Genetic Algorithms and Call Admission to Telecommunications Networks*

C. Rose & R. Yates

May 9, 1995

Department of Electrical and Computer Engineering
Rutgers University
PO Box 909
Piscataway NJ 08855

ABSTRACT

We consider the admission problem for the two class M/M/C/C queueing system. The admission policy is coded as a binary string and strings which produce good policies are found using genetic algorithms. We use three policy coding methods:

- *Direct codes:* The policy consists of a sequence of 2-bit words, corresponding to the admit/deny decision for each class, associated with each state. The resulting binary string represents the policy.
- *Block Coding:* A policy string is constructed from a code book of binary strings and a string which dictates the sequence in which code words appear in the policy string. The genetic algorithm operates on both the code book and the sequence string.
- *Program Codes:* A binary string is supplied as the program for a simple Turing machine to which the system state is supplied as input. The output of the program is the admission policy for that state. This method is used for very large problems where the direct or block codes are prohibitively long.

All three are found to work well for smaller problems and suggested structural properties for good policies. Results were compared to optimal policies derived using a Markov decision process. For very large problems, the program codes easily produced policies with performance comparable to the best known reservation policies.

*Supported by National Science Foundation grant NSF-NCRI-92-06148

1 INTRODUCTION

One of the primary tasks of a telecommunications system is the maintenance of good service quality. Call admission, the acceptance or denial of service requests, is central to this concern. The judicious denial of service based on customer type and/or system state, even when excess capacity exists, can improve the overall service quality (i.e., average blocking probability).

Thus, much research effort has been directed toward the performance evaluation of proposed admission strategies with an eye toward developing optimal strategies. Such research usually takes the following path:

- A particular analytic network model is proposed to address a particular problem.
- An analytic measure of network performance is specified.
- An analytic admission strategy, customized for the problem and model, is proposed and evaluated (usually after much effort).

This research paradigm arises naturally since it is difficult to analytically evaluate call admission policies for reasonably complex communications networks. To decide the optimality of a policy usually requires that the policy/objective function pair yield a system amenable to mathematical analysis; this is rarely the case. For example, it is often debatable whether real telecommunications systems ever display steady state behavior, but since analysis of the system dynamics is generally very difficult, many performance analyses presume that a system reaches a steady state.

Therefore, it is often the case that the set of strategies or policies considered is restricted to those which can be described concisely. Those which are difficult to describe compactly, analytically and/or heuristically are typically ignored.

As a slight change of research paradigm we suggest that the problem of searching for good policies be *separated* from the evaluation of policies. Thus, depending on the particular network model, a policy may be evaluated either analytically or by simulation. However, for purposes of policy search, we need only some means of policy evaluation and an efficient means of searching the set of all policies. We therefore require some means of specifying an arbitrary policy.

To this end we note that a policy either accepts or denies admission to a given call or call type based upon the state of the system [1].[1] If we assume that the network state space is discrete, then a policy for a particular call type is simply a

[1]It is worthwhile to note that the state may be defined as broadly as necessary and may include information about past observations of inputs or even past policy decisions.

finite mapping of network state to a binary yes/no decision.[2] Any such policy can be implemented as one of a finite number of logic functions and this set of logic functions is the policy space.

The separation of policy search from policy evaluation results in three primary benefits:

- The policies we consider can be arbitrary if necessary since we can search for policies directly in the policy space.
- We are not bound to mathematically tractable system models.
- Given good search algorithms, we can automate the policy search procedure.

A number of techniques [4, 5, 6] might be used to search the policy space systematically and efficiently. However, for many relatively small systems of interest, the state space is huge and if the state space is huge, the policy space is even larger. Searching large spaces systematically, even when computationally efficient, is time consuming. In addition, although it is always theoretically possible to specify a policy for any given system, it is often *physically impossible* to specify an arbitrary policy if the state space is large enough.

These practical considerations sometimes force a compromise. When the policy space is large, we must settle for some compact coding of a policy because a policy which cannot be specified is useless. A *compact* coding, by definition, restricts the search to limited *classes* of policies.

Choosing such a code, however, is no different than specifying an analytic or heuristic description of a policy since these too must restrict the number of possible policies. We thus benefit in real terms since the coding chosen need not be analytically tractable AND the classes specified by the code can still be searched systematically.

Unfortunately, even for simple objective functions, the structure of the coded policy space may not be obvious. For example, it is difficult to say *a priori* whether seemingly slight perturbations of a good policy code yield slightly or dramatically different performance; i.e., the notion of "neighborhood" in the Hamming sense may or may not be useful. It is also possible that there are a number of good but very different policies. Moreover, the coded policy space may have many local maxima and minima. Under such conditions, optimization algorithms tailored to unimodal or only slightly multimodal systems may not be effective.

[2] It is also possible to include policies which have probabilistic components as well [2, 3]. For example, the decision specified by the binary policy might be adopted with probability $p < 1$.

We therefore will explore the coded and uncoded policy space using the probabilistic, robust optimization technique of genetic algorithms [7]. Genetic algorithms (GAs) often perform well in situations where little is known about the search space [8]. We will illustrate our method on the well-known two class M/M/C/C queueing system and will find compact good call admission policies. For smaller problems where the optimal policy can be found by traditional means, our policy search methods find near-optimal policies with strong morphological resemblance to the known optimal. Our methods are successful for larger problems where other methods are impractical. Moreover, policies will suggest structural properties amenable to mathematical analysis.

We therefore suspect that our method might be used to both quickly find good policies, and in addition, provide informed analytic insight. That is, the good policies found are not *ad hoc* as might be the case with human analytic supposition, but based on the search over many different solution structures conducted by the genetic algorithm.

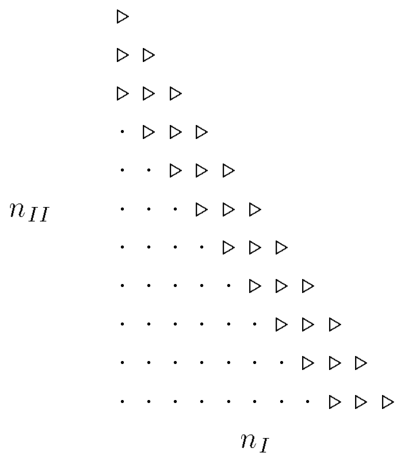
2 BACKGROUND

2.1 System State

We will consider systems that can be described by a finite-dimensional Markovian state vector with a finite number of states. A policy is simply a mapping of the system state to a decision rule.

However, a policy is not used to enforce the state space constraints. The policy acts as a supplement to the physical system. For example, for a queue with capacity C customers, a valid policy may allow admission of a new arrival even when the queue has C customers. Of course, a customer that arrives when the system is full will be blocked. However, it is the system, as opposed to the policy, that blocks the new arrival. As a result, even when the actual system is subject to a set of intricate constraints, the policy space remains unconstrained.

