A Genetic Algorithm For Finding Good Balanced Sequences In A Customer Assignment Problem With No State Information

Wim Hordijk,
SmartAnalytiX.com,
Email: wim@WorldWideWanderings.net

Arie Hordijk,
Mathematical Institute, Leiden University,
Email: arie-hordijk@xs4all.nl

Bernd Heidergott (Principal corresponding author),
Tinbergen Institute,
and Department of Econometrics and Operations Research,
Vrije Universiteit Amsterdam,
Email: bheidergott@feweb.vu.nl

Abstract

In this paper we study the control problem of optimal assignment of tasks to servers in a multi-server queue with inhomogeneous servers. In order to improve the performance of the system, we use a periodic deterministic sequence of job assignments to servers called a *billiard sequence*. We then use a genetic algorithm (GA) for computing a near-optimal billiard sequence. By means of a recent result obtained in the area of ordinal optimization, we show that the solution found by the GA belongs to the top 1% of possible choices for such a billiard sequence. As illustrated by numerical examples, not only is the performance under a billiard sequence better than that of the corresponding randomized policy, the optimal billiard sequence even outperforms the billiard implementation of the optimal randomized policy. The framework we introduce in this paper is suitable

for general optimization problems over (periodic) deterministic decision sequences. Given the significant performance improvement that a switch from randomized policies to billiard sequences yields, this framework is of importance in practical applications. Finally, we show that constrained or multi-objective optimization can be dealt with in our framework as well.

Keywords: Control, Multi-Server Queue, Billiard Sequence, Genetic Algorithm, Ordinal Optimization

1 Introduction

Many real-world problems in the area of operations research, engineering, and computer science can be described by stochastic models that involve decision making. The main example in this paper will be the assignment of jobs to heterogeneous servers in a multi-server queuing system. Here, the decisions are on sending jobs to servers so that the overall stationary waiting time of jobs is minimized. This type of decision making can be formalized within the framework of Markov decision processes (MDP), see, e.g., Feinberg and Schwartz [14] and Puterman [26].

Typically, dynamic programming is used in MDP for computing optimal policies. This results in high-performance policies that depend on the state of the system. However, unfortunately the identification of effective policies via dynamic programming is often impractical, both analytically and numerically, for several reasons. First, the dimensionality of the state space can be very high so that standard dynamic programming algorithms such as value iteration or policy iteration for computing optimal policies become computationally too demanding. Second, many problems have constraints so that the optimal policy is usually found within the class of *randomized policies*, as opposed to the class

of deterministic policies, see e.g. [1]. This provides another reason why standard techniques fail, since they are not tailored to finding the optimal randomized policy. Third, even if powerful computers are used to solve difficult optimization problems, potentially overcoming the previous two issues, the drawback is that the resulting policy often does not have nice structural properties and is difficult to understand and implement, see [7]. These three issues form a serious bottleneck that is currently holding back the widespread implementation of MDPs in many decision support systems. Hence, there is a need for randomized policies that have good performance and yet are simple to understand and implement. For this reason we will, in the following, compute state-independent, deterministic decision sequences representing randomized policies.

We consider, for illustrative purposes, a multi-server queue with J heterogeneous servers, and let a_j denote the decision that a job is sent to server j, for $1 \leq j \leq J$. Let $\mathcal{A} = \{a_j : 1 \leq j \leq J\}$ denote the *alphabet*. An admissible decision sequence is now given by an infinite word α over the alphabet $\mathcal{A}$, i.e., $\alpha \in \mathcal{A}^{\mathbb{N}}$. We call a sequence $\alpha \in \mathcal{A}^{\mathbb{N}}$ *periodic* with period n if, apart from an initial string of at most $n-1$ letters, α is the repetition of a string of n letters from $\mathcal{A}$. The problem we will treat in this paper is that of finding an optimal deterministic sequence α^* such that

$$\alpha^* = \arg\min_{\alpha \in \mathcal{A}} \mathbb{E}[W(\alpha)], \tag{1}$$

where $\mathbb{E}[W(\alpha)]$ denotes the steady-state total mean waiting time of customers for control sequence α. Observe that the system may become unstable for certain $\alpha \in \mathcal{A}$ as too much traffic may be sent to (some) servers. Since we are minimizing the mean steady-state waiting times, and sequences α that lead to instability will lead to infinite mean steady-state waiting times, including sequences leading to an inst-able system will not change the solution of (1).

This way dealing with a constraint optimization problem can be avoided.

To compare the solution of the above optimization problem with a stationary randomized policy, we consider the space of probability vectors $\vec{p} = (p_1, \ldots, p_J)$, where p_j denotes the probability of sending an incoming customer to server j, i.e., the probability of choosing action a_j. We will then consider the randomized version of (1), i.e., the problem of finding an optimal randomized policy sequence:

$$\vec{p}^* = \arg\min_{\vec{p}\in[0,1]^J, \sum_j p_j = 1} \mathbb{E}[W(\vec{p})], \tag{2}$$

where $\mathbb{E}[W(\vec{p})]$ denotes the stationary total mean waiting time of customers for probability vector $\vec{p}$. We will also find the optimal deterministic sequence with a bounded periodicity, since a decision sequence with a large periodicity becomes difficult to communicate and impractical to implement, see point three above. We denote the set of admissible decision sequences with periodicity n by $\mathcal{A}(n)$. With this notation we will find the optimal periodic sequence of predefined periodicity:

$$\alpha^*(n) = \arg\min_{\alpha\in\mathcal{A}(n)} \mathbb{E}[W(\alpha)], \tag{3}$$

for $n \in \mathbb{N}$.

When there are only two decisions to choose from, i.e., $\mathcal{A} = \{1, 2\}$, a particularly nice representation of the density vector $(p, 1-p)$, for $p \in [0, 1]$, exists, called a *balanced sequence*. In [9], stochastic approximation is applied for finding the optimal balanced sequence for control of a call-center. Unfortunately, [9] failed to establish unbiasedness of the gradient estimator. Therefore, we will develop in this paper a novel approach, based on the application of a genetic algorithm (GA), for finding an optimal representation of a deterministic control sequence over an alphabet with more than two letters.

