar{\gamma}_\infty$	γ^*	$\bar{\varepsilon}$	ε_*	ε^*	CPU
1	2.8	(0,1,0,0)	.5968	.5968	0	0	0	23
2	3.5	(1,1,0,0)	.6331	.6338	0.11	1.14	0	39
3	3.9	(1,1,1,0)	.5824	.5824	0	0	0	55
4	5.8	(2,1,1,0)	.6802	.6808	0.09	0.73	0	86
5	8.3	(2,2,1,0)	.6985	.6996	0.16	0.28	0	159
6	6.9	(2,2,1,1)	.7180	.7195	1.14	0.2	0	187
7	12.5	(3,2,2,1)	.7335	.7341	0.18	0.07	0	202
8	9.8	(3,2,2,1)	.7496	.7501	0.43	0.07	0	181
9	9.7	(3,3,2,1)	.7620	.7627	0.68	0.09	0	177
10	13.6	(4,3,2,1)	.7714	7740	1.24	.33	0	261

Table 4: Performance of Algorithm 3.2 for $m-1=5$ niches and various n, exponential processing times with rates $\mu_1=8$, $\mu_2=11$, $\mu_3=14$, $\mu_4=14$, $\mu_5=11$, $\mu_6=8$, failure rates $\beta_i=0.05$ and repair rates $r_i=0.5$, $i=1,\ldots,6$.

n	$\overline{\text{IT}}$	BA	$\bar{\gamma}_\infty$	$\gamma^\dagger$	$\bar{\varepsilon}$	ε_*	ε^*	CPU
2	4.2	(1,0,0,0,1)	5.4935	5.5027	0.17	0.84	0	32.80
4	5.4	(1,0,0,0,1)	5.9245	5.9334	0.15	0.76	0	65.80
6	12	(1,1,0,1,1)	6.2443	6.2555	0.18	0.50	0	156.00
8	13.4	(2,1,0,1,2)	6.5197	6.5253	0.09	0.22	0	198.80
10	25.6	(3,1,1,1,2)	6.7510	6.7589	0.12	0.57	0	386.40
12	49	(4,2,1,2,3)	6.9316	6.9360	0.06	0.11	0	766.40
14	28.2	(4,2,1,2,5)	7.0684	7.0934	0.35	0.79	0	603.60
16	59.2	(5,2,2,2,5)	7.1783	7.1846	0.09	0.26	0	1128.60
18	92.6	(6,2,2,3,5)	7.4149	7.4291	0.19	0.36	0	2048.40

Table 5: Performance of Algorithm 3.2 for $m-1=9$ niches, exponential processing times with rates $\mu_1=8$, $\mu_2=8$, $\mu_3=11$, $\mu_4=14$, $\mu_5=14$, $\mu_6=11$, $\mu_7=8$, $\mu_8=8$, $\mu_9=6$, $\mu_{10}=6$, failure rates $\beta_i=0.05$ and repair rates $r_i=0.5$, $i=1,\ldots,10$.

n	$\overline{\text{IT}}$	BA	$\bar{\gamma}_\infty$	$\gamma^\dagger$	$\bar{\varepsilon}$	ε_*	ε^*	CPU
2	4.00	(0,0,0,0,0,0,0,1,1)	3.8281	3.8749	1.22	3.76	0	110.00
4	11.67	(0,0,0,0,0,0,1,1,2)	4.1160	4.1236	0.18	0.28	0	402.33
6	19.67	(0,1,0,0,0,0,1,2,2)	4.3220	4.3289	0.16	0.24	0	964.00
8	23.33	(0,1,0,0,0,1,1,2,3)	4.5325	4.5420	0.21	0.58	0	1199.67
10	12.67	(1,1,0,0,0,1,2,2,3)	4.6146	4.6426	0.60	1.84	0	1164.67
12	14.33	(1,1,0,0,0,1,2,3,4)	4.7814	4.7946	0.28	0.84	0	1718.67
14	37.00	(1,1,0,0,1,1,2,3,5)	4.8852	4.8895	0.08	0.20	0	3325.00
16	49.33	(1,1,0,1,0,2,2,4,5)	4.9832	4.9891	0.18	0.33	0	5117.00
18	186.67	(2,1,1,0,0,1,3,4,6)	5.0414	5.0638	0.45	1.17	0	20714.00