Consider then a system, S, with N dimensional state vector $s = (s_1, \ldots, s_N)$. For example, the state might simply represent the number of customers of each class in the system. A simple case is the two-class, C-trunk model from telephony where two types of traffic (customers) must share a fixed number of telecommunications channels (say one user per channel) [9]. A server reservation policy for this system can be depicted graphically in Figure 1. As we have indicated, when the system is full the policy admits class I calls even though the system will block all arrivals.



$\triangleright$ *admit only additional class I calls*
$\wedge$ *admit only additional class II calls*
$\cdot$ *admit all classes*
$*$ *admit no additional calls of any class*

Figure 1: *Server reservation policy with 3 servers reserved for class I in a 10-server system.*

2.2 Searching the Policy Space

2.2.1 Markov Decision Processes

For a system described by a finite state Markov chain, the call admission problem can be formulated as a Markov Decision Process, (MDP) a policy optimization technique that can be applied to discrete and continuous time Markov processes [4, 5]. In this subsection, we outline the MDP method and its limitations. In subsection 4.1, we give a detailed description of the MDP formulation of a two class call admission problem.

The MDP model assumes a Markov system and a set of state-dependent controls. A policy is simply a specification of controls. Fixing a policy fixes the state transition matrix as well as a set of rewards that depend on both system state and state transitions.

The two basic methods are policy iteration and value iteration. Under policy iteration, each iteration is a two step procedure. In the first step, the current policy is evaluated. In the second step, this evaluation is then used to find an improved policy. When no such improved policy is found, optimality is assured. For a process that has only a finite number of states and policies, the procedure is guaranteed to terminate in a finite number of iterations. Typically, the number of iterations is very small. The difficulty with policy iteration is that each iterative step requires solving a system of N linear equations where N is the number of *states* in the system. Under value iteration, each possible decision for each possible state must be evaluated. For the call admission problem, this requires $O(N)$ evaluations per iterative step. In this case, the number of iterations tends to be large and also tends to grow rapidly with N. Although value iteration permits the consideration of much larger problems than does policy iteration, the computational effort on a typical workstation still becomes excessive when the number of states is on the order of 10^5.

For more realistic systems, particularly multidimensional systems, the computational burden of either iterative method is simply too great. There is no guarantee that the MDP will produce a policy with a structure that can be described concisely. That is, a policy found by MDP typically must be specified by a complete enumeration of the decisions for every possible state. For large systems, policies found by MDP will be impractical simply because the policy description is too long. Nonetheless, the primary virtue of MDP here is that it allows us to calibrate our probabilistic search procedures for smaller problems.

2.2.2 Genetic Algorithms

Detailed descriptions of genetic algorithms, and the types of problems to which they are applicable can be found elsewhere [7, 10]. Here we simply outline the basic procedures.

We will consider coding of admission policies as binary strings in some detail shortly. For now, however, we simply assume that any binary string of length N can be used to describe one of 2^N unique admission policies. Application of a genetic algorithm to search the space of 2^N policies requires that we form an initial set *population* of different binary strings corresponding to different policies. These strings are usually chosen at random. However, if something is known about the optimal policy structure, specific classes of strings might be used.

Carrying out the biological analogy, if we consider an individual policy as an *organism*, then a string which specifies the policy is composed of its *genes*. The objective function is evaluated for each of these policies and *parent* organisms are selected probabilistically according to their relative *fitness* as given by the objective function. A simple often-used method of selecting parents is to assign each policy a probability proportional to its fitness. Specifically, we let the probability of selecting a given policy as a parent be

$$p_i = \frac{f_i}{\sum_j f_j} \tag{1}$$

where f_i is the fitness of policy i. Then, parents are randomly selected (with replacement) from this distribution akin to spinning a roulette wheel.

The parent organisms are then *mated* to generate a new set of *offspring* organisms. The mating procedure consists of splicing randomly chosen sections from parent gene strings to form offspring. In the case of two parent strings 000000000 and 111111111, a mating might produce the two strings 111000000 and 000111111. The individual bits in the offspring gene strings are then changed with low probability corresponding to *mutation*. For the above example, the two offspring might become 111000100 and 010111111 after mutation. There are also a number of different ways to perform this operation other than the anthropocentric *two-parent* $\rightarrow$ *child* method. In this work however, we only use pairwise mating.

As in natural evolution, the more fit the policy, the more likely that parts of its genome will be propagated through subsequent generations of policies. In fact, successful parts of the genome increase their representation geometrically in subsequent generations [7]. The basic idea is to optimize by natural selection using simple genetic operators.

3 POLICY CODING

Each of the Γ points in the state space must be associated with an admission decision. If D yes/no admission decisions must be made at each point, then the number of bits necessary to specify an admission policy is ΓD. The total number of specifiable policies, P, is then $2^{\Gamma D}$. For smaller P we can simply represent a policy directly by using this policy string. We define this policy coding as the *direct code*.

However, even for seemingly small systems, P can be very large and it might be virtually impossible to search the whole policy space. For slightly larger systems, the state space is so large that describing any particular policy on a state-by-state basis consumes large amounts of memory.

Practical call admission policies usually avoid this problem by ignoring the multiplicity of possible policies as well as the size of a bit-by-bit policy representation. Specifically, heuristics are used to focus on the set of states for which a policy disposition significantly affects the objective function. The fact that this approach is often successful is intriguing. It seems to suggest that it is possible, if not probable, that there exist good policies with compact representations. A compact representation implies in turn that not all policies strings need be considered, but rather, only *classes* of policy strings.

We therefore desire methods of policy representation that can range smoothly from a manageable number of classes to complete specification of any arbitrary policy string. That is, we desire compact codings of "good" policy strings but must be able to code (in theory) any arbitrary string. There are many different ways to code finite length binary strings and there is little reason to choose one over another unless something is known about the characteristics of "good" strings. We consider two general codings below.

3.1 Block Codes

One method of coding a policy is to build a table of 2^G entries with each entry consisting of a distinct K-bit pattern. These K-bit patterns can be concatenated like beads on a string to form full policy strings of length ΓD. We can then specify the policy using a sequence of table indices (G bits each) rather than the sequence of K-bit patterns directly. With $G < K$, we effect a compression of the policy representation. The K-bit *codewords* in the table might be chosen at random or by heuristic and then later refined. With Γ the number of states, a sequence of $R = D\Gamma/K$ codewords would be sufficient to describe the policy where D is the number of bits necessary to specify a decision.