This paper is organized as follows. In Section 2 we provide a brief introduc-

tion to billiard sequences. The genetic algorithm (GA) as used in our framework is described in Section 3. In addition, we explain how a statistical test assessing the quality of the solution found by the GA is used. In Section 4 we describe the multi-server simulation model we use in our experiments. Section 5 then presents the results of the GA-optimization for the multi-server example, with additional results showing how constraints and multiple objectives can be dealt with as well. An interesting observation is that in the numerical experiments $\mathbb{E}[W(\alpha^*)] < \mathbb{E}[W(\alpha^*(n))] < \mathbb{E}[W(\vec{p}^*)]$, for n small. More specifically, our results show that asking for periodic polices with a small periodicity may have a significant (negative) impact on performance. Finally, Section 6 summarizes the main results and conclusions.

2 Billiard Sequences

For an infinite sequence $U = (u_1, u_2, \ldots)$, with $u_i \in \{1, \ldots, J\}$, denote the number of elements equal to j in the sub-sequence of length n beginning at the k-th element of U by

$$s(k, n; j) = \sum_{ji=k}^{k+n-1} 1(u_i = j), \quad k \leq n.$$

We say that $U = (u_1, u_2, \ldots)$ has *density* $\vec{p} = (p_1, \ldots, p_J)$, with $p_j \in [0, 1]$ for $1 \leq j \leq J$ and $\sum_j p_j = 1$, if

$$\lim_{n \to \infty} \frac{s(1, n; j)}{n} = p_j$$

for $1 \leq j \leq J$, provided the limit exits. If U has density $\vec{p}$, then the asymptotic frequency of occurrences of value j in U is p_j. Moreover, it follows for $k = 0, 1, \ldots$

that

$$\lim_{n\to\infty} \frac{s(k,n;j) - np_j}{n} = 0.$$

Intuitively, a small maximal absolute deviation between $s(k,n;j)$ and np_j means that the values $\{1,\ldots,J\}$ are regularly distributed with density $\vec{p}$, a desirable property for U. In fact, it can be shown that for every density $\vec{p}$ there exists U for which the maximal deviation of $|s(k,n;)-np_j|$ is smaller than one for $1 \leq j \leq J$. Such sequences are called *regular*. Following [2], for the special case $J = 2$, a sequence of ones and zeros s is called *balanced* if the difference in the number of ones for sub-sequences of the same length is at most one. Unfortunately, for a given density over an alphabet with three or more letters, deterministic sequences with appropriate densities will, generally, not be balanced. Indeed, the problem of constructing a sequence with given densities such that it is balanced in all letters, is a hard combinatorial problem for $J > 2$ letters.

One way of constructing a deterministic sequence with density $\vec{p}$ is by constructing a *billiard sequence*, see [8, 5]. The construction of a billiard sequence is best explained by the following thought experiment. Imagine a billiard "table" consisting of the J-dimensional cube $[0,1]^J$ with sides numbered $1,...,J$ such that opposite sides have the same number. Given an initial position $x = (x_1,...,x_J) \in [0,1]^J$ and an initial velocity vector $(-p_1,...,-p_J)$ of a billiard ball, now construct the sequence $U = (u_1,u_2,...)$ where the u_i are the consecutive numbers of the sides hit by the billiard ball, breaking ties in an unique way. Then U is called a billiard sequence and by construction the density of U is $\vec{p}$.

The sequence produced by this algorithm depends on the initial value, but in our experiments we keep this initial value fixed at $x = 0$. The billiard sequence algorithm can then be implemented by starting with $k_i = 0, i = 1,\ldots,J$, and at each step choosing that symbol i for which $(x+k_i)/p_i$ is minimal (over all J such values), and increasing the corresponding k_i by one. If there is more than

one i for which $(x+k_i)/p_i$ is minimal, the smallest such i is chosen to break the tie. This is then repeated for the length of the desired sequence. It is, however, worth noting that one could search for "good" initial values x, i.e., search for a value for x that gives the most "balanced" sequence. For the sake of simplicity and efficiency, we have not included this in our algorithm (i.e., we always use $x = 0$).

Example 1. *Let $\vec{p} = (1/3, 2/3)$ and suppose that we want to find a billiard sequence of length six. We initialize $k_1 = 0$ and $k_2 = 0$, and compute in each step k_i/p_i, for $i = 1, 2$. In case of a tie, we choose $i = 1$. Table 1 shows the result of the algorithm.*

	k_1	k_2	k_1/p_1	k_2/p_2	billiard seq.
1	0	0	0	0	1
2	1	0	3	0	12
3	1	1	3	1.5	122
4	1	2	3	3	1221
5	2	2	6	3	12212
6	2	3	6	4.5	122122

Table 1: The algorithm for constructing a billiard sequence for $\vec{p} = (1/3, 2/3)$ of length six.

In [23] it is shown that for rational densities $\vec{p}$, there exists a periodic billiard sequence with these densities which minimizes the total unbalance.

Remark 2. *For any $p \in [0, 1]$ and $\phi \in \mathbb{R}$ a balanced sequence $U = (u_1, u_2, \ldots)$ of density p is obtained by putting*

$$u_j = \lfloor jp + \phi \rfloor - \lfloor (j-1)p + \phi \rfloor \text{ for } j = 1, 2, \ldots,$$

where $\lfloor x \rfloor$ is the largest integer smaller than or equal to x. Sequences constructed in this way are also called lower bracket sequences. Note that the bracket sequence is only defined on two letters. Moreover, for an alphabet of two letters the

billiard sequence of given density becomes the bracket sequence with that density, and it is always balanced.

Balanced sequences have been studied for a long time (see, e.g., [25]) and many properties were derived, but this was not in the context of optimal control. However, in [17] it was proved for a specific admission control problem that the optimal control sequence $U = (u_1, u_2, \ldots)$ is within this subset of sequences with "good balance". Subsequently, control by such sequences have been applied to more scheduling, admission, and routing problems in the area of queuing and discrete-event systems, see, for example, [4, 6, 2, 3, 28].