For example, with $K = 4$, $G = 2$, and a codebook of

index	codeword
00	0001
01	0011
10	0111
11	1111

the codeword index sequence string of 0110111100 would expand to

0011011111111111110001

This method of policy representation can then be characterized as follows:

- Codebook size: $K2^G$ bits
- Code length: $\frac{D\Gamma G}{K}$
- Compression factor: $\alpha = \frac{K}{G}$

Of course, the number of different policies which can be specified is reduced proportionally. Specifically, for a given code book, the total number of representable policies is

$$P_c = 2^{RG} = \left(2^{D\Gamma}\right)^{\frac{1}{\alpha}} \tag{2}$$

Also, given fixed compression ratio, α, block codes still grow linearly with the size of the policy to be coded. Thus, block codes are appropriate for moderate sized problems, but become increasingly unwieldy as the statespace becomes very large.

3.2 Program Codes

One could take an algorithmic approach to policy coding as well. A policy renders a decision based upon a set of state variables. One might consider these states as inputs to a computer program whose output will be the decision. The policy coding would therefore be the bits specifying the program. If the computer is universal [11], then any policy can be represented given sufficient program length and execution time.

Of course, there are usually limits on the amount of complexity we are willing to devote to a given problem in terms of time, effort and computing resources. Thus, we generally seek compact algorithmic representations of policies. Server reservation [12] is an example of a computationally simple and often-used call admission method. The server reservation algorithm is particularly simple. Consider a C-server system with s_1 and s_2 the number of class 1 and class 2 customers respectively. A program which implements a channel reservation policy would take the form,

```
if (s1 + s2 < R)
   admit all classes
else
   admit only class I
```

Any such program can be reduced to a set of (usually) binary machine instructions and such binary digits constitute a policy code. The main utility of this general method is its relative insensitivity to problem size. For example, the only varying quantity for the channel reservation program is the number of bits necessary to represent s_1 and s_2: and this varies as $\log C$.

Of course, there are practical issues. A universal computer is assumed to have infinite time and memory with which to complete a computation, but we are required to limit the program execution time and the memory used. With these restrictions, the problem is changed slightly. The computer is now a finite state machine (FSM) which is allowed a maximum number of transitions per policy disposition. What we

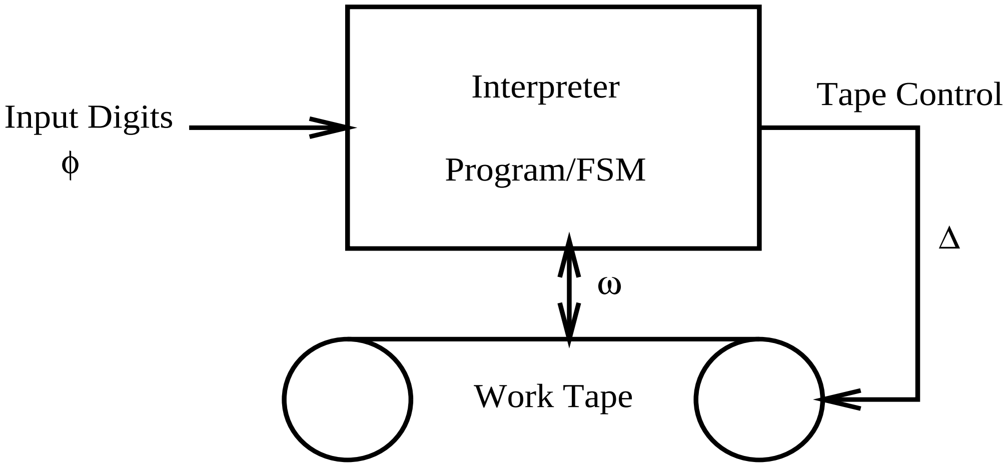


Figure 2: *Turing machine structure. Please, see text for further description.*

seek is the best FSM of a complexity specified by the program string length, which produces an answer within a specified time. This is similar in spirit to evolutionary programming as described in [13] where FSM descriptions are sought as solutions to a particular task. The important difference here is the near universality of the computer for which a program is sought. As such, our search is more akin to modern methods such as [14] where an explicit program for a particular programming language is sought.

For simplicity and generality, we use in our study an approximately classical Turing machine structure [11, 15] as shown in Figure 2. A classical Turing machine consists of a control unit (FSM), an infinite capacity read/writable storage medium (work tape) and external inputs. The FSM actions such as moving/reading/writing the work tape in response to work tape and external input are completely specified by the program. Our machine is only approximately classical since the work tape is assumed finite. However, its length is purposely chosen very long so that the primary limitation on the computation is the allowed execution time and program length.

The input, ϕ, is read along with work tape values, ω. The program specifies the transitions within the FSM. Specifically, for each program state (instruction) S_n, input ϕ_n and work tape value ω_n combination, the FSM provides the address of the next instruction S_{n+1}, an output ω_{n+1} and moves the tape pointer by Δ_n. A single instruction is designated as the terminal or *halt* instruction; execution of this instruction halts execution.

	(ω, ϕ)			
Instruction	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
0	$(s_{01}, \omega_{01}, \Delta_{01})$	$(s_{02}, \omega_{02}, \Delta_{02})$	$(s_{03}, \omega_{03}, \Delta_{03})$	$(s_{04}, \omega_{04}, \Delta_{04})$
1	$(s_{11}, \omega_{11}, \Delta_{11})$	$(s_{12}, \omega_{12}, \Delta_{12})$	$(s_{13}, \omega_{13}, \Delta_{13})$	$(s_{14}, \omega_{14}, \Delta_{14})$
2	$(s_{21}, \omega_{21}, \Delta_{21})$	$(s_{22}, \omega_{22}, \Delta_{22})$	$(s_{23}, \omega_{23}, \Delta_{23})$	$(s_{24}, \omega_{24}, \Delta_{24})$
HALT				

Table 1: *Program table for 4-instruction program with single bit external and work tape inputs. Triplets* $(s_{ij}, \omega_{ij}, \Delta_{ij})$ *represent next program instruction, output and tape increment given the current instruction is* i *and the current input/work tape combination is* j*:* $s_{ij} \in \{0, 1\}$*,* $\omega_{ij} \in \{0, 1\}$ *and* $\Delta_{ij} \in \{-1, 1\}$*.*

Assume there are $\mathbf{P}$ program statements and let $\mathbf{I}$, $\mathbf{O}$ and $\mathbf{T}$ be the number of bits specifying the input, work tape output and tape pointer increments respectively. The program is therefore a binary string of length

$$\mathcal{L} = 2^{\mathbf{I}+\mathbf{O}}(\mathbf{P} - 1)(\log \mathbf{P} + \mathbf{O} + \mathbf{T})\text{bits} \tag{3}$$

Since $\mathcal{L}$ grows rapidly with $\mathbf{I}$ and $\mathbf{O}$, we restrict them to $\mathbf{I} = \mathbf{O} = 1$ and assume the inputs/outputs are read/written sequentially one bit at a time. We also set $\mathbf{T} = 1$ so that $\mathcal{L} = 4(\mathbf{P}-1)(\log \mathbf{P}+2)$. A symbolic table for a program with $P = 4$ instructions and single bit inputs and outputs is shown in Table 1.

Using the FSM, a call admission policy is obtained in the following way. For each system state s, a binary encoding of s is used as input. Each bit of s is presented sequentially (recall, $\mathbf{I} = 1$), one per program step, until all bits have been seen by the FSM. The FSM executes its program until either it terminates or until a time limit is reached. The policy disposition, the set of rules for call admission for the state s, is composed of the last D bits of output. To specify a complete policy for a system, this procedure would be executed for every state s. The policies depicted in Figures 8 and 9 are found by this method. However, to implement a program coded policy in an actual system, the entire policy need not be generated since the Turing machine can be used to make state based call admission decisions online.

4 RESULTS

In this section, we examine call admission policy search techniques for the two class M/M/C/C queue. We consider the performance of policies found by a genetic algorithm using the three policy coding methods of section 3. In addition, we find optimal call admission policies using MDP policy iteration.

In the two class M/M/C/C queue, two types of customers compete for the same set of C servers. Customer type 1 has Poisson arrivals of intensity λ_1 and exponential service rate μ_1. Likewise, customer type 2 has intensity λ_2 and service rate μ_2. At most C customers are allowed in service at any one time. If a customer arrives and all servers are busy, that customer is said to be blocked and immediately exits the system. In addition, the arrivals may also be blocked by a call admission policy κ. Under policy κ, each customer type has an associated blocking probability, $P_{B_1}^{\kappa}$ and $P_{B_2}^{\kappa}$. We will use the average blocking criterion of

$$p^{\kappa} = \frac{\lambda_1 P_{B_1}^{\kappa} + \lambda_2 P_{B_2}^{\kappa}}{\lambda_1 + \lambda_2} \tag{4}$$

Using MDP policy iteration, the optimal p^{κ} will be iteratively calculated. When evaluating a policy κ found by a genetic algorithm, we will estimate p^{κ} via Monte-Carlo simulation.

4.1 MDP Policy Iteration

Using the continuous time Markov process model of [4], we now describe the use of policy iteration for the two class M/M/C/C queue with the average blocking probability criterion (4). We will denote the system state by $\bar{n} = (n_1, n_2)$ where n_i denotes the number of class i customers in the queue. Our controls will permit us to admit or deny service to any new customer. That is, for every state $\bar{n}$, the policy κ must specify the controls $\kappa_1(\bar{n})$ and $\kappa_2(\bar{n})$ where

$$\kappa_i(\bar{n}) = \begin{cases} 1 & \text{admit a class } i \text{ arrival in state } \bar{n} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Note that since departures are never blocked, the empty state is reachable from every state under every call admission policy. Thus, the Markov chain always consists of a single communicating class along with a possible collection of transient states.

Since all customers that are admitted to the queue eventually depart, minimizing the average blocking probability is equivalent to maximizing the average call completion rate

$$g^{\kappa} = (\lambda_1 + \lambda_2)(1 - p^{\kappa}) = \lambda_1(1 - P_{B_1}^{\kappa}) + \lambda_2(1 - P_{B_2}^{\kappa}) \tag{6}$$

To find the policy κ that maximizes the average call completion rate, we associate

```
         ▷
         ▷ ▷
         ▷ ▷ ▷
         ▷ ▷ ▷ ▷
         ▷ ▷ ▷ ▷ ▷
n_II     ▷ ▷ ▷ ▷ ▷ ▷
         ▷ ▷ ▷ ▷ ▷ ▷ ▷
         ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
         · · · · ▷ ▷ ▷ ▷ ▷
         · · · · · · · ▷ ▷ ▷
         · · · · · · · · · ▷ ▷
                  n_I
```

$\triangleright$ *admit only additional class I calls*
$\wedge$ *admit only additional class II calls*
$\cdot$ *admit all classes*
$*$ *admit no additional calls of any class*

Figure 3: *MDP derived optimal policy for* $C = 10$*,* $\lambda_1 = 10$*,* $\lambda_2 = 1$ $\mu_1 = 2.5$ *and* $\mu_2 = 0.25$*. Performance:* $p = 0.0753$.

with each state $\bar{n}$ a reward rate

$$q_{\bar{n}} = n_1\mu_1 + n_2\mu_2 \tag{7}$$

equal to the customer departure rate of state $\bar{n}$, For a particular policy κ, the total expected reward in time t starting from state $\bar{n}$ is $v_{\bar{n}}^{\kappa}(t)$ which is shown [4] to have the asymptotic form $v_{\bar{n}}^{\kappa}(t) = g^{\kappa}t + v_{\bar{n}}^{k}$.

To completely specify the policy iteration, we first define the constraints imposed by the system. We denote the set of feasible states by

$$S = \{\bar{n} \geq 0 | 0 \leq n_1 + n_2 \leq C\} \tag{8}$$

Also, we denote the unit vectors $(1,0)$ and $(0,1)$ by $\bar{e}_1$ and $\bar{e}_2$ respectively. For a system with admission policy κ, the transition rates from state $\bar{n}$ to neighboring states $\bar{n}'$ are

$$a_{\bar{n}\bar{n}'}^{\kappa} = \begin{cases} \lambda_i\kappa_i(\bar{n}) & \bar{n}' = \bar{n} + \bar{e}_i \in S, \quad i = 1,2 \\ n_i\mu_i & \bar{n}' = \bar{n} - \bar{e}_i \in S, \quad i = 1,2 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Given a policy κ, each policy iteration is a two step procedure. In the first step, the current policy κ is evaluated by solving the set of equations

$$g^{\kappa} = q_{\bar{n}} + \sum_{\bar{n}' \neq \bar{n}} a^{\kappa}_{\bar{n}\bar{n}'} \left(v^{\kappa}_{\bar{n}'} - v^{\kappa}_{\bar{n}} \right) \quad (\bar{n} \in S) \tag{10}$$

$$v^{\kappa}_{(0,0)} = 0 \tag{11}$$

In the second step, the evaluation of $v^{\kappa}_{\bar{n}}$ is used to find an improved policy κ' such that for every state $\bar{n}$, κ' maximizes

$$q_{\bar{n}} + \sum_{\bar{n}' \neq \bar{n}} a^{\kappa'}_{\bar{n}\bar{n}'} \left(v^{\kappa}_{\bar{n}'} - v^{\kappa}_{\bar{n}} \right) \tag{12}$$

This maximization step can be expressed simply as

$$\kappa'_i(\bar{n}) = \begin{cases} 1 & v^{\kappa}_{\bar{n}+\bar{e}_i} > v^{\kappa}_{\bar{n}} \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Note that when $\bar{n}$ is a state such that $n_1 + n_2 = C$, the decision rule (13) is undefined; however, in such a state it doesn't matter what policy is specified because the system blocks the new arrival no matter what the policy. When the improved policy κ' is unchanged from the original policy κ, optimality is assured; otherwise we repeat the two step procedure using policy κ'.