Using the concept of multimodularity, [5] gives an overview of control problems for which optimality of such sequences follows. However, quite a number of assumptions and specific types of control policies were needed to apply the concept of multimodularity. First it was used for admission control and then extended to some specific routing and/or polling problems. In this paper we apply these sequences with "good balance" to problems which do not fall within the framework for which multimodularity was established in [5]. Therefore we do not know a priori whether the optimal control sequence is within the subset of sequences with "good balance". From a practical point of view, however, this poses no restriction, as our GA approach can be applied in general. Hence, the effectiveness of a billiard sequence in controlling a given system can be easily investigated through numerical experiments.

Finding optimal billiard sequences is a hard problem. The main reason being that standard optimization algorithms cannot be applied to the problem. To see this, note that typical standard optimization algorithms have a part, called the exploitation phase, where designs are compared to designs close to the candidate solution. Unfortunately, there is no direct interpretation of "closeness" for billiard sequences. Consider, for example, a billiard sequence over two

letters and let $p = (p_1, p_2) = (1/3, 2/3)$ be the candidate solution for the optimal routing. This probability vector can be represented by a billiard sequence $(1, 0, 0, 1, 0, 0, \ldots)$, which is of periodicity 3. Introducing a small change in p, for example, changing p to $\hat{p} = (1/3 + 1/10, 2/3 - 1/10) = (13/30, 17/30)$ will lead to a billiard sequence with period length 30. For example, in a recent paper, stochastic optimization techniques have been applied to finding an optimal balanced sequence [9], but the authors failed in establishing the unbiasedness of the estimator for the sensitivity of the fitness function with respect to the routing probability. The main reason for this is the above illustrated ill behavior of the length of the period of a billiard sequence with respect to small changes in the rates. To overcome this obstacle, we have developed the genetic algorithm based approach presented in the following section.

3 The Genetic Algorithm

A genetic algorithm (GA) [19, 15, 24] is a stochastic search method that is modeled after natural evolution. The idea is to maintain a population of candidate solutions to a given (combinatorial) optimization problem, and to create new "generations" of candidate solutions by applying selection and recombination operators to the current population. This way, better and better solutions are "evolved" over time. GAs are particularly suitable for optimization problems for which there is no analytical solution or efficient (polynomial-time) algorithm available, and they have been applied successfully to many real-world optimization problems to find good approximate (near-optimal) solutions, including for problems closely related to the one investigated here [31, 30].

In a GA, the candidate solutions in the population are represented by a suitable "genetic encoding". A *fitness function* is then used to assign fitness values to the individuals in the current population. The fitness function is

determined by the given optimization problem, and calculates a measure of how well a candidate solution solves the given problem. The better a solution, the better its fitness value (i.e., higher for maximization problems, lower for minimization problems). This way, individuals in the population can be directly compared to each other on the basis of their fitness. The initial population in a GA is usually created at random.

In our framework, the optimization problem is that of minimizing the average waiting time in a multi-server queuing system. Individuals are encoded by a density vector $\vec{p} = \{p_1, p_2, \ldots, p_J\}$ with $p_j \in [0;1]$ and $\sum_j p_j = 1$. Our fitness function translates such a encoding into an assignment policy, and then simulates a multi-server queuing system with J servers and with pre-defined arrival and service times (details on the simulation model are given in the next section). The fitness is then calculated as the average waiting time per arriving customer (or job). We use two variants of our fitness function. In the first version a density vector $\vec{p}$ is converted into a billiard sequence as described in the previous section. This billiard sequence is then used to assign arriving customers to servers, i.e., if the i^{th} symbol in the billiard sequence is j, then the i^{th} arriving customer is assigned to server j. This corresponds to problem (1) in the introduction. In the second version, the densities $\vec{p}$ are used directly to randomly (and independently) assign arriving customers to servers with probabilities p_j. This corresponds to problem (2) in the introduction. So, in both variants we optimize a density vector $\vec{p}$ which represents an assignment policy to minimize the average customer waiting time.

Once the fitness values of all individuals in the current population have been calculated, these individuals are then *selected* to act as "parents" to create the next generation of candidate solutions. Individuals in the current population have a probability of being selected that is proportional to their fitness value

(relative to the rest of the population). In other words, individuals with a relatively "good" fitness value (i.e., low, in our case) will be allowed to contribute their genetic material to the next generation more often (on average) than individuals with a relatively "bad" (high) fitness.

For the selection operator we use a combination of *elitism* and *tournament selection*. With elitism, the best E individuals in the current population are copied unchanged to the next generation. With tournament selection (to fill up the rest of the next generation), in each round two individuals are selected at random from the current population (regardless of fitness value), and with probability p_t the better of the two individuals "wins" the tournament and is selected to act as a parent. With probability $1 - p_t$ the less fit individual wins the tournament and is selected. Note again that since we are dealing with a minimization problem (average waiting time), "better" means a lower fitness value. Note also that this selection process is with replacement, so "good" individuals will (on average) be selected more often to act as parents, and "bad" individuals might not get selected at all.

Once two parents are selected this way, they are allowed to create "offspring". To do this, their genetic material is recombined through *crossover*. For the crossover operator we use one-point crossover on the density vectors. First, a random crossover point is chosen somewhere between the first and last densities. Then, the densities after the crossover point are swapped between the two parents, creating two offspring individuals. Note that this does not necessarily result in valid offspring density vectors (they may not sum up to one). To remedy this, after each crossover event, the offspring density vectors are renormalized. As a simple example, consider the two parent density vectors

$$\vec{p}_1 = \{\mathbf{0.1}, \mathbf{0.1}, \mathbf{0.1}, \mathbf{0.1}, \mathbf{0.6}\}$$
$$\vec{p}_2 = \{0.2, 0.2, 0.2, 0.2, 0.2\}$$

and suppose the crossover point was (randomly) chosen to be between the second and third density. The two offspring individuals will then look like

$$\vec{p'}_1 = \{\mathbf{0.1}, \mathbf{0.1}, 0.2, 0.2, 0.2\}$$
$$\vec{p'}_2 = \{0.2, 0.2, \mathbf{0.1}, \mathbf{0.1}, \mathbf{0.6}\}$$

which are then renormalized to

$$\vec{p'}_1 = \{0.1250, 0.1250, 0.2500, 0.2500, 0.2500\}$$
$$\vec{p'}_2 = \{0.1667, 0.1667, 0.0833, 0.0833, 0.5000\}$$

(rounded to four decimals, for clarity).