Some intuitive feel for characteristics of the optimal policy may be gained by considering the case where the load (ratio of arrival rate to service rate; $\lambda_i/\mu_i = \rho_i$) offered by each customer is equal, but the service rate of one customer type (say type 1) is much slower than the other (say type 2). One can see that if the system is allowed to fill with the slower service rate customers, there will be periods of very high blocking for the more abundant (in time) faster service rate customers.

This propensity is offset somewhat by times when the system is relatively devoid of slower service customers. However, owing to the strong nonlinearity of Erlang blocking rate with queue size [16], and that many more type 2 customers than type 1 request service during a given period, a high blocking rate for type 2 customers severely degrades overall system performance. Thus, it might be advisable to deny admission to slower service rate customers even when the system is not full.

If we let $C = 10$, $\lambda_1 = 1$, $\lambda_2 = 10$ $\mu_1 = .25$ and $\mu_2 = 2.5$ we see that the MDP derived optimal policy, shown in Figure 3, bears out these suspicions. For larger systems ($C = 50$) we were unable to compute the optimal policies owing to inadequate computational resources. For very large systems, the problem becomes infeasible even using the largest computers.

4.2 Genetic Algorithms and the Two Class M/M/C/C Queue

As indicated above, for our GA tests, the standard estimation error of the average blocking probability p will vary from 4% at the start of a run to under 1% near the end. For each generation, g, the performance, p_i, of each policy in the population was obtained and the minimum, p_{min} found. The fitness function used was

$$f_i = \mathbf{e}^{-\alpha(p_i - p_{min})} \tag{14}$$

where $\alpha > 0$ is a scale constant which emphasizes ($\alpha > 1$) or de-emphasizes ($\alpha < 1$) deviations from the best population performance, p_{min}. As a general rule, we found it useful to increase α with increasing generation number, g (akin to an annealing schedule for simulated annealing).

4.2.1 Direct Policy Coding: $C = 10, 50$

Figure 4 shows snapshots in time of a typical population performance evolution for a system with $C = 10$ using the direct code of $\Gamma D = 132$bits. The initial policy bits were chosen randomly with probability 0.5. For comparison, the blocking performances of the *null policy* (admit if not full) was 0.121 and that of the known optimal policy was 0.075. Notice the relatively steady improvement of average population performance as a function of generation number.

For the $C = 50$ system, the best reservation policy provided blocking of 0.072 whereas the null policy achieves blocking of 0.216. The direct codes had genetic descriptions of $\Gamma D = 2652$ bits and required 800 generation to produce a policy with 0.31 blocking and 3600 generations to produce the policy with $p = 0.097$ shown in Figure 5.

4.2.2 Block Codes: $C = 50$

Codebooks with $K = 26$ and $G = 5$ (compression factor $\alpha = 5.2$) and sequence strings were chosen at random for each policy in the population. For the $C = 50$ system, the genetic policy code (codebook and sequence string) contained 1342 bits. The sequence string alone contained 510 bits. The best initial policy is shown in Figure 6.

The genetic algorithm was applied and after 800 generations produced the policy shown in Figure 7 was obtained. This was a substantial improvement in both time

$g = 100\ p = 0.127$

$$
n_{II} \quad
\begin{array}{ccccccccccc}
\triangleright \\
\cdot & \triangleright \\
\cdot & \wedge & \triangleright \\
\cdot & \cdot & \cdot & \triangleright \\
\cdot & \cdot & \wedge & \cdot & \triangleright \\
\cdot & \triangleright & \wedge & \cdot & \triangleright & \triangleright \\
\cdot & \cdot & \triangleright & \triangleright & \triangleright & \cdot & \triangleright \\
\cdot & \triangleright & \triangleright & \cdot & \cdot & \triangleright & \cdot & \triangleright \\
\wedge & \cdot & \triangleright & \cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \wedge & \triangleright & \cdot & \triangleright \\
\cdot & \triangleright & \cdot & \cdot & \triangleright & \cdot & \triangleright & \cdot & \cdot & \cdot & \triangleright
\end{array}
$$

n_I

$g = 200\ p = 0.095$

$$
n_{II} \quad
\begin{array}{ccccccccccc}
\triangleright \\
\wedge & \triangleright \\
\cdot & \cdot & \triangleright \\
\triangleright & \cdot & \wedge & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \triangleright \\
\cdot & \cdot & \cdot & \triangleright & \cdot & \triangleright \\
\cdot & \cdot & \cdot & \triangleright & \cdot & \cdot & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \triangleright & \cdot & \triangleright & \triangleright \\
\cdot & \cdot & \triangleright & \cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \triangleright & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \triangleright
\end{array}
$$

n_I

$g = 300\ p = 0.0776$

$$
n_{II} \quad
\begin{array}{ccccccccccc}
\triangleright \\
\cdot & \triangleright \\
\cdot & \cdot & \triangleright \\
\cdot & \wedge & \wedge & \triangleright \\
\cdot & \triangleright & \triangleright & \triangleright & \triangleright \\
\triangleright & \cdot & \cdot & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \triangleright & \cdot & \cdot & \triangleright \\
\cdot & \cdot & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \triangleright & \cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright \\
\triangleright & \cdot & \triangleright & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \triangleright
\end{array}
$$

n_I

$g = 400\ p = 0.0756$

$$
n_{II} \quad
\begin{array}{ccccccccccc}
\triangleright \\
\triangleright & \triangleright \\
\triangleright & \triangleright & \triangleright \\
\triangleright & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \triangleright \\
\triangleright & \triangleright & \triangleright & \triangleright & \cdot & \triangleright \\
\cdot & \cdot & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright \\
\cdot & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \triangleright & \cdot & \triangleright \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \triangleright & \triangleright & \triangleright & \triangleright & \triangleright
\end{array}
$$

n_I

Figure 4: *Direct Code policy structure for* $C = 10$ *as a function of generation number* g.

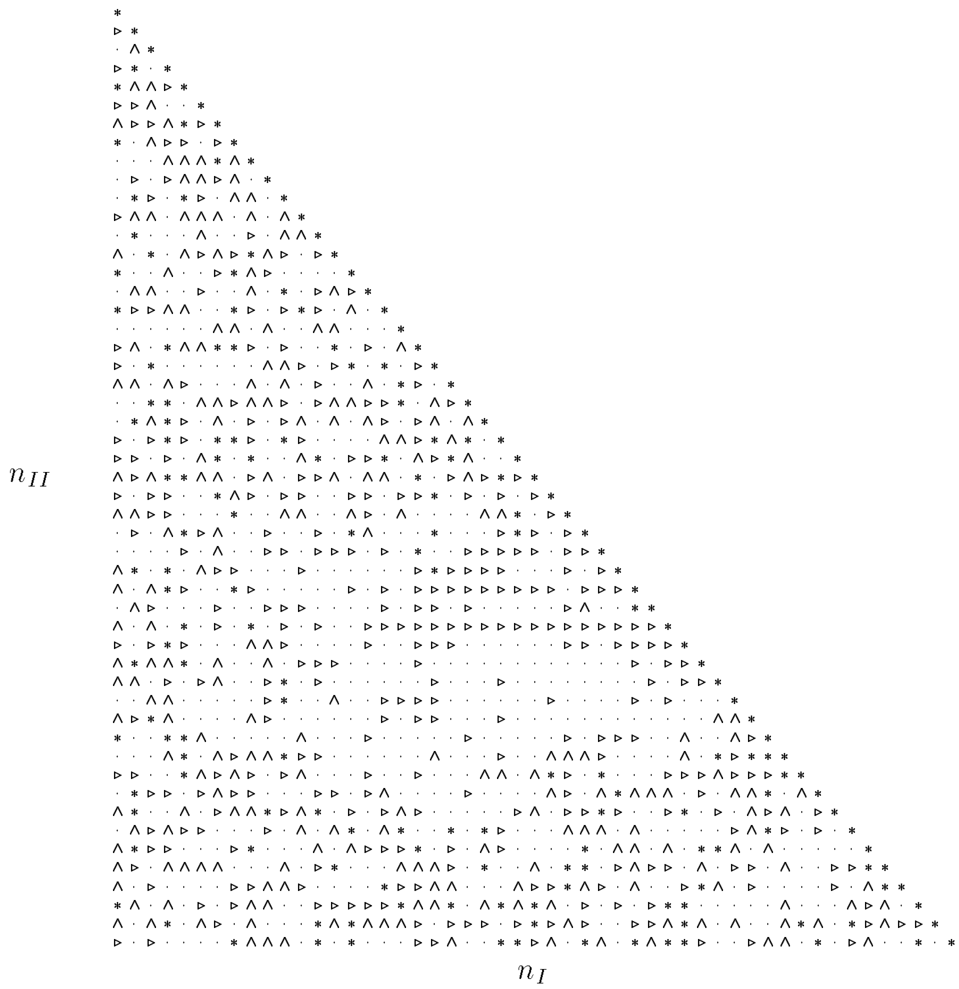


Figure 5: *Best direct code policy for* $C = 50$, $\lambda_1 = 30$, $\lambda_2 = 3$ $\mu_1 = 1$ *and* $\mu_2 = 0.1$. *Blocking performance:* $p = 0.097$.

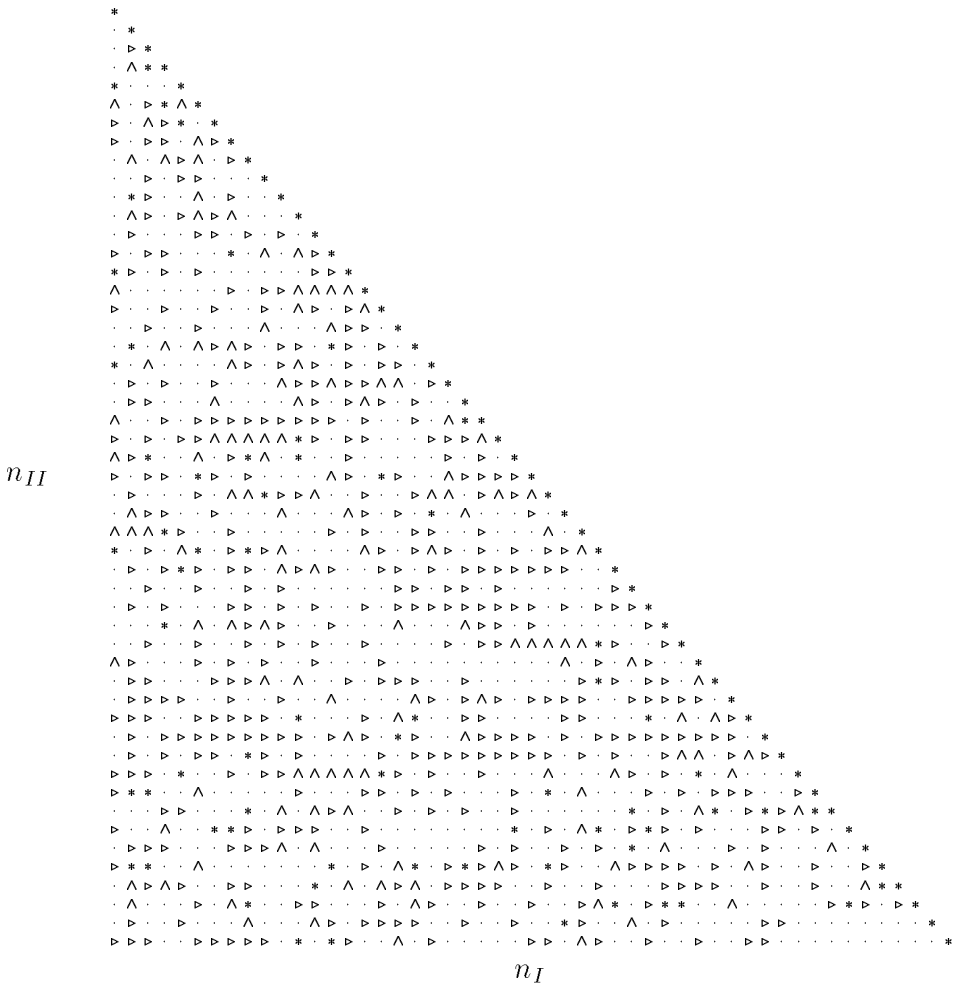


Figure 6: *Best initial block coded policy with $K = 26$ and $G = 5$. Blocking performance: $p = 0.191$.*