Finally, the newly created children are subjected to *random mutations*. In each new offspring density vector, a small random value (drawn uniformly from $[-0.05; 0.05]$) is added to a randomly chosen density (out of the J densities). Here too, the density vector is renormalized after a mutation event.

These selection-crossover-mutation steps are repeated until a new population is filled up with offspring, constituting the next generation. This whole process is then repeated for a given number of generations. To summarize, our GA can be described as follows.

1. **Initialization:** Create an initial population (of size P) of density vectors at random (from a uniform distribution).

2. **Fitness:** Calculate the fitness of each individual (using the multi-server simulation model).

3. **Selection:** Copy the best E individuals directly to the next generation (elitism) and use tournament selection with probability p_t to select $P - E$ parents.

4. **Crossover:** From each next pair of selected parents, create two offspring

individuals using one-point crossover (and renormalization).

5. **Mutation:** Add a small random value (from $[-0.05; 0.05]$) to a randomly chosen density in each newly created offspring individual and renormalize.

6. **Repeat:** Repeat steps 2 to 5 for a given number G of generations.

The particular parameter values used in our GA implementation are a population size of $P = 30$, an elite size of $E = 4$, a tournament selection probability of $p_t = 0.9$, and a number of generations of $G = 50$.

Of course there are many ways in which the various GA operators and the corresponding parameter values could have been implemented and chosen. And, often, the search performance of the GA depends largely on these choices. However, unfortunately there is not much theory yet about how to relate the type or difficulty of a given problem to the particular GA implementation that is best suited for that problem. To a large degree, it comes down to trial and error, and experience with previous (similar) problems. There exist basic guidelines [11, 13], but these are also mostly empirically based.

However, after some experimentation the above described implementation and parameter values seem to give the best overall results for our problem. In the results section below we present a parameter sensitivity analysis that indicates that the chosen parameter values are indeed in an optimal range. In [12] an alternative GA implementation for a similar problem is described.

The overall running time of a GA is mostly determined by the fitness evaluations, and thus scales linearly with the population size and the number of generations. Memory requirements are minimal, as the algorithm mostly needs to store the current population of individuals and their fitness values.

While a GA is a widely used optimization algorithm, one of its main drawbacks is the fact that it is typically not possible to establish convergence of the algorithm. The latter points means that, from a practical point of view,

there are no strict guidelines on when to terminate the algorithm. Of course this is a problem of any heuristics-based algorithm, and providing measures for the quality of the found solution is imperative for acceptance of any GA-based analysis. Fortunately, this drawback can be overcome by using arguments from ordinal optimization [18]. The basic idea is as follows. Rather than considering $\vec{p}$ as a continuous density vector, assume that $\vec{p} \in \Theta \subset [0,1]^J$ with Θ being a discrete set such that $\vec{p} \in \Theta$ implies $\sum_j p_j = 1$. We choose Θ large enough so that Θ becomes a "dense" subset of $[0,1]^J$. The basic idea of ordinal optimization is that if we are interested in finding one of the best 1% densities in Θ, then, independent of the size of Θ, a random sample of 10^4 densities, where the sampling is done uniformly, will with probability of almost one contain one of the top 1% densities. So, the fitness of the best solution found in Θ can be used as a benchmark for testing the quality of a GA solution, i.e., the result obtained by the GA should have at least the fitness of the best fitness of the random sample. More specifically, as shown in Theorem 1 of [27], the type II error of the statistical test on "H_0: the GA solution belongs to the top 1% solutions" and "H_1: the GA solution does not belong to the top 1% solutions" has a type II error of at most 0.05 %; in words, the probability of wrongly accepting the GA solution as one of top 1 % is no larger than 5 %. For details we refer to [27]. For an application of ordinal optimization to generic algorithms, we refer to [29].

4 The Multi-Server Simulation Model

As a practical example, we study a multi-server queuing system with J heterogeneous servers. We have implemented this system as a simulation model and the mean stationary waiting time as the fitness function for the genetic algorithm. The example is taken from [16] where an approximation for the stationary mean

waiting time in a multi-server with general inhomogeneous service time distributions is provided. Note that if complete information on the workloads in the queues is available, then the policy which sends the arriving customer to the queue with shortest waiting time (SWP) is known to be of good quality; see [10]. If only the queue sizes are known, then the structure of the optimal policy is generally not known, see [20]. In this paper we suppose that there is no information on the actual loads of the various servers, hence only open loop policies may be considered. In [22] a similar model is analyzed and bounds on the performance of billiard sequences are derived. In Section 5.5 later on in the text we will perform a comparison with other heuristics.

We assume that there are $J = 10$ servers each with a lognormal service time $s_j \sim \text{LogNorm}(\mu_j, \sigma)$ with the following parameter values: $\mu_1 = 1.0$, $\mu_2 = 1.2$, $\mu_3 = 1.4$, $\mu_4 = 1.6$, $\mu_5 = 1.8$, $\mu_6 = 2.0$, $\mu_7 = 2.2$, $\mu_8 = 2.4$, $\mu_9 = 2.6$, and $\mu_{10} = 2.8$, and $\sigma = 0.5$. The average number of customers served per time unit is thus

$$\sum_{j=1}^{10} \frac{1}{e^{\mu_j + \sigma^2/2}} = 1.55.$$

New customers arrive according to a Poisson process, i.e., the waiting time to the next arrival event $t_e \sim \text{Exp}(\lambda)$ with $\lambda = 1.25$ (on average $\frac{1}{\lambda} = 0.8$ arrivals per time unit). Customers are assigned to queues upon arrival according to a given strategy, and are serviced according to a FIFO discipline. As already mentioned above, we compare two possible assignment strategies: (1) billiard sequences and (2) random assignment, each according to a given density vector $\vec{p}$. We then apply the GA for solving optimization problems (1) and (2). So, the fitness of a candidate solution $\vec{p}$ is the average waiting time per customer in the simulation model using the densities $\vec{p}$ in the assignment strategy. We use 100,000 arrival events in the simulation model to calculate these average waiting times. In the results below, we show that this is enough to reach a stationary

distribution, while still allowing the GA to run fast enough (on the order of minutes). The overall running time of the GA scales linearly with the number of arrival events used in the simulation model.