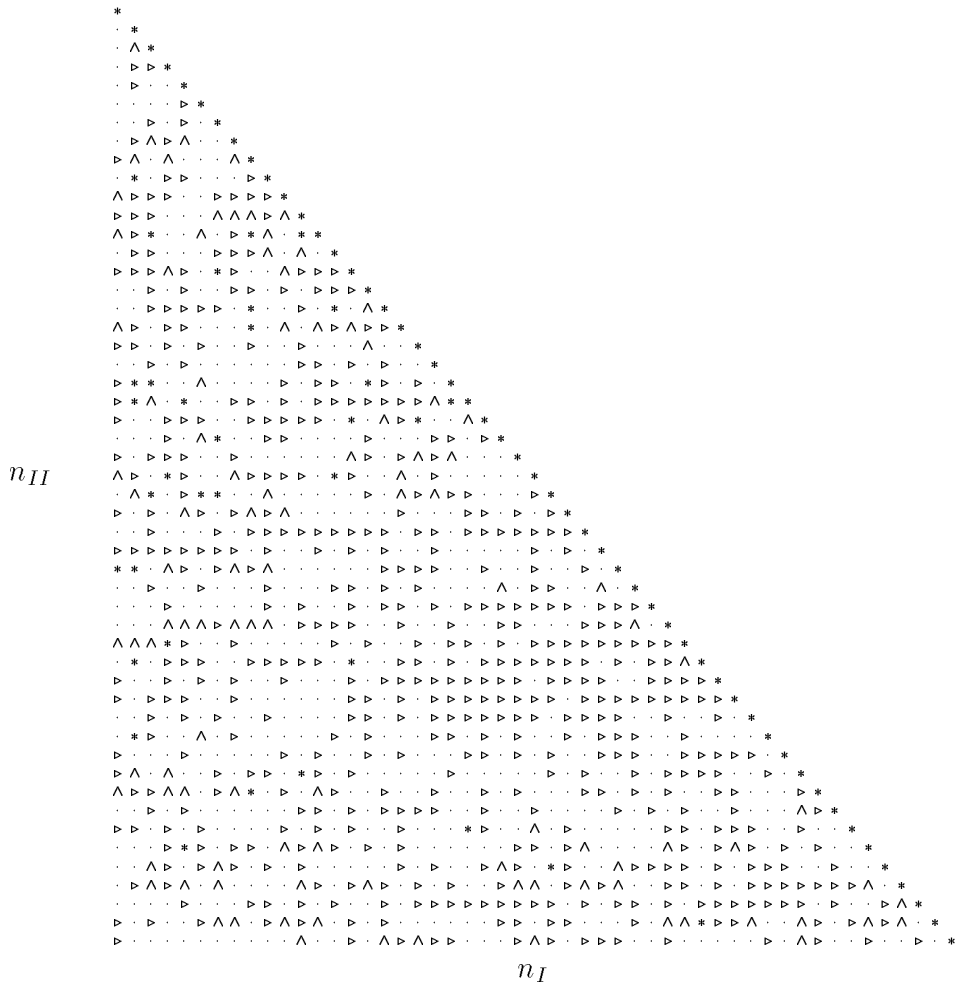


Figure 7: *Best block coded policy after 800 generations with* $K = 26$ *and* $G = 5$. *Blocking performance:* $p = 0.078$.

required and performance obtained over the direct code method.

4.2.3 Program Codes: $C = 10,\ 50\,\&\,500$

A population of randomly chosen strings were evaluated as programs for the simple computer described in section 3.2. We considered random populations of programs with $P = 4,\ 8,\ \&\,16$ instructions. The genetic string lengths for each program were $\mathcal{L} = 80,\ 192,\ \&\,448$ respectively, assuming two bits to specify the tape increment. The input to the program was the system occupancy and the output was the last D

C	Admit All	Best Known.	GA
10	0.121	0.0753 (optimal)	0.076
50	0.216	0.072 (reservation)	0.072
500	0.28	0.062 (reservation)	0.063

Table 2: *Table of results for program codes with $C = 10$, 50 and 500 and comparisons to the best server reservation policy and the admit all policy.*

bits prior to natural or forced termination.[3] As described in section 3.2, a genetic string was provided to specify the order in which input bits were read. The lengths of these strings were 24, 48, & 90 for $C = 10$, 50, & $C = 500$ respectively.

The best initial program codes had performances within 10% of the best server reservation policies. The results for $C = 10$, 50 and 500 are summarized in Table 2. As can be seen, the program codes achieved near-optimal performance and substantial improvements over the null policy in all cases. However, the morphology of the best program code policies differ significantly from server reservation policies as can be seen in Figures 8 and 9. The transition diagram for the best 4-instruction program for $C = 50$ is shown in Figure 10.

Most important, however, is that relatively few generations (typically under 40) were required to produce good policies.

5 DISCUSSION

Empirically, good policies for the two class M/M/C/C system seem to consist of two simply-connected regions. In the first region, all service requests are granted. In the second region, one class is favored over the other. This suggests that although the state space may be large, the complexity of good policies is relatively small. That is, only the boundary between regions needs to be specified rather than a policy disposition for each state.

This intuition was reinforced by our success in using simple coding procedures and genetic algorithms to find compact good policy descriptions. Also of particular note, the structure of policies found seem to suggest that simple restriction of occupancy for a single class provides good average blocking performance under fixed loading. This is an extremely simple policy; less complex than even server reservation,

[3] As mentioned in section 3.2, in order to avoid the problem of decidability, the number of instruction executions was limited and the program forced to terminate if this limit was exceeded.

$P = 4$ $p = 0.0.076$

▷ · ·
· · ·
▷ ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷
· · · · · ·
n_{II} · · · · · · ·
▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
· · · · · · · · · ·
· · · · · · · · · · ·

n_I

$P = 8$ $p = 0.076$

▷
· ·
· · ·
▷ ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷
n_{II} · · · · · ·
· · · · · · ·
▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
· · · · · · · · · ·
· · · · · · · · · · ·

n_I

$P = 16$ $p = 0.077$

▷
· ·
· · ·
▷ ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷
n_{II} · · · · · ·
· · · · ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷
· · · · · · · · ▷ ▷
· · · · · · · · ▷ ▷ ▷

n_I

Figure 8: *Best program code policy for* $C = 10$ *and 4, 8 and 16 program instructions. Blocking performance:* $p = 0.076$.