Note that the analytical approximation provided in [16] breaks down for this particular model, as it generates too many unstable solutions while trying to find the optimal solution, and therefore does not converge to a correct solution. Thus, it is necessary to use heuristic optimization techniques such as a genetic algorithm. We implemented the GA and the simulation model (as described above) in the C programming language, and performed our experiments on a Linux platform. We next describe the results of our heuristic approach.

5 Results

5.1 Billiard sequences vs. randomized policies

Using the GA as described in Section 3 to minimize the average waiting time in the simulation model as described in Section 4, using either billiard sequences or random assignments, gives the results as shown in table 2. First, we ran the GA 20 times on each assignment strategy and recorded the best fitness value in the last (50th) generation. Then, for each assignment strategy we took the best GA solution (density vector) found in these 20 runs, and recalculated its fitness value 20 times (i.e., run the simulation model 20 times with the given solution). The table shows the average and standard deviation of these fitness calculations for both assignment strategies, averaged over the 20 runs (left) and averaged over the best solution's 20 fitness recalculations (right). Clearly, billiard sequences are the better strategy, with the average waiting time per customer for random assignments being more than twice as large as that for billiard sequence assignment, a highly significant difference with a p-value of

less than 10^{-15}. Furthermore, given the extremely low standard deviations, the GA is very robust in performance, both over different runs as well as over different fitness calculations for the same solution.

	Runs		Best solution	
	Billiard	Random	Billiard	Random
Average	10.27	22.33	10.26	22.84
St.dev.	0.126	0.408	0.120	0.571

Table 2: The averages and standard deviations of the best fitness values in the final generation over 20 GA runs ("Runs") and of the best GA solution over 20 fitness recalculations ("Best solution") for billiard sequences and random assignments.

The two GA solutions for the different assignment strategies are very similar, but differ somewhat in their densities. In fact, using the best density vector found by the GA for the random assignment version, and then recalculating its fitness using the billiard sequences version of the fitness function, yields a similar but (statistically significant) slightly higher waiting time.

As already observed in [9] for a call-center model, the optimal balanced sequence has better fitness than the implementation of the optimal random density by means of a balanced sequence. Thus, this phenomenon can also be observed for the multi-server problem, although the improvement of the solution of optimization (1) over the implementation of the solution of (2) by means of a billiard sequence is relatively small.

Finally, to show that 100,000 arrival events are enough to reach stationarity in the simulation model, we took the best solution found by the GA (using billiard sequences) and recalculated again its fitness value, but this time for different numbers of arrival events. Figure 1 shows the average waiting times (solid line) plus or minus one standard deviation (dashed lines). Clearly, convergence is largely reached with 100,000 (1e+05) arrival events.

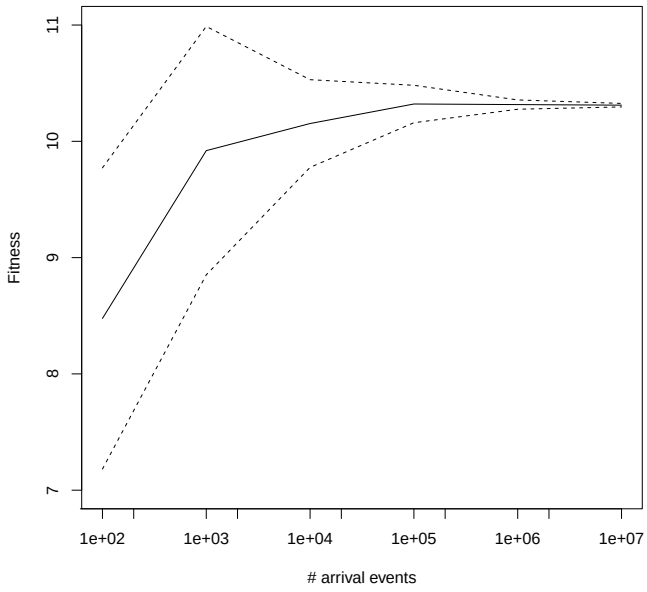

Figure 1: The average fitness (solid line) plus/minus one standard deviation (dashed lines) for the best solution found by the genetic algorithm for different numbers of arrival events.

5.2 The quality of the GA solution

First, we performed a parameter sensitivity analysis to make sure that the chosen GA parameters are indeed in an optimal range. In particular, we varied the value of the three main parameters in our GA implementation (population size P, number of generations G, and tournament selection probability p_t), performed 20 runs of the GA with each alternative parameter value (using billiard sequence assignments), and compared the averages of the best fitness values in the final generation with the average value presented in table 2 (10.27) by performing a t-test. The results are presented in table 3.

Recall that the results in the previous subsection were obtained using parameter values $P = 30$, $G = 50$, and $p_t = 0.9$. Table 3 clearly shows that if any

	Population size		Generations		Selection probability	
	16	50	30	70	0.7	0.95
Average	10.85	10.28	11.13	10.24	10.70	10.26
St.dev.	0.727	0.123	0.560	0.081	0.643	0.085
p-value	0.00187	0.614	0.000001	0.555	0.00691	0.907

Table 3: The averages and standard deviations of the best fitness values in the final generation over 20 GA runs for various alternative GA parameter values.

of these three main GA parameters is too low ($P = 16$, $G = 30$, or $p_t = 0.7$), the results are significantly worse (p-values of 0.00187, 0.000001, and 0.00691, respectively). However, if any of these parameter values is larger ($P = 50$, $G = 70$, or $p_t = 0.95$) than those used to obtain the results presented in table 2 above, there is no improvement (p-values of 0.614, 0.555, and 0.907, respectively). So, the used parameter values indeed seem to be an optimal trade-off between finding good solutions while not using too much unnecessary computing time.

Next, to obtain an indication of the quality of the GA solution, we compared the GA search with a random sample of density vectors $\vec{p}$ drawn uniformly and of size 10,000. Figure 2 shows the fitness value of the best individual in the GA population against the number of fitness function evaluations (solid line).

Starting with an initial population of random density vectors $\vec{p}$, with relatively large fitness values, the best fitness in the population quickly reduces over time, and within a thousand function evaluations the algorithm already converges towards the best solution with an average waiting time of close to 10 time units. Note that the GA was run for only $G = 50$ generations, i.e., 1500 fitness function evaluations in total (given a population size of $P = 30$). However, the curve indicating the best solution over time (solid line) is continued until 10,000 function evaluations for comparison with the random sample.

The dashed line in Figure 2 shows the best fitness value found so far against the number of function evaluations in a random sample of density vectors $\vec{p}$

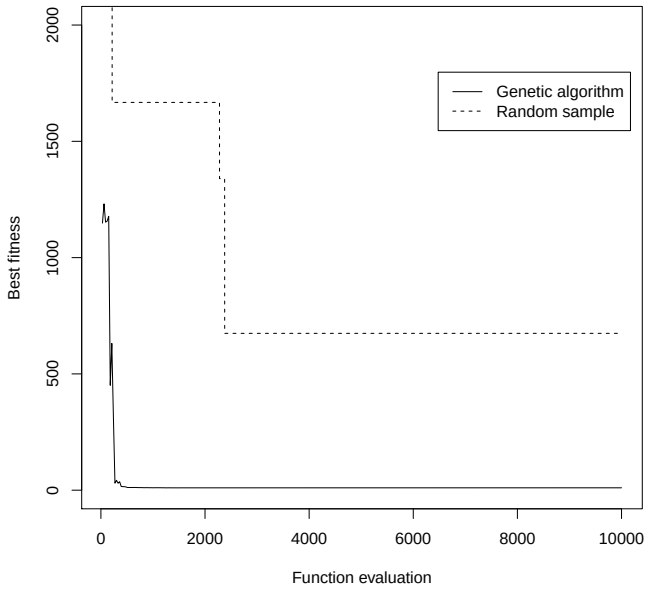

Figure 2: The best fitness against the number of function evaluations for the genetic algorithm (solid line) and for a random sample (dashed line).

of size 10,000. Clearly, the best solution from the random sample does not even come close to the best GA solution. From this, we can conclude that the probability of wrongly assuming that the solution found by the GA belongs to the best 1% of all possible solutions is no larger than 5 % (as argued briefly in Section 3 and in detail in [27]). Note that this does not necessarily mean that the fitness value of the GA solution is less than 1% different from the optimal (best possible) fitness value. After all, it is not know what the optimal fitness value is. But if it were possible to enumerate all possible solutions (density vectors) and order them from best to worst, the solution found by the GA would be within the top 1% of this ranking.

Another way to get more insight into the problem difficulty is to plot the fitness values from the random sample ordered from best to worst, which yields

the so-called *ordered performance curve* [18]. Figure 3 shows this curve. It clearly shows that there are only very few "good" solutions and many "bad" solutions. This makes sense, given that for most (random) density vectors,

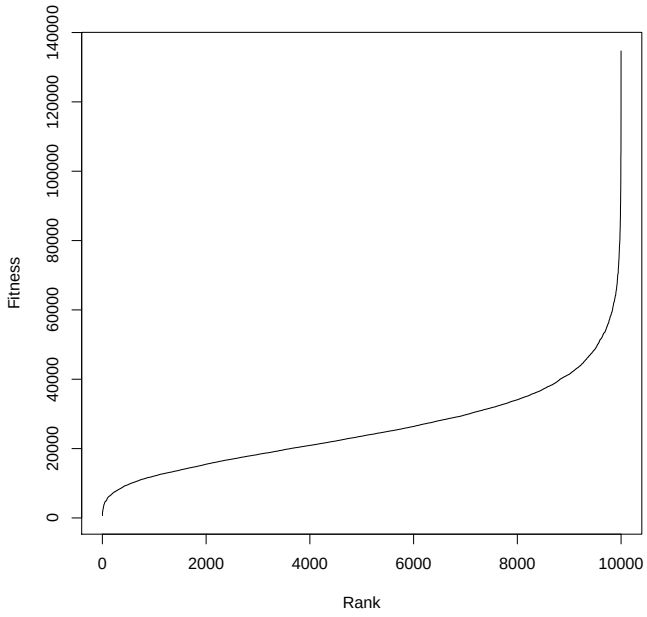

Figure 3: The fitness values of the random sample ordered from best to worst.

arriving customers are not assigned to queues according to the distribution of service times, and one or more of the J queues can easily "explode", resulting in very large waiting times, as Figure 3 shows. So, it seems that the customer assignment problem is not an easy problem, also already indicated by the large difference between the best GA solution and the best solution from the (much larger) random sample.

5.3 Billiard sequences with enforced periodicity

Given that the densities $\vec{p}$ that are optimized with the GA are double-precision real numbers in the multi-server simulation model, it is not expected that the billiard sequence generated with the best GA solution will have a low periodicity. Indeed, if a billiard sequence of length 100,000 is generated with the densities from the best GA solution, there is no periodicity at all (i.e., there are no repeating sub-sequences of length smaller than half the length of the sequence itself). From a practical point of view, in terms of actually implementing the (near-optimal) assignment strategy, this is not desirable. However, it is possible to enforce a given periodicity r in the (candidate) GA solutions by restricting the densities to multiples of $\frac{1}{r}$.

We ran the GA (using the billiard sequence version of the fitness function) again, but this time enforcing different periodicities r, to see how this influences the solution quality. Figure 4 shows the best fitness found by the GA (again averaged over 20 recalculations afterwards) against the enforced periodicity. Clearly, the average waiting time gets worse with smaller periodicities. For periodicities $r = 100$ and $r = 50$, the difference with the unrestricted best solution (represented by the thick horizontal line) is still very small. So, even with an enforced periodicity of $r = 50$ it is still possible to get qualitatively similar solutions as in the unrestricted case. However, the situation quickly becomes increasingly worse for periodicities smaller than $r = 50$. Note, though, that the average waiting times in these cases are still better than the best solution with random assignments! For periodicities smaller than $r = 20$, the GA does not seem to find any reasonable solutions anymore. For example for a periodicity of $r = 15$, the best solution found has an average waiting time of more than 200 time units, i.e., ten times worse than the best solution with random assignments.