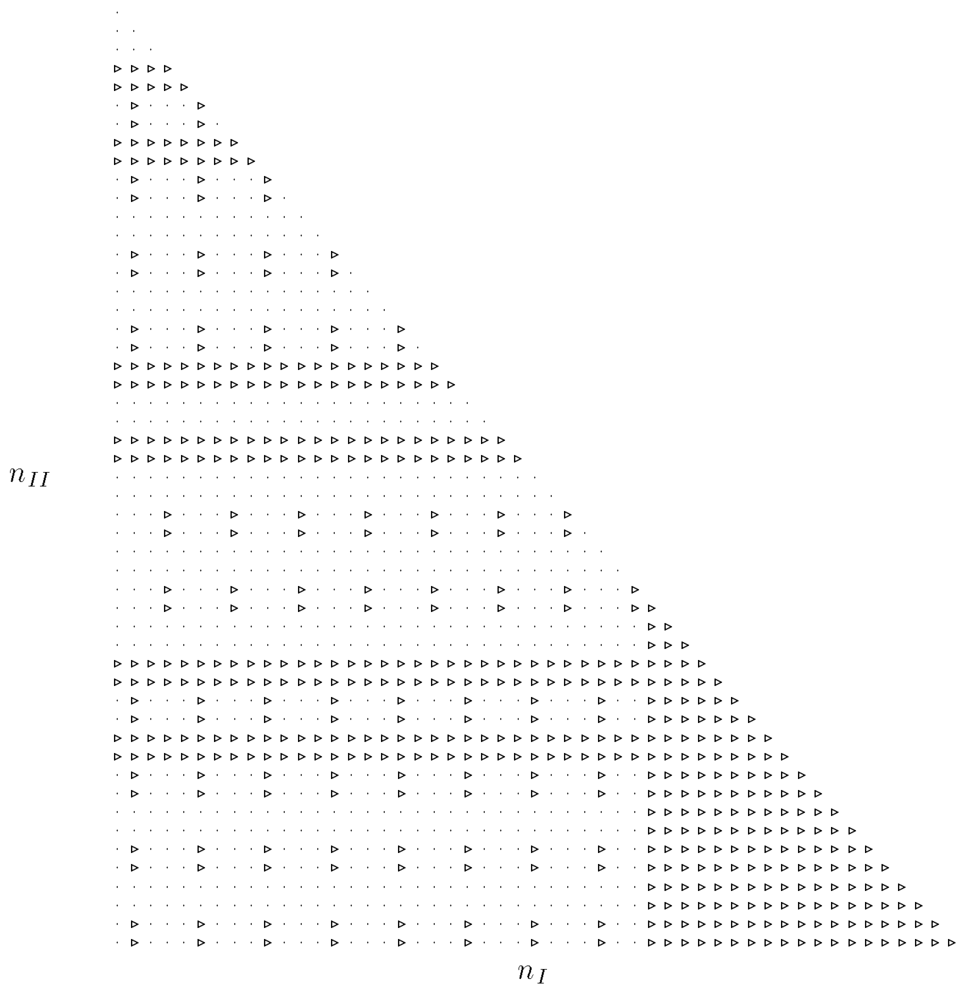


Figure 9: *Best program code policy for $C = 50$, 4-instruction program. Blocking performance: $p = 0.072$.*

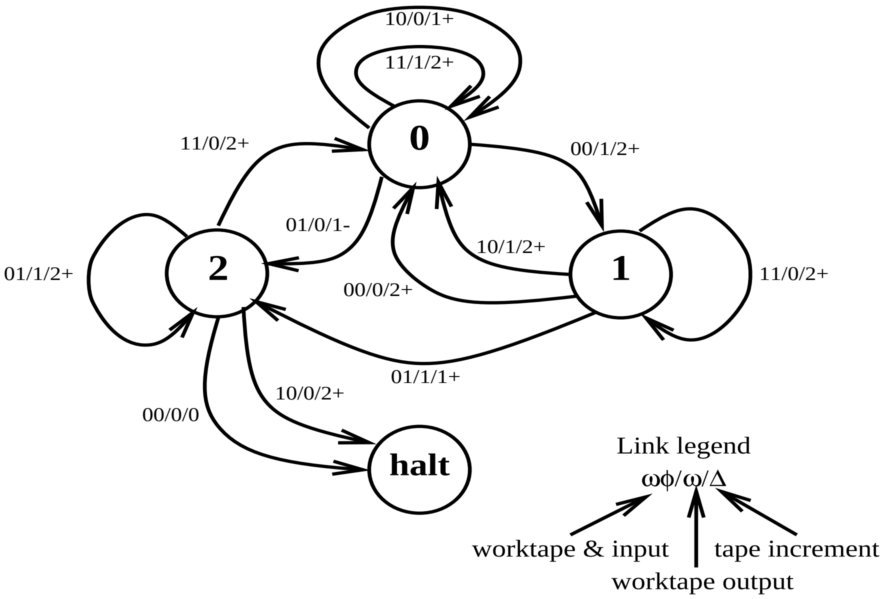


Figure 10: *Transition diagram for best 4-instruction program code with* $C = 50$. *Blocking performance:* $p = 0.072$.

Although this structural result is well-known for the M/M/C/C two class queueing system, it is interesting particularly for the program codes. Consider the general view. We have sought solutions to a problem by evaluating simple computer programs. The structure of the computer was chosen to be general and bears little if any direct resemblance to the problem. Nonetheless, the good programs found have suggested general structural properties of good solutions which could in principle lead to analytic results. This suggests that the method should be applicable to many problems, as both an aid to analytic intuition and a means of finding good solutions.

However, the primary drawback is the amount of computational effort necessary to produce good estimates of policy performance. In this case, the search for good call admission policies could have been made faster by more efficient and speedier methods of queueing system simulation and blocking estimation. Thus, application of pertinent prior work [17, 18] is currently being pursued.

References

[1] D. Tsang and K.W. Ross. Structural properties of optimal policies in an isdn circuit-switched access port. In *Proceedings, Computer Networking Symposium, Washington DC April 11-15*, pages 2–7, 1988.

[2] L. J. Cimini Jr. and G. J. Foschini. Distributed algorithms for dynamic channel allocation in microcellular systems. Submitted to IEEE Transactions on Communications (preprint 1991).

[3] L. J. Cimini Jr. and G. J. Foschini. Call blocking performance of distributed algorithms for dynamic channel allocation in microcells. Submitted to IEEE Transactions on Communications (preprint 1991).

[4] R. A. Howard. *Dynamic Programming and Markov Processes.* M.I.T. Press, 1960.

[5] H. C. Tijms. *Stochastic modeling and analysis : a computational approach.* Chichester; New York: Wiley, 1986.

[6] G.L. Nemhauser, A.H.G. Rinnooy Kan, and M.J. Todd, editors. *Optimization.* North-Holland, Elsevier Science Pub. Co., 1989.

[7] D. E. Goldberg. *Genetic Algorithms: in search, optimization and machine learning.* Addison-Wesley, 1989.

[8] L Davis, editor. *Simple Genetic Algorithms and the Minimal Deceptive Problem*, chapter 6. Morgan Kaufmann, 1987.

[9] G. J. Foschini, B. Gopinath, and J. F. Hayes. Optimum allocation of servers to two types of competing customers. *IEEE Transactions. Commun.*, COM-29:1051–1055, 1981.

[10] L Davis, editor. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann, 1987.

[11] J.M. Brady. *The Theory of Computer Science.* Science Paperbacks, 1977.

[12] A.I. Elwalid and D. Mitra. Effective bandwidth of general markovian traffic sources and admission control of high-speed networks. *IEEE/ACM Transactions on Networking*, 1(3):329, December 1993.

[13] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, Inc., 1966.

[14] J.R. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, 1992.

[15] K.R. Apt and J.W. De Bakker, editors. *"Three Hardest Problems, W.J. Savitch"*, page 111. Mathematical Centre Tracts, Amsterdam, 1976.

[16] L. Kleinrock. *Queueing Systems, Vol.1.* Wiley-Interscience, 1975.

[17] A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani. Algorithms for unboundedly parallel simulations. *ACM Transactions. Compute. Systems*, 9(3):201–221, 1991.

[18] M.R. Frater, T.M. Lennon, and B.D.O. Anderson. Optimally efficient estimation of rare events in queueing networks. *IEEE Transactions on Automatic Control*, 36(12):1395–1405, December 1991.