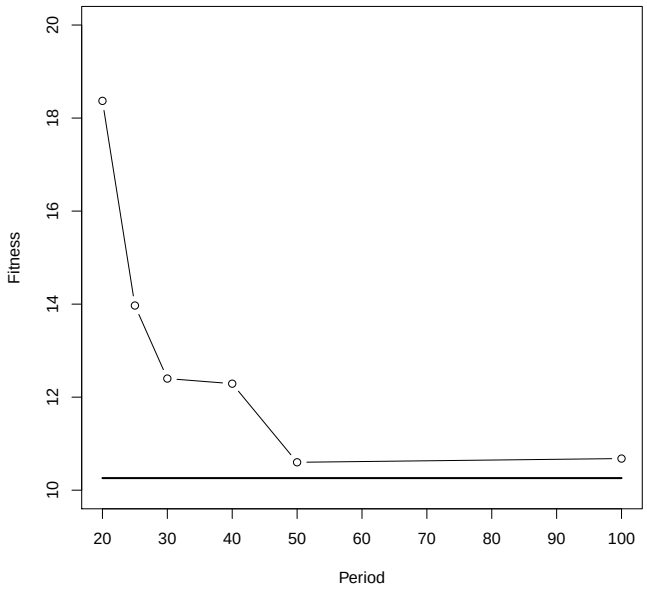

Figure 4: The best fitness value found by the GA against the enforced periodicity of the billiard sequences. The thick horizontal line represents the best fitness value in the unconstrained case.

In conclusion, it is possible to include constraints into the optimization problem, and still have the GA find reasonable solutions. However, the tighter the constraint(s), the more it influences the performance of the best solution in a negative way.

5.4 A multi-objective extension

With the best GA solution, the server occupancy rates (i.e., the fraction of time the servers are actually serving a customer) vary between 72% and 87% among the $J = 10$ servers. So, even though the average waiting time for arriving customers is minimized, the total workload is not equally distributed among the servers.

To investigate if it is possible to get a more balanced workload, we ran the GA again (using the billiard sequence version), but this time with the standard deviation of the agency occupancy rates added to the fitness function (with an appropriate scaling factor). In other words, we not only try to minimize the average customer waiting time, but *simultaneously* the variation in server occupancy rates. It thus becomes a multi-objective optimization problem.

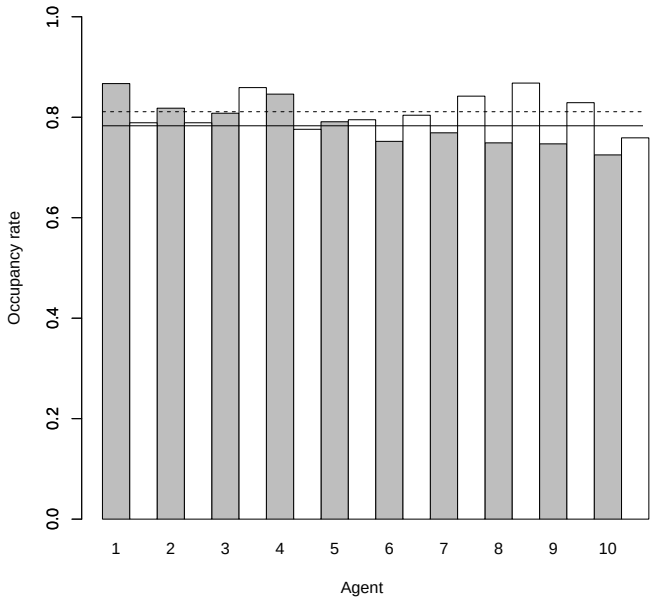

Figure 5: The server occupancy rates for the original GA solution (gray bars) and for the multi-objective GA solution (white bars). The solid line indicates the average server occupancy rate for the first case and the dashed line that for the second case.

Figure 5 shows the occupancy rates of the 10 servers for the original (unrestricted) GA solution (gray bars) and for the multi-objective GA solution (white bars). There is indeed significantly less variation in the occupancy rates for the second case (76% to 86%). Table 4 shows the average and standard deviation in server occupancy rates for the two cases. The standard deviation has been

reduced from 0.047 to 0.036 (i.e., a reduction of more than 20%).

GA	Occupancy rate avg	st.dev.	Waiting time
Original	0.787	0.047	10.26
Multi-obj.	0.811	0.036	11.03

Table 4: The average and standard deviation of the server occupancy rates, and the average customer waiting time, for the original and the multi-objective GA solutions.

However, this reduction in occupancy rate variation comes at a cost. As table 4 shows, the average occupancy rate (over all 10 servers) has increased from 0.787 (indicated by the solid line in Figure 5) to 0.811 (indicated by the dashed line in the same figure). In other words, on average the servers are more busy, which is also reflected in a (statistically significant) increase in average customer waiting time, from 10.26 to 11.03.

So, as with the constraint on the periodicity of the billiard sequences, optimizing multiple objectives simultaneously comes at a cost in terms of an increase in the average customer waiting time. However, these examples do illustrate the possibilities when using a genetic algorithm to optimize the customer assignment problem, showing that it is feasible to optimize multiple objectives simultaneously, or impose constraints in various ways, while still finding reasonable solutions.

5.5 Comparison with other assignment methods

We conclude this series of examples with a comparison of our method with earlier results obtained on a slightly simpler version of the multi-server queuing system. More specifically, we consider a system with three servers, each with exponentially distributed service times with rates $\mu_i, i = 1, 2, 3$, and Poisson λ arrival process. In [21], average waiting times are calculated for the so-called "open loop" customer assignment method for $\mu_1 = 1, \mu_2 = 4, \mu_3 = 7$, and $\lambda = 3$,

6, or 9.

For comparison, we applied a GA to find near-optimal density vectors for either random assignments or billiard sequence assignment. We also considered billiard sequences restricted to a maximum period of 50, roughly corresponding to the periodicities used in the open loop method [21]. Finally, we also considered the shortest waiting-time policy, which always assigns an arriving customer to that queue which will have the shortest waiting time. Note that this policy can, of course, never be implemented in realistic situations, but it represents the best possible solution.

Table 5 shows the results of this comparison. As before, random assignments (Random) are always the worst. Billiard sequence assignments (Billiard) outperform the open loop method (Open loop), also when a maximum periodicity of 50 is imposed (Billiard 50). Obviously the shortest waiting-time policy (SWP) is superior, but not achievable in practice.

λ	Random	Open loop	Billiard	Billiard 50	SWP
3	0.059844	0.030911	0.029317	0.029369	0.006552
6	0.187166	0.119546	0.111929	0.111991	0.043530
9	0.569972	0.411938	0.372741	0.381180	0.180907

Table 5: Comparison of the average waiting time for various assignment methods. Results for "Open loop" are taken from [21].

In short, these results emphasize that our proposed method of using a genetic algorithm to optimize billiard sequences for job or customer assignments in multi-server queuing systems without state information outperforms other known heuristic methods, and optimality of SWP is confirmed.

6 Conclusions

In this paper we addressed the problem of finding near-optimal deterministic decision sequences for a job assignment problem, where an analytical or effi-

cient algorithmic optimal solution is not possible. We followed the principle *naturis artis magistra* and used a genetic algorithm for the optimization. We developed an overall framework for determining a near-optimal deterministic sequence and also addressed the problem of finding a control sequence with periodic behavior. Using recent results from the theory of ordinal optimization, we were able to provide a statistical analysis of the performance of the GA. Hence, we have shown that the approach developed here is useful, and can generate near-optimal solutions while also being able to take additional constraints or multiple objectives into account.

References

[1] E. Altman. *Constrained Markov Decision Processes.* Chapman and Hall, 1999.

[2] E. Altman, B. Gaujal, and A. Hordijk. Balanced sequences and optimal routing. *Journal of the ACM*, 47:4, 2000.

[3] E. Altman, B. Gaujal, and A. Hordijk. Multimodularity, convexity and optimization properties. *Mathematics of Operations Research*, 25(2):324–347, 2000.

[4] E. Altman, B. Gaujal, and A. Hordijk. Regular ordering and applications in control policies. *Discrete Event Dynamic Systems*, 12(2):187 – 210, 2002.

[5] E. Altman, B. Gaujal, and A. Hordijk. *Discrete-Event Control of Stochastic Networks: Multimodularity and Regularity.* Springer-Verlag New York, Inc., ASecaucus, NJ, USA, 2003.

[6] E. Altman, B. Gaujal, A. Hordijk, and G. Koole. Optimal admission, routing and service assignment control: the case of single buffer queues.

In *37th IEEE Conference on Decision and Control*, volume 2, pages 2119–2124, Tampa, FL, USA, 1998.

[7] E. Altman and A. Shwartz. Time-sharing policies for controlled Markov chains. *Operations Research*, 41(6):1116–1124, 1993.

[8] P. Arnoux, C. Mauduit, I. Shiokawa, and J. Tamura. Complexity of sequences defined by billiards in the cube. *Bull. Soc. Math. France*, 122:1–12, 1994.

[9] S. Bhulai, T. Farenhorst-Yuan, B. Heidergott, and D. van der Laan. Optimal control via balanced sequences. *Annals of Operations Research*, (to appear) 2012.

[10] C. Chang, A. Hordijk, R. Righter, and G. Weiss. The stochastic optimality of sept in parallel machine scheduling. *Probability Engineering Informational Sciences*, 8:179–188, 1994.

[11] L.D. Davis. *The Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[12] R. de Goeij. Optimal routing by mixing strategies. Master's thesis, Vrije Universiteit Amsterdam, December 2012.

[13] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[14] E.A. Feinberg and A. Shwartz, editors. *Handbook of Markov Decision Processes*. Kluwer, 2002.

[15] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[16] X. Guo, Y. Lu, and M. Squillante. Optimal probabilistic routing in distributed parallel queues. *SIGMETRICS Performance Evaluation Review*, 32, 2004.

[17] B. Hajek. Extremal splittings of point processes. *Mathematics of Operations Research*, 10(4), 1985.

[18] Y-C. Ho, Q-C. Zhao, and Q-S. Jia. *Ordinal Optimization*. Springer-Kluwer, 2007.

[19] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975. Second edition: MIT Press, 1992.

[20] A. Hordijk and G. Koole. On the assignment of costumers to parallel queue. *Probability in the Engineering and Informational Science*, 6:95–511, 1992.

[21] A. Hordijk, G. Koole, and J.A. Loeve. Analysis of a customer assignment model with no state information. *Probability in the Engineering and Informational Sciences*, 8:419–429, 1994.

[22] A. Hordijk and D. van der Laan. Periodic routing to parallel queues and billiard sequences. *Mathematical Methods of Operations Research*, 59:173–192, 2004.

[23] A. Hordijk and D.A. van der Laan. On the average waiting time for regular routing to deterministic queues. *Mathematics of Operations Research*, 30:521–544, 2005.

[24] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[25] M. Morse and G. Hedlund. Symbolic dynamics II: Sturmian trajectories. *American Journal of Mathematics*, 62:1–42, 1940.

[26] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[27] Z. Shen, Q-C. Zhao, and Q-S. Jia. Quantifying heuristics in the ordinal optimization framework. *Discrete Event Dynamical Systems*, 20:441–471, 2010.

[28] S. Shinya, M. Naoto, and K. Ryohei. m-Balanced words: A generalization of balanced words. *Theoretical computer science*, 314(1):97–120, 2004.

[29] D. P. Song, C. Hicks, and C. F. Earl. An ordinal optimization based evolution strategy to schedule complex make-to-order products. *International Journal of Production Research*, 44:4877–4895, 2006.

[30] Y. Xu, K. Li, L. He, and T. T. Khac. A DAG scheduling scheme on heterogeneous computing systems using double molecular structure-based chemical reaction optimization. *Journal of Parallel Distributional Computation*, 73:1306–1322, 2013.

[31] Y. Xu, K. Li, J. Hu, and K. Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270:255–287, 2014.